Advance Access publication on 7 January 2011
doi:10.1093/comjnl/bxq091

Evaluation and Optimization of Kernel File Readaheads Based on Markov Decision Models

Chenfeng Xu[1,*], Hongsheng Xi[1] and Fengguang Wu[2]

[1]*Department of Automation, University of Science and Technology of China, Hefei, Anhui 230026, China*
[2]*Intel Asia-Pacific Research and Development Ltd., No. 880 Zi Xing Rd., Shanghai Zizhu Science Park, Shanghai 200241, China*
**Corresponding author: xcf@ustc.edu.cn*

Readahead is an important technique to deal with the huge gap between disk drives and applications. It has become a standard in modern operating systems and advanced storage systems. However, it is difficult to develop the common kernel readahead and to achieve full testing coverage of all cases. In this paper, we formulate the kernel read handling, caching and readahead behavior as an absorbing Markov decision process, and present performance evaluations to compare or verify various readaheads. We also introduce algorithms to find optimal prefetching policies for specific read pattern and present the convergence analysis. By exploiting sample-path-based methods, it becomes much easier to evaluate and optimize prefetching policies, which can provide valuable informations to help the design and improvement of practical readaheads. For illustration and verification, we present two examples and some experiments on the readaheads in Linux kernel. The results show that the model-based evaluations agree with the practice and the improved prefetching policy significantly outperforms the original one in Linux kernel for the specified workloads.

Keywords: optimization; operating systems; file systems management; Markov decision processes; simulation; measurements

Received 15 July 2010; revised 16 October 2010
Handling editor: Ing-Ray Chen

1. INTRODUCTION

As the technology of disk drives has a much slower advancement than other hardwares, the disk I/O performances have become key bottleneck problems especially for data-intensive applications. The challenge to bridge the gap between disk drives and applications is an active research problem.

Prefetching, also known as readahead in Linux, is a widely used and effective technique for improving the file system performance. In general, the disk drives suffer from big seek latencies and are better utilized by large accesses. However, applications tend to do many tiny sequential reads. In order to obtain better performance, prefetching is employed to bring in data blocks for upcoming requests and do disk I/Os in big chunks. Prefetching could bring three major benefits [1]. The first is to hide the I/O delays from the applications by overlapping computation with I/O. When an application requests for a data block, it has been prefetched and is ready to use. The second is to improve disk utilization since disk drives are better utilized with the larger disk I/Os. The third is to help to amortize the processing overheads in the I/O path.

There are many researches on prefetching such as heuristic prefetching [2–7] and informed prefetching [8–11]. As prefetching is usually integrated with caching which is another effective technique for improving I/O performance, the integrated prefetching and caching are also studied in [8, 9, 12–15]. It was recently argued that the prefetching policies should be more aggressive [16] to support more complex access patterns because the memory and bandwidth are plenty. However, to identify these complex access patterns is not easy.

Markov models are effective and widely used. There are many Markov predictions and Markov decision models [17] for prefetching [18–28]. Most studies use Markov chain models to predict future I/Os such as data blocks, files or hyperlinks and the prefetching actions are based on the predictions.

Some studies also employ Markov decision process (MDP) models to perform state-based actions but they are often application-specific [27, 28]. In this paper, the Markov decision models are proposed to describe the common readaheads in kernel level. The policy for kernel prefetching usually concerns the temporary access patterns more than the long-run time effects on caching. The Markov decision models can describe the kernel prefetching behaviors very well as they can select actions according to not only the long-run statistical characteristics but also the temporary access patterns. However, there is almost no research on the analysis and optimization of kernel file prefetching based on Markov decision models. The challenge in this field is to deal with the complexity of read handling and caching and to derive the gradient formula theoretically since any read sequence will stop after a short time once the corresponding file descriptor is closed, and it does not agree with the general assumption that every Markov chain under the allowable policies is ergodic.

In this paper an absorbing MDP model for kernel prefetching is proposed to describe various readaheads integrated with the kernel read handling and caching. We also discuss some interesting performance metrics in real-world problems and derive the performance gradient formula. In order to optimize prefetching policies, an optimization algorithm based on the gradient formula is proposed, which is parameterized and sample-path based and needs no knowledge on the transition probabilities. Its convergence is also presented. For illustration, the *legacy* Linux and *ondemand* Linux readaheads are presented.

This paper presents the models first from which the gradient formulas and optimization algorithms are derived. These theoretical results including the convergence analysis make sure that the methods proposed in this paper are correct and available. Because MDPs are used and the actions are state based, it is easy to apply the analysis results and optimized prefetching policies. Our contributions are as follows. First, we model a complex kernel readahead integrated with kernel read handling and caching by using Markov decision models. By using such models, one is able to accurately describe the overall system behaviors and reflect practically interesting performance metrics so that it is possible for the models to be used for analyzing real problems. Second, the performance gradient formula for parameterized absorbing MDSs is derived. The gradient follows the difference formula that we derived, and its existence conditions are also presented. Third, it provides a new way to analyze and optimize prefetching policies which can help the design or optimization of readaheads as shown in our examples and experiments.

In the following sections we first review the related researches in Section 2. Then we formulate the prefetching integrated with read handling and caching by using an absorbing MDP in Section 3. In Section 4, the gradient formula of the absorbing MDP is derived and the gradient policy optimization algorithm is presented. Section 5 illustrates the MDP models by using two real readaheads in Linux kernel. Section 6 gives experiments to approve these ideas. Finally Section 7 concludes this paper.

2. RELATED WORK

Prefetching is a widely used for improving the file system performance. The prefetching can be either heuristic or informed. The heuristic algorithms try to predict the future I/O blocks based on the past ones. The most successful one is the sequential readahead, which has long been a standard practice among operating systems [2, 3]. There are also more comprehensive works to mine correlations between files [4–6] or data blocks [7]. On the other hand, informed prefetching gets hints from the application about its future I/Os. The hints could either be application controlled [8–10] or be automatically generated [11].

Caching is another highly effective technique for improving the I/O performance. It is a common practice to place the prefetched data blocks in the shared caching memory. There are many comprehensive works dealing with the integrated prefetching and caching [8, 9, 12–15] and schemes to dynamically adapt the prefetch memory [29] or depth [30–33].

As the memory and bandwidth are no longer the key bottlenecks, it was recently argued that it should be necessary to employ more aggressive prefetching policies [16]. This is backed by the performance gains in practice, including the tricky AIO [34] and NFS [35, 36] problems. Linux kernel recently introduced a new readahead framework and the *ondemand* readahead algorithm [37, 38]. It implicitly supports some important semi-sequential patterns and its modular framework makes it easier to support new read patterns. However, to support more semi-sequential read sequences, which could not be identified by examining the pattern of accesses, is very difficult since it is hard to find and implement the proper prefetching policy for those access patterns. The caching facility also makes the problem more complex as prefetching has different impacts on different caching replacement algorithms [12] and may have negative effects sometimes. In summary, to improve, develop and test a readahead may be very painful without the help of theoretical discussions especially to improve prefetching policies for some complicated access patterns or applications.

Markov models are effective and widely used for prefetching. There are many Markov predictions used for instruction prefetching [18], file prefetching [19–21] and web or hypermedia prefetching [22–26]. The Markov decision models are also employed to find optimal context-aware prefetching policies for multimedia adaptation or navigation [27, 28]. By using Markov decision models, the read handling can be described very accurately. In order to optimize the prefetching policy, we propose a sample-based algorithm which needs no knowledge on the transition probabilities and works well for large-space problems.

3. PROBLEM FORMULATION

In modern operating systems, the transparent prefetching has been adopted in a widespread manner. It is part of file read handling in kernel side and is integrated with the file caching. So the modeling is very complex and many symbols are used. In order not to cause confusion, Table 1 shows the main symbols for the rest of the paper. This section describes the common read handling and readahead behavior in kernel level. It also provides lots of information that a readahead subsystem needs. These symbols are used to describe such information and other things that the models and optimization algorithms need. Based on these symbols we also present our MDP models in this section.

3.1. Read handling

The read requests from userspace are processed by multiple kernel subsystems before they are actually submitted to the disk. Figure 1 shows how kernel handles the read requests. The default behavior of most operating systems is not to do direct I/Os. The kernel will cache every file as pages which are blocks with a fixed size. Any read access is fed by the file cache. If the demanded pages are not present, then the kernel will indirectly pull them from one or more disks. For the read sequence that meets the sequential characteristic, the kernel also attempts to prefetch continuous pages to reduce the number of ondemand I/Os. Moreover, the kernel has an I/O scheduler to reorder and cluster the disk I/O requests. It can increase the size of a disk I/O, reduce the number of disk requests and decrease the average overhead time of disk accesses.

Readahead can be significantly useful to the sequential accesses to a file since the prefetched pages may be accessed after a short time by the upcoming accesses. If the accesses are not sequential, then the readahead may result in undesired disk I/Os and wastes of memory resources especially when there is less memory for file caching. Therefore, it is very important

TABLE 1. Main symbols.

Symbols	Descriptions	Symbols	Descriptions
$X_r(l)$	The lth read request	$X_p(n)$	The nth page access
$\mathcal{S}_r$	State space of $X_r(l)$	$\mathcal{S}_p$	State space of $X_p(n)$
P_r	$P_r(\cdot)$ is the initial distribution and $P_r(\cdot\|\cdot)$ is the transition matrix of $X_r(l)$	P_p	$P_p(\cdot)$ is the initial distribution and $P_p(\cdot\|\cdot)$ is the transition matrix of $X_p(n)$
RS	The read sequence	**PS**	The page access sequence
$\tilde{\mathbf{RS}}_k$	The kth sample of **RS**	$\tilde{\mathbf{PS}}_k$	The kth sample of **PS**
N_r	The length of the read sequence	$N, N^d, \vec{N}^d$	N is the length of the page access sequence, $N^d = E^d\{N\}$, and $\vec{N}^d = \left(E_s^d\{N\}, s \in \mathcal{S}\right)^T$
$X_{\text{ah}}(n)$	The readahead state	$X_c(n)$	The cache state
$\mathcal{S}_{\text{ah}}$	State space of $X_{\text{ah}}(n)$	$\mathcal{S}_c$	State space of $X_c(n)$
h_{ah}	Evolution law of $X_{\text{ah}}(n)$	h_c^M	Evolution law of $X_c(n)$
h_{ac}^M	Combination of h_{ah} and h_{c}^M	h_M	Rewrite of h_{ac}^M
M	M is the maximal number of memory pages for file caching	E, E^d, E_s^d	Mathematical expectations. E_s^d denotes the expectation under conditions that the initial state is s and the policy is d
$X(n)$	Read state process	$\tilde{X}^k(n), \tilde{N}^k$	The kth sample-path of $X(n)$ and $\tilde{N}^k$ is its length
$\mathcal{S}$	State space of $X(n)$	H^d	$H^d(s)$ is the accumulated performance with initial state s under policy d
$U(n)$	Corresponding action at $X(n)$	g^d	$g^d = H^d - v^d\vec{N}^d$ which is similar to the performance potential defined by [39]
$\mathcal{U}$	Action space of $U(n)$	$\tilde{v}$	Estimate of v^d or v^θ
d	Prefetching policy	θ	Policy parameter vector
$\mathcal{D}$	Prefetching policy space	Θ	Policy parameter vector space
P, P^d, P^θ	$P(\cdot)$ is the initial distribution, $P(\cdot\|\cdot,\cdot)$ is the transition matrix of $X(n)$ and P^d is the one under policy d or θ	π^d	$\pi^d(s) = (N^d(s)/N^d), s \in \mathcal{S}$, where $N^d(s) = E^d\{N(s)\}$ is the mean number of occurrences of state s
f, f^d, f^θ	Performance function. f^d is the one under policy d	$f_{\text{mi}}, f_{\text{wa}}, f_{\text{th}}$	Cost functions defined as performance function f
v^d, v^θ	Mean performance under policy d or θ	$v_{\text{mi}}^d, v_{\text{wa}}^d, v_{\text{th}}^d$	Mean performances defined as v^d with cost functions $f_{\text{mi}}, f_{\text{wa}}, f_{\text{th}}$

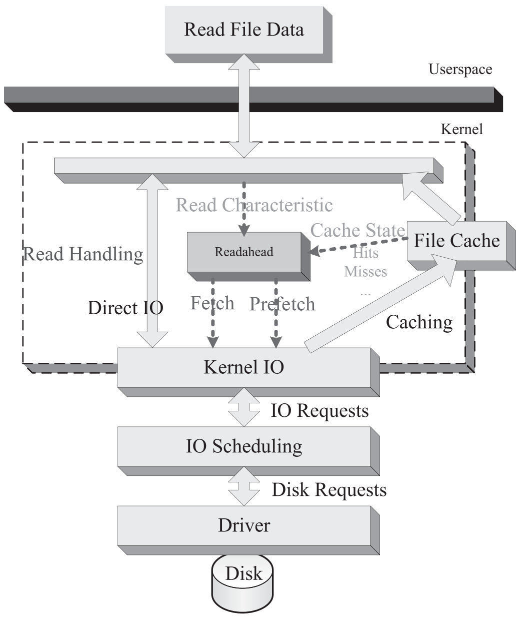


FIGURE 1. Read handling.

to adopt a rational policy for the kernel to decide whether to perform readahead and how many pages to prefetch.

3.2. Read sequence

The read request from userspace is byte based. As the kernel manages a file as pages, it translates the original requests to the page-based ones at first and then deals with them. Once all demanded pages or some consecutive ones are ready in the file cache, the kernel sends the necessary bytes to userspace.

Let $X_r = (\text{offset}, \text{size})$ be a page-based read request, where offset is the zero-based starting page index while size is the number of pages the request demands.

For a strictly sequential read sequence, $\text{offset}_l + \text{size}_l - \text{offset}_{l+1}$ is 0 or 1, where $(\text{offset}_l, \text{size}_l)$ is the lth read request. For a more general one, it is difficult to characterize its access pattern. Let

$$P_r(X_r(l+1) = (j,b) | X_r(l) = (i,a)) \tag{1}$$

be the probability that the next request is (j, b) if the current request is (i, a) on a single file descriptor, which is independent of l. We can consider $P_r((j,b)|(i,a))$ as the long-run frequency of (j, b) when the previous one is (i, a). From this view, the matrix P_r is the file statistical access characteristic for some application, and $X_r(l), l \geq 0$, is a Markov chain with finite state space since the number of pages for any file is fixed. In most cases, the request block has a fixed size, i.e. $\text{size}_l = \text{size}$, where $l = 0, 1, \ldots$. The probability can be rewritten as

$$P_r^{\text{size}}(X_r(l+1) = j | X_r(l) = i). \tag{2}$$

In fact, a read sequence will terminate once the file descriptor is closed. Let

$$P_r(\underline{\text{stop}}|\underline{\text{stop}}) = 1, \tag{3}$$

where $\underline{\text{stop}}$ is the absorbing state, and $X_r(l), l \geq 0$, is an absorbing Markov Chain.

Let $\mathcal{S}_r$ be the state space. Assume the initial state is $s_0^r \in \mathcal{S}_r$. The probability $P_r(s_0^r)$ can be also regarded as a long-run access frequency to a file for some application. To sum up, the read pattern can be characterized by $P_r(\cdot)$ and $P_r(\cdot|\cdot)$. Then the read pattern can be written as

$$\{X_r(l), \mathcal{S}_r, P_r(\cdot), P_r(\cdot|\cdot)\}. \tag{4}$$

Suppose that a read sequence has N_r requests. N_r satisfies

$$N_r = \min\{l : X_r(l) = \underline{\text{stop}}\}. \tag{5}$$

Then a read sequence can be described as

$$\mathbf{RS} = \{X_r(l), l = 0, 1, \ldots, N_r\}. \tag{6}$$

For a specific read sequence, the values of $X_r(l), l \geq 0$, are determinant, which is a sample of **RS**. Let

$$\tilde{\mathbf{RS}}_k = \left\{\tilde{X}_r^k(l), l = 0, 1, \ldots, \tilde{N}_r^k\right\} \tag{7}$$

be the kth sample of **RS**. Let

$$\chi_{i,a}(X_r) = \begin{cases} 1, & X_r = (i,a) \\ 0, & X_r \neq (i,a) \end{cases} \tag{8}$$

Assume that there are K samples; then we have

$$P_r((j,b)|(i,a)) = \lim_{K\to\infty} \frac{\sum_{k=1}^K \chi_{i,a}^{j,b}(\tilde{\mathbf{RS}}_k)}{\sum_{k=1}^K \chi_{i,a}(\tilde{\mathbf{RS}}_k)}, \tag{9}$$

with probability 1, where

$$\chi_{i,a}^{j,b}(\tilde{\mathbf{RS}}_k) = \sum_{l=0}^{\tilde{N}_k - 2} \chi_{i,a}(\tilde{X}_r^k(l)) \cdot \chi_{j,b}(\tilde{X}_r^k(l+1)), \tag{10}$$

$$\chi_{i,a}(\tilde{\mathbf{RS}}_k) = \sum_{l=0}^{\tilde{N}_k - 1} \chi_{i,a}(\tilde{X}_r^k(l)). \tag{11}$$

For the particular case that $\tilde{X}_r(l+1) = \underline{\text{stop}}, l = \tilde{N}_r(k) - 1$, we have

$$P_r(\underline{\text{stop}}|(i,a)) = \lim_{K\to\infty} \frac{\sum_{k=1}^K \chi_{i,a}(\tilde{X}_r^k(l))}{\sum_{k=1}^K \chi_{i,a}(\tilde{\mathbf{RS}}_k)}, \tag{12}$$

with probability 1. For the initial state s_0^r, we have

$$P_r(s_0^r) = \lim_{K\to\infty} \frac{\sum_{k=1}^K \chi_{s_0^r}(X_r^k(0))}{K}, \tag{13}$$

with probability 1.

From this subsection, it is shown that the read handling processes can be described by using probability or stochastic models. However, the OS kernel deals with file read requests in pages so that the readaheads are page-based.

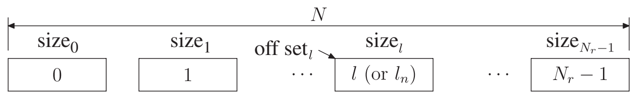


FIGURE 2. Read sequence and page access.

3.3. Page access

Some readahead facilities always check the file cache first and make a decision on prefetching only if the demanded page is missing. For a given system, the page access behavior to a file for some application just follows the access pattern.

As shown in Fig. 2, any page access will belong to some read request. Assume page$_n$ is the nth page access which belongs to the lth read request for a read sequence. For convenience, we use l_n to denote the nth page access of the lth read request, and we have

$$0 \le n_l < \text{size}_{l_n}, \tag{14}$$

$$n = \sum_{k=0}^{l_n-1} \text{size}_k + n_l < \sum_{k=0}^{l_n} \text{size}_k, \tag{15}$$

$$\text{page}_n = \text{offset}_{l_n} + n_l < \text{offset}_{l_n} + \text{size}_{l_n}. \tag{16}$$

where n_l denotes page$_n$ is also the n_lth page access for $(\text{offset}_{l_n}, \text{size}_{l_n})$. These formulas describe the page access process under a given read sequence. Therefore the page access sequence can be written as

$$\mathbf{PS} = \{X_p(n), n = 0, 1, \ldots, N\}, \tag{17}$$

where

$$X_p(n) = (\text{offset}_{l_n}, \text{size}_{l_n}, \text{page}_n), \quad n < N, \tag{18}$$

$$X_p(N) = \underline{\text{stop}}, \quad N = \sum_{k=0}^{N_r-1} \text{size}_k. \tag{19}$$

The page access behavior is

$$P_p(X_p(n+1) = (i', a', j')|X_p(n) = (i, a, j)) = \begin{cases} P_r((i', a')|(i, a)), & j = i + a - 1, j' = i', \\ 1, & j' = j + 1, i' = i, a' = a, \\ 0 & \text{otherwise}, \end{cases} \tag{20}$$

$$P_p\left(s_0^p = (i, a, j)\right) = \begin{cases} P_r((i, a)), & j = i, \\ 0, & j \ne i, \end{cases} \tag{21}$$

where $s_0^p \in \mathcal{S}_p$ is the initial state. Thus **PS** is also an absorbing Markov chain with finite state space. And it is obvious that there exists a $\tilde{\mathbf{PS}}_k$ for any $\tilde{\mathbf{PS}}_k$, where $\tilde{\mathbf{PS}}_k$ is the kth sample of **PS**. Let $\mathcal{S}_p$ be its state space. And the page access pattern can be written as

$$\{X_p(n), \mathcal{S}_p, P_p(\cdot), P_p(\cdot|\cdot)\}. \tag{22}$$

After deriving the page access models, other things that a readahead subsystem needs are presented in the following two subsections.

3.4. Readahead state

In order to trace the read sequence and the previous readahead actions, the readahead state is introduced. The following readaheads depend on it. By using the readahead state, the kernel can choose more rational readahead actions.

Usually, the readahead state just records the last read request and readahead action. Let $X_{\text{ah}}(n)$ be the readahead state and $\mathcal{S}_{\text{ah}}$ be its state space. Most readaheads may maintain the last readahead window and the last page index of the previous read request. For asynchronous readahead, some systems may introduce others into readahead state such as the *ahead_window* in legacy Linux readahead (before kernel version 2.6.22) and the *async_size* in *ondemand* Linux readahead.

Let u be the read or readahead action and $\mathcal{U}$ be its space. The read access and action will turn $X_{\text{ah}}(n)$ to $X_{\text{ah}}(n+1)$, which can be defined as a map

$$h_{\text{ah}} : (s_p, u) \longmapsto x'_{ah}, \tag{23}$$

where $s_p \in \mathcal{S}_p$, $x'_{\text{ah}} \in \mathcal{S}_{\text{ah}}$, $u \in \mathcal{U}$.

In general, the readahead state may be associated with a file descriptor and be initialized when opening the file. Let s_0^{ah} be the initial state; then s_0^{ah} may be always unique for any file under any case for a given readahead facility.

A system with readahead subsystems will need more memory. Common readaheads in kernel level are always integrated with caching. If there is no enough memory, the kernel should perform fewer prefetches. Therefore, the cache state is also very important for a readahead subsystem.

3.5. Cache state

Assume that the file has NP pages. Let $X_{\text{ps}}(n) = (z_0(n), z_1(n), \ldots, z_{\text{NP}-1}(n))$, where $z_i(n)$ is the state of page i. The page state usually shows whether a page is in memory.

We can also assign other flags to the page state. The *ondemand* Linux readahead introduces a flag *PG_readahead* to support prefetching for interleaved sequential reads and concurrent streams since these read patterns will invalidate the readahead state.

Let $X_{\text{fc}}(n)$ be the usage of the file cache. Assume that the file cache has memory to contain M pages. When there is no memory left, the kernel uses some replacement algorithm to get enough space to hold the incoming pages.

To sum up, we can use $X_c(n) = (X_{\text{ps}}(n), X_{\text{fc}}(n))$ as the cache state. Let $\mathcal{S}_c$ be its state space. The read or readahead action and the caching replacement may change the state of a page and the usage of the file cache. For the sake of convenience, we assume that there are only one file descriptor and a fixed amount of memory for our concerned file. The common initial state under this assumption is that no page for the file is in memory and the file cache is empty. In order to describe the more general case, we define the probability of $X_c(0) = s_0^c$ as $P_c(s_0^c)$. Beginning with the initial state, the state $X_c(n)$ turns to $X_c(n+1)$ following some kind of rule. In fact, the evolution law of $X_c(n)$ can be defined as

$$h_c^M : (s_p, s_c, u) \longmapsto s_c', \tag{24}$$

where $s_p \in \mathcal{S}_p$, $s_c, s_c' \in \mathcal{S}_c$ and $u \in \mathcal{U}$. The operator h_c^M denotes the action u makes the cache state turn to s_c' from s_c when the maximal amount of memory for caching is M pages and the page access is s_p.

After the elements of state and their evolution laws are presented, the MDP model is derived in the following subsection.

3.6. Markov decision model

For convenience, let $X_{\text{ac}}(n) = (X_{\text{ah}}(n), X_c(n))$ and $\mathcal{S}_{\text{ac}} = \mathcal{S}_{\text{ah}} \times \mathcal{S}_c$. By combining h_{ah} and h_c^M, define h_{ac}^M as

$$h_{\text{ac}}^M : (s_p, s_{ac}, u) \longmapsto s_{ac}', \tag{25}$$

where $s_p \in \mathcal{S}_p$, $s_{\text{ac}}', s_{\text{ac}} \in \mathcal{S}_{\text{ac}}$ and $u \in \mathcal{U}$.

Let $s = (s_p, s_{\text{ac}})$, $s_p \in \mathcal{S}_p$, $s_{\text{ac}} \in \mathcal{S}_{\text{ac}}$, be the state. Its state space is $\mathcal{S} = \mathcal{S}_p \times \mathcal{S}_{\text{ac}}$. It is obviously large but finite.

Let $P(s'|s,u)$ be the probability that action u in state s will lead to state s'. According to (20), (23) and (24), $P(s'|s,u)$ can be obtained. For the sake of convenience, rewrite h_{ac}^M as h_M, which is

$$h_M : (s,u) \longmapsto s_{\text{ac}}, \tag{26}$$

where $s \in \mathcal{S}$, $u \in \mathcal{U}$, $s_{\text{ac}} \in \mathcal{S}_{\text{ac}}$.

To measure the impact of action u in state s, we define the performance function as

$$f : (s,u) \longmapsto f(s,u), \quad s \in \mathcal{S}, u \in \mathcal{U}. \tag{27}$$

Let $X(n)$ be the state process and $U(n)$ be the corresponding readahead or read action at n. The process will terminate once it reaches the state $\underline{\text{stop}}$. Then we can state that $X(n)$ is an absorbing Markov decision process.

COROLLARY 3.1.

$$\{X(n), \mathcal{S}, \mathcal{U}, P, f\}$$

is a Markov decision process.

Proof. According to (20) and (26),

$$\begin{aligned} &P(X(n+1)|X(n), U(n), \ldots, X(0), U(0)) \\ &= P(X(n+1)|X(n), U(n)) \\ &= P(X_p(n+1)|X_p(n)) \cdot \chi_h(X(n), U(n), X_{ac}(n+1)), \end{aligned} \tag{28}$$

where

$$\chi_h(s,u,s_{\text{ac}}) = \begin{cases} 1, & s_{\text{ac}} = h_M(s,u), \\ 0, & s_{\text{ac}} \neq h_M(s,u). \end{cases}$$ □

For a readahead facility, the action is selected according to the given prefetching policy, which is defined as

$$d : s \longmapsto u, \quad s \in \mathcal{S}, u \in \mathcal{U}. \tag{29}$$

Let $\mathcal{D}$ be the policy space, which is also finite as $\mathcal{S}$ and $\mathcal{U}$ are finite. For $\forall d \in \mathcal{D}$, we have

$$P^d(s'|s) = P(s'|s, u = d(s)). \tag{30}$$

Therefore, $X(n)$ under any policy d is also a Markov chain.

According to (27) and (29), the state s leads to action $d(s)$ and produces cost $f^d(s) = f(s, d(s))$. To measure the performance impact of policy d, we define a mean performance as

$$v^d(s_0) = \frac{E_{s_0}^d \left\{ \sum_{n=0}^{N-1} f^d(X(n)) \right\}}{E_{s_0}^d\{N\}}, \tag{31}$$

where s_0 is the initial state and N is defined as

$$N = \min \left\{ n : X(n) = \underline{\text{stop}} \right\}, \tag{32}$$

which always satisfies (19) and is independent of d. Thus,

$$E_{s_0}^d\{N\} = E_{s_0}\{N\}. \tag{33}$$

Theoretically, accumulated performance might be more interesting than the mean performance, but we are usually more concerned about the metrics such as *Cache Miss Rate* in real-world problems. In fact, they are equivalent from the optimization point of view since N is independent of d.

For initial state $s_0 = (s_0^p, s_0^{\text{ah}}, s_0^c)$, we have

$$P(s_0) = P_p(_0^p)P_c(s_0^c). \tag{34}$$

Let $\mathcal{S}_0$ be the allowable set for initial states, which is defined as

$$\mathcal{S}_0 = \{s_0 : P(s_0) > 0, s \in \mathcal{S}\} \subseteq \mathcal{S}. \tag{35}$$

Then we have

$$v^d = \sum_{s_0 \in \mathcal{S}_0} P(s_0) v^d(s_0) \tag{36}$$

$$= \frac{E^d \left\{ \sum_{n=0}^{N-1} f^d(X(n)) \right\}}{E^d\{N\}}. \tag{37}$$

Define

$$\mathbf{RA} = \{X(n), S_{\text{pp}}, S_{\text{ac}}, U, h_M\} \tag{38}$$

or

$$\mathbf{RA} = \{X(n), S_{\text{pp}}, S_{\text{ah}}, S_c, U, h_{\text{ah}}, h_c^M\} \tag{39}$$

as a readahead facility, where S_{pp} is the set of all possible states for any file and any page access pattern such that $\forall S_p \subseteq S_{\text{pp}} = \bigcup S_p$.

To a given access pattern **PP** for some file, S_p and P_p are certain, $P(X(n+1)|X(n), U(n))$ can be obtained as (28) and $X(n) \in S = S_p \times S_{\text{ac}}$. Let $\mathbf{RA}^d$ be the readahead facility with prefetching policy d. Then $\mathbf{RA}^d$ with a given **PP** is a Markov chain.

3.7. Performance metrics

Readahead is employed to reduce the number of file page cache misses and improve the file system performance by increasing the size of disk I/Os. Therefore, we should employ related metrics to apply our models to real readaheads.

When accessing page j, the kernel checks the file cache first. If j is not cached, then a page fault happens, and a disk I/O request is submitted. For sequential access patterns, minimizing the number of page cache misses can significantly improve the file system performance. Define a cost function as

$$f_{\mathrm{mi}}(s) = \begin{cases} 1 & \text{page } j \text{ is missing,} \\ 0 & \text{otherwise,} \end{cases} \tag{40}$$

where $s = (s_p, s_{\mathrm{ac}}) \in \mathcal{S}$ and $s_p = (i, a, j) \in \mathcal{S}_p$. Then the mean performance $v^d_{\mathrm{mi}}(s_0)$ defined by (31) is the *Cache Miss Rate*.

After a relatively long time, the prefetched pages will be dropped by the caching replacement facility. If they have not been accessed before the kernel drops them, meaningless disk I/Os are raised. Define another cost function as

$$f_{\mathrm{wa}}(s, u) = \mathrm{count}_{\mathrm{wa}}(s_c, s'_c), \tag{41}$$

where $s = (s_p, s_{\mathrm{ah}}, s_c) \in \mathcal{S}$, $s_c \in \mathcal{S}_c$, $u \in \mathcal{U}$ and $s'_c = h^M_c(s_p, s_c, u) \in \mathcal{S}_c$. When the cache state turns to s'_c from s_c, some pages may be scheduled out. Let $\mathrm{count}_{\mathrm{wa}}(s_c, s'_c)$ be the number of those pages that have never been accessed after they were scheduled into the file cache. The mean performance $v^d_{\mathrm{wa}}(s_0)$ defined in (31) is the average number of extra disk I/Os per page access, which is the reverse side of the *Cache Hit Rate*. Less waste means a higher cache hit rate.

There is a very serious situation which will significantly cut the file system performance down. Readahead thrashing happens when accessing a prefetched page that has been scheduled out. Thrashing makes one page access have more than one disk I/Os. Define the third cost function as

$$f_{\mathrm{th}}(s, u) = \mathrm{count}_{\mathrm{th}}(s_c, s'_c), \tag{42}$$

where $s = (s_p, s_{\mathrm{ah}}, s_c) \in \mathcal{S}$, $s_c \in \mathcal{S}_c$, $u \in \mathcal{U}$ and $s'_c = h^M_c(s_p, s_c, u) \in \mathcal{S}_c$. When the cache state turns to s'_c from s_c, some waste pages may be scheduled in. A thrashing page must be a waste page at first. Let $\mathrm{count}_{\mathrm{th}}(s_c, s'_c)$ be the number of those pages. The performance $v^d_{\mathrm{th}}(s_0)$ defined as (31) is the average number of thrashing pages per page access.

In order to measure the whole impact of a readahead facility and its prefetching policy, we can define a combined cost function as

$$f(s, u) = (\alpha_{\mathrm{mi}}, \alpha_{\mathrm{wa}}, \alpha_{\mathrm{th}}) \begin{pmatrix} f_{\mathrm{mi}}(s) \\ f_{\mathrm{wa}}(s, u) \\ f_{\mathrm{th}}(s, u) \end{pmatrix}, \tag{43}$$

where $s \in \mathcal{S}$, $u \in \mathcal{U}$ and $\alpha_{\mathrm{mi}}, \alpha_{\mathrm{wa}}, \alpha_{\mathrm{th}} \in (0, \infty)$.

4. ANALYSIS AND OPTIMIZATION

The comparison, analysis and optimization for a readahead are very complex and difficult in the real world since it costs more in the real world to test a readahead and collect results. The complexity in a real system can also bring many extra factors into the collected results, which is possible to make the comparison and analysis unavailable. By using the models, these things may become easily handled. According to the definition of state, the proposed models have large state space if files are very big. In order to use such models, parameterized and sample-path-based methods are employed, which need no knowledge on the transition probabilities. We also assume that the states in the same subspace use the same decision rule so that the number of parameters does not increase as the size of the state space grows.

4.1. Evaluation and comparison

In order to estimate the performance for a given readahead facility, we can simulate the read handling according to the access pattern h_M and prefetching policy. Suppose that there are K samples for an access pattern PP that are collected from real world or generated according to $P_r(\cdot)$ and $P_r(\cdot|\cdot)$. As (7) shows, $\tilde{\mathbf{RS}}_k$ is the kth sample. Then we can generate K samples of $X(n)$ as

$$\{\tilde{X}^k(n), n = 0, 1, \ldots, \tilde{N}^k\}, \tag{44}$$

where $k = 1, 2, \ldots, K$. And the performance estimation is

$$\tilde{v}^d = \frac{\sum_{k=1}^{K} \sum_{n=0}^{\tilde{N}^k - 1} f^d(\tilde{X}^k(n))}{\sum_{k=1}^{K} \tilde{N}^k}. \tag{45}$$

To compare two given readahead facilities or prefetching policies, we can, respectively, estimate their performances. Suppose that there are two readahead facilities $\mathbf{RA}_1^{d_1}$ and $\mathbf{RA}_2^{d_2}$. If $\tilde{v}_1$ and $\tilde{v}_2$ are their performances and

$$\mathrm{err} = \frac{\tilde{v}_2 - \tilde{v}_1}{\tilde{v}_1} > \xi,$$

we can state that $\mathbf{RA}_1^{d_1}$ is better than $\mathbf{RA}_2^{d_2}$ or d_1 is better than d_2 when they have the same or similar rule of h^M_c.

4.2. Performance gradient formula

Define $N(s)$ as the number of occurrences of state s in the lifetime of $X(n)$ whose initial state follows (34). Let $N(\underline{\mathrm{stop}}) \equiv 0$, $f^d(\underline{\mathrm{stop}}) \equiv 0$, $N^d = E^d\{N\}$ and $N^d(s) = E^d\{N(s)\}$. Then

$$\begin{aligned} v^d &= \frac{E^d\left\{\sum_{n=0}^{N-1} f^d(X(n))\right\}}{E^d\{N\}} \\ &= \sum_{s \in \mathcal{S}} \frac{E^d\{N(s)\}}{E^d\{N\}} f^d(s) = \pi^d f^d, \end{aligned} \tag{46}$$

where $\pi^d = (\pi^d(s), s \in \mathcal{S})$, $f^d = (f(s), s \in \mathcal{S})^T$ and

$$\pi^d(s) = \frac{N^d(s)}{N^d}.$$

Since

$$N^d(s) = P(s) + \sum_{s' \in \mathcal{S}} N^d(s')P^d(s|s'),$$

we have $\pi^d(s) = (1/N^d)P(s) + \sum_{s' \in \mathcal{S}} \pi^d(s')P^d(s|s')$, which yields

$$\pi^d = \frac{1}{N^d}P_0 + \pi^d P^d, \tag{47}$$

where $P_0 = (P(s), s \in \mathcal{S})$ is the initial distribution which is independent of d.

Let $H^d(s) = E_s^d\left\{\sum_{n=0}^{N-1} f^d(X(n))\right\} = v^d(s)N_s^d$. Then we have

$$H^d(s) = f^d(s) + \sum_{s' \in \mathcal{S}} P^d(s'|s)H^d(s'),$$

which yields

$$H^d = f^d + P^d H^d, \tag{48}$$

where $H^d = (H^d(s), s \in \mathcal{S})^T$.

From (47) and (48), we have

$$\begin{aligned} N^{d'}\pi^{d'}[P^{d'} - P^d]H^d &= N^{d'}[\pi^{d'}P^{d'}H^d - \pi^{d'}P^dH^d] \\ &= -P_0H^d + N^{d'}\pi^{d'}f^d \\ &= N^{d'}v^{d'} - N^dv^d - N^{d'}\pi^{d'}[f^{d'} - f^d]. \end{aligned}$$

Thus, we have

$$N^{d'}v^{d'} - N^dv^d = N^{d'}\pi^{d'}[(P^{d'} - P^d)H^d + (f^{d'} - f^d)].$$

By temporarily setting $f^{d'} \equiv f^d \equiv 1$, we also have

$$N^{d'} - N^d = N^{d'}\pi^{d'}[(P^{d'} - P^d)\vec{N}^d],$$

where $\vec{N}^d = (E_s^d\{N\}, s \in \mathcal{S})^{\mathrm{T}}$. Therefore,

$$\begin{aligned} v^{d'} - v^d &= \frac{1}{N^{d'}}[(H^{d'} - H^d) - v^d(N^{d'} - N^d)] \\ &= \pi^{d'}[(P^{d'} - P^d)g^d + (f^{d'} - f^d)], \end{aligned} \tag{49}$$

where

$$g^d = H^d - v^d\vec{N}^d, \tag{50}$$

which means

$$g^d(s) = E_s^d\left\{\sum_{n=0}^{N-1}[f^d(X(n)) - v^d]\right\}.$$

Assumption 1. For any policy $d \in \mathcal{D}$ and initial state $s_0 \in \mathcal{S}_0$, we have

$$\sum_{n=0}^{\infty} P\{X(n+1) = \underline{\text{stop}}, X(n) \neq \underline{\text{stop}}|X(0) = s_0\} = 1.$$

Corollary 4.1. *If Assumption 1 holds, then $N^d < \infty$.*

Proof. Due to $P\{X(n) = \underline{\text{stop}}|X(0) = s_0\} + P\{X(n) \neq \underline{\text{stop}}|X(0) = s_0\} = 1, \forall n = 0, 1, 2, \ldots$ and

$$\begin{aligned} &\lim_{n\to\infty} P\{X(n+1) = \underline{\text{stop}}|X(0) = s_0\} \\ &\quad = \lim_{n\to\infty}\sum_{k=0}^{n} P\{X(k+1) = \underline{\text{stop}}, X(k) \neq \underline{\text{stop}}|X(0) = s_0\} \\ &\quad\quad \cdot P\{X(n+1) = \underline{\text{stop}}|X(k+1) = \underline{\text{stop}}\} = 1, \end{aligned}$$

we have

$$\lim_{n\to\infty} P\{X(n) = \underline{\text{stop}}|X(0) = s_0\} = 0.$$

Therefore, s_0 is a transient state since $\mathcal{S}$ is finite. Then

$$\begin{aligned} &\sum_{n=0}^{\infty} nP\{X(n) = \underline{\text{stop}}, X(n-1) \neq \underline{\text{stop}}|X(0) = s_0\} \\ &\quad = E^d\{N\} < \infty. \end{aligned}$$

□

Parameterize the policy d. Let

$$d(\theta) : s \longmapsto P^\theta(u|s), u \in U, s \in \mathcal{S}, \theta \in \Theta, \tag{51}$$

where Θ is the set of allowable values of θ. And we have

$$P^d = P^\theta, \quad v^d = v^\theta,$$
$$f^d = f^\theta, \quad \pi^d = \pi^\theta.$$

From (49), the gradient with respect to θ can be written as

$$\begin{aligned} \nabla v^\theta &= \sum_s \pi^\theta(s)\sum_{s'}\nabla P^\theta(s'|s)g^\theta(s') \\ &\quad + \sum_s \pi^\theta(s)\nabla f^\theta(s) \\ &= \sum_s\sum_u \pi^\theta(s)\nabla P^\theta(u|s)q^\theta(s,u) \\ &= E^\theta\{F^\theta(X(n), U(n))\} \qquad (52) \\ &= \frac{E^\theta\left\{\sum_{n=0}^{N-1} F^\theta(X(n), U(n))\right\}}{E^\theta\{N\}}, \qquad (53) \end{aligned}$$

where

$$F^\theta(X(n), U(n)) = L^\theta(X'(n), U(n))q^\theta(X(n), U(n)), \tag{54}$$

$$L^\theta(s,u) = \frac{\nabla P^\theta(u|s)}{P^\theta(u|s)}, \tag{55}$$

$$\begin{aligned} q^\theta(s,u) &= f(s,u) - v^\theta + \sum_{s'} P^\theta(s'|s,u)g^\theta(s') \\ &= E_{s,u}^\theta\left\{\sum_{n=0}^{\infty}[f(X(n), U(n)) - v^\theta]\right\}, \qquad (56) \end{aligned}$$

and $X(0) = s$, $U(0) = u$.

ASSUMPTION 2. For every $s', s \in \mathcal{S}$, $P^{\theta}(s'|s)$ and $f^{\theta}(s)$ are bounded, twice differentiable, and have bounded first and second derivatives with respect to each element in θ.

COROLLARY 4.2. *If Assumptions 1 and 2 hold, then v^{θ} and ∇v^{θ} are also bounded.*

4.3. Prefetching policy optimization

Let θ^* be the optimal policy parameter vector that satisfies

$$v^{\theta^*} \leq v^{\theta}, \quad \theta \in \Theta.$$

According to the gradient formula (53), a sample-path-based optimization algorithm follows.

ALGORITHM 1. (Prefetching Policy Algorithm). Suppose that there are many read sequence samples of an access pattern, $\tilde{\mathbf{RS}}_k$, $k = 1, 2, \ldots$, which are collected from the real world or generated according to $P_r(\cdot)$ and $P_r(\cdot|\cdot)$.

(1) Choose values for θ_0, $\epsilon > 0$. Set $k = 1$.
(2) Generate a sample path $(X^k(n), U^k(n))$ according to $\tilde{\mathbf{RS}}$ and $d(\theta_k)$. Compute

$$\tilde{q}_n = \sum_{m=n}^{\tilde{N}^k-1} (f(X^k(m), U^k(m)) - \tilde{v}_{k-1}), \tag{57}$$

$$F_k = \sum_{n=0}^{\tilde{N}^k-1} \tilde{q}_n L^{\theta_{k-1}}(X^k(n), U^k(n)), \tag{58}$$

where $\tilde{v}_{k-1}$ is the estimate of $v^{\theta_{k-1}}$.
(3) Update θ_k:

$$\theta_k = \theta_{k-1} - \gamma_k F_k,$$

where

$$\gamma_k > 0, \quad k = 0, 1, \ldots, \tag{59}$$

$$\sum_{k=0}^{\infty} \gamma_k = \infty, \quad \sum_{k=0}^{\infty} \gamma_k^2 < \infty. \tag{60}$$

(4) Set $k = k + 1$ and go to step 2.

Algorithm 1 is a simple gradient optimization method based on simulations. The estimation of $v^{\theta_{k-1}}$ can follow from (45) which leads to unbiased gradient estimations in Algorithm 1.

ASSUMPTION 3. $E\{\tilde{v}_k|\theta_k\} = v^{\theta_k}$, and F_k, $\tilde{v}_k$ are bounded.

ASSUMPTION 4. $\left(\nabla v^{\theta}\right)^{\mathrm{T}} (\theta - \theta^*) > 0$ for any $\theta \in \Theta, \theta \neq \theta^*$.

THEOREM 4.1.

$$\lim_{k\to\infty} \theta_k = \theta^* \quad \text{w.p.1}$$

if Assumptions 1–4 hold.

Proof. For convenience, the notation 'w.p.1' (with probability 1) is omitted in the proof. As $E\{\tilde{v}_k|\theta_k\} = v^{\theta_k}$, we have

$$E\{F_k|\theta_{k-1}\} = N^{\theta_{k-1}} \nabla v^{\theta_{k-1}}.$$

Let

$$F_k = N^{\theta_{k-1}} \nabla v^{\theta_{k-1}} + \omega_k,$$

where ω_k satisfies $E\{\omega_k|\theta_{k-1}\} = 0$ and $E\left\{||\omega_k||^2 \middle| \theta_{k-1}\right\}$ is bounded since $N^{\theta_{k-1}}$, $\nabla v^{\theta_{k-1}}$, F_k and $\tilde{v}_k$ are all bounded. It is seen that $\sum_{l=1}^{k} \gamma_l \omega_l$ is martingale, and we have $\sum_{l=1}^{k} \gamma_l \omega_l$ converges as $k \to \infty$ according to the martingale convergence theorem:

$$\begin{aligned}
||\theta_k - \theta^*||^2 &= ||\theta_{k-1} - \theta^*||^2 + \gamma_k^2 ||F_k||^2 \\
&\quad - 2\gamma_k F_k^{\mathrm{T}} \left(\theta_{k-1} - \theta^*\right) \\
&= ||\theta_{k-1} - \theta^*||^2 + \gamma_k^2 ||F_k||^2 \\
&\quad - 2\gamma_k \omega_k^{\mathrm{T}} (\theta_{k-1} - \theta^*) \\
&\quad - 2\gamma_k N^{\theta_{k-1}} \left(\nabla v^{\theta_{k-1}}\right)^{\mathrm{T}} \left(\theta_{k-1} - \theta^*\right) \\
&= ||\theta_0 - \theta^*||^2 + \sum_{l=1}^{k} \gamma_l^2 ||F_l||^2 \\
&\quad - 2 \sum_{l=1}^{k} \gamma_l \omega_l^{\mathrm{T}} (\theta_{l-1} - \theta^*) \\
&\quad - 2 \sum_{l=1}^{k} \gamma_l N^{\theta_{l-1}} \left(\nabla v^{\theta_{l-1}}\right)^{\mathrm{T}} \left(\theta_{l-1} - \theta^*\right),
\end{aligned}$$

and

$$\begin{aligned}
E\left\{||\theta_k - \theta^*||^2\right\} &= E\left\{||\theta_0 - \theta^*||^2\right\} + E\left\{\sum_{l=1}^{k} \gamma_l^2 ||F_l||^2\right\} \\
&\quad - 2 \sum_{l=1}^{k} \gamma_l E\left\{N^{\theta_{l-1}} \left(\nabla v^{\theta_{l-1}}\right)^{\mathrm{T}} \left(\theta_{l-1} - \theta^*\right)\right\}.
\end{aligned}$$

Since F_k is bounded, we have $0 \leq \sum_{k=1}^{\infty} \gamma_k^2 ||F_k||^2 < \infty$. If $\lim_{k\to\infty} ||\theta_k - \theta^*|| = \infty$, according to Assumption 4 we have

$$\sum_{k=0}^{\infty} \gamma_k E\left\{N^{\theta_{k-1}} \left(\nabla v^{\theta_{k-1}}\right)^T \left(\theta_{k-1} - \theta^*\right)\right\} > \infty$$

which is contradictory to $E\left\{||\theta_k - \theta^*||^2\right\} \geq 0$. Therefore, there exists an $L > 0$ such that $||\theta_k - \theta^*|| < L$. And we have

$$\left\| \sum_{k=1}^{\infty} \gamma_k \omega_k^{\mathrm{T}} (\theta_{k-1} - \theta^*) \right\| < L \left\| \sum_{k=1}^{\infty} \gamma_k \omega_k \right\| < \infty.$$

Assume $\lim_{k\to\infty} \theta_k \neq \theta^*$. Then there exists a $C > 0$ such that $N^{\theta_{k-1}} \left(\nabla v^{\theta_{k-1}}\right)^T (\theta_{k-1} - \theta^*) > C$. Thus

$$\sum_{k=1}^{\infty} \gamma_k N^{\theta_{k-1}} \left(\nabla v^{\theta_{k-1}}\right)^{\mathrm{T}} \left(\theta_{k-1} - \theta^*\right) > C \sum_{k=1}^{\infty} \gamma_k = \infty,$$

which is contradictory to $||\theta_k - \theta^*||^2 \geq 0$. □

5. EXAMPLES

For illustration, we use our Markov decision model to describe two readaheads in the Linux kernel. One is the legacy Linux readahead, and the other is the *ondemand* Linux readahead, which had replaced the legacy Linux readahead in version 2.6.23.

The Linux kernel uses CLOCK-Pro [40] approximation algorithm for caching replacement, which makes h_c^M so complex. It also employs some variables to limit the readahead size. Let max be the upper limitation. The readahead size will be truncated if it exceeds max. For the legacy readahead, there is also a minimal disk I/O size. Let min be the lower limit, and the readahead size will be never smaller than min.

5.1. The legacy Linux readahead

Let $\mathbf{RA}_1$ denote the legacy Linux readahead facility. Its readahead state is $X_{\text{ah}}^1(n) = (\text{start}_n, \text{size}_n, \text{ahead\_start}_n, \text{ahead\_size}_n, \text{prev\_offset}_n, \text{flags}_n)$. There are two windows in its readahead state. The components start_n and size_n form the *current window*, while ahead_start_n and ahead_size_n form the *ahead window*. According to the algorithm, start_n or ahead_start_n is the page index from which the submitted disk I/O starts. And $size_n$ or ahead_size_n is the corresponding number of pages. *Current window* is constructed by initializing a new readahead or replaced by the *ahead window*. *Ahead window* records the last readahead after the readahead is initialized. The component prev_offset_n is the last visited page index of the last read request. The component flags_n is a partial summary to the cache state. The flag RA_FLAG_MISS implies that thrashing may occur. And the flag RA_FLAG_INCACHE implies that the sequential readahead is not necessary as the file is partly or fully cached. Let $\text{flags}_n = (\text{miss}_n, \text{incache}_n)$, where 1 denotes the flag is set, while 0 means it is not.

The state for any page i is $z_i(n) = 0$ or $z_i(n) = 1$, where 1 implies that the page is cached in memory, while 0 means that it is not. Let $X_c^1(n)$ be the cache state.

Let $\mathcal{U}_1 = \{0, 1, 2, 3\}$. Action 0 does not submit any disk I/O. Action 1 submits a disk I/O whose size is equal to the read request. And action 2 submits more than what the current request needs. Action 3 submits a one-page disk I/O.

Let the state for the legacy readahead be $X_1(n) = (X_p(n), X_{ah}^1(n), X_c^1(n))$. Assume that the current page access is $X_p(n) = (i, a, j)$. The prefetching policy is

$$d_1(s) = \begin{cases} 0, & i \neq j, z_j(n) = 1, \\ 1, & i = j,\ \text{beseq}_1(s) = 0, \\ 2, & i = j,\ \text{beseq}_1(s) = 1, \\ 3, & z_j(n) = 0, \end{cases}$$

where $\text{beseq}_1(s) = 1$ if $\text{incache}_n = 0$ and $i - \text{prev\_offset}_n \leq 1$, otherwise $beseq_1(s) = 0$. This policy will check the sequentiality and try to load the pages before accessing the first page of a read request. It will not recover the window if thrashing occurs. h_{ah}^1 strictly follows the readahead handling process.

5.2. The ondemand Linux readahead

Let $\mathbf{RA}_2$ be the *ondemand* Linux readahead facility where $\mathbf{S}_2$ is its state space. Its readahead state is $X_{\text{ah}}^2(n) = (\text{start}_n, \text{size}_n, \text{async\_size}_n,\ \text{prev\_offset}_n)$. start_n is the page index of the last submitted disk I/O while size_n is the number of pages. The state $z_i(n)$ for any page i is 0, 1 or 2. State $z_i(n) = 0$ denotes that page i is not in memory, while $z_i(n) = 1$ means that it is being cached. State $z_i(n) = 2$ implies that the page is in memory and the flag $PG\_readahead$ is set, which is introduced to support the readahead for interleaved sequential reads and concurrent streams since these read patterns will invalidate readahead states.

Let $\mathcal{U}_2 = \{0, 1, 2\}$. Action 0 does not submit any disk I/O. Action 1 submits a disk I/O whose size is equal to the read request. And action 2 submits more than what the current request needs.

Let $X_c^2(n)$ be the cache state. And let the state for the *ondemand* readahead be $X_2(n) = (X_p(n), X_{\text{ah}}^2(n), X_c^2(n))$. Assume that the current page access is $X_p(n) = (i, a, j)$. The prefetching policy is

$$d_2(s) = \begin{cases} 0, & z_j = 1, \\ 1, & z_j \in \{0, 2\}, \text{beseq}_2(s) = 0, \\ 2, & z_j \in \{0, 2\}, \text{beseq}_2(s) = 1, \end{cases} \tag{61}$$

where $\text{beseq}_2(s) = 1$ if $z_j(n) = 2$, $j - \text{prev\_offset}_n \leq 1$ or $\text{start}_n + \text{size}_n = j$, otherwise $\text{beseq}_2(s) = 0$. The *ondemand* readahead algorithm only does synchronous readahead when a page miss occurs and does asynchronous readahead when visiting the page with the $PG\_readahead$ flag. The operator h_{ah}^2 also strictly follows the handling process of the *ondemand* readahead.

6. EXPERIMENTS

In this section, we will verify the proposed models and then analyze and optimize the prefetching policies for those given workloads. The simulations will imitate the read handling and prefetching behaviors of the Linux kernel except the caching. As the caching facility of the Linux kernel is too complex to simulate, we use a simple LRU algorithm in simulations. It just uses one list and moves the hot pages to the head, while the tail page is the candidate to be evicted. The prefetching policies will convert to prefetching judgment statements before applying them. If the policy is stochastic, the judgment statements should employ random numbers to help in selecting actions by using the same way of simulating a stochastic process. Since the policies are state based, the conversions are very easy and automatic. To collect the information of performances, we also patch the kernel to trace every page's state. The tests for the legacy Linux readahead use kernel 2.6.22 and those for the *ondemand* Linux readahead use kernel 2.6.29.

6.1. Workloads

There are various workloads in the real world. Assume that the request size is BS and a read sequence has N_r requests. Suppose that the file size is $\mathrm{FSIZE} = \mathrm{BS} \times N_f \geq \mathrm{BS} \times N_r$, where N_f is the number of blocks with size BS. Here are four kinds of read sequences used in our experiments. A *Strictly Sequential Sequence* reads a file from the beginning to the end. *Random Sequence* randomly selects a block of the file to read. An *Interleaved Sequence* has K_r threads in an opened file descriptor and every thread is a strictly sequential sequence. A *Mixed Sequence* is a complex access pattern which has sequential reads and random reads mixed together where p_{rg} denotes the probability of sequentiality.

The file used in the following experiments is FSIZE = 128 MB, which has 32 768 pages. Such a file is long enough to illustrate the influence of a readahead facility. Let max = 128 KB (32 pages) and min = 16 KB (4 pages). Suppose that the request size is BS = 64 KB (16 pages), which is smaller than max, so that the normal readahead process works. The number of requests is $N_r = 1638$. The simulations will read the test file as quickly as possible. The number of threads in interleaved workloads is 4. When 4 is selected, the performance is not so good, but it can be improved.

6.2. Model verification

Tables 2 and 3 show the simulation results using the models and the actual results in the Linux kernel when there is enough memory. Tables 4 and 5 show the results when there is a memory of 16 MB. Since it is difficult to ensure that the Linux kernel uses 16 MB for file caching, the memory sizes here are approximate. The numbers after 'Mixed-' are $p_{\mathrm{rg}} \times 100$. The performances v^d_{mi}, v^d_{wa} and v^d_{th} are defined in Section 3.7. And every result is an average of 100 performance evaluations. As shown in these tables, the models meet well for the sequential and interleaved workloads. For the random and mixed ones, the simulation results are not as good as the sequential and interleaved. The caching facility in all simulations is very simple. It does not work the same as the kernel does, and the gap will become more distinct for more complicated workloads. Fortunately the differences here are very small. Therefore, we can conclude that the models can describe the legacy and *ondemand* Linux kernel readahead works very well.

TABLE 2. Model verification: legacy Linux readahead with enough memory.

	v^d_{mi} (%)	v^d_{wa} (%)	v^d_{th} (%)	
Sequential	0.004	0.122	0.000	Kernel
	0.004	0.122	0.000	Simulation
Random	4.30	0.020	0.000	Kernel
	4.30	0.025	0.000	Simulation
Interleaved	6.25	0.000	0.000	Kernel
	6.25	0.000	0.000	Simulation
Mixed-80	1.79	12.9	0.000	Kernel
	1.80	13.0	0.000	Simulation
Mixed-60	2.63	15.6	0.000	Kernel
	2.61	15.6	0.000	Simulation

TABLE 3. Model verification: *ondemand* Linux readahead with enough memory.

	v^d_{mi} (%)	v^d_{wa} (%)	v^d_{th} (%)	
Sequential	0.004	0.122	0.000	Kernel
	0.004	0.122	0.000	Simulation
Random	4.30	0.037	0.000	Kernel
	4.30	0.039	0.000	Simulation
Interleaved	5.77	0.043	0.000	Kernel
	5.77	0.042	0.000	Simulation
Mixed-80	1.40	13.7	0.000	Kernel
	1.44	13.6	0.000	Simulation
Mixed-60	2.42	17.0	0.000	Kernel
	2.46	16.4	0.000	Simulation

TABLE 4. Model verification: legacy Linux readahead with 16 MB memory.

	v^d_{mi} (%)	v^d_{wa} (%)	v^d_{th} (%)	
Sequential	0.004	0.122	0.000	Kernel
	0.004	0.122	0.000	Simulation
Random	5.63	0.040	0.008	Kernel
	5.54	0.034	0.009	Simulation
Interleaved	6.25	0.000	0.000	Kernel
	6.25	0.000	0.000	Simulation
Mixed-80	2.22	28.5	9.07	Kernel
	2.20	27.5	8.39	Simulation
Mixed-60	3.69	36.3	12.4	Kernel
	3.62	34.7	11.2	Simulation

6.3. Performance evaluation

By using the simulations, we can get not only the results in Tables 2, 3, 4 and 5 but also the performances summaries. In order to make the optimized prefetching policy aggressive, the value of α_{mi} must be larger than the ones of α_{wa} and α_{th}. Larger α_{mi} will lead to more prefetches. Also the value of α_{wa} should be small since larger α_{wa} will make the kernel perform fewer prefetches. The parameter α_{th} is used to make sure that the policy is not so aggressive because it will lead to a serious thrashing problem if the policy is too aggressive. Let

TABLE 5. Model verification: *ondemand* Linux readahead with a memory of 16 MB.

	v^d_{mi} (%)	v^d_{wa} (%)	v^d_{th} (%)	
Sequential	0.004	0.122	0.000	Kernel
	0.004	0.122	0.000	Simulation
Random	5.59	0.077	0.012	Kernel
	5.54	0.079	0.015	Simulation
Interleaved	5.72	0.047	0.000	Kernel
	5.80	0.039	0.000	Simulation
Mixed-80	2.05	27.8	8.68	Kernel
	2.03	27.9	8.54	Simulation
Mixed-60	3.60	35.8	12.1	Kernel
	3.58	35.2	11.5	Simulation

$\alpha_{\mathrm{mi}} = 1$, $\alpha_{\mathrm{wa}} = 0.01$ and $\alpha_{\mathrm{th}} = 0.1$. According to Table 6, the performances of sequential and random workloads for both Linux kernel readaheads are similar. Since the *ondemand* readahead supports many semi-sequential access patterns, it works better for the interleaved and mixed workloads. This is why the legacy readahead had been replaced by the *ondemand* readahead in the Linux kernel. As Table 6 shows, the simulation results basically agree with the practice, but the gap gets bigger for more complex workloads because the simulations use a simpler caching facility than the Linux kernel does. And when there is less memory, the caching becomes more active, and so the gap also gets bigger. Therefore the mixed workload with 40% of sequentiality and less caching memory has a big difference (2.5 vs. 0.2). Since we have tested a real Linux kernel by using the same workloads, the confidence intervals of simulations are not discussed here. Instead, we introduce ξ to confirm the availability of comparison. The value of ξ is calculated as follows:

$$\xi = \frac{|v_1 - \tilde{v}_1| + |v_2 - \tilde{v}_2|}{|v_1| + |v_2|}, \tag{62}$$

where v_1, v_2 are the actual performances and $\tilde{v}_1, \tilde{v}_2$ are the estimates for two readaheads. Thus, let $\xi = 2.7 \times 10^{-2}$ in Table 6, which is the maximum among the values calculated according to (62). If a simulation has a result whose err is in $(-2.7 \times 10^{-2}, 2.7 \times 10^{-2})$, we cannot decide which readahead or prefetching policy is better. Therefore, we can have the same conclusions from simulations as those from the real world.

6.4. Prefetching policy analysis

The current *ondemand* Linux kernel readahead divides the state space $\mathcal{S}_2$ into four subspaces, which satisfy $\mathcal{S}_2 = \mathcal{S}_{21} \cup \mathcal{S}_{22} \cup \mathcal{S}_{23} \cup \mathcal{S}_{24}$ and $\mathcal{S}_{21} \cap \mathcal{S}_{22} \cap \mathcal{S}_{23} \cap \mathcal{S}_{24} = \emptyset$. The state $s \in \mathcal{S}_{21}$ denotes that the current visited page is missing and the access is considered to be not sequential. The state $s \in \mathcal{S}_{22}$ denotes that the current page is also missing, but the access is considered to be sequential. The state $s \in \mathcal{S}_{23}$ denotes that the current accessed page has the PG_readahead flag. And the left states are put in $\mathcal{S}_{24}$. Let $\mathcal{U}_{2i}$ be the set of allowable actions for the states in $\mathcal{S}_{2i}$, where $i = 1, 2, 3, 4$. According to the *ondemand* policy (61), the states in the same subspace use the same decision rule. When it is in $\mathcal{S}_{21}$, it just fetches what it needs. In $\mathcal{S}_{22}$ or $\mathcal{S}_{23}$, it performs a readahead. The kernel will do nothing when the state is in $\mathcal{S}_{24}$. Let $\mathcal{U}_{21} = \mathcal{U}_{22} = \{1, 2\}, \mathcal{U}_{23} = \{0, 2\}$ and $\mathcal{U}_{24} = \{0\}$ where the actions are defined as $\mathcal{U}_2$ in Section 5.2. Parameterize the decision rules for $\mathcal{S}_{21}$, $\mathcal{S}_{22}$ and $\mathcal{S}_{23}$. The policy parameter vector is $\theta = (\theta^1, \theta^2, \theta^3) = (1, 0, 0)$, where θ^i is the probability that the readahead is not performed in the state $s \in \mathcal{S}_{2i}$, where $i = 1, 2, 3$. Let $\theta' = (\theta'^1, \theta'^2, \theta'^3) = (0.9999, 0.0001, 0.0001)$ as θ is not well defined.

Table 7 shows the estimated gradients of the given interleaved and random workloads for the *ondemand* Linux readahead with 16 MB memory. From the normalized gradients, we found that the normalized components for θ'^1 are, respectively, 0.999 and -1.000. It means that the interleaved access pattern may want to perform a readahead in the state $s \in \mathcal{S}_1$ but the random access pattern may not.

TABLE 6. Performance evaluation.

	MEM>FSIZE			MEM=16 MB			
	Legacy (10^{-2})	*ondemand* (10^{-2})	err (10^{-2})	legacy (10^{-2})	*ondemand* (10^{-2})	err (10^{-2})	
Sequential	0.005	0.005	0.0	0.005	0.005	0.0	Kernel
	0.005	0.005	0.0	0.005	0.002	0.0	Simulation
Random	4.30	4.30	0.0	5.63	5.59	0.1	Kernel
	4.30	4.30	0.0	5.54	5.54	0.0	Simulation
Interleaved	6.25	5.77	7.7	6.25	5.72	8.5	Kernel
	6.25	5.77	7.7	6.25	5.80	7.2	Simulation
Mixed-80	1.92	1.54	19.8	3.41	3.19	6.4	Kernel
	1.93	1.58	18.1	3.31	3.16	4.5	Simulation
Mixed-60	2.79	2.59	7.2	5.29	5.16	2.5	Kernel
	2.77	2.62	5.4	5.09	5.08	0.2	Simulation

TABLE 7. Estimated gradients: *ondemand* Linux readahead with 16 MB memory.

	Interleaved	Workloads normalized	Random	Workloads normalized
θ'^1	11.6	0.999	−0.127	−1.000
θ'^2	0.003	0.000	0.000	0.001
θ'^3	0.481	0.041	0.000	0.000

Since the interleaved workloads may invalidate the readahead state, many states in which we can perform a readahead are considered to be not sequential. The *ondemand* Linux readahead has already introduced a PG_readahead flag to perform a readahead when the readahead state is invalid. But the estimated gradients show that further improvement is possible. Divide the space $\mathcal{S}_{21}$ to $\mathcal{S}_{211}$ and $\mathcal{S}_{212}$ according to the page states (a state will belong to $\mathcal{S}_{212}$ if the previous page is cached). Let the policy parameters be θ^{11} and θ^{12}. Then, we can optimize them by using Algorithm 1 and confirm our guess that readaheads should be performed for states in $\mathcal{S}_{212}$.

6.5. Prefetching policy optimization

Following Section 6.4, divide the state space $\mathcal{S}_2$ to five subspaces, which are $\mathcal{S}_{211}$, $\mathcal{S}_{212}$, $\mathcal{S}_{22}$, $\mathcal{S}_{23}$ and $\mathcal{S}_{24}$. The policy parameter vector is $\theta = (\theta^{11}, \theta^{12}, \theta^2, \theta^3)$. According to the policy (61) used by the *ondemand* Linux readahead, the current prefetching policy parameter vector is $\theta = (1, 1, 0, 0)$.

Let $\theta_0 = (0.9999, 0.9999, 0.0001, 0.0001)$ be the initial parameter vector for Algorithm 1. It is obvious that θ_0 is well defined and very close to $(1, 1, 0, 0)$. In order to achieve good convergence curves, γ should be adjusted. Figure 3 presents the simulation results of interleaved workloads for the ondemand Linux readahead with 16 MB memory where $\gamma = 0.3/k$. The curves of θ_k and v_k show that the Algorithm 1 converges.

TABLE 8. Performance evaluation: *ondemand* Linux readahead with 16 MB memory.

	$v^{\rm d}_{\rm mi}$ (%)	$v^d_{\rm wa}$ (%)	$v^d_{\rm th}$ (%)	v^d (10^{-2})	
Random	5.59	0.077	0.012	5.59	Original
	5.60	9.14	2.25	5.92	Improved
Interleaved	5.72	0.047	0.000	5.72	Original
	0.027	0.617	0.000	0.033	Improved

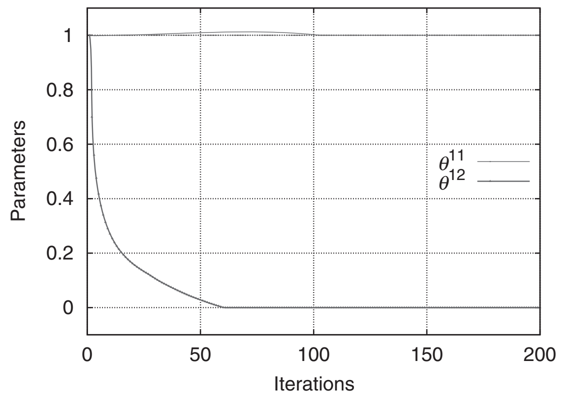
Table 8 shows the performances of the original *ondemand* Linux readahead and its improvement. The original policy of the *ondemand* readahead has performance of 5.72×10^{-2} for the given interleaved workloads and its improvement has 0.033×10^{-2}. The performance for random access pattern is reduced by 6%, while that of interleaved pattern has a significant increase. It is obvious that we have obtained a good improvement.

This improved prefetching policy has been already adopted by official Linux 2.6.31. Feedback shows that there seems to be no big negative impacts, interleaved reads and cooperative IO processes such as NFS reads, and SCST (Generic SCSI Target Subsystem for Linux) can benefit from it.

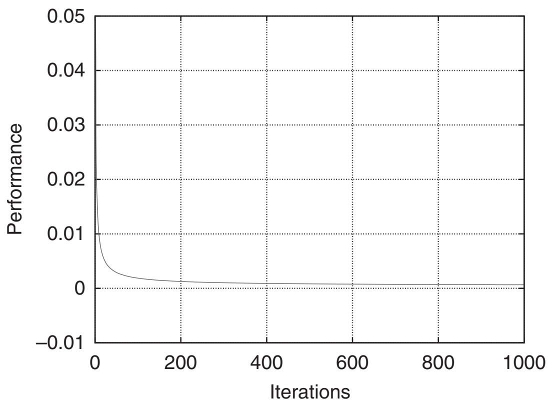
7. CONCLUSION

We have introduced Markov decision models to describe the kernel readahead subsystem integrated with the kernel read handling and caching facilities. By using these models, we can evaluate and compare various readaheads, which can help to reduce many jobs for practical developments of readahead subsystems.

Based on the models, we also derived the performance gradient formula. The analysis of gradient can help us obtain the knowledge of the weaknesses of existent readaheads. Moreover, the gradient optimization algorithm is proposed and



(a) Policy Parameters (θ_k)



(b) Performance ($\tilde{v}_k$)

FIGURE 3. Prefetching policy optimization.

convergence is discussed. The algorithms need no knowledge on the transition matrix as they are all sample-path based. The simulations do not have to maintain the large transition matrix.

For illustration, the *legacy* and *ondemand* Linux kernel readaheads are employed. The results show that the models for both readaheads and the comparison agree well with the practice. We also analyzed the currently used policy for the *ondemand* Linux kernel readahead according to the estimated gradients. Following the analysis, the prefetching policy is optimized by using the proposed algorithm. The tests in practice have shown that the improved policy is better than the original one.

There is a lot of room for further research. The current MDP models are based on a simple assumption of the caching subsystem that there is a cache with a fixed size for an opened file descriptor and no other opened file descriptor shares this cache. In fact, one file may have many opened file descriptions at the same time. It is necessary to bring more information of caching subsystem into the models. For dealing with the relationship between different opened file descriptions, time is essential. The next step would be to construct semi-Markov decision processes to describe these behaviors more exactly. Furthermore, a real readahead subsystem generally tends to use partial information of a state to make a decision. We can develop partially observable Markov decision models to build more general models and construct more general algorithms for low-level readaheads.

FUNDING

This work was partly supported by National Natural Science Foundation of P.R. China under grant No. 60774038, and National 863 HI-TECH Research & Development Plan of P.R. China under grant No. 2008AA01A317.

REFERENCES

[1] Shriver, E., Small, C. and Smith, K. A. (1999) Why Does File System Prefetching Work? *Proc. Annual Conf. USENIX Annual Technical Conf.*, Monterey, CA, June 6–11, pp. 71–84. USENIX Association, Berkeley, CA.

[2] Feiertag, R.J. and Organick, E.I. (1971) The Multics Input/Output System. *Proc. Third ACM Symp. Operating Systems Principles*, Palo Alto, CA, October 18–20, pp. 35–41. ACM, New York, NY.

[3] McKusick, M.K., Joy, W.N., Leffler S.J. and Fabry R.S. (1984) A fast file system for unix. *ACM Trans. Comput. Syst.*, **2**, 181–197.

[4] Kroeger, T.M. and Long, D.D.E. (2001) Design and Implementation of a Predictive File Prefetching Algorithm. *Proc. General Track: 2001 USENIX Annual Technical Conf.*, Boston, MA, June, 25–30 pp. 105–118. USENIX Association, Berkeley, CA.

[5] Paris, J.-F., Amer, A. and Long, D.D.E. (2003) A Stochastic Approach to File Access Prediction. *Proc. Int. Workshop on Storage Network Architecture and Parallel I/Os*, New Orleans, LA, September 28, pp. 36–40. ACM, New York, NY.

[6] Whittle, G.A.S., Paris, J.-F. Amer, A., Long, D.D.E. and Burns R. (2003) Using Multiple Predictors to Improve the Accuracy of File Access Predictions. *Proc. 20th IEEE/11th NASA Goddard Conf. Mass Storage Systems and Technologies* (*MSS'03*), San Diego, CA, April 7–10, pp. 230–240. IEEE Computer Society, Washington, DC.

[7] Li, Z., Chen, Z. and Zhou Y. (2005) Mining block correlations to improve storage performance. *Trans. Storage*, **1**, 213–245.

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

[9] Patterson, R.H., Gibson, G.A., Ginting, E., Stodolsky, D. and Zelenka, J. (1995) Informed prefetching and caching. *ACM SIGOPS Oper. Syst. Rev.*, **29**, 79–95.

[10] Patterson, R.H., Gibson, G.A. and Satyanarayanan, M. (1993) A status report on research in transparent informed prefetching. *ACM SIGOPS Oper. Syst. Rev.*, **27**, 21–34.

[11] Brown, A.D., Mowry, T.C. and Krieger, O. (2001) Compiler-based i/o prefetching for out-of-core applications. *ACM Trans. Comput. Syst.*, **19**, 111–170.

[12] Butt, A.R., Gniady, C. and Hu, Y.C. (2007) The performance impact of kernel prefetching on buffer cache replacement algorithms. *IEEE Trans. Comput.*, **56**, 889–908.

[13] Cao, P., Felten, E.W., Karlin, A.R. and Li, K. (2005) A study of integrated prefetching and caching strategies. *ACM SIGMETRICS Perform. Eval. Rev.*, **33**, 157–168.

[14] Dini, G., Lettieri, G. and Lopriore, L. (2006) Caching and prefetching algorithms for programs with looping reference patterns. *Comput. J.*, **49**, 42-61.

[15] Gill, B.S. and Modha, D.S. (2005) SARC: Sequential Prefetching in Adaptive Replacement Cache. *Proc. Annual Conf. USENIX Annual Technical Conf.*, Anaheim, CA, April 10–15, pp. 293–308. USENIX Association Berkeley, CA.

[16] Papathanasiou, A.E. and Scott, M.L. (2005) Aggressive Prefetching: An Idea Whose Time has Come. *Proc. 10th Conf. Hot Topics in Operating Systems, Vol. 10*, Santa Fe, NM, June 12–15, p. 6 USENIX Association, Berkeley, CA.

[17] Puterman, M.L. (1994) *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, New York.

[18] Joseph, D. and Grunwald, D. (1999) Prefetching using Markov predictors. *IEEE Trans. Comput.*, **48**, 121–133.

[19] Madhyastha, T.M. and Reed, D.A. (1997) Input/Output Access Pattern Classification Using Hidden Markov Models. *Workshop on Input/Output in Parallel and Distributed Systems*, San Jose, CA, November 17, pp. 57–67. ACM, New York.

[20] Bartels, G., Karlin, A., Anderson, D., Chase, J., Levy, H. and Voelker, G. (1999) Potentials and limitations of fault-based Markov prefetching for virtual memory pages. *ACM SIGMETRICS Perform. Eval. Rev.*, **17**, 206–207.

[21] Oly, J. and Reed, D.A. (2002) Markov Model Prediction of IO Requests for Scientific Applications. *Proc. 16th ACM Int. Conf. Supercomputing*, New York, June 22–16, pp. 147–155, ACM, New York.

[22] Sarukkai, R. (2000) Link prediction and path analysis using Markov Chains. *Comput. Netw.*, **33**, 377–386.

[23] Zhu, J., Hong, J. and Hughes, J.G. (2002) Using Markov Models for Web Site Link Prediction. *Proc. Thirteenth ACM Conf.*

Hypertext and Hypermedia, College Park, MD, USA, June 11–15, pp. 169–170. ACM, New York.

[24] Zhu, J., Hong, J. and Hughes, J.G. (2002) Using Markov chains for link prediction in adaptive web sites. *Lect. Notes Comput. Sci.* **2311**, 55–66.

[25] Shi, L, Yao, Y. and Wei, L. (2007) On the Compression of Markov Prediction Model. *Proc. Fourth Int. Conf. Fuzzy Systems and Knowledge Discovery*, Vol. 1, Haikou, Hainan, China, August 24–27, pp. 512–516. IEEE Computer Society, Washington, DC.

[26] Xing, D. and Shen, J. (2002) A new Markov model for web access prediction. *Comput. Sci. Eng.*, **4** 34–39.

[27] Grigoras, R., Charvillat, V. and Douze, M. (2002) Optimizing Hypervideo Navigation Using a Markov Decision Process Approach. *Proc. Tenth ACM Int. Conf. Multimedia*, Juan-les-Pins, France, December 1–6, pp. 39–48. ACM, New York.

[28] Charvillat, V. and Grigoras, R. (2007) Reinforcement learning for dynamic multimedia adaptation. *J. Netw. Comput. Appl.*, **30**, 1034–1058.

[29] Li, C. and Shen, K. (2005) Managing Prefetch Memory for Data-Intensive Online Servers. *Proc. 4th Conf. USENIX Conf. File and Storage Technologies, Vol. 4*, San Francisco, CA, December 13–16, p. 19. USENIX Association, Berkeley, CA.

[30] Gill, B.S. and Bathen, L.A.D. (2007) Optimal multistream sequential prefetching in a shared cache. *Trans. Storage*, **3**, 10.

[31] Li, C., Shen, K. and Papathanasiou, A.E. (2007) Competitive prefetching for concurrent sequential I/O. *ACM SIGOPS Oper. Syst. Rev.*, **41**, 189–202.

[32] Liang, S., Jiang, S. and Zhang, X. (2007) Step: Sequentiality and Thrashing Detection Based Prefetching to Improve Performance of Networked Storage Servers. *ICDCS'07*, Toronto, Canada, June 25–27, p. 64, IEEE Computer Society.

[33] Revel, D., McNamee, D., Steere, D. and Walpole, J. (1998) Adaptive Prefetching for Device Independent File I/O. *Proc. Multimedia Computing and Networking* 1998, San Jose, CA, January 26–28, pp. 139–149. Society of Photo-optical Instrumentation Engineers, Bellingham, WA.

[34] Bhattacharya, S., Tran, J., Sullivan, M. and Mason, C. (2004) Linux Aio Performance and Robustness for Enterprise Workloads. *Proc. Linux Symp.*, Ottawa, Canada, July 21–24, pp. 63–77.

[35] Ellard, D., Ledlie, J., Malkani, P. and Seltzer, M. (2003) Passive NFS Tracing of Email and Research Workloads. *Proc. Second USENIX Conf. File and Storage Technologies*, San Francisco, CA, March 31, pp. 203–216. USENIX Association, Berkeley, CA.

[36] Ellard, D. and Seltzer, M. (2003) NFS Tricks and Benchmarking Traps. *Proc. FREENIX 2003 Technical Conf.*, San Antonio, TX, June 9–14, pp. 101–114. USENIX Association, Berkeley, CA.

[37] Wu, F., Xi, H., Li, J. and Zou, N. (2007) Linux Readahead: Less Tricks for more. *Proc. Linux Symp.*, Ottawa, Canada, June 27–30, pp. 273–284.

[38] Wu, F., Xi, H. and Xu, F. (2008) On the design of a new Linux readahead framework. *ACM SIGOPS Oper. Syst. Rev.*, **42**, 75–84.

[39] Cao, X.R. (2004) The potential structure of sample paths and performance sensitivities of Markov systems. *IEEE Trans. Autom. Control*, **49**, 2129–2142.

[40] Jiang, S., Chen, F. and Zhang, X. (2005) CLOCK-Pro: An Effective Improvement of the CLOCK Replacement. *Proc. USENIX Ann. Technical Conf.*, Anaheim, CA, April 10–15, pp. 323–336. USENIX Association.