Readahead: time-travel techniques for desktop and embedded systems

Michael Opdenacker
Free Electrons
michael@free-electrons.com

Abstract

Readahead techniques have successfully been used to reduce boot time in recent GNU/Linux distributions like Fedora Core or Ubuntu. However, in embedded systems with scarce RAM, starting a parallel thread reading ahead all the files used in system startup is no longer appropriate. The cached pages could be reclaimed even before accessing the corresponding files.

This paper will first guide you through the heuristics implemented in kernelspace, as well as through the userspace interface for preloading files or just announcing file access patterns. Desktop implementations will be explained and benchmarked. We will then detail Free Electrons attempts to implement an easy to integrate helper program reading ahead files at the most appropriate time in the execution flow.

This paper and the corresponding presentation target desktop and embedded system developers interested in accelerating the course of Time.

1 Reading ahead: borrowing time from the future

1.1 The page cache

Modern operating system kernel like Linux manage and optimize file access through the *page cache*. When the same file is accessed again, no disk I/O is needed if the file contents are still in the page cache. This dramatically speeds up multiple executions of a program or multiple accesses to the same data files.

Of course, the performance benefits depend on the amount of free RAM. When RAM gets scarce because of allocations from applications, or when the contents of more files have to be loaded in page cache, the kernel has to reclaim the oldest pages in the page cache.

1.2 Reading ahead

The idea of reading ahead is to speed up the access to a file by preloading at least parts of its contents in page cache ahead of time. This can be done when spare I/O resources are available, typically when tasks keep the processor busy. Of course, this requires the ability to predict the future!

Fortunately, the systems we are dealing with are predictable or even totally predictable in some situations!

- Predictions by watching file read patterns. If pages are read from a file in a sequential manner, it makes sense to go on reading the next blocks in the file, even before these blocks are actually requested.
- System startup. The system init sequence doesn't change. The same executables and data files are always read in the same order. Slight variations can still happen after a system upgrade or when the system is booted with different external devices connected to it.
- Applications startup. Every time a program is run, the same shared libraries and some parts of the program file are always loaded. Then, many programs open the same resource or data files at system startup. Of course, file reading behaviour is still subject to changes, according to how the program was started (calling environment, command arguments...).

If enough free RAM is available, reading ahead can bring the following benefits:

- Of course, reduced system and application startup time.
- Improved disk throughput. This can be true for storage devices like hard disks which incur a high time cost moving the disk heads between random sectors. Reading ahead feeds the I/O scheduler with more I/O requests to manage. This scheduler can then reorder requests in a more efficient way, grouping a greater number of contiguous disk blocks, and reducing the number of disk head moves. This is much harder to do when disk blocks are just read one by one.
- Better overall utilization for both I/O and processor resources. Extra file I/O is performed when the processor is busy. Context switching, which costs precious CPU cycles, is also reduced when a program no longer needs to sleep waiting for I/O, because the data it is requesting have already been fetched.

2 Kernel space readahead

2.1 Implementation in stock kernels

The description of the Linux kernel readahead mechanism is based on the latest stable version available at the time of this writing, Linux 2.6.20.

When the kernel detects sequential reading on a file, it starts to read the next pages in the file, hoping that the running process will go on reading sequentially.

As shown in Figure 1, the kernel implements this by managing two read windows: the *current* and *ahead* one,

While the application is walking the pages in the current window, I/O is underway on the ahead window. When the current window is fully traversed, it is replaced by the ahead window. A new ahead window is then created, and the corresponding batch of I/O is submitted.

This way, if the process continues to read sequentially, and if enough free memory is available, it should never have to wait for I/O.

Of course, any seek or random I/O turns off this readahead mode.

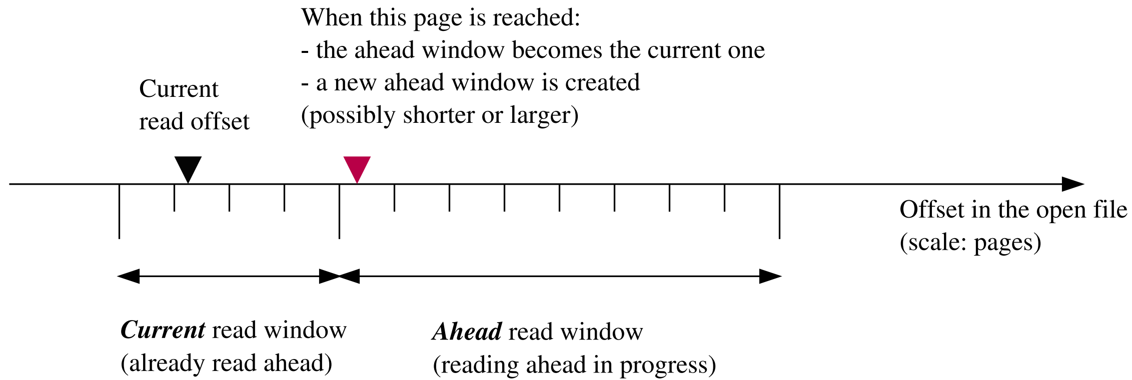

Figure 1: Stock kernel implementation

The kernel actually checks how effective reading ahead is to adjust the size of the new ahead window. If a page cache miss is encountered, it means that some of its pages were reclaimed before being accessed by the process. In this case, the kernel reduces the size of the ahead window, down to `VM_MIN_READAHEAD` (16 KB). Otherwise, the kernel increases this size, up to `VM_MAX_READAHEAD` (128 KB).

The kernel also keeps track of page cache hits, to detect situations in which the file is partly or fully in page cache. When this happens, readahead is useless and turned off.

Implementation details can be found in the `mm/readahead.c` file in the kernel sources.[1]

The initial readahead implementation in Linux 2.6 is discussed in the 2004 proceedings [7] of the Ottawa Linux Symposium.

2.2 Adaptive readahead patches

Many improvements to the kernel readahead mechanism have been proposed by WU Fengguang through the *Adaptive readahead* patchset, since September 2005 (as announced on this LWN article [1].

In addition to the standard sequential reading scenario, this patchset also supports:

- a readahead window which can grow up to 1 MB, depending on the application behaviour and available free memory
- parallel / interleaved sequential scans on one file
- sequential reads across file open/close
- mixed sequential / random accesses
- sparse / skimming sequential read
- backward sequential reading
- delaying readahead if the drive is spinned down in laptop mode

At the time of this writing the latest benchmarks [3] show access time improvements in most cases.

This patchset and its ideas will be described in detail by WU Fengguang himself at this 2007 edition of the Ottawa Linux Symposium.

[1] A very convenient way of studying kernel source files is using a Linux Cross Reference (LXR) website indexing the kernel sources, such as `http://lxr.free-electrons.com`.

3 User space readahead interface

We've seen how the kernel can do its best to predict the future from recent and present application behaviour, to improve performance.

However, that's about all a general purpose kernel can predict. Fortunately, the Linux kernel allows userspace to let it know its own predictions. Several system call interfaces are available.

3.1 The readahead system call

```
#include <fcntl.h>

ssize_t readahead(
        int fd,
        off64_t *offset,
        size_t count);
```

Given an open file descriptor, this system call allows applications to instruct the kernel to readahead a given segment in the file.

Though any `offset` and `count` parameters can be given, I/O is performed in whole pages. So `offset` is rounded down to a page boundary and bytes are read up to the first page boundary greater than or equal to `offset+count`.

Note that `readahead` blocks until all data have been read. Hence, it is typically called from a parallel thread.

See the manual page for the `readahead` system call [6] for details.

3.2 The fadvise system call

Several variants of this system call exist, depending on your system or GNU/Linux distribution: `posix_fadvise`, `fadvise64`, `fadvise64_64`.

They all have the same prototype though:

```
#define _XOPEN_SOURCE 600
#include <fcntl.h>

int posix_fadvise(
    int fd,
    off_t offset,
    off_t len,
    int advice);
```

Programs can use this system call to announce an intention to access file data in a specific pattern in the future, thus allowing the kernel to perform appropriate optimizations.

Here is how the Linux kernel interprets the possible settings for the `advice` argument:

- `POSIX_FADV_NORMAL`: use the default readahead window size.
- `POSIX_FADV_SEQUENTIAL`: sequential access with increasing file offsets. Double the readahead window size.
- `POSIX_FADV_RANDOM`: random access. Disable readahead.
- `POSIX_FADV_WILLNEED`: the specified data will be accessed in the near future. Initiate a non-blocking read of the specified region into the page cache.
- `POSIX_FADV_NOREUSE`: similar, but the data will just be accessed once.
- `POSIX_FADV_DONTNEED`: attempts to free the cached pages corresponding to the specified region, so that more useful cached pages are not discarded instead.

Note that this system call is not binding: the kernel is free to ignore the given advise.

Full details can be found on the manual page for `posix_fadvise` [5].

3.3 The madvise system call

```
#include <sys/mman.h>

int madvise(
    void *start,
    size_t length,
    int advice);
```

The `madvise` system call is very similar to `fadvise`, but it applies to the address space of a process.

When the specified area maps a section of a file, `madvise` information can be used by the kernel to readahead pages from disk or to discard page cache pages which the application will not need in the near future.

Full details can be found on the manual page for `madvise` [4].

3.4 Recommended usage

As the `readahead` system call is binding, application developers should use it with care, and prefer `fadvise` and `madvise` instead.

When multiple parts of a system try to be smart and consume resources while being oblivious to the others, this often hurts overall performance. After all, resource management is the kernel's job. It can be best to let it decide what to do with the hints it receives from multiple sources, balancing the resource needs they imply.

4 Implementations in GNU/Linux distributions

4.1 Ubuntu

Readahead utilities are released through the `readahead` package. The following description is based on Ubuntu 6.10 (Edgy).

Reading ahead is started early in the system startup by the `/etc/init.d/readahead` init script. This script mainly calls the `/sbin/readahead-list` executable, taking as input the `/etc/readahead/boot` file, which contains the list of files to read ahead, one per line.

`readahead-list` is of course started as a daemon, to proceed as a parallel thread while other init scripts run. `readahead-list` doesn't just readahead each specified file one by one, it also *orders* them first.

Ordering files is an attempt to read them in the most efficient way, minimizing costly disk seeks. To order two files, their device numbers are first compared. When their device numbers are identical, this means that they belong to the same partition. The numbers of their first block are then compared, and if they are identical, their inode numbers are eventually compared.

The `readahead-list` package carries another utility: `readahead-watch`. It is used to create or update the list of files to readahead by watching which files are accessed during system startup.

`readahead-watch` is called from `/etc/init.d/readahead` when the `profile` parameter is given in the kernel command line. It starts watching for all files that are accessed, using the `inotify` [8] system call. This is a non trivial task, as `inotify` watches have to

be registered for each directory (including subdirectories) in the system.

readahead-watch eventually gets stopped by the /etc/init.d/stop-readahead script, at the very end of system startup. It intercepts this signal and creates the /etc/readahead/boot file.

For the reader's best convenience, C source code for these two utilities and a copy of /etc/readahead/boot can be found on http://free-electrons.com/pub/readahead/ubuntu/6.10/.

4.2 Fedora Core

Readahead utilities are released through the readahead package. The following description is based on Fedora Core 6.

The readahead executable is /usr/sbin/readahead. Its interface and implementation are similar. It also sorts files in order to minimize disk seeks, with more sophisticated optimizations for the ext2 and ext3 filesystems.

A difference with Ubuntu is that there are two readahead init scripts. The first one is /etc/init.d/readahead_early, which is one of the first scripts to be called. It preloads files listed in /etc/readahead.d/default.early, corresponding to libraries, executables, and files used by services started by init scripts. The second script, /etc/init.d/readahead_later, is one of the last executed scripts. It uses /etc/readahead.d/default.later, which mainly corresponds to files used by the graphical desktop and user applications in general.

Another difference with Ubuntu is that the above lists of files are constant and are not automatically generated from application behaviour. They are just shipped in the package. However, the readahead-check utility (available in package sources) can be used to generate these files from templates and check for non existing files.

Once more, the readahead.c source code and a few noteworthy files can be found on http://free-electrons.com/pub/readahead/fedora-core/6/.

4.3 Benchmarks

The below benchmarks compare boot time with and without readahead on Ubuntu Edgy (Linux 2.6.17-11, with all updates as of Apr. 12, 2007), and on Fedora Core 6 (2.6.18-1.2798.fc6, without any updates).

Boot time was measured by inserting an init script which just copies /proc/uptime to a file. This script was made the very last one to be executed.

/proc/uptime contains two figures: the raw uptime in seconds, and the amount of time spent in the idle loop, meaning the CPU was waiting for I/O before being able to do anything else.

Disabling readahead was done by renaming the /sbin/readahead-list (Ubuntu) or /usr/sbin/readahead programs, so that readahead init scripts couldn't find them any more and exited at the very beginning.

The Fedora Core 6 results are surprising. An explanation is that readahead file lists do not only include files involved in system startup, but also files needed to start the desktop and its applications. Fedora Core readahead is thus meant to reduce the execution time of programs like Firefox or Evolution!

	boot time	idle time
Ubuntu Edgy without readahead	average: 48.368 s std deviation: 0.153	average: 29.070 s std deviation: 0.281
Ubuntu Edgy with readahead	average: 39.942 s (-17.4 %) std deviation: 1.296	average: 22.3 s (-23.3 %) std deviation: 0.271
Fedora Core 6 without readahead	average: 50.422 s std deviation: 0.496	average: 28.302 s std deviation: 0.374
Fedora Core 6 with readahead	average: 59.858 s (+18.7 %) std deviation: 0.552	average: 35.446 (+20.2 %) std deviation: 0.312

Table 1: Readahead benchmarks on Ubuntu Edgy and Fedora Core 6

As a consequence, Fedora Core is reading ahead much more files than needed (even if we disable the `readahead-later` step) and it reaches the login screen later than if readahead was not used. The eventual benefits in the time to run applications should still be real. However, they are more difficult to measure.

4.4 Shortcomings

The readahead implementations that we have just covered are fairly simple, but not perfect though.

4.4.1 Reading entire files

A first limitation is that these implementations always preload *entire files*, while the `readahead` system call allows to fetch only specific sections in files.

It's true that it can make sense to assume that plain data files used in system startup are often read in their entirety. However, this is not true at all with executables and shared libraries, for which each page is loaded only when it is needed. This mechanism is called *demand paging*. When a program jumps to a section of its address space which is not in RAM yet, a page fault is raised by the MMU, and the kernel loads the corresponding page from disk.

Using the `top` or `ps` commands, you can check that the actual RAM usage of processes (`RSS` or `RES`) is much smaller than the size of their virtual address space (`VSZ` or `VIRT`).

So, it is a waste of I/O, time, and RAM to load pages in executables and shared libraries which will not be used anyway. However, as we will see in the next section, demand paging is not trivial to trace from userspace.

4.4.2 Reading ahead too late

Another limitation comes from reading ahead all files in a row, even the ones which are needed at the very end of system startup.

We've seen that files are preloaded according to their location on the disk, and not according to when they are used in system startup. Hence, it could happen that a file needed by a startup script is *accessed before* it is preloaded by the readahead thread.

5 Implementing readahead in embedded systems

5.1 Embedded systems requirements

Embedded systems have specific features and requirements which make desktop implementations not completely appropriate for systems with limited resources.

The main constraint, as explained before, is that free RAM can be scarce. It is no longer appropriate to preload all the files in a row, because some of the read ahead pages are likely to be reclaimed before being used. As a consequence, a requirement is to readahead files just a little while before they are used.

Therefore, files should be preloaded according to the order in which they are accessed. Moreover, most embedded systems use flash instead of disk storage. There is no disk seek cost accessing random blocks on storage. Ordering files by disk location is futile.

Still because of the shortness of free RAM, is it also a stronger requirement to preload only the portions of the files which are actually accessed during system startup.

Last but not least, embedded systems also require simple solutions which can translate in lightweight programs and in low cpu usage.

5.2 Existing implementations

Of course, it is possible to reuse code from readahead utilities found in GNU/Linux distributions, to read ahead a specific list of files.

Another solution is to use the `readahead` applet that we added to the Busybox toolset (`http://busybox.net`), which is used in most embedded systems. Thanks to this applet, developers can easily add readahead commands to their startup scripts, without having to compile a standalone tool.

5.3 Implementation constraints and plans

Updates, code, benchmarks, and documentation will be available through our readahead project page [2].

5.3.1 Identifying file access patterns

It is easy to identify files which are accessed during startup, either by using `inotify` or by checking the `atime` attribute of files (last access time, when this feature is not disabled at mount time). However, it is much more difficult to trace which sections are accessed in a given file.

When the file is just accessed, not executed, it is still possible to trace the `open`, `read`, `seek`, and `close` system calls and deduce which parts of each file are accessed. However, this is difficult to implement.[2]

Anyway, when the file is executed (in the case of a program or a shared library), there doesn't seem to be any userspace interface to keep track of accessed file blocks. It is because demand paging is completely transparent to processes.

That's why we started to implement a kernel patch to log all file reads (at the moment by

[2]Even tracing these system calls is difficult. System call tracing is usually done on a process and its children with the `strace` command or with the `ptrace` system call that it uses. The problem is that `ptrace` cannot be used for the `init` process, which would have allowed tracing on all running processes at once.

Another, probably simpler solution would be to use *C library interposers*, wrappers around the C library functions used to execute system calls.

tracing calls to the `vfs_read` function), and demand paging activity (by getting information from the `filemap_nopage` function). This patch also tracks `exec` system calls, by watching the `open_exec` function, for reasons that we will explain in the next section.

This patch logs the following pieces of information for each accessed file:

- inode number,
- device major and minor numbers,
- offset,
- number of bytes read.

Code and more details can be found on our project page [2].

At the time of this writing, this patch is just meant to assess the usefulness of reading ahead only the used sections in a file. If this proves to be profitable, a clean, long term solution will be investigated with the Linux kernel development community.

5.3.2 Postprocessing the file access dump

We are developing a Python script to postprocess file access information dumped from the kernel.

The main need is to translate inode and device numbers into file paths, as the kernel doesn't know about file names.

This is done by identifying the filesystem the inode belong to thanks to major and minor number information. Then, each filesystem containing one of our files is exhaustively traversed to build a lookup table allowing to find a file path for a given inode.

Of course, this can be very costly, but neither data gathering nor this postprocessing is meant to be run on a production system. This will only be done once during development.

5.3.3 Improving readahead in GNU/Linux distributions

Our first experiment will be to make minor changes to the utilities used in GNU/Linux distributions, so that they can process files lists also specifying which parts to read ahead in each file.

5.3.4 Towards a generic and efficient implementation

While preloading the right file sections is easy once file access information is available, another requirement is to perform readahead at the right time in the execution flow. As explained before, reading ahead mustn't happen too early, and mustn't happen too late either.

Once more, the challenge is to predict the future by using knowledge about the past.

A very basic approach would be to collect time information together with file access data. However, such information wouldn't be very useful to trigger readahead at the right time, as reading ahead accelerates time. Furthermore, as processes spend less time waiting for I/O, the exact ordering of process execution can be altered.

Thus, what is needed is a way to follow the progress of system startup, and to match the actual events with recorded ones.

A simple idea is to use `inotify` to get access notifications for executables involved in system startup, and match these notifications with recorded `exec` calls.

This would be quite easy to implement, as this would just involve a list of files, without having to register recursive directory based notifications.

As shown in the example in Figure 2, our idea is to manage *readahead windows* of a given data size. In this example, when `event11` is recognized, we create a new readahead window starting from this event, corresponding to 1 MB of recorded disk access starting from this event.

Actually, we would only need to start new readahead I/O from the end of the previous window to the end of the new one. This assumes that the window size is large enough to extend beyond the next event. Otherwise, if readahead windows didn't overlap, there would be parts of the execution flow with no readahead at all.

Within a given window, before firing readahead I/O, we would of course need to remove any duplicate read operations, as well as to merge consecutive ones into single larger ones.

Here are the advantages of this approach:

- Possibility to readahead the same blocks multiple times in the execution flow. This covers the possibility that these blocks are no longer in page cache.
- For each specific system, possibility to tune the window size according to the best achieved results.
- The window size could even be dynamically increased, to make sure it goes beyond the next recorded event.
- If window size is large enough, we expect it to compensate for actual changes in the order of events.

5.3.5 Open issues

Several issues have not been addressed yet in this project.

In particular, we would need a methodology to support package updates in standard distributions. Would file access data harvesting be run again whenever a package involved in system startup is updated? Or should each package carry its own readahead information, requiring a more complex package development process?

5.4 Conclusion

Though the proposed ideas haven't been fully implemented and benchmarked yet, we have identified promising opportunities to reduce system startup time, in a way that both meets the requirements of desktop and embedded Linux systems.

If you are interested in this topic, stay tuned on the project page [2], join the presentation at OLS 2007, discover the first benchmarks on embedded and desktop systems, and share your experience and ideas on accelerating the course of Time.

References

[1] Jonathan Corbet. Lwn article: Adaptive file readahead. `http://lwn.net/Articles/155510/`, October 2005.

[2] Free Electrons. Advanced readahead project. `http://free-electrons.com/community/tools/readahead/`.

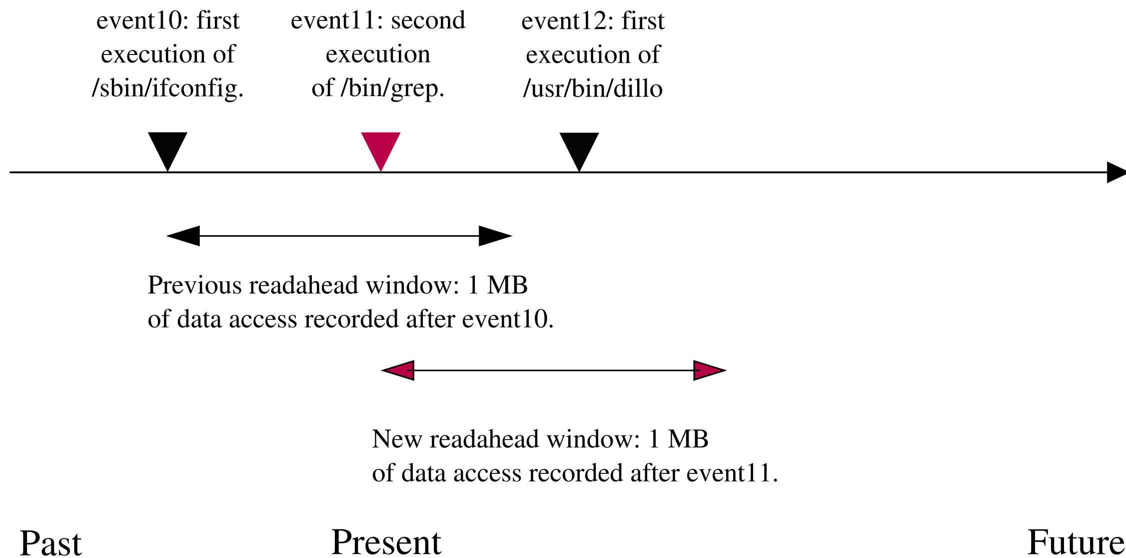

Figure 2: Proposed readahead implementation

[3] WU Fenguang. Linux kernel mailing list: Adaptive readahead v16 benchmarks. `http://lkml.org/lkml/2006/11/25/7`, November 2006.

[4] Linux Manual Pages. madvise(2) - linux man page. `http://www.die.net/doc/linux/man/man2/madvise.2.html`.

[5] Linux Manual Pages. posix_fadvise(2) - linux man page. `http://www.die.net/doc/linux/man/man2/posix_fadvise.2.html`.

[6] Linux Manual Pages. readahead(2) - linux man page. `http://www.die.net/doc/linux/man/man2/readahead.2.html`.

[7] Ram Pai, Badari Pulavarty, and Mingming Cao. Linux 2.6 performance improvement through readahead optimization. In *Ottawa Linux Symposium (OLS)*, 2004. `http://www.linuxsymposium.org/proceedings/reprints/Reprint-Pai-OLS2004.pdf`.

[8] Wikipedia. inotify. `http://en.wikipedia.org/wiki/Inotify`.