Chapter 11
Sequential File Prefetching In Linux

Fengguang Wu
Intel Corporation, China

ABSTRACT

Sequential prefetching is a well established technique for improving I/O performance. As Linux runs an increasing variety of workloads, its in-kernel prefetching algorithm has been challenged by many unexpected and subtle problems; As computer hardware evolves, the design goals should also be adapted. To meet the new challenges and demands, a prefetching algorithm that is aggressive yet safe, flexible yet simple, scalable yet efficient is desired. In this chapter, the author explores the principles of I/O prefetching and present a demand readahead algorithm for Linux. He demonstrates how it handles common readahead issues by a host of case studies. Both static, logic and dynamic behaviors of the readahead algorithm are covered, so as to help readers building both theoretical and practical views of sequential prefetching.

INTRODUCTION

Sequential prefetching, also known as readahead in Linux, is a widely deployed technique to bridge the huge gap between the characteristics of storage devices and their inefficient ways of usage by applications. At one end, disk drives and disk arrays are better utilized by large accesses. At the other, applications tend to do lots of tiny sequential reads. To make the two ends meet, operating systems do prefetching: to bring in data blocks for the upcoming requests, and do so in large chunks.

The Linux kernel does sequential file prefetching in a generic readahead framework that dates back to 2002. It actively intercepts file read requests in the VFS layer and transforms the sequential ones into large and asynchronous readahead requests. This seemingly simple task turns out to be rather tricky in practice (Pai, Pulavarty, and Cao, 2004; Wu, Xi, Li, and Zou, 2007). The wide deployment of Linux --- from embedded devices to supercom-

DOI: 10.4018/978-1-60566-850-5.ch011

puters --- confronts its readahead algorithm with an incredible variety of workloads.

In the mean while, there are two trends that bring new demands and challenges to prefetching. Firstly, the relative cost of disk seeks has been growing steadily. In the past 15 years, the disk bandwidth improved by 90 times, while the disk access latency only saw 3-4 times speedup(Schmid, 2006). As an effective technique for reducing seeks, prefetching is thus growing more and more important. However the traditional readahead algorithm was designed in a day when memory and bandwidth resources are scare and precious. It was optimized for high readahead hit ratio and may be too conservative for today's hardware configuration. We need a readahead algorithm that put more emphasis on "avoiding seeks".

Secondly, I/O parallelism keeps growing. As it becomes more and more hard to increase single-thread performance, the parallel execution of a pool of threads raises to be a new focus. The deserialize of hardware and software means the parallelization of I/O. Prefetching helps parallel I/O performance. In turn it is also challenged by the forms and degree of I/O concurrency: How to detect and keep track of states for multiple I/O streams that are interleaved together? How to maintain good readahead behavior for a sequential stream that is concurrently served by a pool of cooperative threads? How to achieve low time and space overheads in the cases of single thread I/O as well as highly concurrent I/O?

The questions will be answered in the following sections. We will start by the introduction to prefetching with its values for storage devices and roles in I/O optimization. We will give a reference to basic forms of prefetching, discuss the design tradeoffs and argue for more aggressive prefetching policies. We then proceed to introduce the demand readahead algorithm in Linux and explain the differences to the legacy one. At last we demonstrate its dynamic behaviors and advantages by a host of case studies and benchmarks.

PRINCIPLES OF I/O PREFETCHING

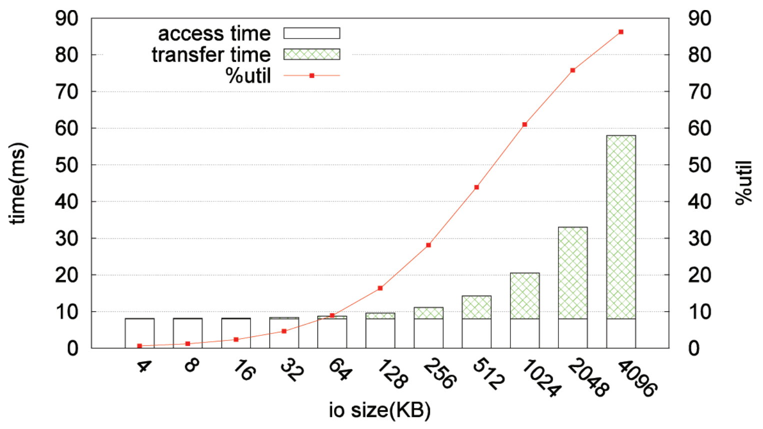
Bandwidth and latency are the two major aspects of I/O performance. For both metrics, there have been huge and growing performance gaps between disk, memory and processor. For example, the Intel$^{(R)}$ QuickPath$^{(TM)}$ Interconnect(QPI) can provide 12.8GB/s bandwidth per direction for processor-to-processor and processor-to-io data transfers. Today's DDR3-1333 memory has a theoretical bandwidth of 10666MB/s and response time of 12 nanoseconds, while a Seagate$^{(R)}$ 7200.11 SATA disk has a maximum sustained transfer rate of 105MB/s and average seek time of 8.5ms. Hence the performance gap as of 2009 is about 10 times for bandwidth and 7e5 times for latency. In this section, we demonstrate how I/O prefetching can help fill the performance gaps.

Storage Devices

The I/O latency is such a dominant factor in disk I/O operations that it can be approximated by a simple I/O model. A typical disk I/O takes two steps: Firstly, the disk head moves to the data track and waits for the data sector to rotate under it; Secondly, data read and transfer start. Correspondingly there are two operational times: the average access time, whose typical value is 8ms; the data transfer time, which roughly equals to the multiplication of I/O size and the disk's sustained transfer rate, which averages to 80MB/s for commodity disks today.

In a full I/O period, only the data transfer time makes real utilization of the disk data channel. The larger I/O size, the more time will be spent in data transfer and less time wasted in seeking, hence the more we can harvest disk utilization ratio and I/O bandwidth. Figure 1 reflects this correlation given the above disk I/O model and representative parameter values. One major function of I/O prefetching is to shift a disk's working point from left to right in the graph, so as to achieve better disk utilization and I/O bandwidth.

Figure 1. Disks can be better utilized with larger I/O sizes



Two exciting advances about cache media are the great abundance of dynamic memory and the general availability of flash memory. One might expect them to bring negative impacts to the relevance of prefetching techniques. When more data can be cached in the two types of memories, there would be less disk I/Os and readahead invocations. However, in this era of information explosion, data set and disk size grow rapidly at the same time. There are increasing I/O intensive applications and I/O parallelism that demand more flexible, robust and aggressive prefetching. Another consequence of the larger memory is, aggressive prefetching becomes a practical consideration for modern desktop systems. A well known example is boot time prefetching for fast system boot and application startup(Esfahbod, 2006).

Flash memory and its caching algorithms fit nicely in one big arena where magnetic disk and its prefetching algorithms are not good at: small random accesses. The Intel$^{(R)}$ turbo memory and hybrid hard drive are two widely recognized ways to utilize the flash memory as a complementary cache for magnetic disks. And apparently the solid-state disk(SSD) is the future for mobile computing. However, the huge capacity gap isn't closing any time soon. Hard disks and storage networks are still the main choice in the foreseeable future to meet the unprecedented storage demand created by the explosion of digital information, where readahead algorithms will continue to play an important role.

The solid-state disks greatly reduced the costly seek time, however there are still non-trivial access delays. In particular, SSD storage is basically comprised of a number of chips operating in parallel, and the larger prefetching I/O will be able to take advantage of the parallel chips. The optimal I/O size required to get full performance from the SSD storage will be different from spinning media, and vary from device to device. So I/O prefetching with larger and tunable size is key even on SSD.

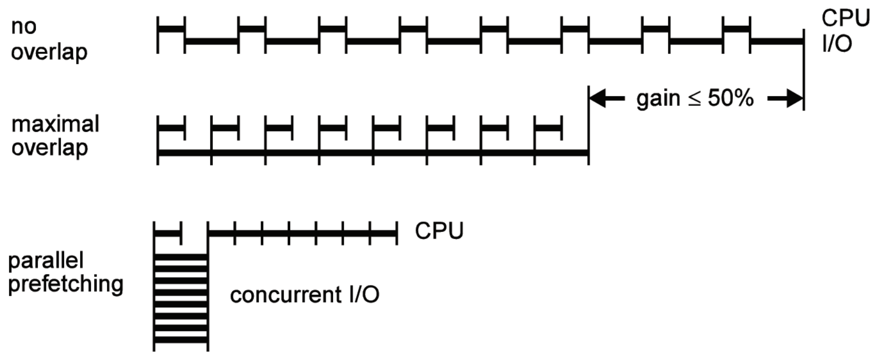
In summary, where there is sequential access patterns, there's arena for I/O prefetching. Whether it be platter-based disk or solid state disk.

2.2 I/O Optimization and Prefetching

According to (Russel, 1997), there are four basic I/O optimization strategies:

Avoidance. The best option is to avoid the costly disk accesses totally or to reduce disk access frequency. This can achieved by file caching. Prefetching is also good at converting small read requests into large ones, which effectively reduces the number of disk accesses and therefore the costly seeks. Another concrete example is the well known

Figure 2. Prefetching helps achieve I/O asynchrony and parallelism



Linux VFS(Virtual File System) mount options noatime and relatime for eliminating undesirable writes triggered by mtime updates.

Sequentiality. Sequential accesses enable sequential prefetching and maximize disk utilization. For concurrent sequential accesses, prefetching plays an essential role in aggregating interleaved tiny I/Os into larger ones. For non-sequential accesses, it still benefits to minimize seek and rotational delays by employing techniques such as smart disk layout management, informed prefetching, I/O queuing and scheduling. To name a few real world examples: the deadline, anticipatory and CFQ(Complete Fairness Queuing) I/O elevators in Linux; the TCQ(Tagged Command Queuing) for SCSI disks and NCQ(Native Command Queuing) for SATA disks; the delayed allocation and pre-allocation features of ext4/xfs; the writeback clustering in xfs; etc.

Asynchrony. Asynchronous accesses improve I/O computing efficiency by pipelining processor and disk activities, and hiding I/O latencies to the application. AIO, non-blocking I/O, writeback and readahead are common facilities for asynchronous I/O.

Parallelism. Disk arrays have been a standard practice for aggregating multiple disks' capacity as well as bandwidth. Beyond the traditional RAID layer, the emerging file systems exemplified by zfs and btrfs target to manage large pool of disks on their own. For large scale cloud computing and high performance computing, large pools of storage servers can be organized with the Google file system, Lustre, pNFS or other cluster file systems. On the other hand, device level parallelism are being employed inside SSD. For example, Intel$^{(R)}$ pioneered 10 parallel NAND flash channels in its SATA solid-state drives to deliver up to 250MB/s read bandwidth and 35,000 IOPS. Concurrent I/O requesting and parallel data transfer are the keys to I/O throughput of the above parallel systems. Aggressive prefetching plays a vital role in this arena: they typically need large and asynchronous prefetching I/O to fill the parallel data channels.

It is obvious that prefetching plays an important role in each of the four I/O optimization strategies. Prefetching brings benefits to applications, storage devices and storage pools, and even processing resources(Shriver, Small, and Smith, 1999). Applications can run smoother and faster by masking the low level I/O delays. Disks can be better utilized by large I/O size. Storage pools can be better parallelized. The larger I/O unit also helps amortize processing overheads in the whole I/O path.

Basic Approaches To Prefetching

Prefetching algorithms can be either heuristic or informed. The heuristic algorithms try to predict I/O blocks to be accessed in the near future based on the past ones. The most successful one is sequential readahead, which has long been a standard practice among operating systems (Feiertag and Organick, 1971; McKusick, Joy, Leffler, and Fabry, 1984). There are also more comprehensive works to mine correlations between files (Kroeger and Long, 2001; Paris, Amer, and Long, 2003; Whittle, Paris, Amer, Long, and Burns, 2003) or data blocks (Li, Chen, and Zhou, 2005).

On the other hand, informed prefetching works with hints from individual applications about their future I/O operations. The hints could either be explicitly controlled by application (Cao, Felten, Karlin, and Li, 1996; Patterson, Gibson, Ginting, Stodolsky, and Zelenka, 1995; Patterson, Gibson, and Satyanarayanan, 1993) or be automatically generated (Brown, Mowry, and Krieger, 2001).

Caching is another ubiquitous performance optimization technique. It is a common practice to share prefetching memory with cache memory, this opens the door to interactions between prefetching and caching. Besides the basic defensive support for readahead thrashing discussed in this chapter, there are also comprehensive works dealing with integrated prefetching and caching (Butt, Gniady, and Hu, 2005; Cao, Felten, Karlin, and Li, 1995; Cao, Felten, Karlin, and Li, 1996; Dini, Lettieri, and Lopriore, 2006; Gill and Modha, 2005; Patterson, Gibson, Ginting, Stodolsky, and Zelenka, 1995; Itshak and Wiseman, 2008) and schemes to dynamically adapt the prefetching memory (Li and Shen, 2005) or prefetch depth (Gill and Bathen, 2007; Li, Shen, and Papathanasiou, 2007; Liang, Jiang, and Zhang, 2007).

Design Tradeoffs Of Prefetching

The prefetching size can greatly impact I/O performance and is considered as the main I/O parameter. One must tradeoff between throughput and latency on determining its value. The general guideline is: prefetching size should be large enough to deliver good I/O throughput and small enough to prevent undesirable long I/O latencies. Different storage devices, disk array configurations and workloads have different optimal I/O size. Some applications(e.g. FTP) are not sensitive to I/O latency and therefore can safely use a large readahead size; Some others(e.g. VOD) may be sensitive to I/O latency and should use a more conservative I/O size.

Besides the tradeoff between throughput and latency, prefetching hit ratio is another common design consideration. It is defined as: for all the data pages faulted in by the prefetching algorithm, how much of them are actually accessed before being reclaimed. To keep prefetching hit ratio high, adaptive readahead size shall be employed. This is because even we are sure that an application is doing sequential reads, we have no idea how long the read sequence will last. For example, it is possible that an application scans one file from beginning to end, while only accesses the first two pages in another file.

Fortunately, the common I/O behaviors are somehow inferable. Firstly, the number of pages to be read(not considering the end-of-file case) and the number of pages have been accessed are normally positively correlated. Secondly, the larger read size, the more possibility it will be repeated. Because the large read size implies an optimization made by the developer for an expected long run of reads. Based on the above two empirical rules, the possibility of the current access pattern being repeated can be estimated, upon which an adaptive prefetching size can be calculated.

The prefetching hit ratio has served as one major goal in the design of prefetching algorithms. A low hit ratio means waste of memory and disk I/O resources. The cost was high and unacceptable in a day when these resources are scare and precious. So traditional prefetching algorithms tend to do prefetching for only strictly sequential reads. They employ conservative policies on prefetching size as well as sequential detection, seeking for a high prefetching hit ratio. For example, Linux 2.6 has been using a conservative 128KB readahead size since its infancy.

However, with the rapid evolvement of computer hardware, we are now facing new constraints and demands. The bandwidth and capacity of memory and disk both improved greatly, while the disk access times remain slow and become more and more an I/O bottleneck. As a consequence, the benefit of readahead hits goes up. It not only adds importance to prefetching, but also favors aggressive prefetching.

The cost of prefetching misses has been much lowered, reducing the performance relevance of prefetching hit ratio. There are two major costs of prefetching miss: Firstly, wasted memory cache. When more memory is taken by prefetching, less is available for caching. This may lower the cache hit ratio and increase number of I/O. However, in the context of memory abundance, the impact of cache size on cache hit ratio should be limited. Secondly, wasted disk bandwidth. It is relatively easy to estimate the impact of prefetching misses on I/O bandwidth. For an I/O intensive application, if a page being asynchronously prefetched is accessed later, it typically means one less I/O operation, assume that the saved disk seek time is 5ms; on the other hand, a prefetching miss page costs about 4KB/80MBps=0.05ms extra disk transfer time. From these two numbers alone, we could say that a prefetching hit ratio larger than 1% would be beneficial if not optimal.

So it deserves to use more aggressive prefetching polices. It can improve overall I/O performance even when sacrificing the hit ratio. The prefetching hit ratio of a workload depends on its accuracy of pattern recognition and pattern run length estimation. For sequential workloads, loosing the pattern recognition accuracy means that the strict page-after-page pattern matching for read requests is no longer necessary, which will allow the detection of more interleaved and semi-sequential patterns; loosing the run length estimation accuracy means larger prefetching size.

It is argued in (Papathanasiou and Scott, 2005) that more aggressive prefetching policies be employed to take advantage of new hardware and fulfill increasing I/O demands. Our experiences are backing it: there are risks of regression in extending the sequential detection logic to cover more semi-sequential access patterns, however we received no single negative feedback since the wide deployment of the new policies in Linux 2.6.23. It's all about performance gains in practice, including the tricky retry based AIO reads(Bhattacharya, Tran, Sullivan, and Mason, 2004) and locally disordered NFS reads(Ellard, Ledlie, Malkani, and Seltzer, 2003; Ellard and Seltzer, 2003), which will be discussed in this chapter.

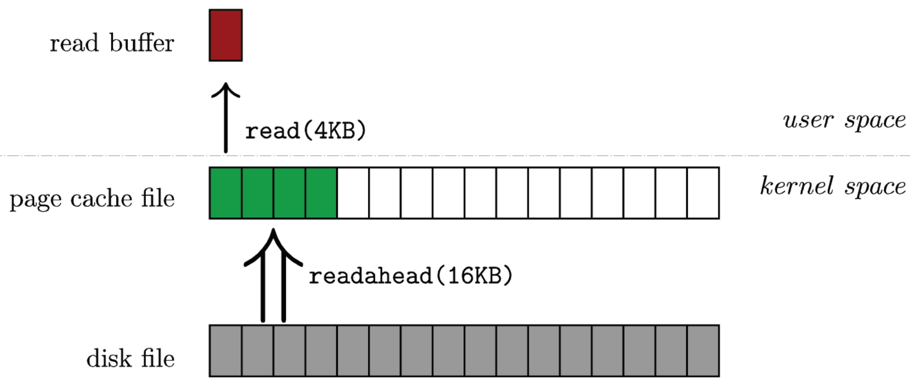
READAHEAD IN LINUX

Linux 2.6 does autonomous file level prefetching in two ways: read-around for memory mapped file accesses, and read-ahead for general buffered reads. The read-around logic is mainly designed for executables and libraries, which are perhaps the most prevalent mmap file users. They typically do a lot of random reads that exhibit strong locality of reference. So a simple read-around policy is employed for mmap files: on every mmap cache miss, try to pre-fault some more pages around the faulting page. This policy also does a reasonable good job for sequential accesses in either forward or backward directions.

The readahead logic for buffered reads, including read(), pread(), readv(), sendfile(), aio_read() and the Linux specific splice() system calls, is designed to be a general purpose one and hence is more comprehensive. It watches those read requests and tries to discover predictable patterns in them, so that it can prefetch data in a more I/O efficient way. Due to the highly diversity of access patterns and wide range of system dynamics, it has been hard to get right, and only the sequential access patterns are supported for now.

The read-around and read-ahead algorithms are either dumb or heuristic, in various cases these in-kernel logics can be aided with some kind of user/

Figure 3. Page cache oriented read and readahead: when an empty page cache file is asked for the first 4KB data, a 16KB readahead I/O will be triggered



application provided I/O hints. Linux provides a per-device tunable max_readahead parameter that can be queried and modified with command blockdev. As for now it defaults to 128KB for hard disks and may be increased in the future. To better parallelize I/O for disk arrays, it defaults to 2*stripe_width for software RAID.

Linux also provides madvise(), posix_fadvise() and the non-portable readahead() system calls. The first two calls allow applications to indicate their future access patterns as normal, random or sequential, which correspondingly set the read-around and read-ahead policies to be default, disabled or aggressive. The APIs also make application controlled prefetching possible by allowing the application to specify the exact time and location to do readahead. Mysql is a good example to make use of this facility in carrying out its random queries.

The Page Cache

Figure 3 shows how Linux transforms a regular read() system call into an internal readahead request. Here the page cache plays a central role: user space data consumers do read()s which transfer data from page cache, while the in-kernel readahead routine populates page cache with data from the storage device. The read requests are thus decoupled from real disk I/Os.

This layer of indirection enables the kernel to reshape "silly" I/O requests from applications: a huge sendfile(1GB) request will be served in smaller max_readahead sized I/O chunks; while a sequence of tiny 1KB reads will be aggregated into up to max_readahead sized readahead I/Os.

The readahead algorithm does not manage a standalone readahead buffer. Prefetched pages are put into page cache together with cached pages. It also does not take care of the in-LRU-queue life time of the prefetched pages in general. Every prefetched page will be inserted not only into the per-file radix tree based page cache for ease of reference, but also to one of the system wide LRU queues managed by the page replacement algorithm.

This design is simple and elegant in general; however when memory pressure goes high, the memory will thrash (Wiseman, 2009), (Jiang, 2009) and the interactions between prefetching and caching algorithms will become visible. On the one hand, readahead blurs the correlation between a page's position in the LRU queue with its first reference time. Such correlation is relied on by the page replacement algorithm to do proper page aging and eviction. On the other hand, in a memory hungry system, the page replacement algorithm may evict readahead pages before they are accessed by the application, leading to readahead thrashings. The latter issue will be revisited in section 5.3.

Readahead Windows And Pipelining

Each time a readahead I/O decision is made, it is recorded as a "readahead window". A readahead window takes the form of (start, size), where start is the first page index and size is the number of pages. The readahead window produced from this run of readahead will be saved for reference in the next run.

Pipelining is an old technique to enhance the utilization of the computer components (Wiseman, 2001). Readahead pipelining is a technique to parallelize CPU and disk activities for better resource utilization and I/O computing performance. The legacy readahead algorithm adopts dual windows to do pipelining: while the application is walking in the current_window, I/O is underway asynchronously in the ahead_window. Whenever the read request is crossing into ahead_window, it becomes current_window, and a readahead I/O will be triggered to make the new ahead_window.

Readahead pipelining is all about doing asynchronous readahead. The key question is how early should the next readahead be started asynchronously? The dual window scheme cannot provide an exact answer, since both read request and ahead_window are wide ranges. As a result it is not able to control the "degree of asynchrony".

Our solution is to introduce it as an explicit parameter async_size: as soon as the number of not-yet-consumed readahead pages falls under this threshold, it is time to start next readahead. async_size can be freely tuned in the range [0, size]: async_size = 0 disables pipelining, whereas async_size = size opens full pipelining. It avoids maintaining two readahead windows and decouples async_size from size.

Figure 4 shows the data structures. Note that we also tag the page at start + size - async_size with PG_readahead. This newly introduced page flag is one fundamental facility in our proposed readahead framework. It was originally intended to help support the interleaved reads: It is more stable than the per-file-descriptor readahead states, and can tell if the readahead states are still valid and dependable in the case of multiple readers in one file descriptor. Then we quickly find it a handy tool for handling other cases, as an integral part of the following readahead call convention.

Call Convention

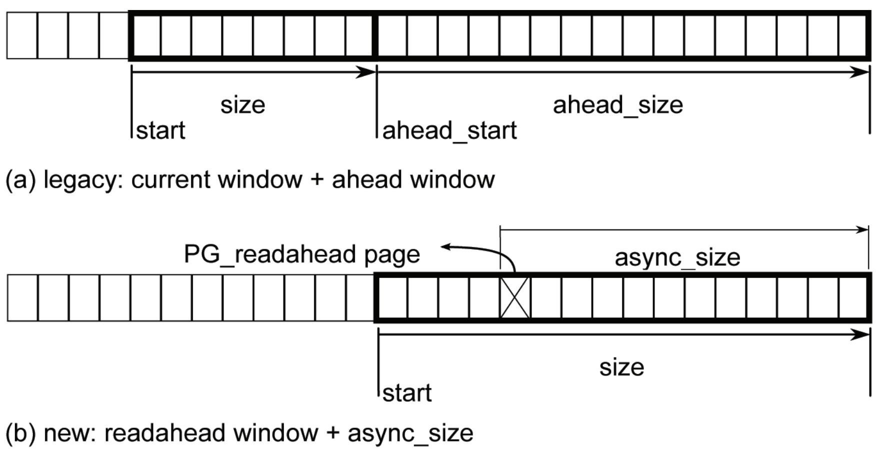
The traditional practice is to feed every read request to the readahead algorithm, and let it handle all possible system states in the process of sorting out the access patterns and making readahead decisions. The handling everything in one interception routine approach leads to unnecessary invocations of readahead logic and makes it unnecessarily complex.

There are also fundamental issues with read requests. They may be too small(less than 1 page) or too large(more than max_readahead pages) that require special handling. What's more, they are unreliable and confusing: the same pages may be requested more than one times in the case of readahead thrashing and retried sequential reads.

The above observations lead us to two new principles: Firstly, trap into the readahead heuristics only when it is the right time to do readahead; Secondly, judge by the page status instead of the read requests and readahead windows whenever feasible. These principles yield a modular readahead framework that separates the following readahead trigger conditions with the main readahead logic. When these two types of pages are read, it is time to do readahead:

1. *cache miss page*: it's time for synchronous readahead. An I/O is required to fault in the current page. The I/O size could be inflated to piggy back more pages.
2. *PG_readaheadpage*: it's time for asynchronous readahead. The PG_readahead tag is

Figure 4. The readahead windows


size
start
ahead_size
ahead_start
(a) legacy: current window + ahead window
PG_readahead page
async_size
size
start
(b) new: readahead window + async_size

set by a previous readahead to indicate the time to do next readahead.

Sequentiality

Table 1 shows the common sequentiality criteria. The most fundamental one among them is the "consecutive criterion" in the last line, where page access offsets are incremented one by one. The legacy readahead algorithm enforces the sequentiality criteria on each and every read request, so one single seek will immediately shutdown the readahead windows. Such rigid policy guarantees high readahead hit ratio. However it was found to be too conservative to be useful for many important real life workloads.

Instead of demanding a rigorous sequentiality, we propose to do readahead for reads that have good probability to be sequential. In particular, the following rules are adopted:

1. Check sequentiality only for synchronous readahead triggered by a missing page. This ensures that random reads will be recognized as the random pattern, while a random read in between a sequential stream wont interrupt the stream's readahead sequence.
2. Assume sequentiality for the asynchronous readahead triggered by a PG_readahead page. Even if the page was hit by a true random read, it indicates two random reads that are close enough both spatially and temporally. Hence it may well be a hot accessed area that deserves to be readahead.

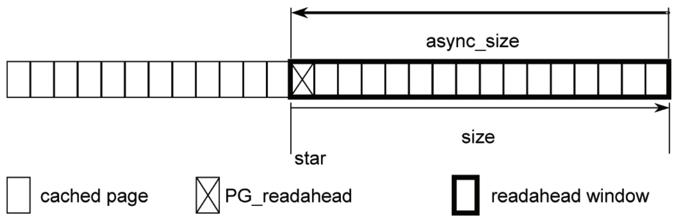
Concurrent Streams

The consecutive criterion in table 1 demands both time and space continuity. However when multiple threads do sequential reads on the same file in parallel, the read requests may appear interleaved in

Table. 1. Common sequentiality criterions

criterion	case
read_size > max_readahead	oversize read, common in sendfile() calls
offset == 0 && prev_offset == -1	first read on start of file
offset == prev_offset	unaligned consecutive reads
offset == prev_offset + 1	trivial consecutive reads

Figure 5. For a typical readahead window, async_size equals to size, and PG_readahead lies in the first page



time but still continuous in their respective access spaces. Such kind of interleaved pattern used to be a great challenge to the readahead algorithm. The interleaving of several sequential streams may be perceived as random reads.

In theory, it is possible to track the state of each and every stream. However the cost would be unacceptable: since each read request may start a new stream, we will end up remembering every read in the case of random reads. ZFS takes this approach but limits the number of streams it can track to a small value. The implementation also has to deal with the complexity to dynamically allocate, lookup, age and free the stream objects.

Linux takes a simple and stupid approach: to statically allocate one readahead data structure for each file descriptor. This is handy in serving the common case of one stream per file descriptor, however it leaves the support for multiple interleaved streams an open question.

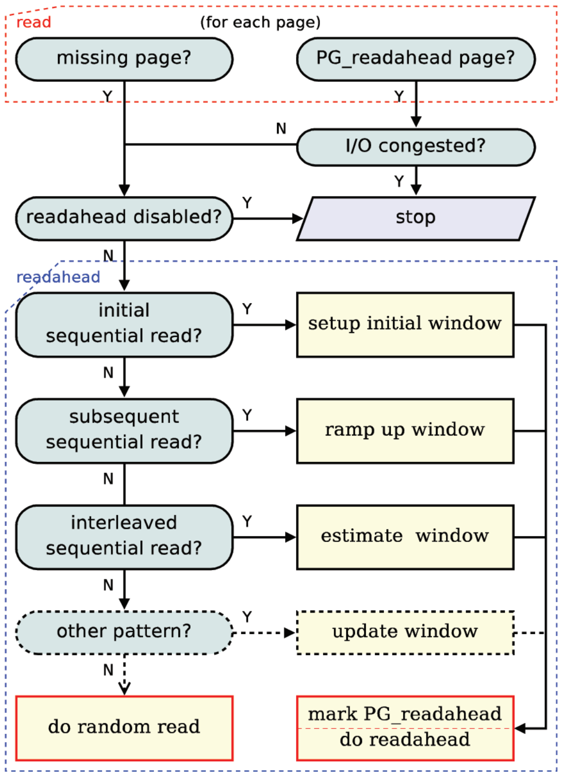
When there are multiple streams per file descriptor, the streams will contend the single space slot for read offset and readahead window, and end up overwriting each other's state. This gives rise to two problems. Firstly, given an unreliable prev_offset, how do we know the current read request is a sequential one and therefore should be prefetched? Secondly, given an unreliable readahead window, how do we know it is valid for the current stream and if not, how can we recover it? The first problem has in fact been addressed in the previous subsection. It is also straightforward to validate the ownership of a readahead window: if it contains the current PG_readahead page, then it is the readhead window for the current stream. Otherwise we have to seek alternative ways for finding out the lost information about readahead window.

Fortunately, we found that the lost readahead window information can be inferred from the page cache. Figure 5 shows the typical status of a page cache. The application hits the PG_readahead page and is triggering an interleaved readahead. To recover the readahead window information, we scan forward in the page cache from the current page to the first missing page. The number of pages in between is exactly async_size. As we know that async_size equals to size for all subsequent readaheads and at least size/2 for the initial readahead, we can safely take async_size as size, and calculate the next readahead size from it.

Readahead Sequence

Assume the sequence of readahead for a sequential read stream to be A0, A1, A2, … We call A0 the "initial readahead" and A1, A2, … the "subsequent readaheads". The initial readahead is typically synchronous in that it contains the currently requested page and the reader have to stop and wait for the readahead I/O. The subsequent readaheads can normally be asynchronous in that the readahead pages are not immediately needed by the application.

Figure 6. The readahead framework in Linux 2.6.24



A typical readahead sequence starts with a small initial readahead size, which is inferred from the read size, and then ramps up exponentially to max_readahead. However in the case of a large sendfile() call, the readahead window can immediately expand to its full size. The resulting readahead size is adaptive to the current read size as well as the accumulated sequential read sizes.

We conclude this section with the block diagram of the demand readahead in Linux 2.6.24.

CASE STUDY

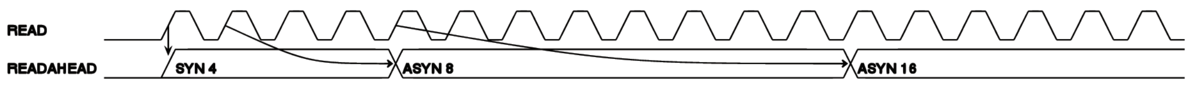
A Typical Readahead Stream

Take the "cp" command as an example, it does 4KB page sized sequential reads covering the whole file. Table 2 and figure 7 demonstrates the first three readahead invocations. The initial read triggers a readahead that is four times the read size, whose initial async_size is such that the second sequential read request will immediately trigger the next readahead asynchronously. The goal of the rules is to start asynchronous readahead as soon as possible, while trying to avoid useless readahead for applications that only examine the file head.

Readahead Cache Hits

Linux manages kernel level "page cache" to keep frequently accessed file pages in memory. A read request for an already cached page is called a "cache hit", otherwise it is a "cache miss". If a readahead request is made for an already cached page, it makes a "readahead cache hit". Cache hits are good whereas readahead cache hits are

Figure 7. Readahead I/O triggered by "cp"



evil: they are undesirable and normally avoidable overheads. Since cache hits can far outweigh cache misses in a typical system, it is important to shutdown readahead on large ranges of cached pages to avoid excessive readahead cache hits.

The classical way to disable readahead for cached page ranges is to set a RA_FLAG_INCACHE flag after 256 back-to-back readahead cache hits and to clear it after the first cache miss. Although this scheme works, it is not ideal. Firstly, the threshold does not apply to cached files that are smaller than 1MB, which is a common case; Secondly, during the in-cache-no-readahead mode, the readahead algorithm will be invoked for *every page* instead of every read request or max_readahead pages, to ensure on time resume of readahead(yet it still misses readahead for the first cache miss page), so it's converting one type of overhead into another hopefully smaller one.

Our proposed readahead trigger scheme can handle readahead cache hits automatically. This is achieved by taking care that PG_readahead be only set on a newly allocated readahead page and get cleared on the first read hit. So when the new readahead window lies inside a range of cached pages, PG_readahead won't be set, disabling further readaheads. As soon as the reader steps out of the cached range, there will be a cache miss which immediately restarts readahead. If the whole file is cached, there will be no missing or PG_readahead pages at all to trigger an undesired readahead. Another merit is that the in-kernel readahead will automatically stop when the user space is carrying out accurate application controlled readahead.

Readahead Thrashing

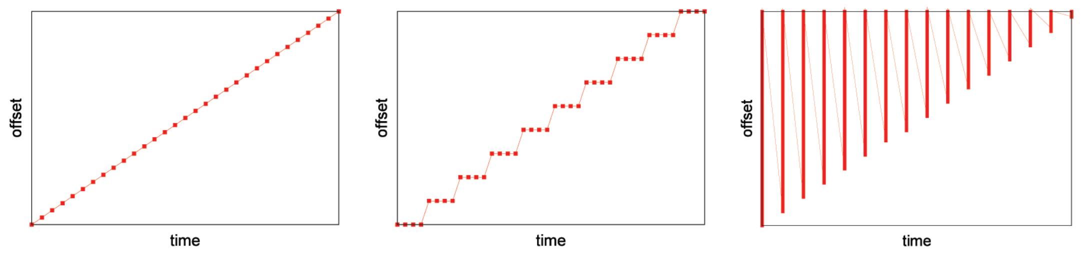
We call a page the "readahead page" if it was populated by the readahead algorithm and is waiting to be accessed by some reader. Once being referenced, it is turned into a "cached page". Linux manages readahead pages in inactive_list together with cached pages. For the readahead pages, inactive_list can be viewed as a simple FIFO queue. It may be rotated quickly in a loaded and memory tight server. Readahead thrashing happens when some readahead pages are shifted out of inactive_list and reclaimed, before a slow reader is able to access them in time.

Readahead thrashing can be easily detected. If a cache miss occurs inside the readahead windows, a thrashing happened. In this case, the legacy readahead algorithm will decrease the next readahead size by 2. By doing so it hopes to adapt readahead size to the "thrashing threshold", which is defined as the largest possible thrashing safe readahead size. As the readahead size steps

Table. 2. Readahead invocations triggered by "cp"

offset	trigger condition	readahead type	size	async_size
0	missing page	initial/sync readahead	4*read_size = 4	size-read_size = 3
1	PG_readahead page	subsequent/async readahead	2*prev_size = 8	size = 8
4			2*prev_size = 16	size = 16

Figure 8 The trivial, unaligned, and retried sequential reads



slowly off to the thrashing threshold, the thrashings will fade away.

However, once the thrashings stop, the readahead algorithm immediately reverts back to the normal behavior of ramping up the window size by 2 or 4, leading to a new round of thrashings. On average, about half of the readahead pages will be lost and re-read from disk.

Besides the wastage of memory and bandwidth resources, there are much more destructive disk seeks. When thrashing is detected, the legacy readahead takes no action to recover the lost pages inside the readahead windows. The VFS read routine then has to fault them in one by one, generating a lot of tiny 4KB I/Os and hence disk seeks. Overall, up to half pages will be faulted in this destructive way.

Our proposed framework has basic safeguards against readahead thrashing. Firstly, the first read after thrashing makes a cache miss, which will automatically restart readahead from the current position. Therefore it avoids the catastrophic 1-page tiny I/Os suffered by the legacy readahead. Secondly, the size ramp up process may be starting from a small initial value and keep growing exponentially until thrashing again, which effectively keeps the average readahead size above half of the thrashing threshold. If necessary, more fine grained control can be practiced after thrashing.

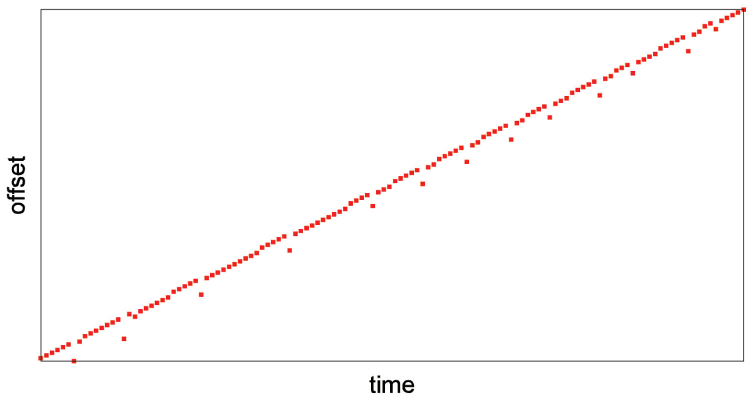
Non-Trivial Sequential Reads

Interestingly, sequential reads may not look like sequential. Figure 8 shows three different forms of sequential reads that have been discovered in the Linux readahead practices. For the following two cases, the consecutive test offset == prev_offset + 1 can fail even when an application is visiting data consecutively.

Unaligned Reads. File reads work on byte ranges, while readahead algorithm works on pages. When a read request does not start or stop at page boundaries, it becomes an "unaligned read". Sequential unaligned reads can access the same page more than once. For example, 1KB sized unaligned reads will present the readahead algorithm with a page series of {0, 0, 0, 0, 1, 1, 1, 1, …} To cover such cases, this sequentiality criterion has been added: offset == prev_offset.

Retried Reads. In many cases --- such as non-blocking I/O, the retry based Linux AIO, or an interrupted system call --- the kernel may interrupt a read that has only transferred partial amounts of data. A typical application will issue "retried read" requests for the remaining data. The possible requested page ranges could be: {[0, 1000], [4, 1000], [8, 1000], …}. Such pattern confuses the legacy readahead totally. They will be taken as oversize reads and trigger the following readahead requests: {(0, 64), (4, 64), (8, 64), …}. Which are overlapped with each other, leading to a lot of readahead cache hits and tiny 4-page I/Os. The new call convention can mask off the retried parts perfectly, in which readahead is triggered by the real accessed pages instead of spurious read requests. So the readahead heuristics won't even be aware of the existence of retried reads.

Figure 9 Sequential accesses over NFS can be out of order when reaching the readahead routine



Locally Disordered Reads

Access patterns in real world workloads can deviate from the sequential model in many ways. One common case is the reordered NFS reads. The pages may not be served at NFS server in the same order they are requested by a client side application. They may get reordered in the process of being sent out, arriving at the server, and finally hitting the readahead logic. Figure 9 shows a trace of NFS reads observed by the readahead logic.

The in-depth discussion for such complex kind of workload is beyond the scope of this book. However the readahead framework does offer clean and basic support for semi-sequential reads. Its call convention and sequential detection logics can help a readahead sequence to start and go on in the face of random disturbances:

Startup. It's easy to start an initial readahead window. It will be opened as soon as two consecutive cache misses occur. Since semi-sequential reads are mostly consecutive, it happens very quickly.

Continuation. It's guaranteed that one readahead window will lead to another in the absence of cache hits. So a readahead sequence won't be interrupted by some wild random reads. A PG_readahead tag will be set for each new readahead window. It will be hit by a subsequent read and unconditionally trigger the next readahead. It does not matter if that read is a non-consecutive one.

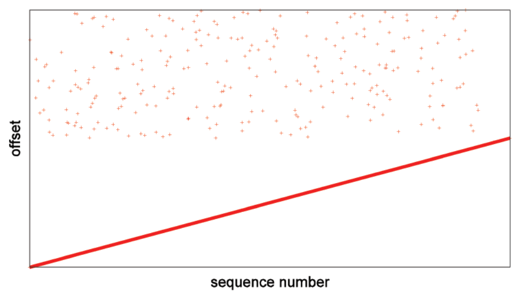
PERFORMANCE EVALUATION

In this section we explore the readahead performances of Linux 2.6.24 and 2.6.22 side by side, which implements the new and legacy readahead algorithms respectively. The max readahead size is set to 1MB for better I/O performance. The testing platform is a single Hitachi(R) Deskstar(TM) T7K250 160GB 7200RPM 8MB hard disk and Intel(R) Core(TM)2 E6800 2.93GHz CPU. The selected comparison experiments illustrate how much impact readahead algorithms can have on single disk I/O performance. One can expect much larger differences for disk arrays.

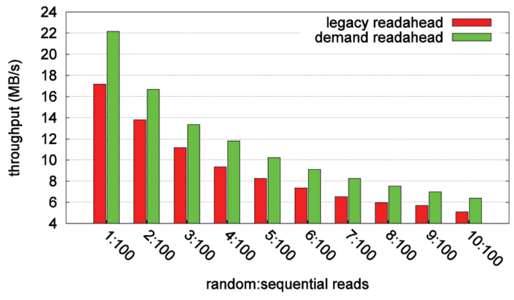
Intermixed Random And Sequential Reads

We created a 200MB file to carry out the intermixed 4KB random and sequential reads. The sequential stream begins at start of file and stops at middle of file, while the random reads land randomly in the second half of the file. We created ten access patterns where the amount of sequential reads are fixed at 100MB and the amount of random reads increase from 1MB to 10MB. Figure 10(a)

Figure 10. Comparison of the readahead performances under random disturbs



(a) mixed random/sequential access pattern



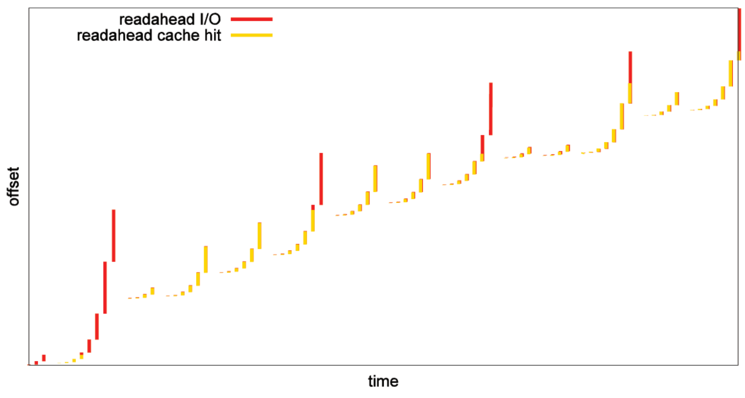
(b) throughput of the readahead algorithms

describes the first access pattern. Figure 10(b) shows the I/O throughputs for each of the 10 access patterns. The in-disk readahead function is disabled to highlight the performance impact of OS readahead.

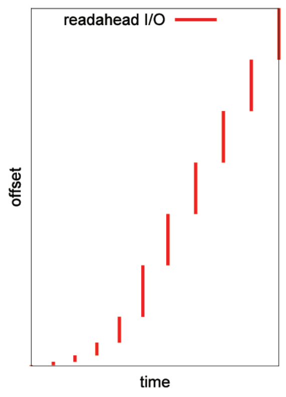
Not surprisingly the I/O throughput decreases with more and more random reads. Also notably, the new readahead algorithm maintains a stable lead over the legacy one. In the case of 1:100 random:sequential bytes ratio, the throughputs are 17.18MB/s and 22.15MB/s respectively, with the new readahead showing an edge of 28.9% over the legacy one. When the ratio goes up to 10:100, the throughputs decrease to 5.10MB/s and 6.39MB/s, but still keeps a performance edge of 25.4%.

The performance edges originate from the different readahead behaviors under random disturbs. Figure 11 demonstrates the readahead sequences submitted for the first 1600 pages by the two readahead algorithms. Figure 11(a) shows vividly how the legacy readahead sequences are interrupted by random reads. On each and every random read, the readahead window will be shutdown. And then followed by a new readahead size ramp up process. Due to the existence of async readahead pages, the new readahead sequence will be overlapping with the old one, leading to series of readahead cache hits. In contrast, the demand readahead algorithm ramps up and pushes forward the

Figure 11 Comparison of readahead sequences under random disturbs



(a) legacy readahead



(b) demand readahead

readahead window steadily and is not disturbed by the random reads.

Interleaved Sequential Streams

To validate readahead performance on concurrent streams, we created 10 sequential streams S_i, i=1,2,…,10, where S_i is a sequence of 4KB reads that start from byte (i-1)*100MB and stop at i*100MB. By interleaving the first n streams, we get C_n=interleave(S_1,S_2,…,S_n), n=1,2,…,10. Where C_1=S_1 is a time and space consecutive read stream, and C_n, n=2,3,…,10 is an interleaved access pattern that has n sequential streams. Figure 12 shows a segment of C_5.

Figure 13 plots the I/O throughputs for C_1-C_{10}. The single stream throughputs are close, with 24.95MB/s for legacy readahead and 28.32MB/s for demand readahead. When there are 2 and 3 interleaved streams, the legacy readahead quickly slows down to 7.05MB/s and 3.15MB/s, while the new one is not affected. When there are 10 interleaved streams, the legacy readahead throughput crawls along at 0.81MB/s, while the demand readahead still maintains a high throughput of 21.78MB/s, which is 26.9 times the former one.

We also measured performance with disk readahead enabled. As showed in figure 13(b), the in-disk readahead have very positive effects on the performance. However due to limited disk cache and capability, it can only support a limited number of streams. As the number of streams increase, the influence of the in-disk readahead decreases. When there are 10 streams, the throughputs decrease to 12.29MB/s and 35.30MB/s, in which the demand readahead is 2.87 times fast. When it comes to hundreds of concurrent clients, which is commonplace for production file servers, it effectively renders the in-disk readahead useless. So figure 13(a) may be a more realistic measurement of the readahead performance at high concurrency.

Figure 14 plots the total application visible I/O delays in each benchmark. When the number of streams increases, the legacy readahead delays increase rapidly from 3.36s for single stream to 122.35s for 10 streams, while the demand readahead delays increase slowly from 2.89s to 3.44s. There is a huge gap of 35.56 times for the 10 streams case. This stems from the fact that the legacy readahead windows can hardly ramp up to the ideal size, leading to smaller I/O and more seeks, increasing the total disk I/O latencies. Because the async readahead size will also be reset and limited to small values, it can hardly hide I/O latencies to the upper layer.

Figure 12. Five interleaved sequential streams

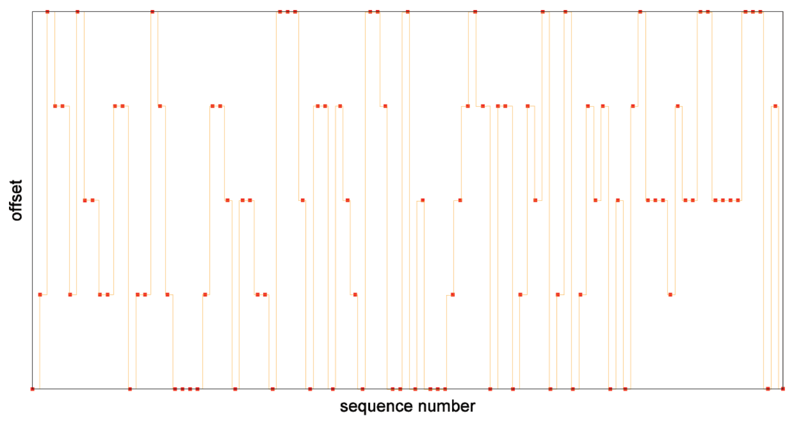
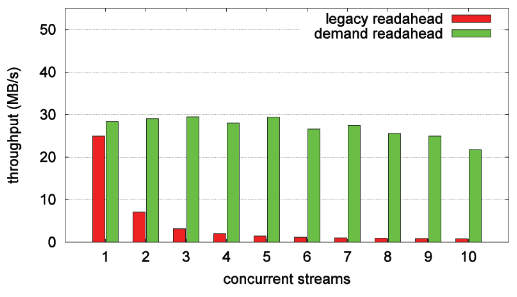
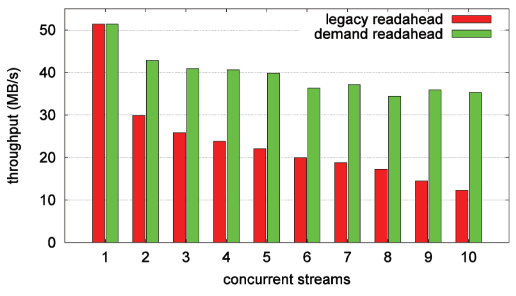


Figure 13 Comparison of readahead performances on interleaved reads



(a) disk drive readahead disabled



(b) disk drive readahead enabled

Readahead Thrashing

We boot the kernel with mem=128m single, and start one new 100KB/s stream on every second. Various statistics are collected and showed in figure 15. The thrashing begins at 20 second. The legacy readahead starts to overload the disk at 50 second, and eventually achieved 5MB/s maximum network throughput. With the new framework, throughput keeps growing and the trend is going up to 15MB/s. That's three times better. The average I/O size also improves a lot. It used to drop sharply to about 5KB, while the new behavior is to slowly go down to 40KB under increasing loads. Correspondingly, the disk quickly goes 100% utilization for legacy readahead. It is actually overloaded by the storm of seeks as a result of the tiny 1-page I/Os.

CONCLUSION

Sequential prefetching is a standard function of modern operating systems. It tries to discover application I/O access pattern and prefetch data pages for them. Its two major ways of improving I/O performance are: increasing I/O size for better throughput; facilitating async I/O to mask I/O latency.

The diversity of application behaviors and system dynamics sets high standards to the adaptability of prefetching algorithms. Concurrent and interleaved streams are also big challenges for their capabilities. These challenges became more imminent by two trends: the increasing relative cost of disk seeks and the prevalence of multi-core processors and parallel computing.

Based upon experiences and lessons gained in the Linux readahead practices, we designed a de-

Figure 14. Application visible I/O delays on 100MB interleaved reads

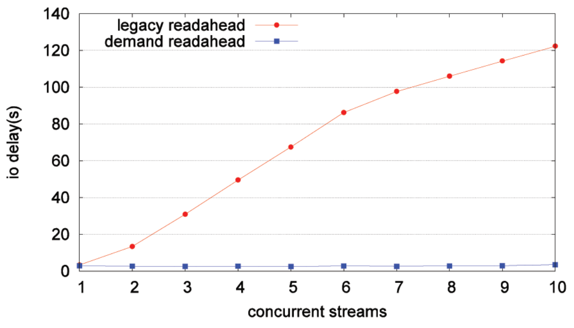
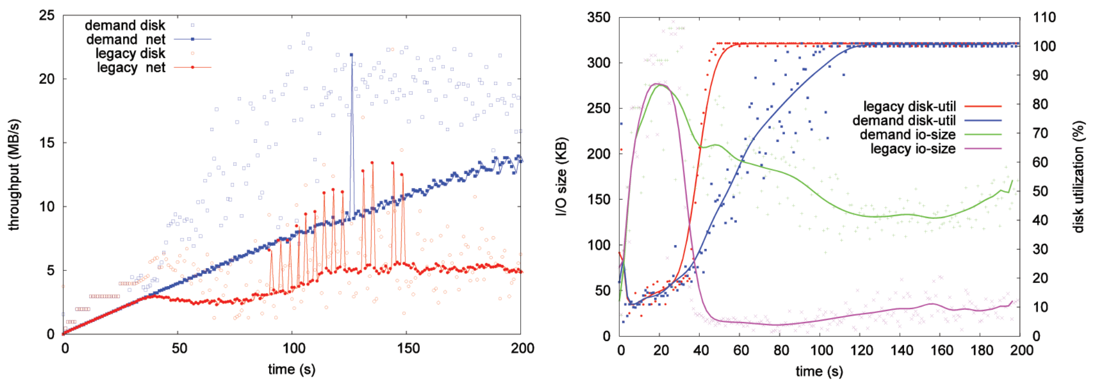


Figure 15. I/O performance on readahead thrashing



(a) disk/net throughput (b) average I/O size and disk utilization

mand readahead algorithm with flexible heuristics that can cover varied sequential access patterns and support interleaved streams. It also enjoys great simplicity by handling most abnormal cases in an implicit way. Its power stems from the relaxed criteria on sequential pattern recognition and the exploitation of page and page cache states. The new design guidelines seem to work well in practice. Since its wide deployment with Linux 2.6.23, we have not received any regression reports.

REFERENCES

Bhattacharya, S., Tran, J., Sullivan, M., & Mason, C. (2004). Linux AIO Performance and Robustness for Enterprise Workloads. In . *Proceedings of the Linux Symposium*, *1*, 63–78.

Brown, A. D., Mowry, T. C., & Krieger, O. (2001). Compiler-based i/o prefetching for out-of-core applications. *ACM Transactions on Computer Systems*, *19*, 111–170. doi:10.1145/377769.377774

Butt, A. R., Gniady, C., & Hu, Y. C. (2007). The Performance Impact of Kernel Prefetching on Buffer Cache Replacement Algorithms. *IEEE Transactions on Computers*, *56*(7), 889–908. doi:10.1109/TC.2007.1029

Cao, P., Felten, E. W., Karlin, A. R., & Li, K. (1995). A study of integrated prefetching and caching strategies. In *Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, (pp. 188-197).

Cao, P., Felten, E. W., Karlin, A. R., & Li, K. (1996). Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, *14*, 311–343. doi:10.1145/235543.235544

Dini, G., Lettieri, G., & Lopriore, L. (2006). Caching and prefetching algorithms for programs with looping reference patterns. *The Computer Journal*, *49*, 42–61. doi:10.1093/comjnl/bxh140

Ellard, D., Ledlie, J., Malkani, P., & Seltzer, M. (2003). Passive NFS Tracing of Email and Research Workloads. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST'03)*, (pp. 203-216).

Ellard, D., & Seltzer, M. (2003). NFS Tricks and Benchmarking Traps. In *Proceedings of the FREENIX 2003 Technical Conference*, (pp. 101-114).

Esfahbod, B. (2006). *Preload - An Adaptive Prefetching Daemon*. PhD thesis. Graduate Department of Computer Science, University of Toronto, Canada.

Feiertag, R. J., & Organick, E. I. (1971). The multics input/output system. In *Proceedings of the third ACM symposium on Operating systems principles*, (pp. 35-41).

Gill, B. S., & Bathen, L. A. D. (2007). Optimal multistream sequential prefetching in a shared cache. *ACM Transactions on Storage*, *3*(3), 10. doi:10.1145/1288783.1288789

Gill, B. S., & Modha, D. S. (2005). Sarc: sequential prefetching in adaptive replacement cache. *Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*, (pp. 33-33).

Itshak, M., & Wiseman, Y. (2008). AMSQM: Adaptive Multiple SuperPage Queue Management. In *Proc. IEEE Conference on Information Reuse and Integration (IEEE IRI-2008)*, Las Vegas, NV, (pp. 52-57).

Kroeger, T. M., & Long, D. D. E. (2001). Design and implementation of a predictive file prefetching algorithm. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, (pp. 105-118).

Li, C., & Shen, K. (2005). Managing prefetch memory for data-intensive online servers. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, (pp. 19).

Li, C., Shen, K., & Papathanasiou, A. E. (2007). Competitive prefetching for concurrent sequential i/o. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, (pp. 189-202).

Li, Z., Chen, Z., & Zhou, Y. (2005). Mining block correlations to improve storage performance. *ACM Transactions on Storage*, *1*, 213–245. doi:10.1145/1063786.1063790

Liang, S., Jiang, S., & Zhang, X. (2007). STEP: Sequentiality and Thrashing Detection Based Prefetching to Improve Performance of Networked Storage Servers. *27th International Conference on Distributed Computing Systems (ICDCS'07)*, (p. 64).

McKusick, M. K., Joy, W. N., Leffler, S. J., & Fabry, R. S. (1984). A fast file system for unix. *ACM Transactions on Computer Systems*, *2*(3), 181–197. doi:10.1145/989.990

Pai, R., Pulavarty, B., & Cao, M. (2004). Linux 2.6 performance improvement through readahead optimization. In . *Proceedings of the Linux Symposium*, *2*, 391–402.

Papathanasiou, A. E., & Scott, M. L. (2005). Aggressive prefetching: An idea whose time has come. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS).*

Paris, J.-F. Amer, A. & Long, D. D. E. (2003). A stochastic approach to file access prediction. In *Proceedings of the international workshop on Storage network architecture and parallel I/Os*, (pp. 36-40).

Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., & Zelenka, J. (1995). Informed prefetching and caching. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, (pp. 79-95).

Patterson, R. H., Gibson, G. A., & Satyanarayanan, M. (1993). A status report on research in transparent informed prefetching. *SIGOPS Operating Systems Review*, *27*(2), 21–34. doi:10.1145/155848.155855

Patterson III Russel. H. (1997). *Informed Prefetching and Caching*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Schmid, P. (2006). *15 years of hard drive history: Capacities outran performance*. Retrieved from http://www.tomshardware.com/reviews/15-years-of-hard-drive-history,1368.html

Shriver, E., Small, C., & Smith, K. A. (1999). Why does file system prefetching work? In *Proceedings of the Annual Technical Conference on 1999 USENIX Annual Technical Conference*, (pp. 71-84).

Whittle, G. A. S., Paris, J.-F., Amer, A., Long, D. D. E., & Burns, R. (2003). Using multiple predictors to improve the accuracy of file access predictions. In *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*, (pp. 230).

Wu, F., Xi, H., Li, J., & Zou, N. (2007). Linux readahead: less tricks for more. In . *Proceedings of the Linux Symposium*, *2*, 273–284.

Wu, F., Xi, H., & Xu, C. (2008). On the design of a new linux readahead framework. *ACM SIGOPS Operating Systems Review*, *42*(5), 75–84. doi:10.1145/1400097.1400106