Readahead

Time Travel Techniques

For Desktop and Embedded Systems

Michael Opdenacker
Bootlin
https://bootlin.com

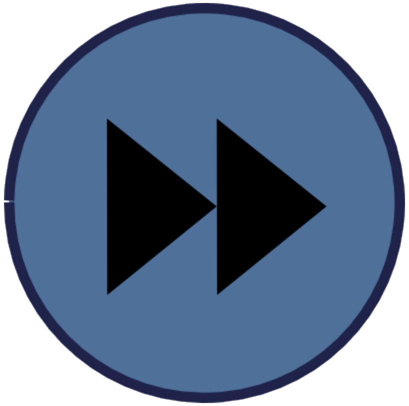
Created with OpenOffice.org 2.2

Question

Ever dreamed of a device to accelerate the course of Time?

To watch the end of the “Lost” TV show
before you retire

To build faster CPUs moving electrons at Warp 9.99



To configure sendmail in 2 minutes

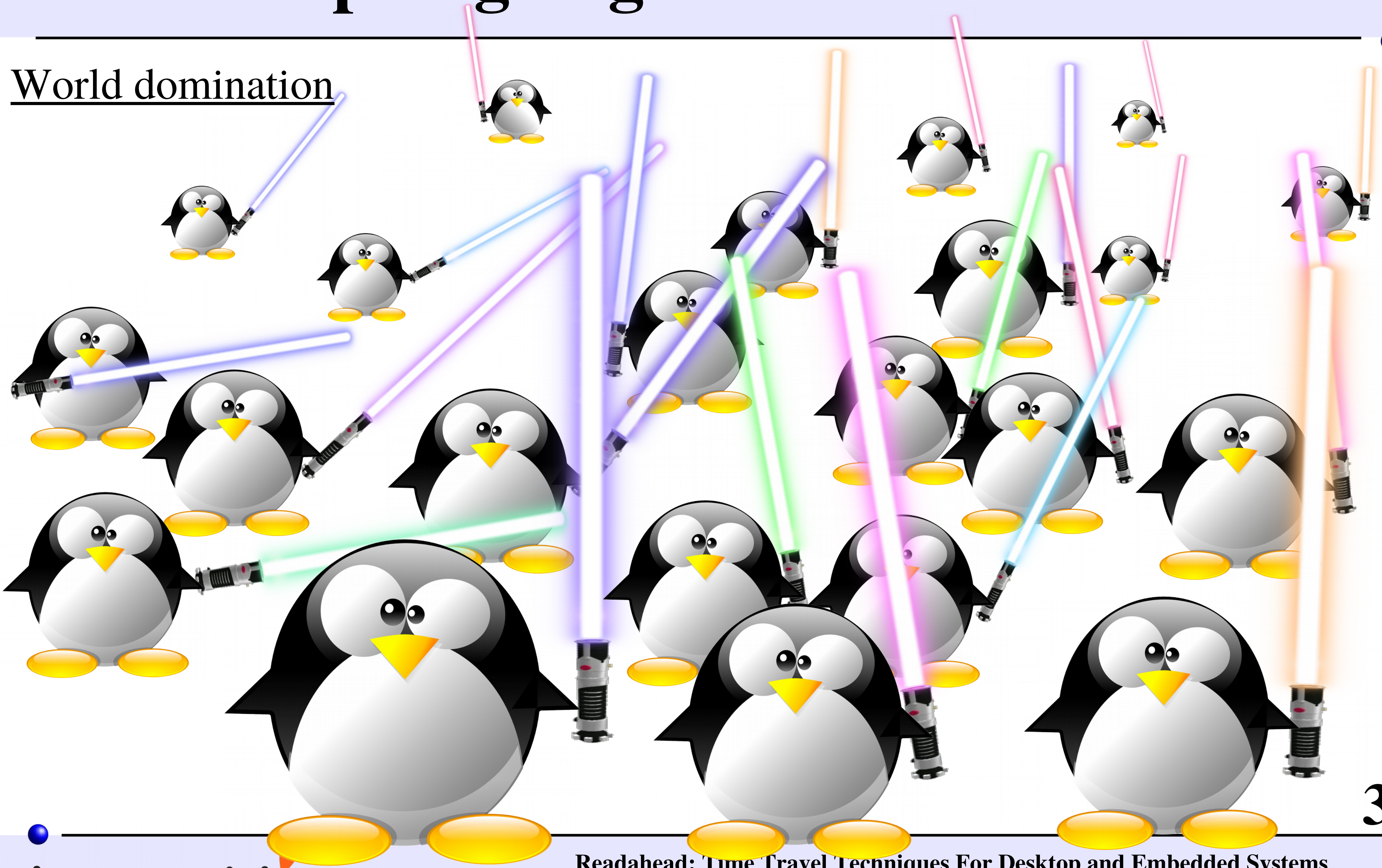
To remove all bugs from your Perl scripts
before you stop understanding your own code.

To pass US immigration in just 2 hours 😉

Such a device exists: it's called **readahead**.

Oops... going forward too fast

World domination



Contents

Introduction

Readahead in kernel space

Readahead in user space

Trying to improve user-space implementations

- Applications for embedded systems

Readahead techniques

Introduction to readahead

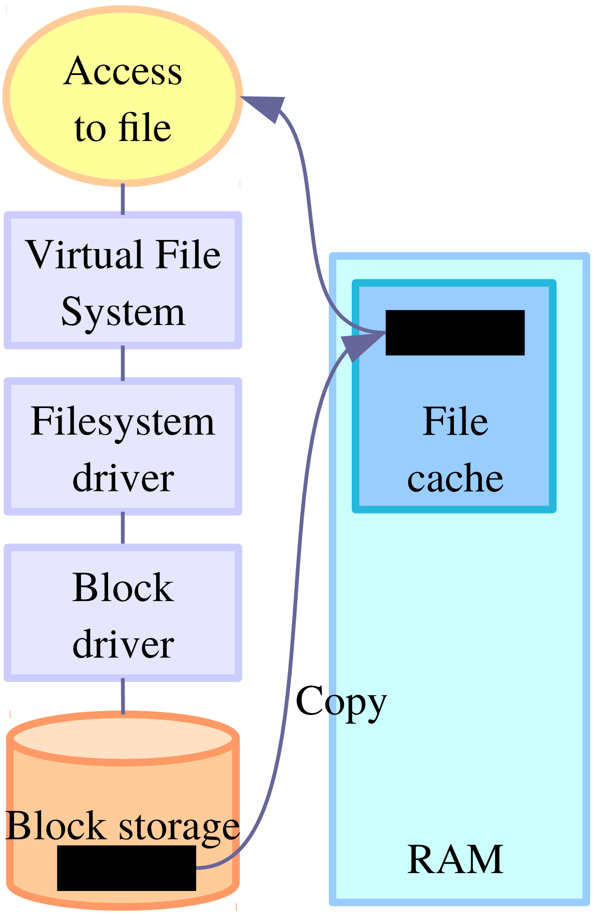
The page cache

Linux copies each file into the *file cache* before accessing it.

Useful to speed up subsequent accesses to the same file. No more I/O wait.

Yields best results when plenty of free RAM is available.

Otherwise, Linux only keeps the most recently accessed pages.



Reading ahead

Idea: to accelerate access to files

Preload files in the file cache (entirely or in parts)
before they are actually used.

When each file or program is accessed,
there is no more I/O wait. Rescheduling not needed.
Much better performance.

Best done when spare I/O resources are available,
typically when tasks keep the processor busy.

But this requires the ability to predict the future!

Possible predictions

Fortunately, our systems are predictable in some cases

File blocks being accessed in a sequential way:
the next blocks are very likely to be read too.

System startup:
the same executables and data files are read in the same order.

Application startup:
the same parts of program text, shared libraries,
resource or input files are accessed.

Benefits of reading ahead

Reading ahead *can* bring the following benefits:

Reduced system and application startup time

Improved disk throughput:

- more requests fed to the I/O scheduler, which can do a better job at reordering requests to minimize disk head moves.

Better utilization of CPU resources:

- less I/O wait, so less rescheduling and context switching.

Readahead techniques

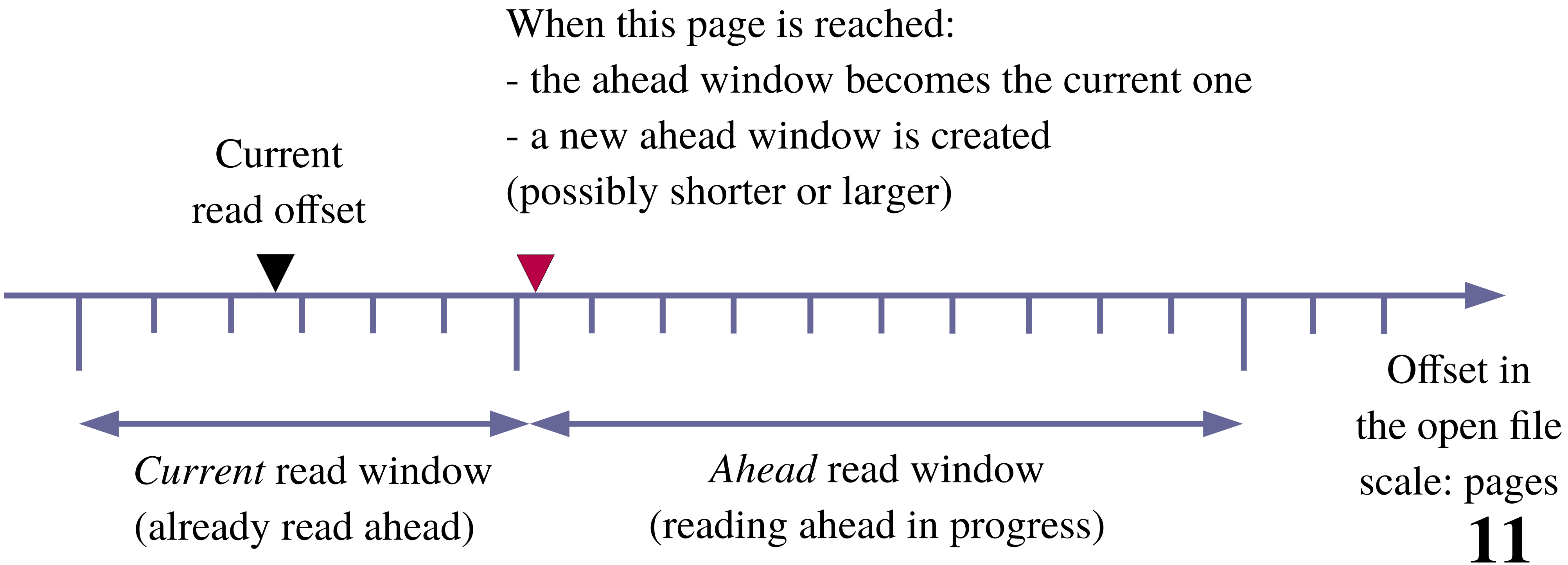
Kernel space readahead

Readahead in the standard kernel (1)

Status in Linux 2.6.20

Implemented with 2 read windows, when sequential file read is detected.

If sequential reading continues, the process never waits for I/O.



When this page is reached:
- the ahead window becomes the current one
- a new ahead window is created
(possibly shorter or larger)

Current read offset

Current read window
(already read ahead)

Ahead read window
(reading ahead in progress)

Offset in the open file
scale: pages

Readahead in the standard kernel (2)

The kernel automatically checks how effective readahead is

If page misses are encountered:

some of the pages were reclaimed before being accessed by the process.

Action: reduce the window size, down to `VM_MIN_READAHEAD` (16 KB)

Otherwise:

Increase the window size, up to `VM_MAX_READAHEAD` (128 KB)

If page cache hits are encountered during readahead:

the file is partly or fully in page cache.

Readahead is then useless. Disable it.

Implementation details found on `mm/readahead.c` in kernel sources.

Adaptive readahead patchset

Proposed by Fengguang Wu
since September 2005.
Posted in the Linux kernel mailing list.

Added features:

Readahead window
which can grow up to 1 MB

Parallel, interleaved sequential scans
on 1 file

Sequential reads across
file open / close

Mixed sequential / random access

Sparse / skimming sequential read

Backward sequential reading

Delaying readahead if the disk is
spinned down in laptop mode.

The last benchmarks show access time
improvements in most cases.

Also released a simpler patchset:
“on-demand readahead”.

Fengguang Wu is here in Ottawa!
Don't miss his presentation
on Friday (11:00 - 11:45).

Readahead techniques

User space readahead interface

Need for a userspace interface

Though the kernel can do a great job predicting file access from current and recent application behavior, there are things a general purpose kernel cannot predict.

Fortunately, system developers can use system calls to let the kernel know about their own predictions:

`readahead`: load file blocks into the file cache

`fadvise`: announce file access patterns

`madvise`: announce memory access patterns

readahead system call

Make the kernel load count bytes in the file, starting from offset.

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

I/O is performed in whole pages.

- `offset` and `offset+count` are rounded to page boundaries

The function blocks until all data have been read.

- Typically called in a parallel thread.

fadvise system call

Several variants:

`posix_fadvise`, `fadvise64`, `fadvise64_64`.

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

Here's how the kernel interprets the advice setting:

- `POSIX_FADV_NORMAL`: use default readahead window size
- `POSIX_FADV_SEQUENTIAL`: double window size
- `POSIX_FADV_RANDOM`: disable readahead
- `POSIX_FADV_WILLNEED`: reads the specified region in file cache
- `POSIX_FADV_NOREUSE`: same, but data just used once
- `POSIX_FADV_DONTNEED`: free the corresponding cached pages.

Note that the kernel is free to ignore this advise.

madvise system call

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

Similar to `fadvise`, but corresponding to the address space of a process.

Using the userspace interface

The readahead system call should be used with care!

Caution: the readahead system call is binding. It forces the kernel to load pages in RAM. There shouldn't be multiple parts of the system trying to be smart like this.

Best to use the non binding interfaces: `fadvise` and `madvise`, and let the kernel arbitrate between resource requests.

Readahead techniques

Implementations in GNU/Linux distributions

Ubuntu

Observed on Edgy 6.10

`/etc/readahead/boot`:

list of files which are accessed during the boot process.

`/sbin/readahead-list`:

Orders these files according to their disk order (first block at least), and reads them ahead. No more runs as a background job.

`/sbin/readahead-watch`:

Uses `inotify` to get the list of files accessed during startup. Stopped by `/etc/init.d/stop-readahead`.

Fedora Core 6

`/usr/sbin/readahead`

Similar implementation and interface. It also orders files to reduce disk seeks, but with special optimizations for ext2 and ext3.

`/etc/readahead.d/default.early`

Files accessed by init scripts.

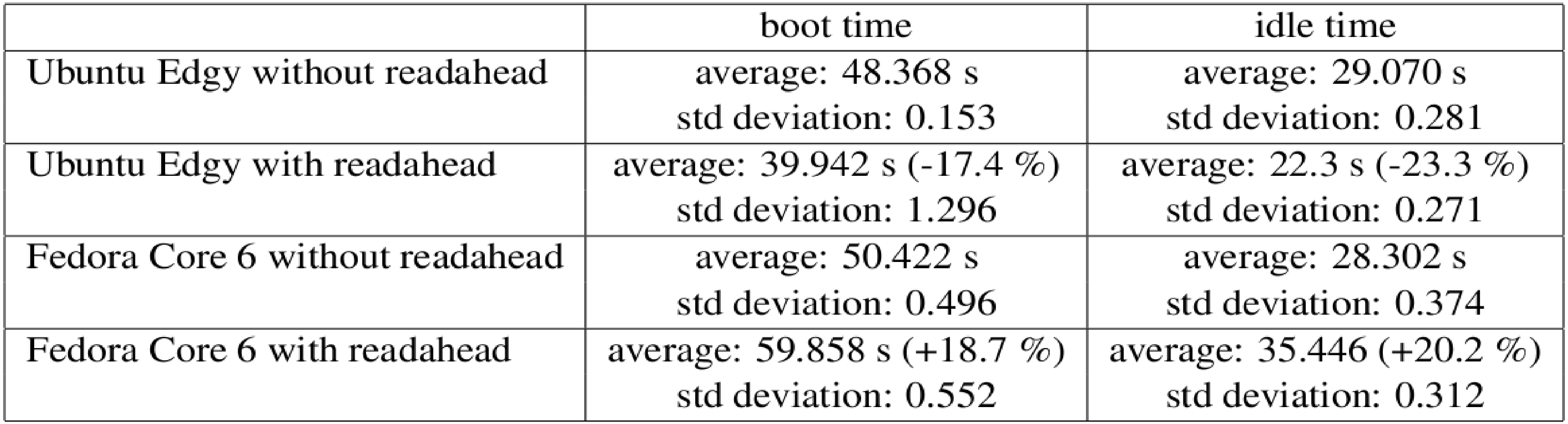
`/etc/readahead.d/default.later`

Files used by the graphical desktop and user applications.

No utility is given to update these files, which are generated from templates.

Benchmarks

Measuring time in the last init script



	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

Comments:

Ubuntu almost achieves a 20% boot time reduction

Fedora Core: more difficult to measure, because it also tries to speed up application startup.

Shortcomings

Things that could be improved

Reading entire files: too much in some cases.

- Demand paging: the kernel only loads the pages of programs and shared libraries which are actually accessed.
- This wastes I/O time and RAM.

Not taking into account file access order.

- It could be that a file is read ahead after it is actually used.

Readahead techniques

Implementing readahead in embedded systems

Embedded system requirements

RAM is scarce and perhaps slow to access

Shouldn't readahead file pages which are not needed.

Shouldn't load pages too much in advance.
They could be reclaimed even before they are used.

Slow CPU and limited storage

Need for simple and fast implementations, typically in C.

BusyBox readahead applet

Available on `http://busybox.net`

Usage:

```
readahead <files>
```

Make it easy to add manual readahead to your startup scripts, in particular when your embedded system already uses BusyBox!

Tracing accessed file blocks

Not trivial to do. Apparently no interface do do it.

Can't be done with `inotify`. We just now about accessed files.

Intercepting `read` and `exec` system calls, through C library interposers. However, no information on demand paging for libraries, executables and mapped files.

A solution: tracing kernel filesystem access functions

- `vfs_read`: reading from files
- `filemap_nopage`: demand paging activity
- `open_exec`: tracing all executions (explained later)

Any other idea?

Tracing implementation

Could have been done easily with SystemTap.

But SystemTap should be complicated to use in embedded systems (user space tools to install, `arm` and `mips` not fully supported).

Quick and dirty solution for desktop and embedded systems:

a kernel patch dumping information on the kernel log.

Seems to be too much information for `klogd`.

(looses the first messages).

Best to dump the messages on a serial console.

The patch is available on the project page.

Dumped raw data

```
...
RAINFO exec 621770 3 3
RAINFO read 621770 3 3 0 128
RAINFO nopage 924793 3 3 8036352 4096
RAINFO nopage 585470 3 3 1933312 4096
RAINFO nopage 456803 3 3 1835008 4096
RAINFO nopage 456803 3 3 8278016 4096
RAINFO nopage 456803 3 3 1691648 4096
RAINFO nopage 456803 3 3 483328 4096
RAINFO nopage 456803 3 3 786432 4096
```

event inode offset size

device major

device minor

readahead-blocks-digest

Our Python script to digest raw data:

Turns inodes into file paths

(done by recording inode numbers for each file in the filesystem)

Merges consecutive blocks

Outputs statistics

Output file statistics

Ubuntu Edgy 6.10 excerpt

```
/sbin/udevd : 53248 / 55232 (96%)
/lib/libselinux.so.1 : 42460 / 75228 (56%)
/lib/libsepol.so.1 : 19232 / 203552 (9%)
/sbin/dhclient3 : 256956 / 433084 (59%)
/lib/tls/i686/cmov/libnss_nis-2.4.so : 17936 / 34320 (52%)
/lib/tls/i686/cmov/libnsl-2.4.so : 26336 / 75488 (34%)
/bin/sed : 39208 / 39208 (100%)
/etc/init.d/checkroot.sh : 9875 / 9875 (100%)
/lib/init/vars.sh : 113 / 113 (100%)
/etc/default/rcS : 261 / 261 (100%)
/lib/lsb/init-functions : 7744 / 7744 (100%)
/etc/lsb-base-logging.sh : 3770 / 3770 (100%)
...
Total file size : 98477782
Total read size : 46655799(47%)
```

Output block information

Ubuntu Edgy 6.10 excerpt

```
/sbin/udevd 0 53248
/lib/libselinux.so.1 0 16384 32768 40960 45056 49152 57344
65536 69632 75228
/lib/libsepol.so.1 0 12288 184320 188416 200704 203552
/sbin/dhclient3 0 16384 20480 57344 90112 122880 135168
155648 172032 176128 184320 196608 204800 208896 212992
225280 229376 233472 282624 294912 299008 319488 327680
348160 352256 356352 368640 376832 380928 389120 393216
433084
/lib/tls/i686/cmov/libnss_nis-2.4.so 0 8192 16384 20480
28672 34320
/lib/tls/i686/cmov/libnsl-2.4.so 0 16384 65536 75488
/bin/sed 0 39208
/etc/init.d/checkroot.sh 0 9875
...
```

readahead-blocks program

The program running on the target system

Coded in C

Based on the `readahead-list` code found in Ubuntu (from Robin Hugh Johnson and Erich Schubert)

Takes a block information file as input.

Reads ahead file blocks following system startup order.

Ubuntu Edgy statistics

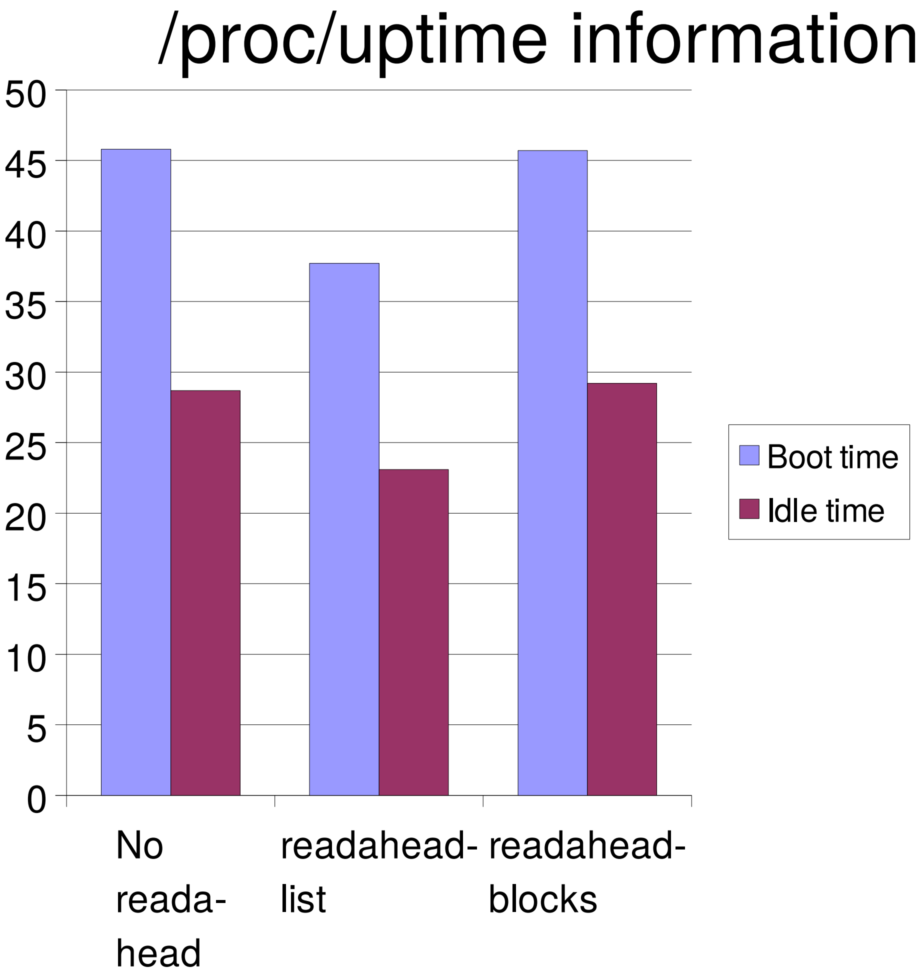
Statistics:

Total file size: 93.9 MB

Read file size: 44.5 MB

Only 47% of Ubuntu readahead I/O is useful!

Ubuntu Edgy first results



Disappointing results!

I did check that `readahead-blocks` is actually doing something.

`readahead-blocks` reads less data, but the benefits are canceled by more disk head moves.

Will soon update `readahead-blocks-digest` to write blocks in on-disk order.
It will beat `readahead-list` for sure!

New victim: HP iPAQ h2200

Perfectly supported by Linux
http://handhelds.org/projects/h2200

Running the Familiar 0.8.4 distribution
(not at all optimized for boot time: easy victim?)

CPU: pxa255 (400 MHz)

RAM: 64 MB

Booting from Compact Flash (IDE):
Sandisk Ultra II

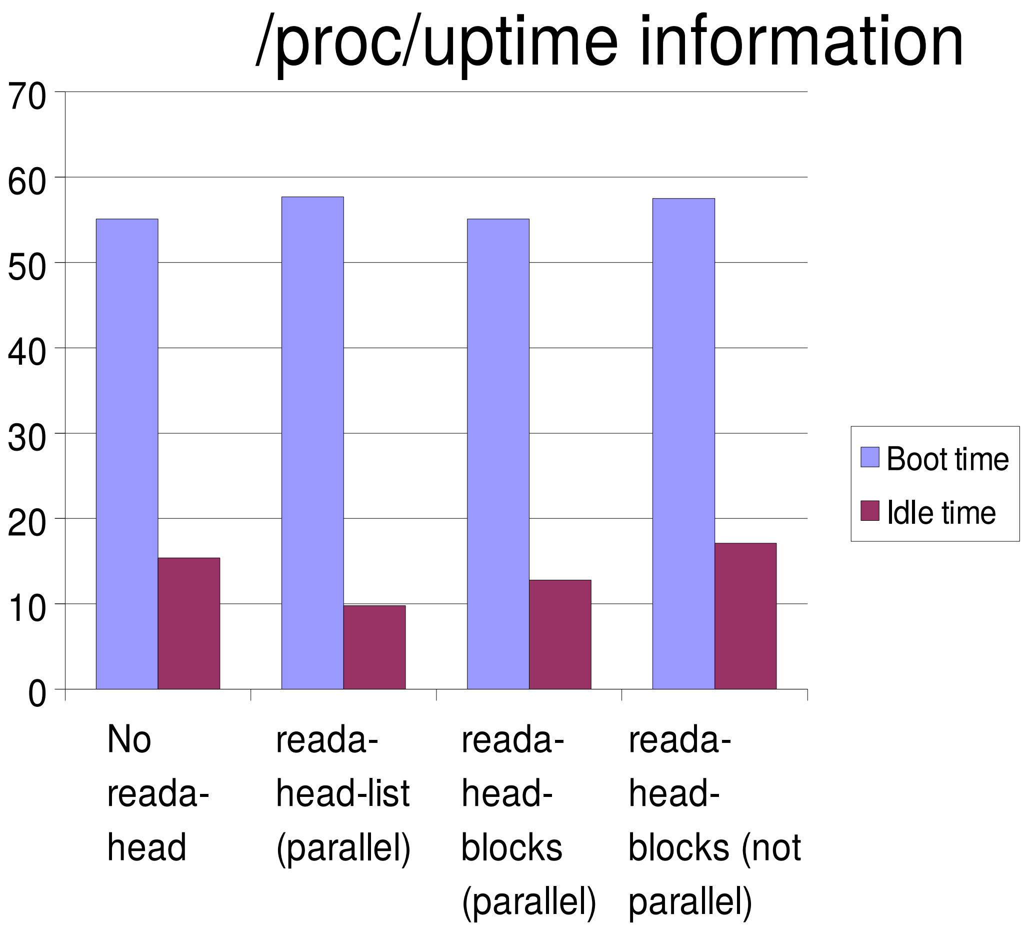
Familiar 0.8.4 statistics

Statistics:

Total file size: 20.4 MB

Read file size: 9.3 MB (45 %)

Familiar 0.8.4 results



Disappointing results too!

I did check that `readahead-blocks` is actually doing something.

No disk head moves: `readahead-list` now has a handicap. Kernel readahead also has less handicap.

Need to try with slower Compact Flash storage. Could expect more significant gains.

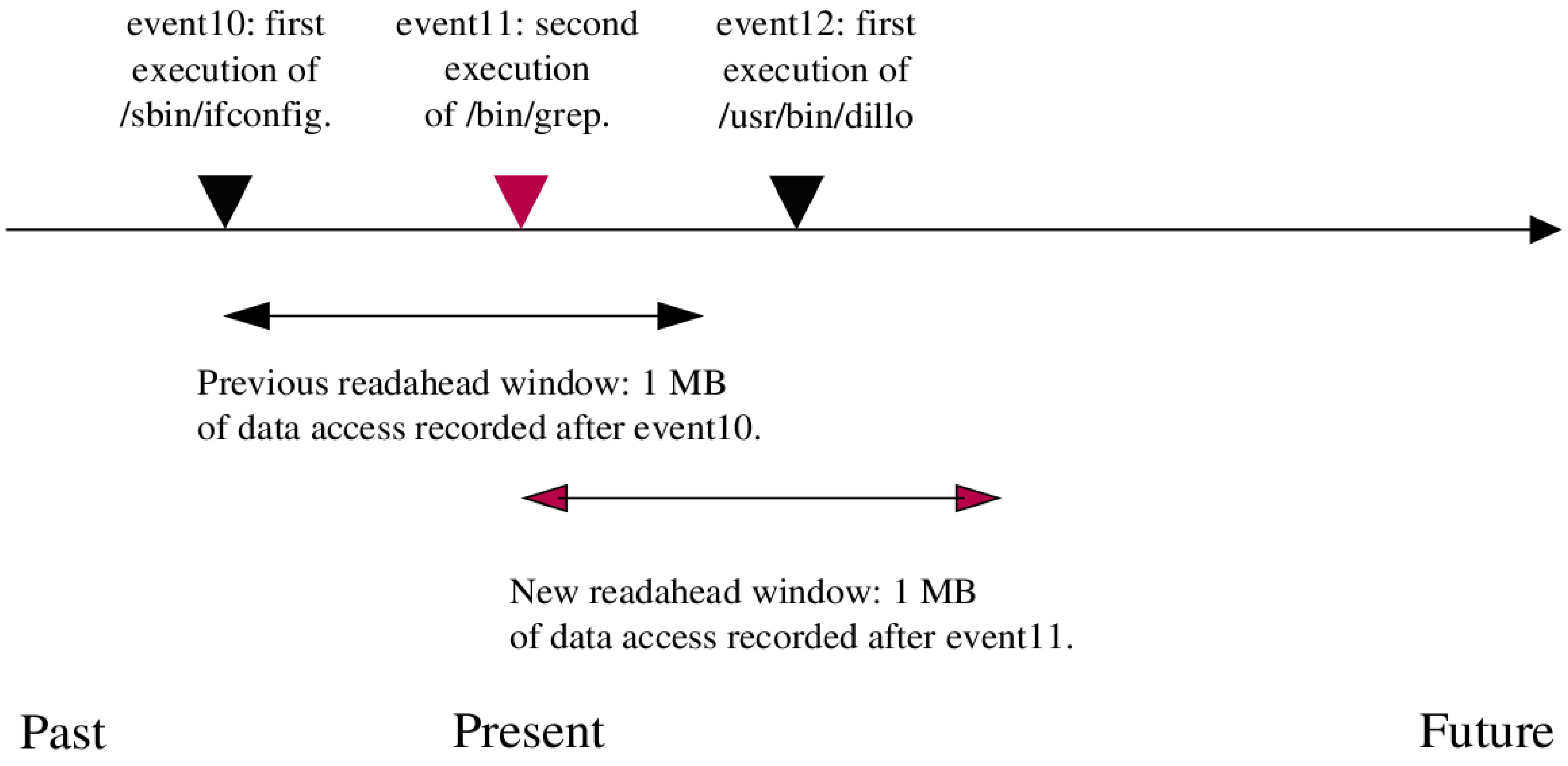
The CPU latency / storage latency ratio is greater too. Less idle time and reduced benefit outlook.

RAM access is slower too.

Event triggered read-ahead

Proposed for embedded systems with very little RAM.

Implemented in the future 😉



event10: first execution of /sbin/ifconfig.

event11: second execution of /bin/grep.

event12: first execution of /usr/bin/dillo

Previous readahead window: 1 MB of data access recorded after event10.

New readahead window: 1 MB of data access recorded after event11.

Lessons learned (1)

`readahead-list`: reads way too much
(because of demand paging)

Hard disk storage: minimizing disk head seeks saves much more time than reading only the needed blocks.

Embedded systems with flash storage:
benefits more difficult to achieve, unless flash access is very slow.
Slower CPU and RAM access could be bottlenecks.

Lessons learned (2)

Decision criterion:

- Measure idle time before investigating readahead solutions.
- A low idle time / boot time ratio means less to win.

Better to be humble, and let the kernel do its job.

- However, feed it with as much advise as possible, and it will make the best decisions, without sacrificing performance.

Future work

Hard disk storage: read blocks by on-disk order.

This will be better than today's readahead-list implementation.

Try with `fadvise` instead of `readahead`

Embedded systems: confirm greater benefits on slow flash.

Embedded systems with very little RAM:

implement the event-triggered readahead window.

Other ways of accelerating time

Disable console output

Disable delay loop calibration

Starting system services in parallel

Prelinking

Reduce forking in shells.

Use the standalone shell option of Busybox

Faster filesystems (SquashFS!)

See http://elinux.org/Boot_Time for details and more!

References

Our readahead project page:

https://bootlin.com/community/tools/readahead/

Even more interesting in the future!

More details and references on our paper:

https://bootlin.com/pub/readahead/doc/ols2007-readahead.pdf

Linux Readahead: Less Tricks For More

Fengguang Wu, Room Emperor, 11h00 - 11h45 on Friday

Thank you!

Questions or suggestions?