Linux solution for prefetching necessary data during application and system startup

Krzysztof Lichota
lichota@mimuw.edu.pl

What is prefetching and why it is needed?

The problem

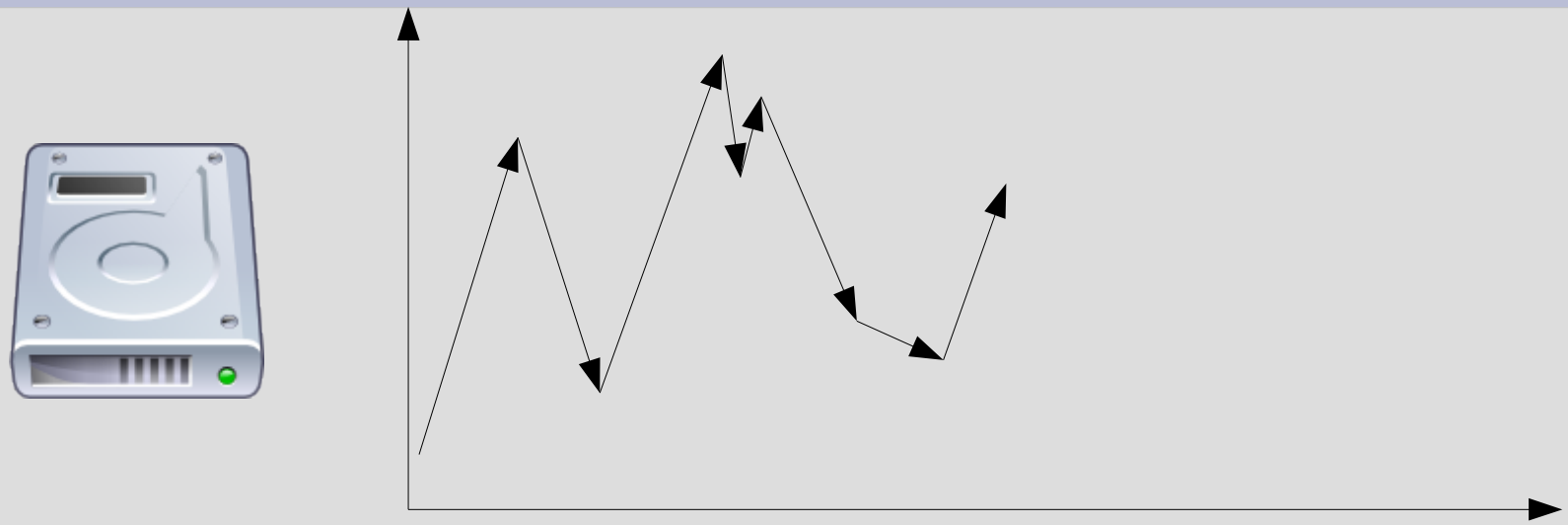
In modern computers

- CPUs – fast
- Memory – fast
- Disk – slooooow (by orders of magnitude)
  - Disk access: ~8 ms = $8*10^{-3}$
  - Memory access: ~8 ns = $8*10^{-9}$
  - Difference: 10^{6} = 1 000 000 times

Application start – demand paging

- Modern operating systems introduced paging on demand
- Great idea, but...
  - Load one page from executable file (8 ms)
  - Execute (0.1 ms)
  - Need one more page – wait (8 ms)
  - Execute (0.1 ms)
  - Need next page (8 ms)
  - etc.

Scattered files



- Many scattered files cause a lot of disk seeks
- Seek time is ~proportional to distance between disk cylinders

The effect

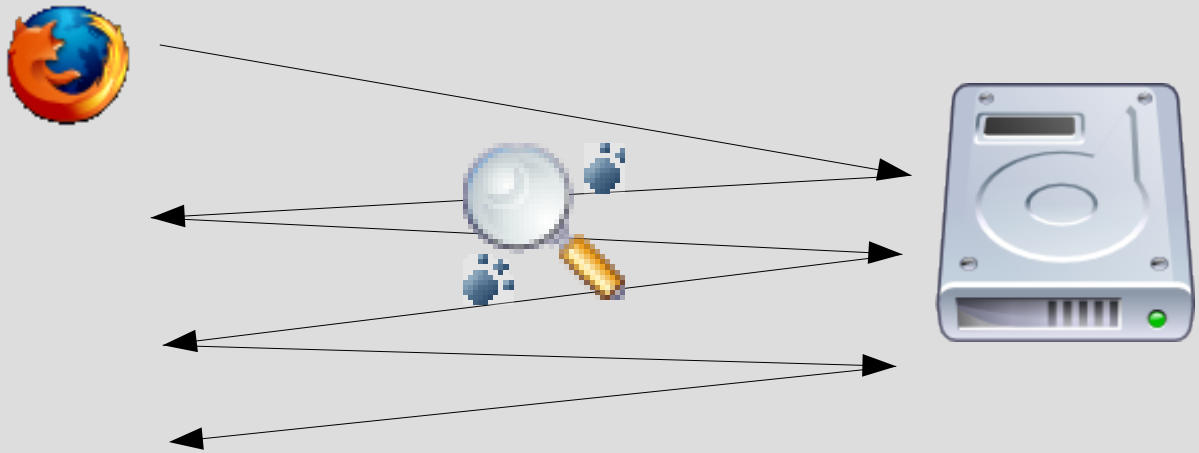
- ~15 seconds to start OpenOffice on Linux
- ~7 seconds to start Firefox
- Note not all of this is caused by disk seeks: other problems also apply (like linker problems, which hopefully have been already solved)

What can be done

- Prefetch all necessary file pages before application even requests it
- Group files in one place on disk:
  - Avoids seeks
  - Disk works better when sending large chunks of data

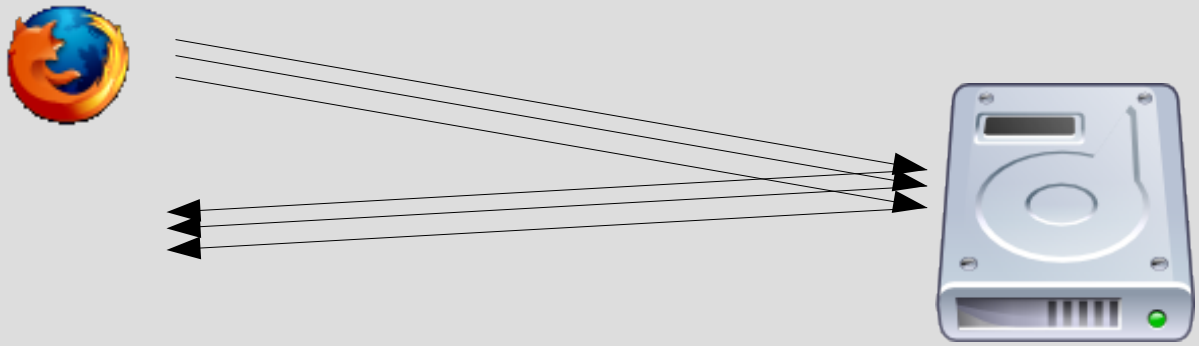
The question: how to know what to prefetch and when?

Application start analysis



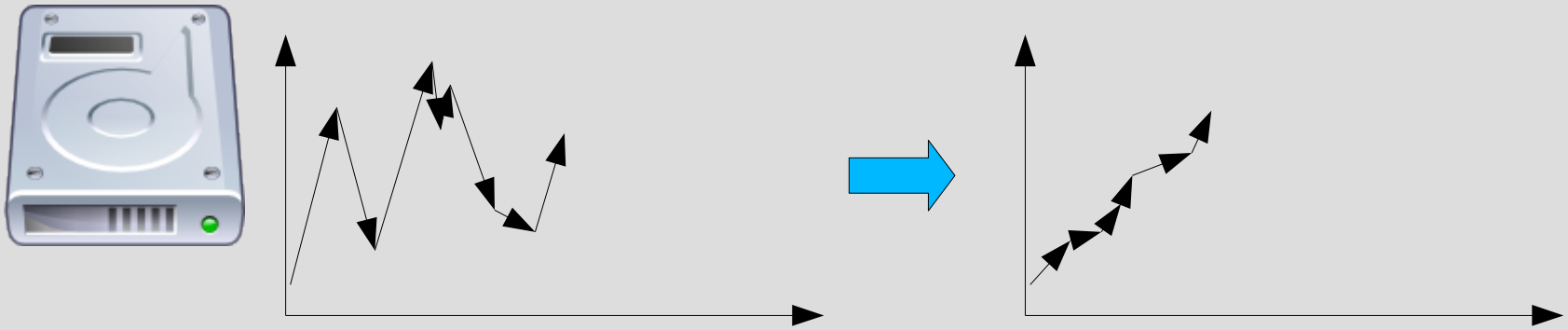
- Monitor first application start (or system boot)
- Write down which files it fetches and in which order
- Predict which files will be used next time (based on history)

Prefetch necessary files



- Prefetch files when application starts next time
- At the same time monitor if new files are used and others stop to be used

Laying out files



- Group files in one place on disk
- Order them by access order

Current state of the art

Prefetching in desktop operating systems

- Windows XP/Vista
  - analyzes applications start and system boot
  - fetches necessary files on boot and application start
  - Vista tries to predict when you will use application
  - details not known (closed source)
- Mac OS X - BootCache
- Linux – *almost nothing*

Previous attempts of prefetching

- There were several attempts to tackle prefetching problem in Linux
- None of them was completely successful
- All of them required manual intervention of user

Ubuntu boot readahead

- Consists of boot scripts which can analyze and prefetch files during boot
- User must manually run analyzing process upon boot
- Analyzing boot is done using inotify and has high overhead, so it is not suitable for use on every boot
- When analysis is done, prefetching is not performed, so user notices slowdown at boot

Ubuntu boot readahead (2)

- It works on whole files, not on only relevant parts, so it has higher memory requirements
- This causes problems on machines with less RAM and might even slow down boot on such machines
- It does not notice order of read files, files to prefetch are sorted by disk position and fetched all at once at boot
- It works purely in userspace
- Does not address application prefetching

Preload

- Developed as part of Google Summer of Code 2005
- Aimed to provide preloading of file based on statistical analysis by corellation of applications (possibly multiple) and files they use
- Uses /proc/pid/maps as source of information which files application uses
- Thus does not notice files accesses using other methods than mmap (like read())

Preload (2)

- It runs as daemon, wakes up every 20 seconds to see if files should be preloaded. It cannot react to application starting in this 20 seconds interval
- Daemon analyzes what applications are running together and fetches their files
- It might work for applications which are started during login as this is predictable
- It does not work well for applications which are started on user demand, like Firefox

Bootcache/filecache

- Developed as part of Google Summer of Code 2006
- It concentrates on kernel side of prefetching by providing facilities for faster readahead and analysis of page cache

Bootcache/filecache (2)

- It contains some interesting features:
  - Adds open-by-inode to Linux kernel which allows faster readahead (without directory lookups)
  - Contains some improvements to ioprio (I/O prioritization) to make readahead have smaller impact on currently running applications
  - Adds dumping state of file cache for processes, which is later used for checking which files to prefetch
  - It contains "poor man's defrag" to group files on disk, using "copy to directory and hardlink in previous position" trick

Bootcache/filecache (3)

- Problems:
  - It does not intercept automatically application startup, so user must manually set up prefetching and analyzing
  - Poor man's defrag is not complete defragging solution, it works only on whole files and has limited capabilities of laying out files as it relies on behaviour of old and new kernel blocks allocator. It also can create only one group of files.

Bootcache/filecache (4)

- Open-by-inode allowed for userspace is a security risk
- Files can be purged from cache before analyzer notices they were read (especially for boot analysis)
- It does not take into account order of files being read
- It uses user-level threads to do prefetching, they have to fight for processor with all others, slowing down prefetching effectiveness and using CPU for context switches

Conclusions

- **Linux needs prefetching to compete effectively with other desktop systems**
- Currently available solutions do not provide complete and automatic solution:
  - None of them is able to intercept application startup automatically, analyze its behaviour and prefetch necessary files in efficient manner
  - There is no complete defragging solution to lay out files on disk
  - None of them provides lightweight tracing facility which can be used during each boot

Prefetch implementation for Linux

Overview

- Developed during Google Summer of Code 2007
- Provides:
  - automatic application start tracing and prefetching
  - boot tracing and prefetching
  - reordering of files (highly experimental)

Overview (2)

- Consists of:
  - kernel patches which provide tracing and prefetching facilities
  - boot scripts which control kernel tracing and prefetching
  - utility to reorder files upon shutdown

Tracing and prefetching kernel facilities

Tracing

- Main problem – distinguishing disk accesses caused by prefetching and those caused by application
- Tracing just disk accesses does not work properly in such case
- Solution – check “page referenced” bit in Linux VM subsystem
- Based on filecache code to walk all pages in system

Tracing (2)

- Also notices pages released by VM subsystem, for greater resolution
- Still misses some accesses (checked using blktrace) – in investigation
- Even with missed accesses provides enough information for effective use
- Kernel part provides generic tracing facility which can be used concurrently by many facilities (currently boot tracing and application tracing)

Tracing – implementation details

- Simple buffer where trace records are added
- Trace record contains:
  - device number
  - inode number
  - start of area (in page units)
  - length of area (in page units)
- Hook in __remove_from_page_cache() which adds released pages to buffer

Tracing – implementation details (2)

- Module can request walk of all pages in system
- On first walk page referenced bits are cleared
- During next walks pages referenced are added to buffer during the walk
- Buffer is freed when all modules declare they no longer want to trace accesses
- Trace can be saved to disk using provided functions
- Time of pages walk is very small (0.002s for clearing, 0.02s for recording with 256 MB RAM)

Prefetching

- Module requests prefetching of given set of records
- Function is provided to read trace from disk
- Records are processed in order
- Devices are opened using their numbers (tricky)
- Files are opened using their inode numbers
- Cache is populated using force_page_cache_readahead()
- Possible synchronous and asynchronous prefetching mode

Application startup tracing and prefetching

Application tracing and prefetching

- Hooks into exec() call and checks if there is trace for executed application
- Application is identified as part of filename and hash of path
- If there is the trace, reads trace from file and starts prefetching (synchronous)
- If application is on tracing whitelist, starts tracing
- Schedules “end startup” handler

Application tracing and prefetching (2)

- After scheduled startup time (by default 10 seconds, configurable) startup end handler is run
- Handler finishes tracing, if it was enabled, and writes new trace to /.prefetch directory
- It also checks if application used a lot of IO during startup (using delayacct_blkio_ticks())
  - if the application reached certain threshold, it adds it to tracing whitelist
  - if it did not reach threshold, removes it from tracing whitelist

Application tracing and prefetching (3)

- Only last trace of application startup is used
- Trace is for all files accesses, not only for traced application
- This creates possibility of reading too much
- On the other hand it solves problem of prefetching files used by related applications, needed for startup
- In practice works quite well
- It might be improved by computing intersection of a few historical traces

Application startup time measurement

- OpenOffice used as metric (due to long startup time)
- Problem: erratic behaviour (high variance) – solved by averaging results over many runs
- Problem: OpenOffice contains its own prefetching tool, manually crafted – had to disable it for reliable results
- Startup time measured by loading document with macro which has written startup time to file

OpenOffice startup results

- Startup time:
  - without any prefetching: 14.38s
  - with built-in prefetching (pagein): 12.74s (1.64s difference)
  - with automatic prefetch: 11.01s alone, 11.07s with pagein
- Improvement: 3.36s (23%) to none, 1.67s (13%) to pagein

Boot tracing and prefetching

Boot tracing and prefetching

- Kernel module which provides /proc interface for boot scripts
- Boot init scripts control tracing and prefetching “phases”
- Phases:
  1. From boot from root partition to mounting all partitions
  2. From mounting all partitions till GUI is started (i.e. display manager)
  3. Since GUI is started for 60 seconds (does not detect user login yet)

Boot tracing and prefetching (2)

- Each phase has separate tracing and prefetching
- Phases determined by tests – this split gives best results
- First 2 phases use synchronous prefetching (i.e. wait until prefetching finishes before proceeding)
- GUI phase is prefetched asynchronously after 2 phase – gives best results

Boot tracing and prefetching (3)

- Each phase trace is saved into separate file
- A few historical traces for each phase are kept
- After boot is finished script computes logical sum of last 3 traces (separately for each phase) and writes it as trace used for next boot
- Boot scripts can be modified to have other phases – the interface in kernel is generic

Boot time measurement

- Problem: erratic behaviour (high variance) – solved by averaging results over many runs
- Has to watch out for periodic maintenance tasks (fsck), network discovery, etc.
- Startup time measured by starting a script as part of auto-login and recording uptime
- Simulation of changing boot process done by running OpenOffice as part of boot (before uptime is recorded)

Boot prefetching results

- Boot time with Ubuntu kernel and Ubuntu readahead: 61.21s
- Boot time with prefetch kernel and boot prefetching: 54.91s
- Improvement: 6.31s (10%)

With OpenOffice as part of boot:

- Prefetch kernel and prefetch: 65.53s
- Ubuntu kernel and readahead: 81.01s
- Improvement: 15.48s (19%)

Readahead does not adapt to changes, prefetch does

File reordering

File reordering tool

- **Highly experimental, might eat your data, do not use yet**
- Works only for ext2/3, uses libext2fs for on-disk manipulation (the same as used in e2fsck and tune2fs)
- Has nothing to do with e2defrag (do not use it, it is dangerous!)
- Works on unmounted volume, modifies physically disk device blocks, similar as fsck

File reordering tool (2)

- Reads planned order of files from input file
- Finds contiguous disk area which can hold all blocks of file
- Relocates blocks belonging to inodes, including indirect blocks, in specified order
- Updates bitmaps

Reordering during shutdown

- Reordering of files for faster startup is done during system shutdown
- Script transforms prefetching boot traces into file order input file used by reordering tool
- Last shutdown script before power-off runs reordering tool

Reordering during shutdown - problems

Problems with reordering during shutdown:

- Reordering tool should run on unmounted volume
- Root volume cannot be unmounted
- Reordering tool cannot use disk

Solution (hackish, better would be welcome):

- Cache the tool and needed files by reading them
- Force reordering on read-only mounted volume

Reordering results

- Reordering run takes about 14-20s (might be improved)
- Boot time without reordering, with prefetching: 52.68s
- Boot time with reordering: 47.75s
- Improvement over just prefetching: 4.93s (9%)

Summary

What is done

- Initial prefetching facility for Linux is implemented
- Application startup prefetching works and gives about ~10% improvement in startup time
- Boot startup works and can give ~10% to ~20% improvement in startup time
- File reordering is not yet ready for production use, but can give further ~10% improvement in boot time

Unsolved (yet) problems

- Still to do:
  - Applications loaded by IPC (e.g. kdeinit)
  - „Loader” applications which load other using dlopen() (e.g. kcmshell --lang pl. --embed 0x123 displayconfig)
  - How to lay out files shared among many applications effectively?
  - Detecting user login

More information, current versions and precompiled kernel for Ubuntu available at “Prefetch” project on Google Code:

http://code.google.com/p/prefetch