On the Design of a New Linux Readahead Framework

WU Fengguang, XI Hongsheng, XU Chenfeng
University of Science and Technology of China
Dept. Automation, Hefei 230027
wfg@mail.ustc.edu.cn, xihs@ustc.edu.cn, johnx@mail.ustc.edu.cn

ABSTRACT

As Linux runs an increasing variety of workloads, its in-kernel readahead algorithm has been challenged by many unexpected and subtle problems. To name a few: readahead thrashings arise when readahead pages are evicted prematurely under memory pressure; readahead attempts on already cached pages are undesirable; interrupted-then-retried reads and locally disordered NFS reads that can easily fool the sequential detection logic. In this paper, we present a new Linux readahead framework with flexible and robust heuristics that can cover varied sequential I/O patterns. It also enjoys great simplicity by handling most abnormal cases in an implicit way. We demonstrate its advantages by a host of case studies. Network throughput is 3 times better in the case of thrashing and 1.8 times better for large NFS files. On serving large files with lighttpd, the disk utilization is decreased by 26% while providing 17% more network throughput.

Categories and Subject Descriptors

D.4.3 [**Operating Systems**]: File Systems Management; D.4.8 [**Operating Systems**]: Performance; I.5.1 [**Pattern Recognition**]: Models

Keywords

Linux, operating systems, I/O performance, prefetching, readahead, access pattern, sequentiality, caching, thrashing

1. INTRODUCTION

Sequential prefetching, also known as readahead in Linux, is a widely deployed technique to bridge the huge gap between the characteristics of disk drives and their inefficient ways of usage by applications. At one end, disk drives suffers from big seek latencies and 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 and disk drives do prefetching: to bring in data blocks for the upcoming requests, and do so in big chunks.

Prefetching could bring three major benefits[25]. Firstly, I/O delays are effectively hidden from the applications. When an application requests for a page, it has been prefetched and is ready to use. Secondly, disks are better utilized with the larger prefetching requests. Lastly, it helps to amortize processing overheads in the I/O path.

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[19, 27]. The wide deployment of Linux - from embedded devices to supercomputers - confronts readahead algorithms with an incredible variety of workloads. Almost every early stage assumptions have been invalidated:

(A) Sequential accesses do not necessary translate into consecutive page indexes: unaligned reads, retried reads[1], reordered NFS reads[7], concurrent reads.

(B) Readahead should not always be performed on sequential reads: readahead cache hits, congested I/O queue.

(C) Readahead may not always succeed: out of memory, full I/O queue.

(D) Readahead pages may be reclaimed before being accessed: readahead thrashing.

It follows from (A) that the sequential detection logic should be extended to cover more semi-sequential patterns. When dealing with sequentiality, the legacy readahead algorithm has been over-conservative. Next, the issues in (B,C,D) are not related to access patterns. They arise out of various system states and should be handled in an isolated way.

In this paper, we present a design and an implementation of readahead framework for the Linux 2.6 kernel. It enables sequential and semi-sequential I/O patterns to be served in a uniform way. The modular design frees the readahead algorithms from the chores of abnormal system dynamics. We concentrate on how the new principles and techniques differ from the conventional wisdom, and illustrate how the various challenges are addressed in the new framework. The algorithms, practical implications and I/O performance will be compared with the legacy readahead framework.

The rest of this paper is organized as follows. Sections 2 and 3 describe the design principles and case studies of the readahead algorithms. The I/O performances are evaluated in section 4. Finally, we introduce related work and conclude the paper in sections 5, 6.

2. READAHEAD ALGORITHMS

2.1 Readahead in Linux

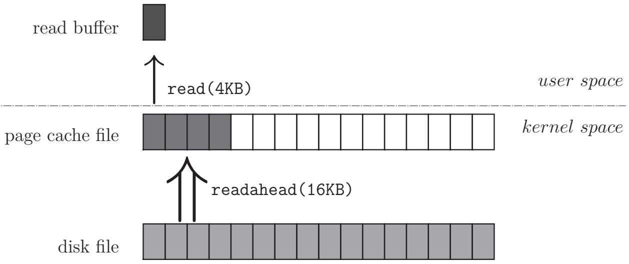


Figure 1: Page cache oriented read and readahead: when an empty page cache file is asked for the first 4KB data, a 16KB readahead I/O will be triggered

Linux 2.6 does autonomous file level prefetching in two ways: read-around for memory mapped file accesses, and read-ahead for 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 both 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 a number of reasons described in section 1, it has been hard to get right, and only sequential reads are supported for now.

The read-around and read-ahead algorithms are dumb and heuristic, and can be helped with some kind of hints. Some per-device parameters `max_readahead` are accessible via command `blockdev`. They default to 128KB for hard disks, `2*stripe_width` for software RAID, and `rsize*15` for NFS. Linux also provides `madvise()`, `posix_fadvise()` and the non-portable `readahead()` system calls for applications to alter the default behavior of the read-around and readahead algorithms, or even to do application controlled prefetching.

Figure 1 shows how Linux transforms a regular `read()` call into an internal readahead request. Here 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 data from the storage device into page cache. The read requests are thus decoupled from the real disk I/Os. This layer of indirection enables the kernel to reshape 'silly' I/O requests from applications: a huge `sendfile(1GB)` will be broken into smaller `max_readahead` sized chunks, while many tiny 1KB sequential reads will be aggregated into up to `max_readahead` sized readahead requests.

The readahead algorithms don't manage a standalone readahead buffer, prefetched pages are put into page cache together with cached pages. Neither will they guard the life time of the prefetched pages. Every prefetched page will be inserted not only into the per-file page cache, but also to one of the system wide LRU queues to be managed by the page replacement algorithm. The design is simple and elegant in general. However when memory pressure goes high the interactions between prefetching and caching algorithms 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, the page replacement algorithm may evict readahead pages before they are accessed by the application. The latter issue will be revisited in section 3.2.

2.2 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 referencing in the next run.

Readahead pipelining is a technique to parallelize the CPU and disk activities for better resource utilization and system 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`.

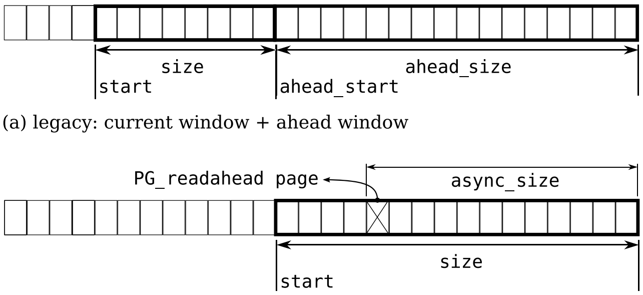
Readahead pipelining is all about doing asynchronous readahead. The key question is how early should the next readahead be started asynchronously? The dual window scheme cannot provide an exact answer, since both read request and `ahead_window` are wide ranges. As a result it is not able to control the *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 for readahead. `async_size` can be freely tuned: `async_size=0` disables pipelining, whereas `async_size=readahead_size` opens full pipelining. It avoids maintaining two readahead windows and decouples `async_size` from `readahead_size`.

Figure 2 shows the data structures. Note that we also tag the page at `start+size-async_size` with `PG_readahead`. The page flag is more stable than other readahead states. It can tell if the readahead states are still valid and dependable.

2.3 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. Figure 3 shows the issues that the legacy readahead routine `page_cache_readahead()` has to deal with.

It is a waste of time to blindly and excessively call into `page_cache_readahead()`, since no actions are required in most times. For example, when doing 256-page readaheads for 1-page sequential reads, one readahead invocation will be adequate for every 256 reads; for a cached file, readahead



(a) legacy: current window + ahead window

(b) new: readaead window + async size

Figure 2: The readahead windows

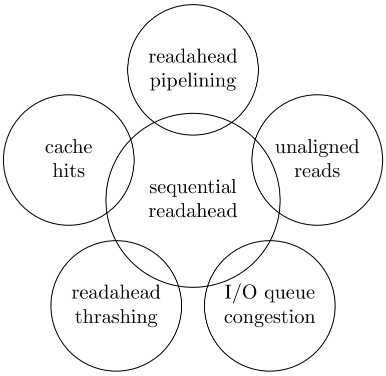


Figure 3: The task list of legacy readahead routine

actions are not desirable at all. Moreover, we know from (B,C,D) in section 1 that seeing read requests or submitting readahead requests does not necessarily mean the corresponding pages will be transfered or populated. They are not reliable criteria, and have lead to convoluted feedbacks on detecting and handling the abnormalities happened to read or readahead requests in legacy readahead.

The above observations lead us to two new principles.

1. Trap into the readahead heuristics only when it is the right time to do readahead.
2. Whenever feasible, judge by the page status instead of the read requests and readahead windows.

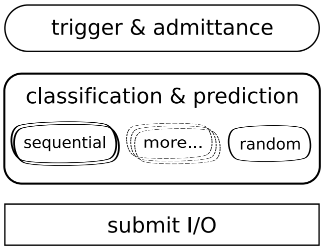


Figure 4: The new modular framework: readahead heuristics are called only when necessary and simply to handle access patterns

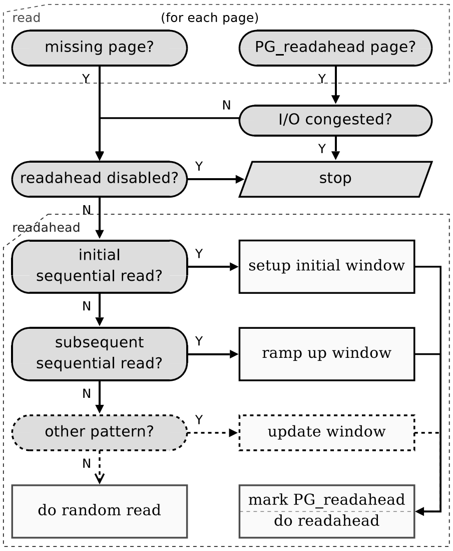


Figure 5: Flow chart of the new framework

criterion	case
`offset == 0`	start of file
`offset == prev_offset + 1`	trivial sequential
`offset == prev_offset`	unaligned sequential
`read_size > readahead_max`	big read(e.g. sendfile)

Table 1: The sequentiality criteria

That yields a modular readahead framework(figure 4,5). It introduces two simple *readahead triggers*. Whenever the following pages are accessed, it's time to do readahead:

- *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.
- *PG_ readahead page*: it's time for asynchronous readahead. The `PG_readahead` tag is set by a previous readahead to indicate the time to do next readahead.

On designing this readahead framework, our original goals were to support readahead thrashing and interleaved reads: the first trigger enables us to re-establish the readahead window as soon as it is thrashed; the second one helps to keep track of unlimited number of concurrent read streams in one file. Later on we are much amused to find it addressing more challenges from cache hits and retried/reordered reads. They could be good testimonies to the new design principles.

2.4 Sequentiality

Table 1 shows the sequentiality criteria used by both readahead algorithms. The most fundamental one among them is the *consecutive criterion*, where accessed page 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 has been 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 the new readahead framework, the following rules are employed:

1. `prev_offset` points to the last accessed page. It used to be the last requested page in `current_window`. However we know that neither read requests nor readahead windows are dependable.
2. 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 won't interrupt the stream's readahead sequence.
3. 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 could be a hot accessed area that deserves to be readahead.

2.5 Readahead Sizes

Assume the sequence of readahead I/Os performed for a sequential stream to be

$$\{A_i = (start_i, size_i) : \; i = 0, 1, \cdots, M\} \qquad (1)$$

We call A_0 the *initial readahead*; $A_1, A_2, \ldots$ the *subsequent readaheads*. Their sizes are computed as follows:

1. Initial readahead: initialized from the read size.

   `size = read_size * scale0`, where `scale0` is 2 or 4

   `async_size = size - read_size`
2. Subsequent readahead: ramp up sizes exponentially.

   `size = prev_size * 2`

   `async_size = size`
3. Always enforce the maximum allowed I/O size.

   `size = min(size, max_readahead)`

   `async_size = min(async_size, size)`

The legacy algorithm triggers asynchronous readahead as soon as the read request crossed into `ahead_window`. Due to the uncertainty of the read size, its '`async_size`' is a random value that lies in range `[max_readahead,2*max_readahead)`.

3. CASE STUDIES

3.1 Readahead Cache Hits

Linux manages a *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 evil: they are nothing more than 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.

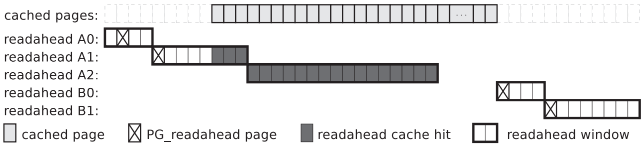


Figure 6: Avoiding readahead cache hits: readahead stops at A2 by not setting the `PG_readahead` tag; restarts from B0 on the first cache miss

The legacy readahead counts continuous readahead cache hits in `cache_hit`. Whenever it goes up to `VM_MAX_CACHE_HIT(=256)`, the flag `RA_FLAG_INCACHE` will be set. It disables further readahead until a cache miss happens, which indicates that the application have walked out of the cached segment. The whole process goes in the following steps:

1. Call `page_cache_readahead()` on each read request;
2. Disable readahead after 256 cache hits;
3. Call `page_cache_readahead()` on each accessed page;
4. Enable readahead on cache miss.

That scheme works, but is not satisfactory.

1. It only works for large files. If a fully cached file is smaller than 1MB, it won't be able to see the light of `RA_FLAG_INCACHE`. This tend to be a common case. Imagine a web server that caches a lot of Web pages and images, or desktop systems that are dominated by small files.
2. Pretend that it happily enters cache-hit-no-readahead mode for a `sendfile(100M)` and avoids extra page cache lookups. Now another overhead rises: `page_cache_readahead()` that used to be called once every `max_readahead` pages will be called on each page to ensure in time restarting of readahead after the first cache miss.

In the new framework, we stop readahead cache hits by taking care that `PG_readahead` be only set on a newly allocated readahead page and get cleared on the first hit. Figure 6 shows how the rules work out. When the new 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 automatically restarts the readahead. If the whole file is cached, there will be no missing or tagged pages at all to trigger readahead.

3.2 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 the cached pages. For the readahead pages, `inactive_list` can be viewed as a simple FIFO queue. It could be rotated quickly in a loaded and memory tight server. *Readahead thrashing* happens when the readahead pages are shifted out of `inactive_list` and reclaimed, before the 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 the largest possible thrashing safe readahead size. As the readahead size steps 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.

Besides the wastage of memory and bandwidth resources, it could be even more destructive for disk I/O. Suppose that the pages in `current_window` are reclaimed when an application is walking in the middle of it. Figure 7 shows the disk I/Os that follow the thrashing. The legacy readahead logic will be notified via `handle_ra_miss()`. However it merely sets a flag `RA_FLAG_MISS` which will take effect in the next readahead. There's no immediate action to recover the to be accessed pages inside `current_window`. The VFS read routine `do_generic_mapping_read()` then starts to fault them in one by one, generating a lot of disk seeks. Overall, up to half pages will be faulted in with tiny 4KB I/Os.

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. Hence it avoids catastrophic 1-page 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 close to the thrashing threshold.

3.3 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 practices of Linux readahead. In the following two cases, the test `offset==prev_offset+1` could fail even when an application visits 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 $0, 0, 0, 0, 1, 1, 1, 1, \ldots$ To cover such cases, this sequentiality criterion has been added in: `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 pages could be: $[0, 1000]$, $[4, 1000]$, $[8, 1000]$, ... Such pattern confuses the legacy readahead totally(figure 9). They will be taken as huge random reads and trigger the following readahead requests: $(0, 32)$, $(4, 32)$, $(8, 32)$, ... Which are overlapped with each other, leading to a lot of readahead cache hits and tiny 4-page I/Os. We get rid of the retried read problem with the new call convention, where readahead is triggered by the real accessed pages instead of spurious read requests. The readahead heuristics won't even be aware of the existence of retried reads.

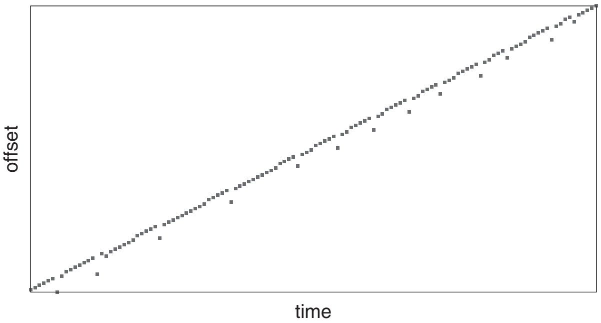


Figure 10: Sequential accesses over NFS can become out of order when reaching the readahead routine

3.4 Semi-sequential 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 order they are requested by a client side application. They could get reordered in the process of being sent out, arriving at the server, and finally hitting the readahead logic. Figure 10 shows a trace of NFS reads observed by the readahead logic.

Ellard et al. proposed the term *δ-consecutive* to describe the `nfsd` reads where any page is within δ pages of its predecessor[7]. They also proposed the *SlowDown* algorithm[8], in which the sequentiality criterion is extended to cover the *δ-consecutive* pattern and the readahead size will be halved step by step instead of being directly reset to 0 on each random read. The *SlowDown* algorithm can handle the NFS case very good and also makes a sequential stream less susceptible to random disturbs.

This paper offers a clean and general readahead framework for semi-sequential reads. Its call convention and sequential detection logics work just fine in face of random disturbances, thanks to the following two properties:

- *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.

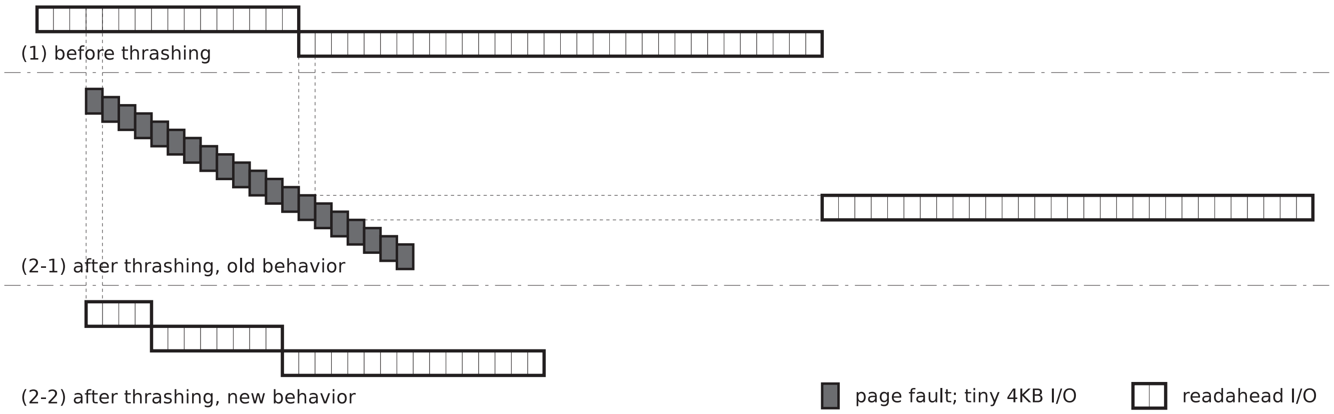

(1) before thrashing
(2-1) after thrashing, old behavior
(2-2) after thrashing, new behavior
page fault; tiny 4KB I/O
readahead I/O

Figure 7: Disk I/Os on readahead thrashing: (1) last two readahead I/Os before thrashing; (2-1) legacy framework: continue the original readahead sequence, the lost pages are fault in with lots of 1-page I/Os; (2-2) new framework: start a new readahead sequence from the current read position

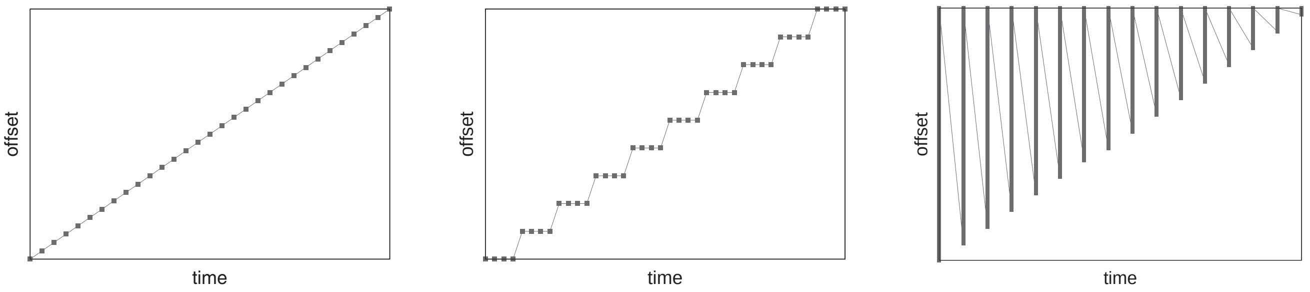


Figure 8: The trivial/unaligned/retried sequential reads

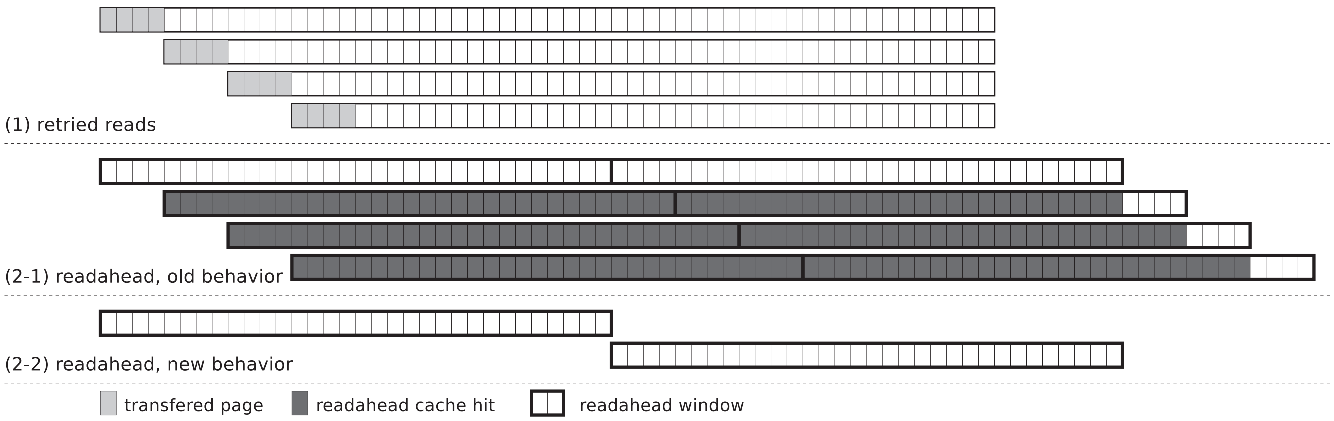


Figure 9: Readahead on retried reads: (1) retried reads, each read only transfered 4 pages; (2-1) legacy framework: each read triggers two back-to-back readaheads, which are mostly overlapped, the real I/O sizes are 64, 4, 4, 4 pages; (2-2) new framework: two normal 32-page readahead I/Os

case	legacy	new	delta
sequential re-read in 4KB	20.30	20.05	−1.2%
sequential re-read in 1MB	37.68	36.48	−3.2%
small files re-read (tar/lib)	49.13	48.47	−1.3%
sequential read on sparse file	389.26	387.97	−0.3%
random read on sparse file	80.44	81.17	+0.9%

Table 2: Comparison of CPU overheads: each task is big enough to show stable numbers in seconds

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.

4. PERFORMANCE EVALUATION

4.1 Overheads in Common Workloads

In order to avoid unnecessary regressions, the new framework inherits almost the same readahead policies from the legacy one for normal workloads. The resulted disk I/O behaviors and performances are close if not identical. However there are good differences in implementation and hence CPU overheads. To demonstrate them, we ran some common workloads on cached files and sparse files, where no disk I/O is involved. The results are shown in table 2. The basic setup is

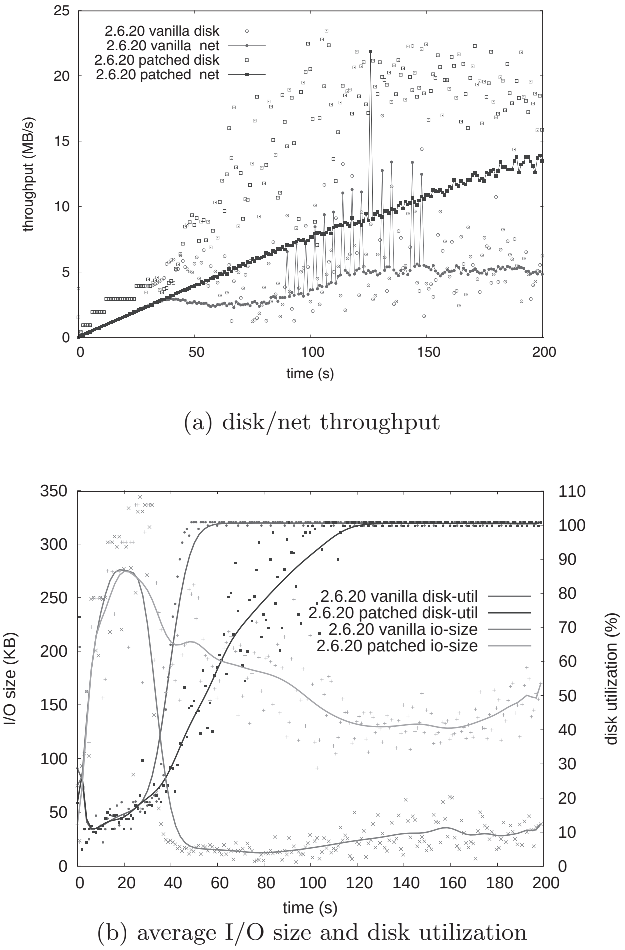
- Linux 2.6.20 (vanilla and patched)
- 1MB max readahead size
- 2.9 GHz Intel® Core 2 CPU
- 2GB memory

Cache hot sequential reads on a huge file are now faster by 1.2% for 1-page reads and by 3.2% for 256-page reads. The improvements come from the fact that the new framework will stop working on continuous cache hits, while the legacy one has to actively trap into the readahead routine for *each* cached page. Cache hot reads on small files (`tar/lib`) see 1.3% speed up. In this case the new framework is able to totally avoid the overheads of readahead cache hits.

We also measured the maximum possible overheads on the trivial random and sequential reads. The scenario is to do 1-page sized reads on a huge sparse file. It is 0.9% worse for random reads, and 0.3% better for sequential ones. The random reads are slightly slower because the actual I/O will be started a bit later - the legacy algorithm submits I/O upon receiving the request, while the new one will do so after looking up the page cache and make sure the page has not been cached. Anyway, the differences are trivial enough to be lost in noises when the costly disk I/Os are involved.

4.2 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 11. The thrashing begins



(a) disk/net throughput

(b) average I/O size and disk utilization

Figure 11: I/O performance on readahead thrashing

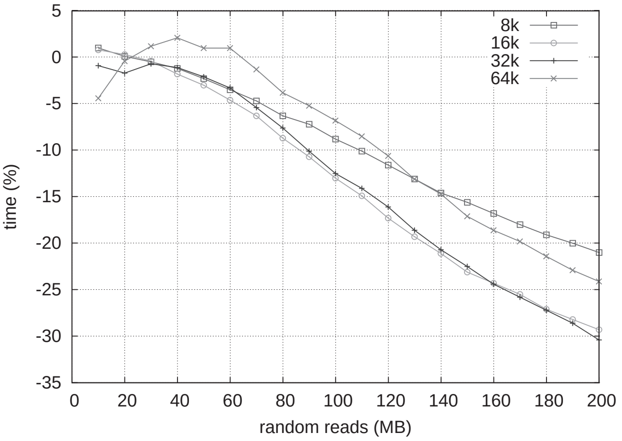


Figure 12: Timing overlapped random reads

	2k transactions run		10k transactions run	
threads	trans/sec	gain	trans/sec	gain
1	38.04	+1.3%	64.56	+2.8%
2	38.99	+1.5%	70.93	+4.4%
4	46.45	+2.3%	85.87	+5.0%
8	52.36	+1.4%	97.89	+3.5%
16	55.18	+1.5%	104.68	+5.7%
32	54.49	+4.5%	104.28	+8.7%
64	54.61	+0.9%	103.68	+7.5%

Table 3: MySQL OLTP performance gains

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.

4.3 Random Reads

The proposed framework may improve performance for high density random reads. For example, we saw pure performance gains when carrying out a series of OLTP tests on MySQL with the sysbench tool(table 3). The database file is accessed mostly by 4-page sized random reads. The reads are not uniformly distributed, there are hot areas that are accessed more frequently than others. Sometimes there will be 16-page readahead triggered by two consecutive reads. They are likely to reside in one of the hot areas, if so, their `PG_readahead` pages are more likely to be accessed later on, and trigger further readahead pages. In general, the more read density, the more readahead and readahead hit ratio, hence the more performance gain.

Hot areas could be detected in two opportunistic ways: a random read that hits a `PG_readahead` page, or a random read that starts from a cached page. In either cases it indicates that some nearby pages have been accessed recently. As long as the workload is a stationary process, doing some readahead could be profitable: the readahead pages will have good probability to be accessed in their life time, saving some possible future seeks.

To verify the effectiveness of the above theory, we fabricated some overlapped random reads that will trigger more and more readaheads as the read density grows. In the benchmark, some 8/16/32/64KB sized random reads are performed on a 100MB file. The reads are page aligned and could be overlapping with each other. On every 10MB reads, the total seconds elapsed are recorded. Figure 12 shows the difference of time between the two readahead frameworks in a progressive way.

The performance gain goes up in general as the read density grows. The largest gain is about 30% for 16/32KB reads. The largest regression is about 2% for 64KB reads. The regression is mainly due to the fact that the new readahead framework will not start I/O as early as the legacy one for random reads that start from a cached page. For example, for an 8-page random read whose first 4 pages are cache hits, the legacy algorithm will submit readahead I/O as early as it sees the request, whereas the new framework will do so after the first 4 cached pages have been transfered to user space. So the user visible I/O latencies are increased. It mainly hurts the throughput for one thread. The overall system throughput on high-concurrency servers won't be affected.

	legacy	new	
avg readahead size(pages)	15	228	x15
cpu %iowait	25.20	21.00	-16.7%
disk %util	13.03	9.62	-26.2%
network bandwidth(MB/s)	37.00	43.40	+17.3%
disk bandwidth(MB/s)	28.18	36.46	+29.4%

Table 4: lighttpd retried reads performance

4.4 User Reports

Besides the home made benchmarks, we'd like to present some user reports coming from real life workloads. They may not be as scientific or rigorous. The numbers only show the possible strength of the new readahead framework.

NFS file serving The server is equipped with big RAID and runs 16 `nfsd` serving 100MB sized files. The sequential read throughput was 260MB/s in 2.6.18, and bumped up to 470MB/s for 2.6.23, which features the new framework.

lighttpd file serving It's an AMD Opteron 250 server with 16G mem, the lighttpd is serving big files to about 1200 concurrent clients. Its trace of `sendfile()` system calls are obviously retried reads: it requests more than 18MB data from the current position to the end of file, but each time only less than 37KB data are transfered.

```
sendfile(188, 1921, [1478592], 19553028) = 37440
sendfile(188, 1921, [1516032], 19515588) = 28800
sendfile(188, 1921, [1544832], 19486788) = 37440
sendfile(188, 1921, [1582272], 19449348) = 14400
sendfile(188, 1921, [1596672], 19434948) = 37440
sendfile(188, 1921, [1634112], 19397508) = 37440
```

Table 4 shows the performance numbers. With 1MB max readahead size, the average readahead I/O size goes from 15 pages with legacy readahead to 228 pages with our new framework, that's 15 times better. As a result, the disk utilization rate dropped by 26.2% and CPU iowaits dropped by 16.7%, while the network throughput increased by 17.3%.

5. RELATED WORK

Prefetching is a well established technique for improving I/O performance. It can be either heuristic or informed. The heuristic algorithms try to predict I/O blocks to be accessed in the future based on the past ones. The most successful one is the sequential readahead, which has long been a standard practice among operating systems[10, 18]. There are also more comprehensive works to mine correlations between files[13, 23, 26] or data blocks[16]. In the other hand, informed prefetching gets hints from the application about its future I/O operations. The hints could either be application controlled[5, 21, 22] or be automatically generated[2].

It was recently argued that more aggressive prefetching policies should be employed[20]. Our experiences are backing it: there are risks of regressions in extending the sequential detection logic to cover more semi-sequential access patterns, however we received no single negative feedback since its wide deployment in Linux 2.6.23. It's all about performance gains in practice, including the tricky AIO[1] and NFS[7, 8] problems.

Caching is another ubiquitous performance technique. It is a common practice to share the prefetch memory with cache memory, hence there could be strong interactions between prefetching and caching. Our readahead framework has basic defensive support for readahead thrashing. There are more comprehensive works dealing with integrated prefetching and caching[3, 4, 5, 6, 12, 21] and schemes to dynamically adapt the prefetch memory[14] or depth[11, 15, 17, 24].

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 bing 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, another consequence of the bigger 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[9].

Flash memory and its caching algorithms fit nicely in one big arena where magnetic disk and its readahead algorithms are not good at: small random accesses. The Intel 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 eliminate 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 IO triggered by readahead will be able to take advantage of the parallel chips. The size of the readahead window required to get full performance from the SSD storage will be different from spinning media, and vary from device to device. So readahead with tunable `max_readahead` is key even on SSD.

6. CONCLUSIONS

We have analyzed major challenges to readahead algorithms and introduced a new Linux readahead framework with new call conventions, revised data structures, relaxed sequential detection logic and simplified readahead heuristics.

The modular design greatly simplified the readahead algorithms. We successfully eliminated the complexity of dual windows, cache hit/miss, unaligned reads and retried reads. The system dynamics are now automatically handled by the framework. Readahead algorithms only need to care about the read patterns and readahead states. We believe that this framework makes a better platform for future work.

It inherits the good I/O behavior and performance for common sequential/random access patterns. The overheads on cache hits are eliminated. Performance on readahead thrashing, retried reads, and reordered NFS reads are greatly improved. We also demonstrate that readahead on high density random reads can be benefitial.

The readahead framework described in this paper has been merged into Linux 2.6.23 with slight changes. Feedbacks from the community are encouraging. It's worth noting that no negative reports have been received till this writing. We conclude that the relaxed readahead heuristics have been successful in improving performance on many workloads while avoiding undesirable side effects in others.

7. FUTURE WORK

There are a lot room for further improvements. The obvious next step would be to support more access patterns. Stride reads occur when an application is reading a matrix in row order that was stored in column order, a typical case in scientific arenas. Backward reads are also possible. And things get more complicated when multiple threads are accessing the same file. Concurrent sequential streams on the same file may become interleaved. In the case of Linux, the readahead state variables are static allocated for each opened file descriptor. When one file descriptor is used for multiple concurrent streams, the states could be overwritten by one another. The newly introduced page flag `PG_readahead` provides a reliable way to identify each stream and tell if the readahead states are valid for the current stream.

The current readahead framework works on logical blocks inside files. It disregards possible file fragmentations, or the stripe boundaries in RAID like configurations, or locking boundaries in cluster file systems. It may be desirable to provide an interface to the underlying file systems for better alignment management. It might also be worthwhile to do block level readahead for meta data operations.

File servers serving large number of concurrent clients may want better thrashing threshold estimation to adapt readahead sizes accordingly. Our framework is merely doing the fundamentals right so that system performance won't fall sharply on thrashing. Also in disk arrays, storage networks and clustered file systems, it is desirable to adaptively scale up the readahead size and async readahead size so that disks and storage servers can be better utilized in parallel.

Acknowledgments

The authors are grateful to Paul E. McKenney, Chris Mason and the anonymous reviewers for careful reviews and valuable comments. We owe a lot to the linux community for warm supports and feedbacks. We also enjoyed great discussions with many attendees of Linux Symposium 2007, 2008 China Linux Developer Symposium, AKA 2007 Kernel Developers Conference and the Intel OTC people. This work was supported by Intel research project 4507345522, National Natural Science Foundation of P.R. China under grant No. 60774038, and National 863 Program under grant No. 2006AA01Z114.

8. REFERENCES

[1] S. Bhattacharya, J. Tran, M. Sullivan, and C. Mason. Linux aio performance and robustness for enterprise workloads. In *Proceedings of the Linux Symposium*, 2004.

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

[3] A. R. Butt, C. Gniady, and Y. C. Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. *SIGMETRICS Perform. Eval. Rev.*, 33:157–168, 2005.

[4] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. 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*, pages 188–197, Ottawa, Ontario, Canada, 1995. ACM.

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

[6] G. Dini, G. Lettieri, and L. Lopriore. Caching and prefetching algorithms for programs with looping reference patterns. *The Computer Journal*, 49:42–61, 2006.

[7] D. Ellard, J. Ledlie, P. Malkani, and M. Seltzer. Passive NFS tracing of email and research workloads. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST'03)*, pages 203–216, San Francisco, CA, March 2003.

[8] D. Ellard and M. Seltzer. Nfs tricks and benchmarking traps. In *Proceedings of the FREENIX 2003 Technical Conference*, pages 101–114, San Antonio, TX, June 2003.

[9] B. Esfahbod. Preload – an adaptive prefetching daemon. Master's thesis, Graduate Department of Computer Science, University of Toronto, 2006.

[10] R. J. Feiertag and E. I. Organick. The multics input/output system. In *Proceedings of the third ACM symposium on Operating systems principles*, pages 35–41, Palo Alto, California, United States, 1971. ACM.

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

[12] B. S. Gill and D. S. Modha. Sarc: sequential prefetching in adaptive replacement cache. In *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*, pages 33–33, Berkeley, CA, USA, 2005. USENIX Association.

[13] T. M. Kroeger and D. D. E. Long. Design and implementation of a predictive file prefetching algorithm. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 105–118, Berkeley, CA, USA, 2001. USENIX Association.

[14] C. Li and K. Shen. Managing prefetch memory for data-intensive online servers. In *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, pages 19–19, Berkeley, CA, USA, 2005. USENIX Association.

[15] C. Li, K. Shen, and A. E. Papathanasiou. Competitive prefetching for concurrent sequential i/o. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 189–202, Lisbon, Portugal, 2007. ACM.

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

[17] S. Liang, S. Jiang, and X. Zhang. Step: Sequentiality and thrashing detection based prefetching to improve performance of networked storage servers. In *ICDCS*, page 64, 2007.

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

[19] R. Pai, B. Pulavarty, and M. Cao. Linux 2.6 performance improvement through readahead optimization. In *Proceedings of the Linux Symposium*, 2004.

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

[21] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 79–95, Copper Mountain, Colorado, United States, 1995. ACM.

[22] R. H. Patterson, G. A. Gibson, and M. Satyanarayanan. A status report on research in transparent informed prefetching. *SIGOPS Oper. Syst. Rev.*, 27(2):21–34, 1993.

[23] J.-F. Paris, A. Amer, and D. D. E. Long. A stochastic approach to file access prediction. In *Proceedings of the international workshop on Storage network architecture and parallel I/Os*, pages 36–40, New Orleans, Louisiana, 2003. ACM.

[24] D. Revel, D. McNamee, D. Steere, and J. Walpole. Adaptive prefetching for device independent file I/O. Technical Report CSE-97-005, 18, 1997.

[25] E. Shriver, C. Small, and K. A. Smith. Why does file system prefetching work? In *ATEC'99: Proceedings of the Annual Technical Conference on 1999 USENIX Annual Technical Conference*, pages 71–84, Berkeley, CA, USA, 1999. USENIX Association.

[26] G. A. S. Whittle, J.-F. Pâris, A. Amer, D. D. E. Long, and R. Burns. Using multiple predictors to improve the accuracy of file access predictions. In *MSS '03: Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*, page 230, Washington, DC, USA, 2003. IEEE Computer Society.

[27] F. Wu, H. Xi, J. Li, and N. Zou. Linux readahead: less tricks for more. In *Proceedings of the Linux Symposium*, 2007.