# Proceedings of the Linux Symposium

June 27th–30th, 2007
Ottawa, Ontario
Canada

## Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

## Review Committee

Andrew J. Hutton, *Steamballoon, Inc.*
Dirk Hohndel, *Intel*
Martin Bligh, *Google*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat, Inc.*
C. Craig Ross, *Linux Symposium*

## Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
Gurhan Ozen, *Red Hat, Inc.*
John Feeney, *Red Hat, Inc.*
Len DiMaggio, *Red Hat, Inc.*
John Poelstra, *Red Hat, Inc.*

# Linux readahead: less tricks for more

Fengguang Wu Hongsheng Xi Jun Li
*University of Science and Technology of China*
wfg@ustc.edu, {xihs,Ljun}@ustc.edu.cn

Nanhai Zou
*Intel Corporation*
nanhai.zou@intel.com

## Abstract

The Linux 2.6 readahead has grown into an elaborate work that is hard to understand and extend. It is confronted with many subtle situations. This paper highlights these situations and proposes alternative solutions to them. By adopting new calling conventions and data structures, we demonstrate that readahead can be made more clean, flexible and reliable, which opens the door for more opportunities.

## 1 Introduction

### 1.1 Background

Readahead is a widely deployed technique to bridge the huge gap between the characteristics of disk drives and the inefficient usage by applications. At one end, disk drives are good at large sequential accesses and bad at seeks. At the other, applications tend to do a lot of tiny reads. To make the two ends meet, modern kernels and disk drives do readahead: to bring in the data before it is needed and try to do so in big chunks.

Readahead brings three major benefits. Firstly, I/O delays are effectively hidden from the applications. When an application requests a page, it has been prefetched and is ready to use. Secondly, disks are better utilized by large readahead requests. Lastly, it helps amortize request processing overheads.

Readahead typically involves actively detecting the access pattern of all read streams and maintaining information about them. Predictions based on on where and how much data will be needed in the near future are made. Finally, carefully selected data is read in before being requested by the application.

There exist APIs (`posix_fadvise(2)`, `madvise(2)`) for the user-space to inform the kernel about its access pattern or more precise actions about data, but few applications bother to take advantage of them. They are mostly doing sequential or random reads, and expecting the kernel to serve these common and simple cases right.

So the kernel has to guess. When done right, readahead can greatly improve I/O throughput and reduce application visible I/O delays. However, a readahead miss can waste bandwidth and memory, and eventually hurt performance.

### 1.2 A brief history

Linux 2.6 implements a generic readahead heuristic in its VFS layer.

The unified readahead framework was first introduced [1] in the early 2.5 time by Andrew Morton. It features the current/ahead windows data structure; readahead/read-around heuristics; protection against readahead thrashing, aggressive cache hits [2] and congested queues [3]. The mmap read-around logic was later taken out by Linus Torvalds[4]. The separation of read-ahead and read-around yields better solutions for both. The rather chaotic mmap reads from executables can now be prefetched aggressively, and the readahead logic can concentrate on detecting sequential reads from random ones.

Handling sequential and random reads right, however, turned out to be a surprisingly hard mission. One big

challenge comes from some mostly random database workloads. In three years, various efforts were made to better support random cases [5, 6, 7, 8, 9]. Finally, Steven Pratt and Ram Pai settled it down by passing the read request size into `page_cache_readahead()` [10, 11]. The addition of read size also helped another known issue: when two threads are doing simultaneous `pread()`s, they will be overwriting each other's readahead states. The readahead logic may see lots of 1-page reads, instead of the true `pread()` sizes. The old solution, in the day when `page_cache_readahead()` was called once per page, was to take a local copy of readahead state in `do_generic_mapping_read` [12, 13].

### 1.3 Moving on

Linux 2.6 is now capable of serving common random/sequential access patterns right. That may be sufficient for the majority, but it sure can do better.

So comes the adaptive readahead patch [14], an effort to bring new readahead capabilities to Linux. It aims to detect semi-sequential access patterns by querying the page-cache context. It also measures the read speed of individual files relative to the page-cache aging speed. That enables it to be free from readahead thrashing, and to manage the readahead cache in an economical way.

The lack of functionality is not an issue with regard to readahead, in fact the complexity of the code actually prevents further innovation. It can be valuable to summarize the experiences we learned in the past years, to analyze and reduce the source of the complexity is what motivated us to do the work of required for this paper.

We propose a readahead algorithm that aims to be clean. It is called on demand, which helps free readahead heuristics from most of the chores that the current algorithm suffers from. The data structure is also revised for ease of use, and to provide exact timing information.

### 1.4 Overview of the rest of the paper

In Section 2, we first discuss the readahead algorithm as in Linux 2.6.20, then proceed to discuss and propose new solutions to the three major aspects (data structure, call scheme, and guideline) of readahead. In Section 3, we analyze how the old/new readahead algorithms work out in various situations. Finally, Section 4 gives benchmark numbers on their overheads and performance.

## 2 Readahead Algorithms

### 2.1 Principles of 2.6 readahead

Figure 1 is a brief presentation of the readahead algorithm in Linux 2.6.20. The heuristics can be summarized in four aspects:

**sequential detection** If the first read at the start of a file, or a read that continues from where the previous one ends, assume a sequential access. Otherwise it is taken as a random read.

The interface demands that it be informed of every read requests. `prev_page` is maintained to be the last page it saw and handled. An oversize read (`read_size > max_readahead`) will be broken into chunks of no more than `max_readahead` and fed to the readahead algorithm progressively.

**readahead size** There are three phases in a typical readahead sequence:

initial When there exists no `current_window` or `ahead_window`, the size of initial readahead is mainly inferred from the size of current read request. Normally `readahead_size` will be 4 or 2 times `read_size`.

ramp-up When there is a previous readahead, the size is doubled or x4.

full-up When reaching `max_readahead`.

It is possible to jump directly into full-up phase, if the request size is large enough (e.g. `sendfile(10M)`).

**readahead pipelining** To maximally overlap application processing time and disk I/O time, it maintains two readahead windows: `current_window` is where the application expected to be working on; `ahead_window` is where asynchronous I/O happens. `ahead_window` will be opened/renewed in advance, whenever it sees a sequential request that

- is oversize
- has only `current_window`
- crossed into `ahead_window`

```
do_generic_mapping_read:
    call page_cache_readahead
    for each page
        if is prev_page + 1
            call page_cache_readahead
        if page not cached
            report cache miss
            leave cache hit mode

page_cache_readahead:
    handle unaligned read
    set prev_page to current page index
    if in cache hit mode
        return
    shift prev_page to the last requested page,
        but no more than max_readahead pages
    if is sequential read and no current_window
        make current_window
        call blockable_page_cache_readahead
        if is oversize read
            call make_ahead_window
    elif is random read
        clear readahead windows
        limit size to max_readahead
        call blockable_page_cache_readahead
    elif no ahead_window
        call make_ahead_window
    elif read request crossed into ahead_window
        advance current_window to ahead_window
        call make_ahead_window
    ensure prev_page do not overrun ahead_window

make_ahead_window:
    if have seen cache miss
        clear cache miss status
        decrease readahead size by 2
    else
        x4 or x2 readahead size
        limit size to max_readahead
    call blockable_page_cache_readahead

blockable_page_cache_readahead:
    if is blockable and queue congested
        return
    submit readahead io
    if too many continuous cache hits
        clear readahead windows
        enter cache hit mode
```

Figure 1: readahead in 2.6.20

**cache hit/miss** A generic *cache hit/miss* happens when a page to be accessed is found to be cached/missing. However, the terms have specific meanings in 2.6 readahead.

A *readahead cache hit* happens when a page to be readahead is found to be cached already. A long run of readahead cache hits indicates an already cached file. When the threshold `VM_MAX_CACHE_HIT(=256)` is reached, readahead will be turned off to avoid unnecessary lookups of the page-cache.

A *readahead cache miss* happens when a page that was brought in by readahead is found to be lost on time of read. The page may be relaimed prematurely, which indicates readahead thrashing. The readahead size can be too large, so decrease it by 2 for the next readahead.

### 2.2 Readahead windows

The 2.6 readahead adopts dual windows to achieve readahead pipelining: while the application is walking in the `current_window`, I/O is underway in the `ahead_window`. Although it looks straightforward, the implementation of this concept is not as simple as one would think.

For the purpose of pipelining, we want to issue I/O for the next readahead before the not-yet-consumed readahead pages fall under a threshold, *lookahead size*. A value of `lookahead_size = 0` disables pipelining, whereas `lookahead_size = readahead_size` opens full pipelining.

The current/ahead windows scheme is one obvious way to do readahead pipelining. It implies `lookahead_size` to be `readahead_size - read_size`[1] for the initial readahead. It then ranges from `readahead_size` to `readahead_size + read_size` for the following ones. It is a vague range due to the fact that 2.6.20 readahead pushes forward the windows as early as it sees the read request (instead of one realtime page read) crossing into the `ahead_window`.

However, the scheme leads to obscure information and complicated code. The lookahead size is implicitly coded and cannot be freely tuned. The timing information for the previous readahead may be too vague to be useful. The two windows bring three possible combinations of on/off states. The code has to probe for the existence of `current_window` and/or `ahead_window` before it can do any query or action on them. The heuristics also have to explicitly open `ahead_window` to start readahead pipelining.

Now let's make a study of the information necessary for a sequential readahead:

1. To work out the position of next readahead, that of the previous one will be sufficient: We normally apply a simple size ramp up rule: `(offset, size) => (offset+size, size*2)`.
2. Optionally, in case the previous readahead pages are lost, the timing information of their first enqueue to `inactive_list` would be helpful. Assume the reader is now at `offset`, and was at page `la_index` when the lost pages were first brought in, then `thrashing_threshold = offset - la_index`.
3. To achieve pipelining, indicating a *lookahead page* would be sufficient: on reading of which it should be invoked to do readahead in advance.

We revised the data structure (Figure 2) to focus on the previous readahead, and to provide the exact timing information. The changes are illustrated in Figure 3 and compared in Table 1.

### 2.3 Readahead on demand

The 2.6 readahead works by inspecting *all* read requests and trying to discover sequential patterns from them. In theory, it is a sound way. In practice, it makes a lot of fuss. Why should we call excessively into `page_cache_readahead()` only to do nothing? Why handle cache hits/misses in convoluted feedback loops?

In fact, there are only two cases that qualify for a readahead:

**sync readahead on cache miss** A cache miss occurred, the application is going to do I/O anyway. So try readahead and check if some more pages should be piggybacked.

[1] assume `read_size <= max_readahead`

```
 struct file_ra_state {
-   unsigned long start;            /* Current window */
-   unsigned long size;
-   unsigned long ahead_start;      /* Ahead window */
-   unsigned long ahead_size;
-   unsigned long cache_hit;        /* cache hit count */
-   unsigned long flags;            /* RA_FLAG_MISS | RA_FLAG_INCACHE */
+   pgoff_t la_index;               /* enqueue time */
+   pgoff_t ra_index;               /* begin offset */
+   pgoff_t lookahead_index;        /* time to do next readahead */
+   pgoff_t readahead_index;        /* end offset */
    unsigned long prev_page;        /* Cache last read() position */
    unsigned long ra_pages;         /* Maximum readahead window */
 };
```

Figure 2: revising readahead data structure

```
(a) 2.6.20 readahead

    |----- size ---->|------------ ahead_size ------------->|
 ===|================|=====================================|=====
    ^start           ^ahead_start

(b) on-demand readahead

                     |------------------- readahead size -->|
          |---- reader on the way --->|-- lookahead size -->|
 ===|=====#==========|================#=====================|=====
          ^          ^                ^                     ^
        la_index   ra_index  lookahead_index       readahead_index
```

Figure 3: tracking readahead windows

| question | on-demand readahead | 2.6.20 readahead |
|---|---|---|
| when to do next readahead | `lookahead_index` | around `ahead_start` or `start` |
| where to do next readahead | `readahead_index` | `start + size` or `ahead_start + ahead_size` |
| the time of previous readahead | `la_index` | maybe/roughly `start` |
| the size of previous readahead | `readahead_index - ra_index` | `size` or `ahead_size` |

Table 1: deciding the next readahead

**async readahead on lookahead page** The application is walking onto a readahead page with flag `PG_readahead`, or a *lookahead mark*. It indicates that the readahead pages in the front are dropping to `lookahead_size`, the threshold for pipelining. So do readahead in advance to reduce application stalls.

When called on demand, the readahead heuristics can be liberated from a bunch of special cases. The details about them will be covered in Section 3.

### 2.4 The new algorithm

Figure 4 shows the proposed on-demand readahead algorithm. It is composed of a list of condition-action blocks. Each condition tests for a specific case (Table 2), and most actions merely fill the readahead state with proper values (Table 3).

**random** A small, stand-alone read. Take it as a random read, and read as is.

**lookahead** It is lookahead time indicated by the readahead state, so ramp up the size quickly and do the next readahead.

**readahead** It is readahead time indicated by the readahead state. We can reach here if the lookahead mark was somehow ignored (queue congestion) or skipped (sparse read). Do the same readahead as in lookahead time.

**initial** First read on start of file. It may be accessing the whole file, so start readahead.

**oversize** An oversize read. It cannot be submitted in one huge I/O, so do it progressively as a readahead sequence.

**miss** A sequential cache miss. Start readahead.

**interleaved** A lookahead hit without a supporting readahead state. It can be some interleaved sequential streams that keep invalidating each other's readahead state. The lookahead page indicates that the new readahead will be at least the second one in the readahead sequence. So get the initial readahead size and ramp it up once.

The new algorithm inherits many important behaviors from the current one, such as random reads, and the size ramp up rule on sequential readahead. There are also some notable changes:

1. A new parameter `page` is passed into `ondemand_readahead()`. It tells whether the current page is present. A value of `NULL` indicates a synchronous readahead, otherwise an asynchronous one.

2. A new parameter `begin_offset` is introduced to indicate where the current read request begins. `prev_page` now simply tracks the last accessed page of previous request. Hence the new sequential access indicator becomes: `sequential = (begin_offset - prev_page <= 1)`.

3. I/O for overlapped random reads may not be submitted as early. Suppose a 8-page random read, whose first 4 pages are overlapped with a previous read. The 2.6 readahead will emit request for all of the 8 pages before accessing the first page. While the on-demand readahead will ask for the remaining 4 pages on accessing the 5th page. Hence it avoids some unnecessary page-cache lookups, at the cost of not being able to overlap transfer of the leading cached pages with I/O for the following ones.

4. Linux 2.6.20 only does readahead for sequential read requests. In the new design, we loosen the criteria a bit: the lookahead hit alone can trigger the next readahead. It enables detection of interleaved reads.

It is not as safe to ignore sequentialness, but the risk is pretty low. Although we can create a set of random reads to trigger a long run of readahead sequences, it is very unlikely in reality. One possible candidate may be stride reads. But it cannot even cheat the algorithm through the size ramp-up phase, where the lookahead pages distribute in a non-uniform way.

The support of interleaved reads is minimal. It makes no extra efforts to detect interleaved reads. So the chances of discovering them is still low. Interleaved sequential reads may or may not be readahead, or may be served intermittently.

```
do_generic_mapping_read:
    for each page
        if page not cached
            call ondemand_readahead
        if page has lookahead mark
            call ondemand_readahead
    set prev_page to last accessed page

ondemand_readahead:
    if is asynchronous readahead and queue congested
        return
    if at start of file
        set initial sizes
    elif is small random read in wild
        read as is
        return
    elif at lookahead_index or readahead_index
        ramp up sizes
    else
        set initial sizes
        if has lookahead mark
            ramp up size
    fill readahead state
    submit readahead io
    set lookahead mark on the new page at new lookahead_index
```

Figure 4: on-demand readahead algorithm

| case | description | condition |
|---|---|---|
| initial | read on start of file | `!offset` |
| oversize | random oversize read | `!page && !sequential && size > max` |
| random | random read | `!page && !sequential` |
| lookahead | lookahead hit | `offset == ra->lookahead_index` |
| readahead | readahead hit | `offset == ra->readahead_index` |
| miss | sequential cache miss | `!page` |
| interleaved | lookahead hit with no context | `page` |

Table 2: detecting access patterns

| case | `ra_index` | `ra_size` | `la_size` |
|---|---|---|---|
| random | `offset` | `size` | `0` |
| lookahead,readahead | `ra->readahead_index` | `get_next_ra_size(ra)` | |
| initial,oversize,miss | `offset` | `get_init_ra_size(size,max)` | `1` |
| interleaved | `offset + 1` | `get_init_ra_size(...) * 4` | `1` |

Table 3: deciding readahead parameters

## 3 Case Studies

In this section, we investigate special situations the readahead algorithm has to confront:

1. Sequential reads do not necessary translate into incremental page indexes: multi-threaded reads, retried reads, unaligned reads, and sub-page-size reads.
2. Readahead should not always be performed on sequential reads: cache hits, queue congestion.
3. Readahead may not always succeed: out of memory, queue full.
4. Readahead pages may be reclaimed before being read: readahead thrashing.

### 3.1 Cache hits

Ideally, no readahead should ever be performed on cached files. If a readahead is done on a cached file, then this can cost many pointless page cache lookups. In a typical system, reads are mostly performed on cached pages. Cache hits can far outweigh cache misses.

The 2.6 readahead detects excessive cache hits via `cache_hit`. It counts the continuous run of readahead pages that are found to be already cached. 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.

In summary,

1. Always call `page_cache_readahead()`;
2. Disable readahead after 256 cache hits; and
3. Enable readahead on cache miss.

That scheme works, but is not satisfactory.

1. It only works for large files. If a file is fully cached but smaller than 1*MB*, it won't be able to see the light of `RA_FLAG_INCACHE`, which can be a common case. Imagine a web server that caches a lot of small to medium `html/png` files and desktop systems.
2. Pretend that it happily enters cache-hit-no-readahead mode for a `sendfile(100M)` and avoids page-cache lookups. Now another overhead arises: `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.

The above issues are addressed in on-demand readahead by the following rules:

1. Call `ondemand_readahead()` on cache miss;
2. Call `ondemand_readahead()` on lookahead mark; and
3. Only set lookahead mark on a newly allocated readahead page.

Table 4 compares the two algorithms' behavior on various cache hit situations. It is still possible to apply the threshold of `VM_MAX_CACHE_HIT` in the new algorithm, but we'd prefer to keep it simple. If a random cached page happens to disable one lookahead mark, let it be. It would be too rare and non-destructive to ask for attention. As for real-time applications, they need a different policy—to persist on cache hits.

### 3.2 Queue Congestion

When the I/O subsystem is loaded, it becomes questionable to do readahead. In Linux 2.6, the load of each disk drive is indicated by its request queue. A typical request queue can hold up to `BLKDEV_MAX_RQ(=128)` requests. When a queue is $7/8$ full, it is flagged as *congested*; When it is completely full, arriving new requests are simply dropped. So in the case of a congested queue, doing readahead risks wasting the CPU/memory resources: to scan through the page-cache, allocate a bunch of readahead pages, get rejected by the request system, and finally free up all the pages—a lot of fuss about nothing.

Canceling readahead requests on high I/O pressure can help a bit for the time being. However, if it's only about breaking large requests into smaller ones, the disks will be serving the same amount of data with much more seeks. In the long run, we hurt both I/O throughput and latency.

| | ondemand_readahead() | page_cache_readahead() |
|---|---|---|
| large cached chunk | not called | called on *every* page to recheck |
| full cached small files<br>prefetched chunks | | called to do readahead |
| small cached chunk | may or may not be called | |
| cache miss | called and do readahead *now* | restart readahead *after* first miss |

Table 4: readahead on cache hits

So the preferred way is to defer readahead on a congested queue. The on-demand readahead will do so for asynchronous readaheads. One deferred asynchronous readahead will return some time later as a synchronous one, which will always be served. The process helps smooth out the variation of load, and will not contribute more seeks to the already loaded disk system.

The current readahead basically employs the same policy. Only that the decisions on whether to force a readahead are littered throughout the code, which makes it less obvious.

### 3.3 Readahead thrashing

In a loaded server, the page-cache can rotate pages quickly. The readahead pages may be shifted out of the LRU queue 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, readahead thrashing happened. In this case, the current readahead decreases the next readahead size by 2. By doing so it hopes to adapt to the thrashing threshold. Unfortunately, the algorithm does not remember it. Once it steps slowly off to the thrashing threshold, the thrashings stop. It then immediately reverts back to the normal behavior of ramping up the window size by 2 or 4. Which starts a new round of thrashings. On average, about half of the readahead pages can be thrashed.

It would be even more destructive for disk throughput. Suppose that `current_window` is thrashed when an application is walking in the middle of it. The 2.6 readahead algorithm will be notified via `handle_ra_miss()`. But it merely sets a flag `RA_FLAG_MISS`, and takes no action to recover the `current_window` pages to be accessed. `do_generic_mapping_read()` then starts to fault in them one by one, generating a lot of disk seeks. Overall, up to half pages may be faulted in this crude way.

The on-demand readahead takes no special action against readahead thrashing. Once thrashed, an initial readahead will be started from the current position. It does not cut down the number of thrashed pages, but does avoid the catastrophic seeks. Hence it performs much better on thrashing.

### 3.4 Unaligned reads

File operations work on byte ranges, while the readahead routine works on page offsets. When an application issues 10000B sized reads, which do not align perfectly to the 4K page boundary, the readahead code will see an $offset + size$ flow of $0+3, 2+2, 4+3, 7+2, 9+3, 12+2, 14+3, 17+2, 19+3, \ldots$. Note that some requests overlap for one page. It's no longer an obvious sequential pattern.

Unaligned reads are taken care of by allowing `offset == prev_page` [15] in 2.6 readahead and on-demand readahead.

### 3.5 Retried reads

Sometimes the readahead code will receive an interesting series of requests[16] that looks like: $0+1000, 10+990, 20+980, 30+970, \ldots$. They are one normal read followed by some retried ones. They may be issued by the retry-based AIO kernel infrastructure, or retries from the user space for unfinished `sendfile()`s.

This pattern can confuse the 2.6.20 readahead. Explicit coding is needed to ignore the return of reads that have already been served.

The on-demand readahead is not bothered by this issue. Because it is called on the page to be accessed now, instead of the read request.

| case | 2.6.20 | on-demand | overheads | reasoning |
|---|---|---|---|---|
| sequential re-read in 4*KB* | 20.30 | 20.05 | −1.2% | no readahead invocation |
| sequential re-read in 1*MB* | 37.68 | 36.48 | −3.2% | |
| small files re-read (`tar /lib`) | 49.13 | 48.47 | −1.3% | no page-cache lookup |
| random reading sparse file | 81.17 | 80.44 | +0.9% | one extra page-cache lookup per cache miss |
| sequential reading sparse file | 389.26 | 387.97 | −0.3% | less readahead invocations |

Table 5: measuring readahead overheads

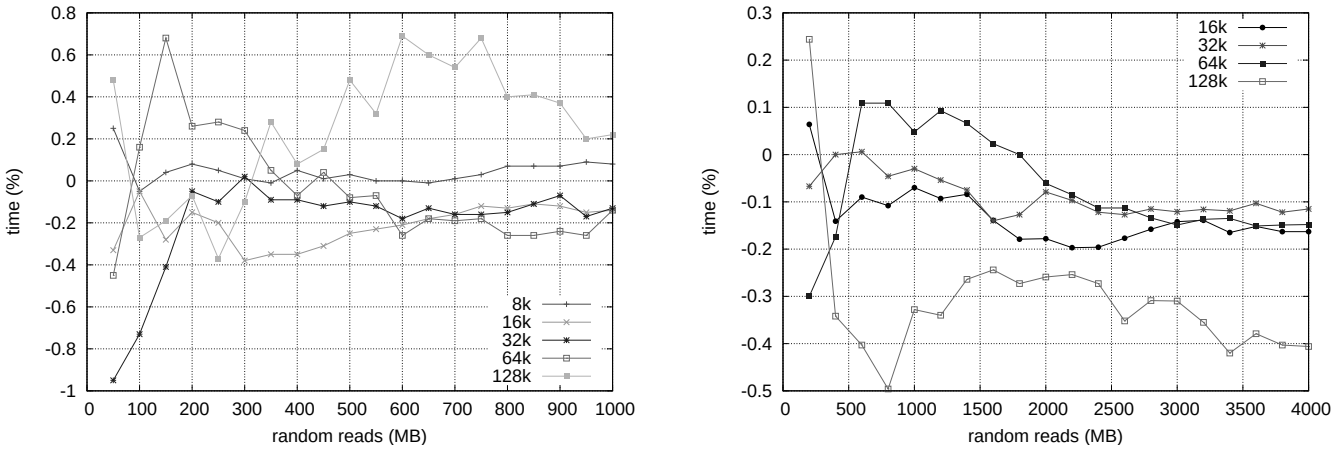



Figure 5: timing overlapped random reads

### 3.6 Interleaved reads

When multiple threads are reading on the same file descriptor, the individual sequential reads get interleaved and look like random ones to the readahead heuristics. Multimedia files that contain separated audio/video sections may also lead to interleaved access patterns.

Interleaved reads will totally confuse the current readahead, and are also beyond the mission of on-demand readahead. However, it does offer minimal support that may help some interleaved cases. Take, for example, the 1-page requests consisting of two streams: $0, 100, 1, 101, 2, 102, \ldots$. The stream starting from 0 will get readahead service, while the stream from 100 will still be regarded as random reads. The trick is that the read on page 0 triggers a readahead (the initial case), which will produce a lookahead mark. The following reads will then hit the lookahead mark, make further readahead calls and push forward the lookahead mark (the lookahead case).

## 4 Performance

### 4.1 Benchmark environment

The benchmarks are performed on a Linux 2.6.20 that is patched with the on-demand readahead. The basic setup is

- 1*MB* max readahead size
- 2.9GHz Intel Core 2 CPU
- 2*GB* memory
- 160G/8M Hitachi SATA II 7200 RPM disk

### 4.2 Overheads

Table 5 shows the max possible overheads for both algorithms. Each test is repeated sufficient times to get the stable result. When finished, the seconds are summed up and compared.

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. Cache hot reads on small files (`tar /lib`) see a 1.3% speed up.

We also measured the maximum possible overheads on random/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. But don't take the two numbers seriously. They will be lost in the background noise when doing large sized reads, and doing it on snail-paced disks.

### 4.3 Overlapped random reads

We benchmarked $8/16/32/64/128KB$ random reads on a $500/2000MB$ file. The requests are aligned to small $4KB$ boundaries and therefore can be overlapping with each other. On every $50/200MB$ read, the total seconds elapsed are recorded and compared. Figure 5 demonstrates the difference of time in a progressive way. It shows that the $128KB$ case is unstable, while others converge to the range $(-0.2\%, 0.1\%)$, which are trivial variations.

### 4.4 iozone throughput

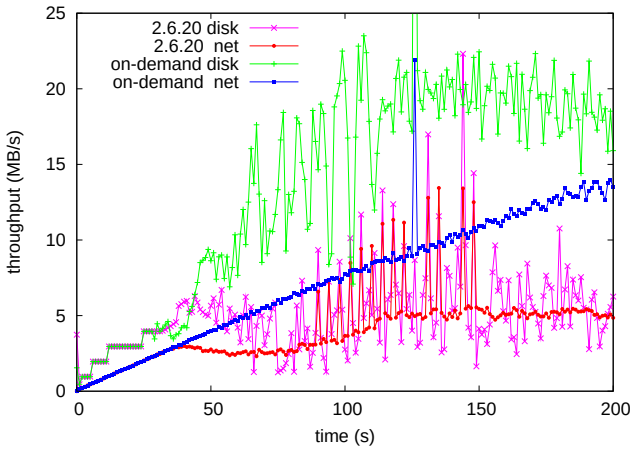
We ran the iozone benchmark with the command `iozone -c -t1 -s 4096m -r 64k`. That's doing $64KB$ non-overlapping reads on a $4GB$ file. The throughput numbers in Table 6 show that on-demand readahead keeps roughly the same performance.

| access pattern | 2.6.20 | on-demand | gain |
|---|---|---|---|
| Read | 62085.61 | 62196.38 | +0.2% |
| Re-read | 62253.49 | 62224.99 | −0.0% |
| Reverse Read | 50001.21 | 50277.75 | +0.6% |
| Stride read | 8656.21 | 8645.63 | −0.1% |
| Random read | 13907.86 | 13924.07 | +0.1% |
| Mixed workload | 19055.29 | 19062.68 | +0.0% |
| Pread | 62217.53 | 62265.27 | +0.1% |

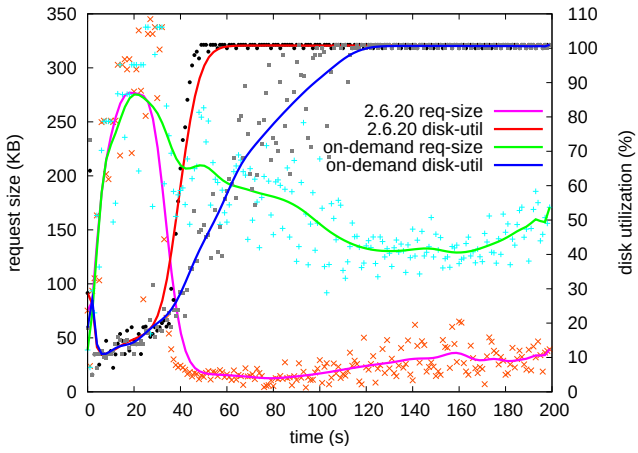
Table 6: iozone throughput benchmark (KB/s)

### 4.5 Readahead thrashing

We boot the kernel with `mem=128m single`, and start a $100KB/s$ stream on every second. Various statistics are collected and showed in Figure 6. The thrashing begins at $20sec$. The 2.6 readahead starts to overload the disk at $40sec$, and eventually achieved $5MB/s$ maximum network throughput. The on-demand readahead throughput keeps growing, and the trend is going up to $15MB/s$. That's three times better.




(a) disk/net throughput on loaded disk




(b) average I/O size and disk utilization

Figure 6: performance on readahead thrashing

## 5 Conclusion

This work greatly simplified Linux 2.6 readahead algorithm. We successfully eliminated the complexity of dual windows, cache hit/miss, unaligned reads, and retried reads. The resulting code is more clean and should be easier to work with. It maintains roughly the same behavior and performance for common sequential/random access patterns. Performance on readahead thrashing and cache hits are improved noticeably.

## 6 Future Work

The algorithm is still young and imperfect. It needs more benchmarks, real-world tests, and fine tuning. Look-ahead size may be a bit smaller, especially for the initial readahead. It does not impose strict sequential

checks, which may or may not be good. The overlapped random reads may also be improved.

Then we can embrace all the fancy features that were missed for so long time. To name a few: interleaved reads from multimedia/multi-threaded applications; clustered random reads and chaotic semi-sequential reads from some databases; backward reads and stride reads in scientific arenas; real thrashing prevention and efficient use of readahead cache for file servers. Sure there are more. They can be packed into an optional kernel module for ease of use and maintenance.

## References


[1] Andrew Morton, *[PATCH] readahead*, git commit `b546b96d0969de0ff55a3942c71413392cf86d2a` [2]

[2] Andrew Morton, *[PATCH] readahead optimisations*, git commit `213f035476c932921d6281e4d5d39585f214a2eb`

[3] Andrew Morton, *[PATCH] rework readahead for congested queues*, git commit `10d05dd588a3879f9b40725a9073bc97fcd44776`

[4] Linus Torvalds, *[PATCH] Simplify and speed up mmap read-around handling*, git commit `d5cfe1b35c4e81f4c4dc5139bd446f04870ebf90`

[5] Andrew Morton, *[PATCH] Allow VFS readahead to fall to zero*, git commit `71ddf2489c68cf145fb3f11cba6152de49e02793`

[6] Ram Pai, *[PATCH] readahead: multiple performance fixes*, git commit `2bb300733b3647462bddb9b993a6f32d6cbcdbbc`

[7] Ram Pai, *[PATCH] speed up readahead for seeky loads*, git commit `ef12b3c1abce83e8e25d27bdaab6380238e792ff`

[8] Suparna Bhattacharya, Ram Pai, *[PATCH] adaptive lazy readahead*, git commit `87698a351b86822dabbd8c1a34c8a6d3e62e5a77`

[9] Ram Pai, Badari Pulavarty, Mingming Cao, *Linux 2.6 performance improvement through readahead optimization*, `http://www.linuxsymposium.org/proceedings/reprints/Reprint-Pai-OLS2004.pdf`

[10] *Simplified Readahead*, `http://groups.google.com/group/linux.kernel/browse_thread/thread/e5f475d4a11759ba/697d85a0d86458d3?&hl=en#697d85a0d86458d3`

[11] Steven Pratt, Ram Pai, *[PATCH] Simplified readahead*, git commit `e8eb956c01529eccc6d7407ab9529ccc6522600f`

[12] *Linux: Random File I/O Regressions In 2.6*, `http://kerneltrap.org/node/3039`

[13] Andrew Morton, *[PATCH] readahead: keep file->f_ra sane*, git commit `2ea7dd3fc9bc35ad0c3c17485949519cb691c097`

[14] *Linux: Adaptive Readahead*, `http://kerneltrap.org/node/6642`

[15] Oleg Nesterov, *[PATCH] readahead: improve sequential read detection*, git commit `03a554d2325ef5f3160514359330965fd7640e81`

[16] Suparna Bhattacharya, John Tran, Mike Sullivan, Chris Mason, *Linux AIO Performance and Robustness for Enterprise Workloads*, `http://www.linuxsymposium.org/proceedings/reprints/Reprint-Bhattacharya-OLS2004.pdf`


[2] all git commits are accessible from `http://git.kernel.org/?p=linux/kernel/git/torvalds/old-2.6-bkcvs.git`