SARC: Sequential Prefetching in Adaptive Replacement Cache

Binny S. Gill and Dharmendra S. Modha
IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120
Emails: {binnyg,dmodha}@us.ibm.com

***Abstract*—** Sequentiality of reference is an ubiquitous access pattern dating back at least to Multics. Sequential workloads lend themselves to highly accurate prediction and prefetching. In spite of the simplicity of the workload, design and analysis of a good sequential prefetching algorithm and associated cache replacement policy turns out to be surprisingly intricate. As first contribution, we uncover and remedy an anomaly (akin to famous Belady's anomaly) that plagues sequential prefetching when integrated with caching. Typical workloads contain a mix of sequential and random streams. As second contribution, we design a self-tuning, low overhead, simple to implement, locally adaptive, novel cache management policy SARC that dynamically and adaptively partitions the cache space amongst sequential and random streams so as to reduce the read misses. As third contribution, we implemented SARC along with two popular state-of-the-art LRU variants on hardware for IBM's flagship storage controller Shark. On Shark hardware with 8 GB cache and 16 RAID-5 arrays that is serving a workload akin to Storage Performance Council's widely adopted SPC-1 benchmark, SARC consistently and dramatically outperforms the two LRU variants shifting the throughput-response time curve to the right and thus fundamentally increasing the capacity of the system. As anecdotal evidence, at the peak throughput, SARC has average response time of 5.18ms as compared to 33.35ms and 8.92ms for the two LRU variants.

I. Introduction

Moore's law indicates that processor speed grows at an astounding 60% yearly rate. In contrast, disks which are electro-mechanical devices have improved their access times at a comparatively meager annual rate of about 8%. Moreover, disk capacity grows 100 times per decade, implying fewer available spindles for the same amount of storage [1]. These trends dictate that a processor must wait for increasingly larger number of cycles for a disk read/write to complete. A huge amount of performance literature has focused on hiding this I/O latency for disk bound applications.

A. Caching

Caching is a fundamental technique in hiding I/O latency and is widely used in storage controllers (IBM Shark, EMC Symmetrix, Hitachi Lightning), databases (IBM DB2, Oracle, SQL Server), file systems (NTFS, EXT3, NFS, CIFS), and operating systems (UNIX variants and Windows). SNIA (www.snia.org) defines a cache as "A high speed memory or storage device used to reduce the effective time required to read data from or write data to a lower speed memory or device." We shall study cache algorithms for a storage controller wherein fast, but relatively expensive, random access memory is used as a cache for slow, but relatively inexpensive, disks. A modern storage controller's cache typically contains volatile memory used as a *read cache* and a non-volatile memory used as a *write cache*.

The effectiveness of read cache depends upon *hit ratio*, that is, the fraction of requests that are served from the cache without necessitating a disk trip (*miss*). We shall focus on improving the performance of the read cache that is increasing the hit ratio or equivalently minimizing the miss ratio. Typically, cache is managed in uniformly sized units called *pages*.

B. Demand Paging

Demand paging is a classical assumption used to study and design cache algorithms [2] wherein a page is brought in from the slower memory to the cache only on a miss. Demand paging precludes speculatively prefetching pages. Under demand paging, the only question of interest is: When the cache is full, and a new page must be inserted in the cache, which page should be replaced? The best, offline cache replacement policy is Belady's MIN that replaces the page whose next access is farthest in the future [3]. In practice, variants of LRU that replaces the least recently used page [4], [5], [6] are often used. For a recent detailed survey of cache replacement policies, see [7], [8].

C. Prefetching

A deeper dent can be made in I/O latency by speculatively prefetching or prestaging pages even before they are requested [9].

To prefetch, a prediction of likely future data accesses based on past accesses is needed. The accuracy of prediction plays an important role in reducing cache pollution and in increasing the utility of prefetching. The accuracy is generally dependent upon the amount of history that can be maintained and mined on-line, and on the stationarity of the access patterns.

A number of papers have focused on predictive approaches to prefetching, for example, [10] used relationship graph models, [11] used a prediction scheme based on classical information-theoretic Lempel-Ziv algorithm, [12] used a scheme based on associative memory, and [13] used a scheme based on partitioned context modeling. Commercial systems have rarely used very sophisticated prediction schemes. There are several reasons for this gap between research and practice. To be effective, sophisticated prediction schemes need extensive history of page accesses which is cumbersome and expensive to maintain for real-life systems. For example, a high-end storage controller may serve many tens of thousands of I/Os per second. Recording and mining such data online and in real-time is a non-trivial challenge. Furthermore, to be effective a prefetch must complete before the predicted request. This requires sufficient prior notice. However, long-term predictive accuracy is generally very low to begin with and becomes worse with interleaving of a large number of different workloads. A further degradation in predictive accuracy happens if the workloads do not exhibit stationarity which is the founding axiom behind many predictive approaches. Finally, for a disk subsystem operating near its peak capacity, average response time increases drastically with the increasing number of disk fetches. Thus, low accuracy predictive prefetching, which results in an increased number of disk fetches, can in fact worsen the performance.

In a well-known paper, [14] proposed an approach to significantly increase the predictive accuracy of prefetching by letting applications disclose their knowledge of future accesses to enable informed caching and prefetching. Building on [14], [15] utilized idle processor cycles while an application is stalled on I/O to speculatively pre-execute application's code to garner information about its future read accesses.

D. Sequential Prefetching

Sequentiality is a characteristic of workloads which reference consecutively numbered pages in ascending order without gaps. Sequential file accesses have been known at least since Multics [16]. Sequentiality naturally arises in video-on-demand, database scans, copy, backup, and recovery that may read a large number of files sequentially. Evidence of sequentiality abounds in database workloads, for example, [17], [18], [19], [20], [21]. The world of database and storage systems performance is largely dominated by benchmarks. The Transaction Processing Performance Council (TPC) benchmarks TPC-D [21], [22] and TPC-H exhibit a significant amount of sequentiality. Similarly, more recent Storage Performance Council (SPC)'s first benchmark SPC-1 is designed to be a mix of random and sequential workloads [23], [24]. The importance of sequential access patterns is further underscored by the fact that forthcoming SPC-2 benchmark will focus entirely on many concurrent sequential clients [25].

In contrast to sophisticated forecasting methods, detecting sequentiality is easy, requiring very little history information, and can attain nearly 100% predictive accuracy. An important trend is that sequential bandwidth of the disk has been increasing at a respectable annual rate of 40% while seek time have improving only at a meager annual rate of 8%. This implies that the additional cost of read-ahead on a seek is becoming progressively smaller. For these reasons, all UNIX variants [26], most modern day file systems [27], [28], databases such as DB2 [29] and Oracle [30], and high-end storage controllers such as IBM Shark [31], EMC Symmetrix all employ sequential detection and prefetching.

E. Our Contributions

We make several contributions towards design and implementation of a self-tuning, low overhead, high performance cache replacement algorithm for a real-life system that deploys sequential prefetching. For a seemingly much studied, well understood, and "simple" technique, design and analysis of a good sequential prefetching algorithm and associated cache replacement policy turns out to be surprisingly tricky. We summarize our main novel contributions, and outline the paper:

1) (Section II) For a class of state-of-the-art sequential prefetching schemes that use LRU, we point out an anomaly akin to Belady's anomaly that results in more misses when larger cache is allocated. We propose a simple technique to eliminate the anomaly.
2) (Section III) It is common to separate sequential data and random data into two LRU lists. We develop elegant analytical and empirical models for dynamically and adaptively trading cache space between the two lists with the goal of maximizing the overall hit ratio and, consequently, minimizing the average response time. Towards this end, we synthesize a new algorithm, namely, Sequential Prefetching in Adaptive Replacement Cache (SARC), that is self-tuning, low overhead, and simple to implement. SARC improves performance for a wide range of workloads that may have a varying mix of sequential and random data streams and may possess varying temporal locality of the random data streams.
3) (Section IV) Shark is IBM's flagship high-end storage controller [31]. We implemented SARC and two commonly used LRU variants on Shark (Model 800) hardware. On a widely adopted SPC-1 storage benchmark, SARC convincingly outperforms the two LRU variants. As anecdotal evidence, at the same throughput, SARC has average response time of 5.18ms as compared to 33.35ms and 8.92ms for the two LRU variants. [1]

II. Sequential Prefetching

The goal of sequential read-ahead is to keep the cache pre-loaded and ready with data for upcoming I/O operations, thus preventing potential misses. We outline a state-of-the-art sequential prefetching scheme whose variants are used in many commercial systems, for example, DB2 [29], Oracle [30], and Shark [32]. We also point out an anomaly akin to Belady's anomaly [3] that plagues such sequential prefetching schemes when used in conjunction with LRU-based caching. We also offer a simple and elegant remedy.

We manage the lists in the cache in terms of *tracks*, where a track is a set of up to eight 4K *pages*.

Before a sequential prefetching policy can be deployed, sequential access must be detected *somehow*. In many mainframe type applications, the client often indicates sequential accesses. In the absence of such hints, a necessary first step is to effectively detect sequences. Next, we discuss one such detection scheme; if desired, other schemes can be readily substituted without changing the essence of our analysis.

A. Sequential Detection

The goal of sequential detection is to automatically uncover a sequential access pattern. Such a pattern is not obvious to discover because of possible interleaving between various sequential streams and pauses between consecutive requests by a single stream, namely, the *think time* between requests.

The basic idea is to maintain a sequential detect counter with every track, and to use a parameter seqThreshold that can be set differently depending upon whether aggressive or conservative sequential detection is desired. The counter is updated as follows. On a hit or miss to track n whose counter is uninitialized, if track $n-1$ is present in the cache, then set the counter of track n to minimum of seqThreshold or one plus the counter value for track $n-1$. If track $n-1$ is not present in the cache, set the counter for track n to 1. Once initialized, the counter value for a track is not changed unless it reenters the cache after an eviction. When the counter equals seqThreshold, the track is termed as a *sequential track*. If a track gets designated as a sequential track on a miss, then we say that a *sequential miss* has occurred.

B. Synchronous and Asynchronous Prefetching

The simplest sequential read-ahead strategy is *synchronous prefetching* which on a sequential miss on track n simply brings tracks n through $n+m$ into the cache, where m is known as the degree of sequential read-ahead and is a tunable parameter [33]. As mentioned in the introduction, the additional cost of read-ahead on a seek is becoming progressively smaller. The number of sequential misses decrease with increasing m, if (i) all the read-ahead tracks are accessed and (ii) not discarded before they are accessed. Consider the well known OBL (one block look-ahead) scheme [20] that uses $m = 1$. This scheme reduces the number of sequential misses by $1/2$. Generalizing, with m track look-ahead, number of sequential misses decrease by $1/(m+1)$. To eliminate misses purely by using synchronous prefetching, m needs to become infinite. This is impractical, since prefetching too far in advance, will cause cache pollution, and will ultimately degrade performance. Also, depending upon the amount of cache space available to sequential data, not all tracks can be used before they are evicted. Hence, by simply increasing m, it is not always possible to drive the number of sequential misses to zero. The behavior of sequential misses for synchronous prefetching is illustrated in the left-hand panel of Figure 1.

To effectively eradicate misses, asynchronous prefetches can be used [29], [32]. An *asynchronous prefetch* is carried out in the absence of a sequential miss, typically, on a hit. The basic idea is to read-ahead a first group of tracks synchronously, and after that when a preset fraction of the prefetched group of tracks is accessed, *asynchronously* (meaning in the absence of a sequential miss) read-ahead next group of tracks, and so forth and so on. Typically, asynchronous prefetching is done on an *asynchronous trigger*, namely, a special track in a prefetched group of tracks. When the asynchronous trigger track is accessed, the next group of tracks is asynchronously read-ahead and a new asynchronous trigger is set. The intent is to exploit sequential structure to continuously stage tracks ahead of their access without incurring a single additional miss other than the initial sequential miss. A good analogy is that of an on-going relay race where each group of tracks passes the baton on to the next group of tracks and so on.

To summarize, in state-of-the-art sequential prefetching, synchronous prefetching is used initially when the sequence is first detected. After this bootstrapping stage, asynchronous prefetching can sustain itself as long as all the tracks within the current prefetched group are accessed before they are evicted. As a corollary, the asynchronous trigger track will also be accessed, and in turn, will prefetch the next group of tracks amongst which will also be the next asynchronous trigger.

C. Combining Caching and Prefetching for Sequential Data

So far, we have discussed two crucial aspects of sequential prefetching: (i) What to prefetch? and (ii) When to prefetch? We now turn our attention to the next issue, namely, management of prefetched data in the

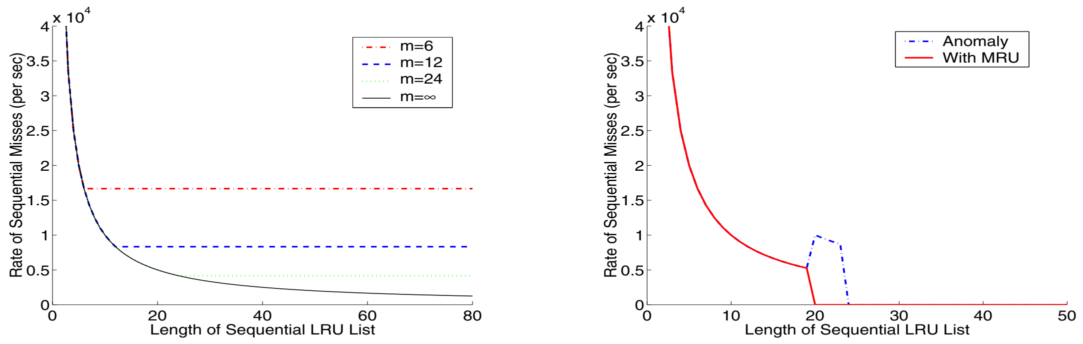

Fig. 1. Both of the above graphs were obtained via a simulation for a single sequential stream. The left-hand panel depicts behavior of sequential misses for synchronous prefetching. The lower most hyperbolic curve corresponds to an idealized situation with an infinite degree of read-ahead. It can be seen that for a fixed, finite degree of read-ahead m, sequential misses initially follow the hyperbolic curve, and, then become constant. Moreover, higher the degree of read-ahead, the smaller the minimum constant attained. The right-hand panel depicts behavior of sequential misses for synchronous+asynchronous prefetching with and without an anomaly. The anomalous curve initially follows the hyperbolic curve, but has a bump before going to zero.

cache. Given a fixed amount of cache, prefetched data cannot be kept forever and must eventually be replaced. Prefetching and caching are intertwined, and cannot be studied in isolation.

In practice, the most widely used online policy for cache management is LRU that maintains a doubly-linked list of tracks according to recency of access. In the context of sequential prefetching, when tracks are prefetched or accessed, they are placed at the MRU (most recently used) end of the list. And, for cache replacement, tracks are evicted from the LRU end of the list.

We now describe an interesting situation that arises when the above described synchronous+asynchronous prefetching strategy is used along with the LRU-based caching. Suppose that the asynchronous trigger track in a currently active group of prefetched tracks is accessed. This will cause an asynchronous prefetch of the next group of tracks. In an LRU-based cache, these newly fetched group of tracks along with the asynchronous trigger track will be placed at the MRU end of the list. The unaccessed tracks within the current prefetch group remain where they were in the LRU list, and, hence, in some cases, could be near the LRU end of the list. However, observe that these unaccessed tracks within the current prefetch group will be accessed before the tracks in the newly prefetched group. Hence, depending upon the amount of cache space available for sequential data, it can happen that some of these unaccessed tracks may be evicted from the cache before they are accessed resulting in a sequential miss. Furthermore, this may happen repeatedly, thus defeating the purpose of employing asynchronous prefetching. This observation is related to "Do No Harm" rule of [34] in the context of offline policies for integrated caching and prefetching.

In a purely demand-paging context, LRU is a *stack algorithm* [2]. Quite surprisingly, when LRU-based caching is used along with the above prefetching strategy, the resulting algorithm violates the stack property. As a result, when the amount of cache space given to sequentially prefetched data increases, sequential misses do not necessarily decrease. This anomaly is illustrated in the right-hand panel of Figure 1. As will be seen in the sequel, a stack property is a crucial ingredient in our algorithm.

At the cost of increasing sequential misses, both of the above problems can be hidden if (i) only synchronous prefetching is used or (ii) both synchronous+asynchronous prefetching are used and the asynchronous trigger is always set to be the last track in a prefetched group. The first approach amounts to a relapse in which we forego all potential benefits of asynchronous prefetching. The second approach is attractive in principle; however, if the track being prefetched is accessed before it is in the cache, then the resulting sequential miss will not be avoided. To avoid this sequential miss, in real life, the asynchronous trigger track is set sufficiently before the last prefetched track so that the next prefetched group arrives in cache before it is actually accessed.

Unlike the above two approaches, we are interested in fixing the anomalous behavior without incurring additional sequential misses. To this end, we now propose a simple to implement and elegant algorithmic enhancement. As mentioned above, in an LRU-based cache, the newly prefetched group of tracks along with the asynchronous trigger track in the current group of tracks are placed at the MRU end of the list. We propose, in addition, to also move all unaccessed tracks in the current group of tracks to the MRU end of the list. As can be seen in the right-hand panel of Figure 1, this enhancement retains all benefits of asynchronous prefetching while ridding it of its anomalous behavior.

III. SARC: Sequential Prefetching in Adaptive Replacement Cache

So far, we have focused primarily on designing an effective sequential prefetching strategy along with an LRU-based caching policy for housing sequential data. Typical workloads contain a mix of sequential and random streams. We now turn our attention to designing an integrated cache replacement policy that manages the cache space amongst both of these classes of workloads so as to minimize the overall miss rate.

A. Prior Work

A number of approaches have been proposed for integrated caching of sequential and random data. Almost all of them employ an LRU variant. One simple approach [20] that we call LRU-Top is to maintain a single LRU list for both sequential and random data, and whenever tracks are prefetched or accessed, they are placed at the MRU end of the list. The tracks are evicted from the LRU end of the list. Another approach [35], [36] that we call LRU-Bottom is to maintain a single LRU list for both sequential and random data; however, while random data is inserted at the MRU end of the list, sequential data is inserted near the LRU end. For another interesting LRU variant, see [37]. [1] suggested holding sequential data for a fixed amount of time, while [29] suggested giving sequential data a fixed, predetermined portion of the cache space. [34] studied offline, optimal policies for integrated caching and prefetching. In a recent work, [38] focused on general demand prepaging and noted that the amount of cache space devoted to prefetched data is "critical, and its ideal value depends not only on the predictor and the degree, but also on the main memory size, the application, and the reference behavior of the process." In other words, no strategy that is independent of the workload characteristics is likely to be universally useful.

In the context of demand paging, in addition to LRU, a number of cache replacement policies have been studied, see, for example, LFU, FBR, LRU-2, 2Q, MQ, LRFU, and ARC. For a detailed overview of these algorithms, see [7], [8]. Our context is different than that of these algorithms, since we are interested in integrated policies for caching and prefetching. Previously, [14] have considered adaptively balancing cache amongst three partitions: LRU cache, hinted cache, and prefetch cache. It is not clear how to efficiently extend the algorithm in [14] in presence of potentially a very large number of sequential streams.

Our algorithm, namely, Sequential Prefetching in Adaptive Replacement Cache (SARC), is closely related to–but distinct from–Adaptive Replacement Cache (ARC). In particular, the idea of two adaptive lists in SARC is inspired by ARC. There are several differences between the two algorithms: (i) ARC is applicable only in a demand paging scenario, whereas SARC combines caching along with sequential prefetching. (ii) While ARC maintains a history of recently evicted pages, SARC does not need history and is also simpler to implement.

B. Our Approach

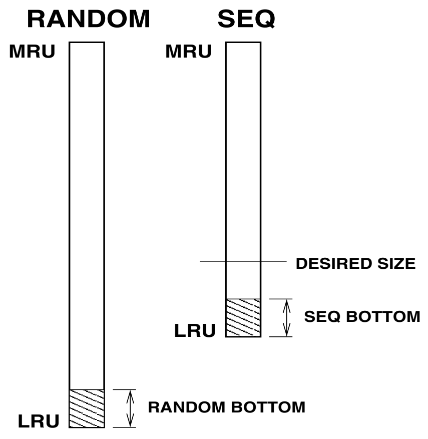

Fig. 2. SARC separates sequential and random data into two lists, and maintains a desired size parameter for the sequential list. The desired size is continually adapted in response to a dynamic, changing workload. Specifically, if the bottom portion of SEQ list is found to be more valuable than the bottom portion of RANDOM list, then the desired size is increased; otherwise, the desired size is decreased.

We shall develop an adaptive, self-tuning, low overhead algorithm that dynamically partitions the amount of cache space amongst sequential and random data so as to minimize the overall miss rate. Given the radically different nature of sequentially prefetched pages and the random data, it is natural to separate these two types of data. As shown in Figure 2, we will manage each class in separate LRU lists: RANDOM and SEQ. One of our goals is to avoid *thrashing* [34] that happens when more precious demand-paged random pages are replaced with less precious prefetched pages (cache pollution) or when prefetched pages are replaced too early before they are used.

A key idea in our algorithm is to maintain a desired size (see Figure 2) for SEQ list. The tracks are evicted from the LRU end of SEQ, if its size (in pages) is larger than the desired size; otherwise, the tracks are evicted from the LRU end of RANDOM. The desired size is continuously adapted. We now explain the intuition behind this adaptation.

Assuming that both the lists satisfy the LRU stack property (see Section II-C), the optimum partition is one that equalizes the marginal utility of allocating additional cache space to each list. We design a *locally* *adaptive* algorithm that starts from any given cache partitioning and gradually and dynamically tweaks it to gear it towards the optimum. As an important step towards this goal, we first derive an elegant analytical model for computing the marginal utility of allocating additional cache space to sequentially prefetched data. In other words, how does the number of sequential misses experienced by SEQ change as the size of the list changes. Similarly, as a second step, we empirically estimate the marginal utility of allocating additional cache space to random data. Finally, as the third step, if during a certain time interval the marginal utility of SEQ list is higher than that of RANDOM list, then the desired size is increased; otherwise, the desired size in decreased.

C. *Single Sequential Stream*

For simplicity, assume that there is only one request to each track. Multiple consecutive requests do not change the final algorithm in any way.

Every track in cache has a time stamp that is updated with the current time whenever the track is placed at the MRU position of either list. Let T denote the *temporal length*, that is, the time difference between the MRU and LRU time stamps of SEQ.

Let s_1 and s_1^a denote the rates of sequential misses of one stream when synchronous and synchronous+asynchronous prefetching, respectively, are employed.

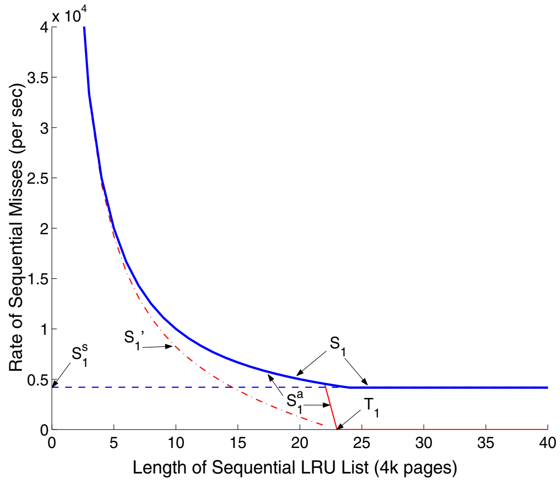

Fig. 3. Via simulation for a single sequential stream, we depict behavior of s_1 (rate of sequential misses for synchronous prefetching), s_1^a (rate of sequential misses for synchronous+asynchronous prefetching), and s_1' (an approximation to s_1^a). The hyperbolic curve s_1 flattens out at point (T_1, s_1^s).

Figure 3 displays the behavior of s_1 and s_1^a as the temporal length varies. As discussed in Section II-B, under synchronous prefetching, sequential misses or its rate is inversely proportional to the degree of read-ahead. If, however, the read-ahead are discarded before they are accessed, the *effective degree of read-ahead* decreases. Whenever he effective degree of read-ahead is less than the actual degree, it is proportional to the temporal length of the list. Hence, we have that

$$s_1 = \begin{cases} (\text{constant})/T & 0 \le T \le T_1 \\ s_1^s & T_1 < T, \end{cases} \quad (1)$$

where $s_1^s > 0$ is the smallest attainable rate of sequential misses (which is inversely proportional to the actual degree of read-ahead) and T_1 is the smallest temporal length that attains s_1^s. As previously discussed in Section II-B and depicted in Figure 3, s_1^a drops to zero when s_1 flattens out:

$$s_1^a = \begin{cases} s_1 & 0 \le T \le T_1 \\ 0 & T_1 < T. \end{cases} \quad (2)$$

For later use, we define the curve s_1' as:

$$s_1' = \begin{cases} s_1 - T(s_1^s)/(T_1) & 0 \le T \le T_1 \\ 0 & T_1 < T. \end{cases} \quad (3)$$

Observe that the curve s_1' distributes the discontinuity of s_1^a throughout the interval $[0, T_1]$.

D. *Marginal Utility of* SEQ

When we have multiple streams, their think times will in general be different. Also, in general, each client will have different number of accesses to each track before moving onto the next track. Fortunately, these differences do not enter our analysis below.

Let us suppose that there are ℓ streams. The parameter ℓ will not appear anywhere in our algorithm. Let s_i^a, $1 \le i \le \ell$, denote the rate of sequential misses of stream i for synchronous+asynchronous prefetching. Observe that these individual numbers are generally not easily observable. However, their sum

$$s = \sum_{i=1}^{\ell} s_i^a \quad (4)$$

is simple to observe in a real system.

Let L denote the length of the list SEQ in number of 4K pages. To compute the marginal utility of SEQ, we examine how the overall rate of sequential misses changes, namely, Δs, in response to a small change ΔL in L. By the stack property of SEQ, as L increases (respectively, decreases) s decreases (respectively, increases). Hence, $\Delta s/\Delta L$ is always negative. The marginal utility is measured by the magnitude of this quantity. We shall lower bound this negative quantity, and, in turn, upper bound the marginal utility.

$$\begin{aligned}
\frac{\Delta s}{\Delta L} &= \frac{\Delta T}{\Delta L}\frac{\Delta s}{\Delta T} \\
&\overset{(a)}{=} \frac{\Delta T}{\Delta L}\sum_{i=1}^{\ell}\frac{\Delta s_i^a}{\Delta T} \\
&\overset{(b)}{\geq} \frac{\Delta T}{\Delta L}\sum_{i=1}^{\ell}\frac{\Delta s_i'}{\Delta T} \\
&\overset{(c)}{=} \frac{\Delta T}{\Delta L}\sum_{i=1}^{\ell} I(T < T_i)\cdot\left(\frac{\Delta s_i}{\Delta T} - \frac{s_i^s}{T_i}\right) \\
&\overset{(d)}{=} \frac{\Delta T}{\Delta L}\sum_{i=1}^{\ell} I(T < T_i)\cdot\left(-\frac{s_i}{T} - \frac{s_i}{T}\left(\frac{T}{T_i}\right)^2\right) \\
&\overset{(e)}{>} \frac{\Delta T}{\Delta L}\sum_{i=1}^{\ell}\left(-2\frac{s_i^a}{T}\right) \\
&\overset{(f)}{=} -2\frac{\Delta T}{\Delta L}\frac{s}{T} \\
&= -2\frac{L\Delta T}{T\Delta L}\frac{s}{L} \\
&\overset{(g)}{=} -2\frac{s}{L}, \qquad (5)
\end{aligned}$$

where (a) follows from (4); (b) follows since, by definitions in (2) and (3), s_i' that is steeper than s_i^a throughout the continuous region $[0, T_i)$; (c) follows from (3) and I is the indicator function that takes value 1 or 0 depending upon whether $T < T_i$ or not; (d) follows from (1); (e) follows since $T < T_i$ and the indicator function $I(T < T_i)$ can be removed since when I is zero s_i^a is also zero; (f) follows from (4); and (g) follows since, in practice, there is linear relationship between L and T. In other words, as the length of the list increases (respectively, decreases) the time difference between the MRU and LRU time stamps of the list increases (respectively, decreases) proportionately for any given workload.

Let T_i, $1 \leq i \leq \ell$, denote the times at which various streams attain zero misses (see Figure 3). Mathematically, the above chain of inequalities is valid at all points in $[0, \infty)$ except at T_i, $1 \leq i \leq \ell$, since at T_i, $\Delta s_i^a/\Delta T$ is not defined. However, our choice of the approximating curve s_i' is such that it cleverly distributes the magnitude of the drop in s_i^a at T_i evenly throughout the interval $[0, T_i)$, and, hence, in practice, the inequalities are applicable. This approximation smoothes out the changes in Δs in relation to ΔL at the discontinuities and paves the way for designing a locally adaptive algorithm such as ours.

We now have the following bounds

$$-\frac{s}{L} \geq \frac{\Delta s}{\Delta L} \geq -2\frac{s}{L},$$

where the upper bound follows from (1) and (2) and the lower bound follows from (5). In other words, the marginal utility lies somewhere between s/L and $2s/L$, but closer to latter in practice. In this paper, we have chosen $2s/L$ to approximate the marginal utility of SEQ.

E. *Marginal Utility of* RANDOM

The RANDOM list contains essentially only demand-paged, non-sequential data. For demand paged data, LRU is a stack algorithm. In other words, increasing (respectively, decreasing) the length of the list leads to smaller (respectively, larger) number of misses.

Let r denote the rate of cache misses in RANDOM. To compute the marginal utility of RANDOM, we examine r changes, namely, Δr, changes in response to a small change ΔL in L. By the stack property, as L increases (respectively, decreases) r decreases (respectively, increases). Hence, $\Delta r/\Delta L$ is always negative. The marginal utility is measured by the magnitude of this quantity. Unlike SEQ list where an analytical model was necessary, for the RANDOM list the quantity $\Delta r/\Delta L$ can be estimated directly from real-time cache introspection.

We will monitor ΔL cache space at the bottom of RANDOM list (see Figure 2), and observe rate of hits Δr in this region. A bottom hit is an indication that a miss has been saved due to the cache space at the bottom of RANDOM. In other words, if cache space ΔL at the bottom was not allocated to RANDOM, then the rate of misses r would increase by Δr. We assume that $\Delta r/\Delta L$ is a constant in a small region (generally much smaller than even ΔL) at the bottom of the list where our locally adaptive algorithm is active. Hence, as a corollary, if a small amount of cache space were added to RANDOM, then the misses would decrease in proportion to $\Delta r/\Delta L$.

However, allocating more cache space to RANDOM will take away equal amount from SEQ, and, hence, needs to be carefully weighed. The optimum is attained when marginal utilities of both the lists are equalized.

F. *Equalizing Marginal Utilities*

Figure 2 depicts the structure of our algorithm. Fix the size of RANDOM bottom ΔL to a small constant fraction of the cache size. The essence of the algorithm is to compare the absolute values of

$$\frac{\Delta s}{\Delta L}\left(\approx \frac{2\cdot s}{L}\right) \text{ and } \frac{\Delta r}{\Delta L}.$$

If $(2s)/L$ is larger than the magnitude of $\Delta r/\Delta L$, then it is more advantageous to transfer cache space from the bottom of RANDOM to the bottom of SEQ and hence we increase the desired size of SEQ; otherwise, we decrease the desired size of SEQ.

So far, the time interval for sampling the rates Δr and s has not been fixed. The smaller the time interval the more adaptive the algorithm, while larger the time interval the smoother and slower the adaptation. Our algorithm implicitly selects a time interval based on cache hits. Thus, the rate of adaptation is derived from and adapts to the workload itself. Specifically, we select the time interval to be the time difference between two successive hits in the bottom of the RANDOM list. In this case, Δr is just a constant 1, and we measure s during the same interval. Thus, we now need to simply compare

$$\frac{2 \cdot s}{L} \text{ and } \frac{1}{\Delta L}, \text{ or, equivalently, } \frac{2 \cdot s \cdot \Delta L}{L} \text{ and } 1. \tag{6}$$

G. SARC

We now weave together the above analysis and insights into a concrete, adaptive (self-tuning) algorithm that dynamically balances the cache space allocated to RANDOM and SEQ data under different conditions such as different number of sequential/random streams, different data layout strategies, different think times of sequential/random streams, different cache-sensitivities/footprints of random streams, and different I/O intensities. The complete algorithm is displayed in Figure 4. SARC is extremely simple to implement and has virtually no computational/space overhead over an LRU-based implementation.

In our experiments, we have set ΔL to be 2% of the cache space. Any other small fraction would work as well.

In the algorithm, we will need to determine if a hit in one of the lists was actually in its bottom (see lines 6 and 12). Of course, one could maintain separate fixed sized bottom lists for both the lists. However, with an eye towards simplification and computational efficiency, we now describe a time-based approximation to achieve nearly the same effect. We now describe how to determine if the a hit in SEQ lies in its bottom ΔL. Let T_{MRU} and T_{LRU} denote the time stamp of the MRU and LRU tracks, respectively, in SEQ. Let L denote the size of SEQ in pages. Let T_{HIT} denote the time stamp of the hit track. Now, if

$$(T_{\mathsf{HIT}} - T_{\mathsf{LRU}}) \leq \frac{\Delta L}{L}(T_{\mathsf{MRU}} - T_{\mathsf{LRU}}),$$

then we say that a bottom hit has occurred. The same calculation can also be used for determining the bottom hits in RANDOM.

The algorithm uses the following constants: m (lines 18 and 32) denotes the degree of synchronous and asynchronous read-ahead; g (lines 18 and 32) denotes the number of disks in a RAID group; triggerOffset (line 49) is the offset from the end of a prefetched group of tracks and is used to demarcate the asynchronous trigger; LargeRatio (line 13) is a threshold, say, 20, to indicate that the ratio has become much larger than 1; and FreeQThreshold (line 52) denotes the desired size of the FreeQ. Shark uses RAID arrays (either RAID-5 or RAID-10) at the back-end. Our machine (see Section IV-B) was configured with RAID-5 (6 data disks + parity disk + spare disk) meaning that $g = 6$. RAID allows for parallel reads since logical data is striped across the physical data disks. To leverage this, setting m as a multiple of g is meaningful. We set $m = 24$. Finally, we chose triggerOffset to be 3.

The algorithm is self-explanatory. We now briefly explain important portions of the algorithm in Figure 4 in a line-by-line fashion.

Lines 1-3 are used during the initialization phase only. The counter seqMiss tracks the number of sequential misses between two consecutive bottom hits in RANDOM, and is initialized to zero. The variable desiredSeqListSize is the desired size of SEQ, and is initially set to zero meaning adaptation is shut off. The adaptation will start only after SEQ is populated (see lines 69-73). The variable adapt determines the instantaneous magnitude and direction of the adaptation to desiredSeqListSize.

Lines 4-50 describe the cache management policy. The quantity ratio in line 4 is derived from (6). Lines 5-10 deal with the case when a track in RANDOM is hit. If the hit is in the bottom portion of the RANDOM list (line 6), then seqMiss is reset to zero (line 7) since we are interested in number of sequential misses between two successive hits in the bottom of RANDOM. Line 8, sets the variable

$$\mathsf{adapt} = \frac{2 \cdot \mathsf{seqMiss} \cdot \Delta L}{L} - 1.$$

Note that in the steady state the rate of new tracks being added to any cache is the same as the rate of old tracks being demoted from the cache. This rate is an upper bound on the rate at which the size of either SEQ or RANDOM can change while keeping the total number of tracks in the cache constant. Since adapt is used to change the desiredSeqListSize, it is therefore not allowed to exceed 1 or be less than -1. Observe that by the test prescribed in (6), if adapt is greater than zero, then we would like to increase desiredSeqListSize, else we would like to decrease it. This increase or decrease is executed in line 70. Also, observe that when the inequality between the marginal utilities of SEQ and RANDOM is larger, the magnitude of adapt is larger, and, hence, a faster rate of adaptation is adopted, whereas when the two marginal utilities are nearly equal, adapt will be close to zero, and a slower rate of adaptation is adopted. Finally, observe that line 70 (which carries out the actual adaptation) is executed

```
INITIALIZATION: Set the adaptation variables to 0.

1:  Set seqMiss to $0$
2:  Set adapt to $0$
3:  Set desiredSeqListSize to $0$
```

```
CACHE MANAGEMENT POLICY:

Track $x$ is requested:

4:  Set ratio = $(2 \cdot \text{seqMiss} \cdot \Delta L)/\text{seqListSize}$

5:  case i: $x \in$ RANDOM (HIT)
6:      if $x \in$ RANDOM BOTTOM then
7:          Reset seqMiss = $0$
8:          Set adapt = max( -1, min(ratio - 1, 1) )
9:      endif
10:     Mru($x$, RANDOM)

11: case ii: $x \in$ SEQ (HIT)
12:     if $x \in$ SEQ BOTTOM then
13:         if (ratio $>$ LargeRatio) then
14:             Set adapt = 1
15:         endif
16:     endif
17:     if $x$ is AsyncTrigger then
18:         ReadAndMru([$x + 1$, $x + m - x\%g$], SEQ)
19:     endif
20:     Mru($x$, SEQ)
21:     if track $(x - 1) \in$ (SEQ $\bigcup$ RANDOM) then
22:         if (seqCounter$(x - 1)$ == $0$) then
23:             Set seqCounter$(x)$ = max (seqThreshold,
                    seqCounter$(x - 1) + 1$)
24:         endif
25:     else
26:         Set seqCounter$(x)$ = $1$
27:     endif

28: case iii: $x \notin$ (SEQ $\bigcup$ RANDOM) (MISS)
29:     if $(x - 1) \in$ (SEQ $\bigcup$ RANDOM) then
30:         if seqCounter$(x - 1)$ == seqThreshold then
31:             seqMiss++
32:             ReadAndMru([$x$, $x + m - x\%g$], SEQ)
33:             Set seqCounter$(x)$ = seqThreshold
34:         else
35:             ReadAndMru([$x, x$], RANDOM)
36:             Set seqCounter$(x)$ =
                    seqCounter$(x - 1) + 1$
37:         endif
38:     else
39:         Set seqCounter$(x)$ = $1$
40:     endif
```

```
CACHE MANAGEMENT POLICY (CONTINUED):

ReadAndMru( [start, end], listType )
41: foreach track $t$ in [start, end]; do
42:     if $t \notin$ (SEQ $\bigcup$ RANDOM) then
43:         grab a free track from FreeQ
44          read track $t$ from disk
45:     endif
46:     Mru($t$, listType)
47: done
48: if (listType == SEQ)
49:     Set AsyncTrigger as (end $-$ triggerOffset)
50: endif
```

```
FREE QUEUE MANAGEMENT:

FreeQThread()
51: while (true) do
52:     if length(FreeQ) $<$ FreeQThreshold then
53:         if (seqListSize $< \Delta L$
            or randomListSize $< \Delta L$) then
54:             if (lru track of SEQ is older than
                lru track of RANDOM) then
55:                 EvictLruTrackAndAdapt(SEQ)
56:             else
57:                 EvictLruTrackAndAdapt(RANDOM)
58:             endif
59:         else
60:             if (seqListSize $>$ desiredSeqListSize) then
61:                 EvictLruTrackAndAdapt(SEQ)
62:             else
63:                 EvictLruTrackAndAdapt(RANDOM)
64:             endif
65:         endif
66:     endif
67: endwhile

EvictLruTrackAndAdapt( listType )
68: evict lru track in listType and add it to FreeQ
69: if (desiredSeqListSize $>$ 0) then
70:     Set desiredSeqListSize += adapt / 2
71: else
72:     Set desiredSeqListSize = seqListSize
73: endif
```

Fig. 4. Algorithm for Sequential Prefetching Adaptive Replacement Cache. This algorithm is completely self-contained, and can directly be used as a basis for an implementation. The algorithm starts from an empty cache.

only when a track is actually evicted from one of the lists. In a steady state, tracks are evicted from the cache at the rate of cache misses. Hence, a larger (respectively, a smaller) rate of misses will effect a faster (respectively, a slower) rate of adaptation. Hence, SARC adapts not only the sizes of the two lists, but also the rate at which the sizes are adapted.

Lines 11-27 deal with the case when a track in SEQ is hit. If the hit is in the bottom portion of the SEQ list (line 12) and ratio has become large (line 13), in other words, no hit has been observed in the bottom of the RANDOM list while comparatively large number of sequential misses have been seen on the SEQ list, then set adapt to 1 (line 14) meaning that increase desiredSeqListSize at the fastest rate possible. Now, if the hit track is an asynchronous trigger track (line 17), then asynchronously read-ahead the next sequential group of tracks (line 18). Lines 21-27 describe how the sequential detection mechanism in Section II-A is implemented.

Lines 28-40 deal with a cache miss. For a sequential miss (lines 29-31), synchronously read-ahead a sequential group of tracks (line 32). The remaining lines deal with sequential detection mechanism in Section II-A.

Lines 41-50 (i) read the missing tracks from a given range of tracks; (ii) places all tracks in the given range at the MRU position; and (iii) set the asynchronous trigger.

Lines 51-73 implement the cache replacement policy and carry out the adaptation. As is typical in multithreaded systems, we assume that these lines run on a separate thread (line 51). If the size of the free queue drops below some predetermined threshold (line 52), then tracks are evicted from SEQ if it exceeds desiredSeqListSize and tracks are evicted from RANDOM otherwise. In either case, the evicted tracks are placed on the free queue. Observe that SARC becomes active (lines 60-64) only if the sizes of both the SEQ and the RANDOM list exceed ΔL. Otherwise, a simple LRU eviction (lines 54-58) is done. Whenever the utility of one of the two lists becomes so small when compared to the utility of the other list that its size eventually drops below ΔL, we do not want to waste even ΔL amount of cache, and revert back to LRU. Whenever both list sizes exceed ΔL, SARC takes over. Finally, lines 68-73 evict the LRU track from the desired list, and effect an adaption as already described above.

Our description of SARC is now complete.

IV. System Implementation, Workload, and Results

We implemented SARC and two well known LRU variants, namely, LRU Top and LRU Bottom (see Section III-A), on IBM Shark (formally, TotalStorage Enterprise Storage Server Model 800) hardware. We compare the performance of SARC to the LRU variants on an SPC-1 Like benchmark workload in different configurations.

A. Shark

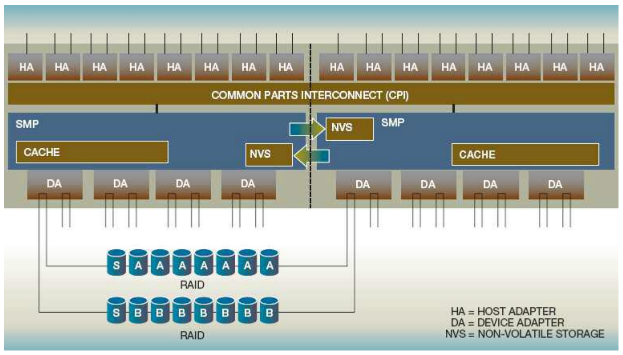

Fig. 5. A conceptual representation of IBM Shark (Enterprise Storage Server 800) [39].

Figure 5 outlines the architecture of Shark. The architecture can support upto 16 host adapters (HA) in four host adapter bays. The host adapters can have fiber channel, ESCON, or SCSI ports. Shark has two active cluster processors with symmetrical multiprocessors (SMP) for performance, reliability, and availability. Each host adapter is connected to both the SMP clusters via the Common Parts Interconnect (CPI). Either cluster is able to handle IOs from any host adapter. Both the clusters have multiple SMPs in processor drawers and have an I/O drawer which provide PCI connections for access to non-volatile, battery-backed memory (denoted as NVS in the figure) and the device adapters (denoted as DA in the figure). The processor drawer also contains up to 32GB cache per cluster. For read data, the host adapter directs the request to the appropriate cluster. For write data, the data is written to both the clusters: on one it resides in the SMP RAM and on the other cluster it resides in the NVS memory. At the back-end, Shark uses RAID arrays (or arrays) that can be configured as RAID-5 or RAID-10. For further details on Shark, please see [31], [39].

B. Our Experimental Setup

Our Shark was equipped with: 8 GB cache (per cluster), 2 GB NVS (per cluster), four 600 MHz PowerPC/RS64IV CPUs (per cluster), and 16 RAID-5 (6 + parity + spare) arrays with 72 GB, 10K rpm drives. We use only one cluster since enhancements provided by dual cluster are not necessary to study effectiveness of cache algorithms. In a single cluster mode, the behavior of our experimental software on Shark does not change other than the fact that writes now go to both the NVS memory as well as the SMP RAM in the same cluster.

	Warm-up Phase	Measurement Phase					
Time (in minutes)	120	30	30	30	30	30	30
Load (percentages)	100	100	97.5	95	80	50	10
High Load (scaled IOPS)	25000	25000	23750	22500	20000	12500	2500
Low Load (scaled IOPS)	11364	11364	10795	10227	9091	5682	1136

TABLE I. The structure of the SPC-1 Like benchmark on two different loads. For both loads, the warm-up phase is run for 120 minutes followed by 6 measurement phases of 30 minutes each. The purpose of the warm-up phase is to fill-up the cache and to brings it to a steady-state. Observe that in the measurement phase, we gradually decrease the load in proportion 100%, 97.5%, 95%, 80%, 50%, and 10%, where 100%, represents the highest load. This allows studying the behavior of the storage controller under a wide range of load conditions.

The essence of our results does not change with a larger or a smaller cache size.

As a client, we use a powerful AIX host with the following configuration: 16 GB RAM, 2-way SMP with 1GHz PowerPC/Power4 CPUs. The host is connected to the Shark through two fiber channel cards which can support more than sufficient bandwidth for our all experiments.

C. SPC-1 Like Workload

SPC-1 is a synthetic, but sophisticated and fairly realistic, performance measurement workload for storage subsystems used in business critical applications. The benchmark simulates real world environments as seen by on-line, non-volatile storage in a typical server class computer system. SPC-1 measures the performance of a storage subsystem by presenting to it a set of I/O operations that are typical for business critical applications like OLTP systems, database systems and mail server applications. For extensive details on SPC-1, please see: [23], [24]. We used SPC-1 Like that is an earlier prototype implementation of SPC-1 benchmark by Bruce McNutt who was one of the chief architects of the official SPC-1 benchmark.

The SPC-1 Like workload synthesizes a community of users targeting I/Os to the storage that is logically organized in the form of three Application Storage Units (ASU). ASU-1 represents a "Data Store", ASU-2 represents a "User Store", and ASU-3 represents a "Log/Sequential Write". Of the total amount of available back-end storage, 45% is assigned to ASU-1, 45% is assigned to ASU-2, and remaining 10% is assigned to ASU-3 as per SPC-1 specifications. We shall furnish more details on sizes of various ASUs in Section IV-D.

The SPC-1 Like workload is specified in units of Business Scaling Units (BSU). One BSU corresponds to a community of users who collectively generate up to 50 IOPS. The overall composition of a BSU, and, that of SPC-1 Like, is specified by the following simple matrix, where all numbers are in percentages:

	Read	Write	All
Random	29	32	61
Sequential	11	28	39
All	40	60	100

The workload is scaled by using more BSUs that in effect increases the number of users being simulated.

In this paper, due to the commercial nature of the system involved, we will not use IOPS, but rather use *scaled IOPS* which are obtained by multiplying the true IOPS by a constant (a non-integer rational number) that is not revealed in the paper.

We shall use two different load schedules for SPC-1 Like in Table I.

D. Footprint of the Workload

While modern storage controllers can make available immense amount of space, in a real-life scenario, workloads actively use only a fraction of the total available storage space known as the *footprint*. Generally speaking, for random workloads, for a given cache size, the larger the footprint, the smaller the hit ratio, and vice versa. In this paper, we will use the following two different back-end configurations in conjunction with the SPC-1 Like workload:

	ASU-1	ASU-2	ASU-3
(percentages)	45	45	10
cache-sensitive (GB)	45	45	10
cache-insensitive (GB)	1443	1443	320

E. Data Layout: Striping

RAID stripes data across its constituent disks. In addition, our AIX host permits striping data across RAID arrays. There are essentially three striping models: (i) wide striping; (ii) narrow striping; and (iii) no striping. Wide striping lays out contiguous data across all the arrays whereas narrow striping lays out data across only a subset of the arrays. For a detailed study comparing wide striping to narrow striping, please see [40]. For a very useful practical introduction to the mechanics of implementing striping, please see [41]. For random streams, striping is useful in reducing "hot spots" (points of contention) leading to a reduction in the average response time for random seeks.

While wide striping is extremely useful for random clients, it is not friendly to sequential clients. Because striping is done at the host level and is not visible to Shark, when wide striped, each sequential access stream could appear as multiple, although slower, sequential access streams. Narrow striping is likely to have moderate number of hot spots for random clients, but is friendlier to sequential clients. In this paper, we have used narrow striping that stripes across 4 RAID arrays. Our insights, analysis, and algorithm do not change with wide striping or with no striping.

F. Results

We now compare performance of SARC to two state-of-the-art LRU variants, namely, LRU Top and LRU Bottom (see Section III-A), using the SPC-1 Like benchmark on Shark hardware that is running our experimental software implementations.

SARC minimizes the number of misses. The effect of such minimization can be studied from two perspectives, namely, that of the client and that of the storage controller. To a client, at a fixed load, the most important metric is the average response time. To study the performance seen by a client over a large spectrum of real-life operating scenarios, for the SPC-1 Like benchmark, we compare throughput versus average response time curves for all the three algorithms. To a storage controller, a crucial metric is the internal load on the RAID arrays. Within Shark, we measure the rate of tracks being staged to the cache due to read requests (both random and sequential). We will demonstrate how SARC convincingly outperforms both the LRU variants from the perspective of the client as well as the storage controller.

1) Throughput vs. Response Time: The two plots in the left column of Figure 6 show the throughput (in scaled IOPS) versus average response time (in ms) for all three algorithms by using the SPC-1 Like workload. Each displayed data point is an average of 27 numbers, each number being an overall response time average for read and write requests over a minute. According to SPC-1 specification, the numbers corresponding to the first three minutes of a measurement phase are discarded.

The top, left plot is obtained on a cache-sensitive configuration (see Section IV-D) for which, due to relatively high cache hit ratio, a High Load schedule (see Table I) is required to saturate the machine. The bottom, left plot is observed on a cache-insensitive configuration for which, due to relatively low cache hit ratio, a Low Load schedule is sufficient. LRU Bottom generally allocates more cache space to RANDOM than to SEQ when compared to LRU Top. Hence, LRU Bottom performs better than LRU Top in the cache-sensitive configuration where RANDOM list has more utility, whereas, the reverse is true in the cache-insensitive configuration. However, in both the cases, SARC significantly and dramatically outperforms both the LRU variants by judiciously and dynamically partitioning the cache space amongst the two lists. Due to its self-tuning nature, SARC achieves this without any *a priori* knowledge of the different workloads resulting from different configurations and load levels. For the cache-sensitive configuration (resp. cache-insensitive), at the peak throughput, the overall average response times for LRU Top, LRU Bottom, and, SARC are, respectively, 33.35ms, 8.92ms, and 5.18ms (resp. 8.62ms, 15.26ms, and 6.87ms).

To facilitate a more detailed analysis of the performance improvements due to SARC, Table II provides the break-up of overall average response time into read and write components. At the peak throughput in the cache-sensitive configuration, SARC provides 83.7% and 39.0% read response time reduction over LRU Top and LRU Bottom, respectively. Even for the cache-insensitive configuration at the peak throughput, SARC provides 16.1% and 46.9% read response time reduction over LRU Top and LRU Bottom, respectively.

SARC improves read response times directly by reducing the misses, serendipitously, the resultant reduction in the back-end load also improves the performance for the concurrent writes. At the peak throughput in the cache-sensitive configuration, SARC provides 85.2% and 44.8% write response time reduction over LRU Top and LRU Bottom, respectively. Once again, even for the cache-insensitive configuration at peak throughput, SARC provides 28.6% and 66.7% write response time reduction over LRU Top and LRU Bottom, respectively. Also observe in Table II that although none of the LRU strategies works well in both cache-sensitive and cache-insensitive configurations, SARC outperforms the better of the two LRU variants fairly consistently across all load levels for both reads and writes.

2) Power of the Storage Controller: We now provide an alternate viewpoint for studying the throughput versus average response time curves. Typically, at a low (resp. high) throughput one observes a low (resp. high) response time. Thus, there is a trade-off between the two quantities. [42] combined them into a single performance measure *power* that is defined as the ratio of throughput to average response time. The two plots in the middle column of Figure 6 display overall throughput versus power for the three algorithms. The visualization helps us observe the relative performance of the algorithms even at low load levels, where the throughput-response time plots may seem to overlap. In

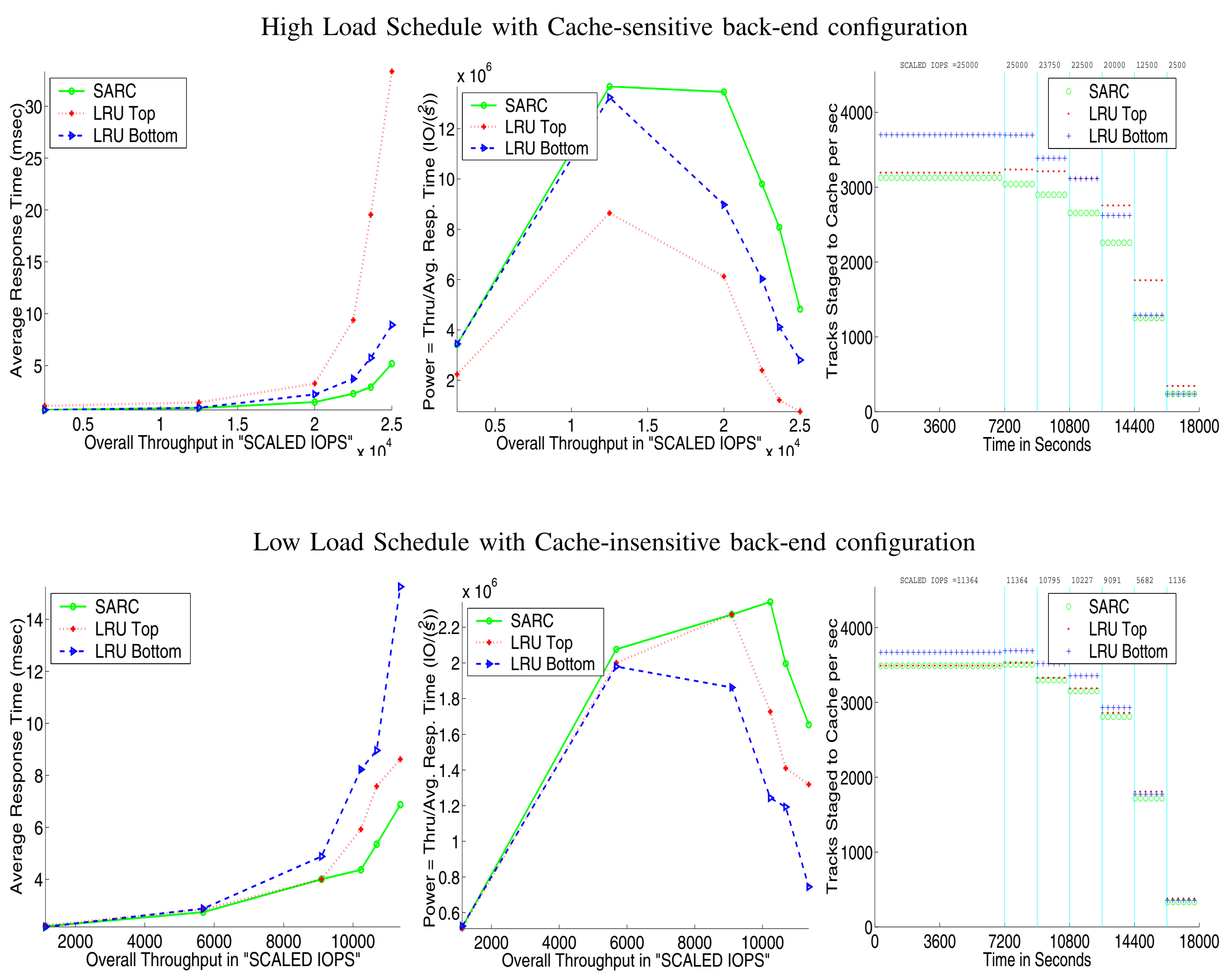

Fig. 6. A comparison of LRU Top, LRU Bottom, and SARC. The top (resp. bottom) three panels correspond to SPC-1 Like in cache-sensitive (resp. cache-insensitive) configuration. For both the configurations, the left column displays throughput versus overall average response times, the middle column displays throughput versus power, and the right column displays the evolution of the rate of tracks staged to cache. The vertical lines demarcate the load schedules in Table I.

Low Load Schedule with
Cache-insensitive back-end configuration

Scaled IOPS	LRU-Top read/write	LRU-Bottom read/write	SARC read/write
1136	5.30/**0.18**	**5.15/0.18**	5.21/**0.18**
5682	6.71/**0.26**	6.77/0.27	**6.45/0.26**
9091	8.62/**0.92**	9.74/1.64	**8.58**/0.95
10227	11.13/2.45	13.98/4.39	**8.86/1.37**
10795	13.23/3.81	14.79/5.07	**10.15/2.15**
11364	14.36/4.79	22.72/10.28	**12.05/3.42**

High Load Schedule with
Cache-sensitive back-end configuration

Scaled IOPS	LRU-Top read/write	LRU-Bottom read/write	SARC read/write
2500	2.51/**0.19**	**1.53/0.19**	1.54/**0.19**
12500	3.00/0.41	1.79/0.38	**1.76/0.35**
20000	5.22/1.96	3.20/1.58	**2.38/0.89**
22500	12.71/7.18	4.95/2.92	**3.34/1.60**
23750	24.84/16.00	7.32/4.69	**4.06/2.17**
25000	41.70/27.78	11.10/7.46	**6.77/4.12**

TABLE II. A comparison of average read/write response times for LRU Top, LRU Bottom, and SARC. The left (resp. right) table corresponds to cache-insensitive (resp. cache-sensitive) configuration. All the response time numbers are in ms. The best numbers for each load point are in bold.

both configurations, the power of the storage controller with the SARC algorithm envelopes the power from the other algorithms.

3) Rate of Stages to Cache: The two plots in the right column of Figure 6 display the rate (per second) of tracks being brought (or staged) to the cache in response to read misses or sequential prefetches. In the cache-sensitive configuration (top right), we observe that LRU Top is better than LRU Bottom for the higher load levels, while the opposite is true for the lighter load levels. In contrast, SARC is consistently better than both the LRU variants for all load levels.

To understand the importance of this metric, note that from a client's perspective, in the cache-sensitive configuration, as seen in the top, left panel of Figure 6, LRU Bottom consistently outperforms LRU Top by delivering a lower average response time. However, from a storage controller's perspective, LRU Bottom outperforms LRU Top for lower loads while the converse is true for higher loads. In contrast, SARC consistently outperforms both the LRU variants from both the perspectives. In other words, not only does SARC deliver a better performance to a client, it does so without unduly stressing the server.

Similar observations also hold for the cache-insensitive configuration, albeit, to a smaller extent.

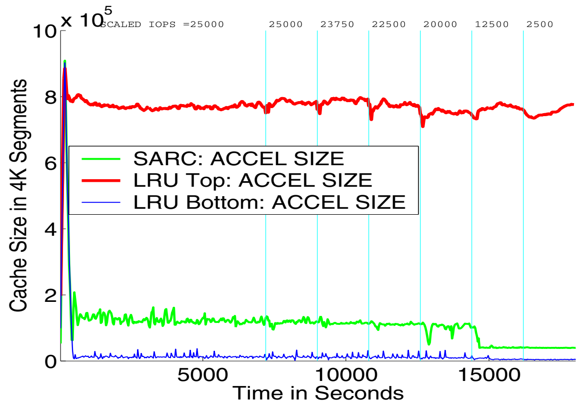

Fig. 7. A comparison of cache space allocated to SEQ list by LRU Top, LRU Bottom, and SARC for the SPC-1 Like workload in cache-sensitive configuration. The vertical lines demarcate the high load schedule in Table I.

4) Adaptive nature of SARC*:* The Figure 7 plots the sizes of the SEQ list versus the time (in seconds) for all the three algorithms. It can be seen that LRU Top allocates too much cache space to SEQ list, LRU Bottom allocates too little cache space to SEQ list, while SARC adaptively allocates just the right amount of cache space to SEQ so as improve the overall performance. It can also be seen that SARC adapts to the evolving workload and selects a different size for the SEQ list at different loads.

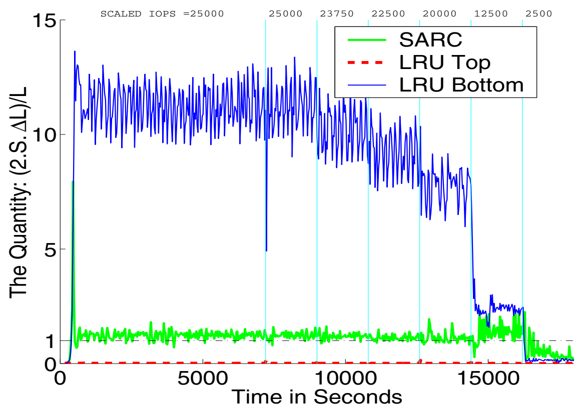

Fig. 8. A comparison of the quantity $(2 \cdot s \cdot \Delta L)/L$ for LRU Top, LRU Bottom, and SARC for the SPC-1 Like workload in cache-sensitive configuration. The vertical lines demarcate the high load schedule in Table I.

The Figure 8 plots the quantity $(2 \cdot s \cdot \Delta L)/L$ for the three algorithms. This quantity is denoted by variable ratio in line 4 of the algorithm in Figure 4. It can be seen that for LRU Bottom keeps ratio too large meaning that it would benefit by further increase in cache space devoted to sequential data. Similarly, LRU Top keeps ratio too small meaning that it would benefit by a decrease in cache space devoted to sequential data. Whereas SARC keeps ratio to roughly the idealized value of 1 meaning that it essentially gets very close to the optimum. In other words, SARC will not benefit by further trading of cache space between SEQ and RANDOM, whereas the other two algorithms would. This plot clearly explains why SARC outperforms the LRU variants in the top three panels of Figure 7. Furthermore, quite importantly, this plot also indicates that any other algorithm that does not directly attempt to drive ratio to 1 as SARC does will not be competitive with SARC. For example, any algorithm that keeps sequential data in the cache for a fixed time [1] or allocates a fixed, constant amount of space to sequential data [29], cannot in general outperform SARC.

Remark IV.1 For heavy sequential workloads, SARC keeps SEQ list just as large enough as useful and does not starve the random workload if present. If there is no random workload then the entire cache will be devoted to the sequential workload and vice versa. When both heavy sequential and heavy random workloads are present, SARC computes the marginal utility to dynamically divide the cache between the two workloads.

V. Conclusions

We have designed a powerful sequential prefetching strategy that combines virtues of synchronous and asynchronous prefetching while avoiding the anomaly that arises when prefetching and caching are integrated, and is capable of attaining zero misses for sequential streams.

We have introduced a self-tuning, low overhead, simple to implement, locally adaptive, novel cache management policy SARC that dynamically and adaptively partitions the cache space amongst sequential streams and random streams so as to reduce the read misses. SARC is doubly adaptive in that it adapts not only the cache space allocated to each class but also the rate at which the cache space is transferred from one class to another. It is extremely easy to convert an existing LRU variant into SARC.

We have implemented SARC along with two popular state-of-the-art LRU variants on Shark hardware. By using the most widely adopted storage benchmark, we have demonstrated that SARC consistently outperforms the LRU variants, and shifts the throughput versus average response time curves to the right thus fundamentally increasing the capacity of the system. Furthermore, SARC deliver better performance to a client, without unduly stressing the server.

We believe that the insights, analysis, and algorithm presented in this paper are widely applicable. Due to its adaptivity, we expect SARC to work well across (i) a wide range of workloads that may have a varying mix of sequential and random clients and may possess varying temporal locality of the random clients and varying number of sequential and random streams with varying think times; (ii) different back-end storage configurations; and (iii) different data layouts.

Endnotes

1. The numbers reported in this paper cannot be used to draw inferences about IBM's products, since (i) Shark (ESS Model 800) does not use any of the above three algorithms; (ii) Shark does not use the sequential prefetching algorithm described above; (iii) our software implementation on Shark hardware is experimental and academic; (iv) we use only one of the two available clusters on Shark; and (v) we have scaled throughput (IOPS) numbers. This paper is not intended to be an official SPC-1 submission, but is an academic study geared to demonstrate that SARC is better than LRU variants.

Acknowledgements

We are indebted to Michael Benhase, Joseph Hyde, Steven Lowe, and Thomas (Chip) Jarvis for numerous discussions on Shark. We are grateful to Dr. Mustafa Uysal, our shepherd, for detailed comments that greatly improved the readability of the paper.

References

[1] J. Gray and P. J. Shenoy, "Rules of thumb in data engineering," in *ICDE*, pp. 3–12, 2000.

[2] J. E. G. Coffman and P. J. Denning, *Operating Systems Theory*. Englewood Cliffs, NJ: Prentice-Hall, 1973.

[3] L. A. Belady, "A study of replacement algorithms for virtual storage computers," *IBM Sys. J.*, vol. 5, no. 2, pp. 78–101, 1966.

[4] M. J. Bach, *The Design of the UNIX Operating System*. Englewood Cliffs, NJ: Prentice-Hall, 1986.

[5] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems: Design and Implementation*. Prentice-Hall, 1997.

[6] A. Silberschatz and P. B. Galvin, *Operating System Concepts*. Reading, MA: Addison-Wesley, 1995.

[7] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proc. FAST Conf.*, pp. 115–130, 2003.

[8] N. Megiddo and D. S. Modha, "Outperforming LRU with an adaptive replacement cache algorithm," *IEEE Computer*, vol. 37, no. 4, pp. 58–65, 2004.

[9] S. P. Vanderwiel and D. J. Lilja, "Data prefetch mechanisms," *ACM Comput. Surv.*, vol. 32, no. 2, pp. 174–199, 2000.

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

[11] K. M. Curewitz, P. Krishnan, and J. S. Vitter, "Practical prefetching via data compression," in *SIGMOD Conference*, pp. 257–266, 1993.

[12] M. L. Palmer and S. Zdonik, "Fido: A cache that learns to fetch," in *Proc. VLDB Conf.*, Sep 1991.

[13] T. M. Kroeger and D. D. E. Long, "Design and implementation of a predictive file prefetching algorithm," in *USENIX Annual Technical Conference*, pp. 105–118, 2001.

[14] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed prefetching and caching," in *Proc. SOSP Conf.*, December 1995.

[15] F. W. Chang and G. A. Gibson, "Automatic I/O hint generation through speculative execution," in *Operating Systems Design and Implementation*, pp. 1–14, 1999.

[16] R. J. Feiertag and E. I. Organisk, "The Multics input/output system," in *Proc. 3rd SOSP*, 1971.

[17] J. Rodriguez-Rosell, "Empirical data reference behavior in data base systems," *IEEE Computer*, vol. 9, pp. 9–13, November 1976.

[18] P. Hawthorn and M. Stonebraker, "Performance analysis of a relational data base management system," in *Proc. SIGMOD Conf.*, pp. 1–12, May 1979.

[19] B. T. Zivkov and A. J. Smith, "Disk cache design and performance as evaluated in large timesharing and database," in *Proc. Comput. Measurement Group Conf.*, pp. 639–658, Dec 1997.

[20] A. J. Smith, "Sequentiality and prefetching in database systems," *ACM Trans. Database Systems*, vol. 3, no. 3, pp. 223–247, 1978.

[21] W. W. Hsu, A. J. Smith, and H. C. Young, "I/O reference behavior of production database workloads and the TPC benchmarks - an analysis at the logical level," *ACM Trans. Database Syst.*, vol. 26, no. 1, pp. 96–143, 2001.

[22] W. W. Hsu, A. J. Smith, and H. C. Young, "Characteristics of production database workloads and the TPC benchmarks," *IBM Sys. J.*, vol. 40, no. 3, pp. 781–802, 2001.

[23] B. McNutt and S. Johnson, "A standard test of I/O cache," in *Proc. Comput. Measurements Group's 2001 Int. Conf.*, 2001.

[24] S. A. Johnson, B. McNutt, and R. Reich, "The making of a standard benchmark for open system storage," *J. Comput. Resource Management*, no. 101, pp. 26–32, Winter 2001.

[25] Storage Performance Council, "SPC Benchmark-2: Public Review Draft Specification, 0.8.0," November 2003.

[26] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A fast file system for UNIX," *ACM Trans. Computer Systems*, vol. 2, pp. 181–197, August 1984.

[27] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout, "Measurements of a distributed file system," in *Proc. SOSP Conf.*, pp. 198 –212, 1991.

[28] J. K. Ousterhout, H. D. Costa, D. Harrison, J. A. Kunze, M. D. Kupfer, and J. G. Thompson, “A trace-driven analysis of the UNIX 4.2 BSD file system,” in *Proc. 1oth SOSP*, pp. 15–24, 1985.

[29] J. Z. Teng and R. A. Gumaer, “Managing IBM database 2 buffers to maximize performance,” *IBM Sys. J.*, vol. 23, no. 2, pp. 211–218, 1984.

[30] P. Mead, “Oracle Rdb buffer management, www.oracle.com/technology/products/rdb/pdf/2002_tech_forums/rdbtf_2002_buffer.pdf,” 2002.

[31] G. Castets, P. Crowhurst, S. Garraway, and G. Rebmann, “IBM TotalStorage Enterprise Storage Server Model 800.” IBM Redbook, October 2002.

[32] B. C. Beardsley, M. T. Benhase, J. S. Hyde, T. C. Jarvis, D. A. Martin, and R. L. Morton, “Method and system for staging data into cache.” US Patent 06381677, issued on April 30, 2002.

[33] E. A. M. Shriver, A. Merchant, and J. Wilkes, “An analytic behavior model for disk drives with readahead caches and request reordering,” in *SIGMETRICS*, pp. 182–191, 1998.

[34] P. Cao, E. W. Felten, A. R. Karlin, and K. Li, “A study of integrated prefetching and caching strategies,” in *SIGMETRICS*, pp. 188–197, 1995.

[35] N. Ragaz and J. Rodriguez-Rosell, “Empirical studies of storage management in a data base system,” tech. rep., RJ 1834, IBM Research Labs, San Jose, October 1976.

[36] J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, “A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references,” in *Proc. OSDI Conf.*, pp. 119–134, 2000.

[37] B. C. Beardsley, M. T. Benhase, D. A. Martin, R. L. Morton, and M. A. Reid, “Data management method in cache, involves selecting one of least recently used data entries for demoting it during new data entry.” US Patent 6141731, filed on August 19, 1998, issued on October 10, 2000.

[38] S. F. Kaplan, L. A. McGeoch, and M. F. Cole, “Adaptive caching for demand prepaging,” in *MSP/ISMM*, pp. 221–232, 2002.

[39] M. H. Hartung, “IBM TotalStorage Enterprise Storage Server: A designer’s view,” *IBM Sys. J.*, vol. 42, no. 2, pp. 383–396, 2003.

[40] V. Sundaram, P. Goyal, P. Radkov, and P. Shenoy, “Evaluation of object placement techniques in a policy-managed storage system,” tech. rep., TR03-38, Dept. Comput. Sci., Univ. Mass., Nov 2003.

[41] D. Martin, “Configuring the IBM Enterprise Storage Server for Oracle OLTP applications,” tech. rep., IBM, April 2003.

[42] L. Kleinrock, “On flow control in computer networks,” in *Proc. Int’l. Conf. Commun., Toronto, Ontario, Canada*, vol. II, pp. 27.2.1–27.2.5, June 1978.