Analyzing and improving GNOME startup time

Lorenzo Colitti

RIPE NCC

Abstract

The startup time of open source desktop applications and environments does not compare well with other systems, especially closed-source systems. We perform a detailed analysis of the startup of the GNOME desktop environment using a mixture of available open-source tools and ad-hoc measuring techniques, identifying bottlenecks and examining strategies to improve performance. Our results show that startup is I/O bound and dominated by disk seeks, and that substantial improvements can be made at relatively little cost. We validate the results of the analysis with proof-of-concept code modifications which provide a 40% reduction in measured login time; many of these suggestions have been adopted by the developers and implemented in the current releases of the applications in question. We also evaluate the impact of library loading strategies to startup time, finding that trivial changes to the dynamic linker can provide a 10% reduction in the startup time of large applications.

1 Introduction

Despite substantial advances in hardware, the time it takes to log in to the GNOME desktop environment [1] has not improved much in recent years. GNOME startup time also does not compare favourably with other systems, especially closed-source systems. In this paper, we analyse the causes of slow startup time and what can be done to address the problem.

The paper is organised as follows: Section 2 discusses the preliminary evaluations performed and the methodologies that emerged from these preliminary investigations and briefly touches on related work; Section 3 describes the measurement and analysis tools used, and Section 4 presents the performance problems found. One of the most important bottlenecks, dynamic library load time, is not due to GNOME but is a systemwide problem that affects all large applications; we discuss it in Section 5. Section 6 provides a summary of the proposed changes and their impact and Section 7 contains conclusions and ideas for future work.

2 Methodology

An initial investigation of GNOME startup shows that it heavily I/O bound: a simple stopwatch test shows that logging in on a system that has just booted up takes more

than 30 seconds, while subsequent logins performed immediately after the first only take about 5 seconds. Therefore, any attempt to reduce startup time must focus on reducing I/O. Also, when measuring, care must be taken to defeat caching, both by the system buffer cache and by the hard disk. Unfortunately, this is not as easy as it seems. In particular, since we know of no way to programmatically clear the Linux buffer cache, we must resort to workarounds such as loading large files into memory.

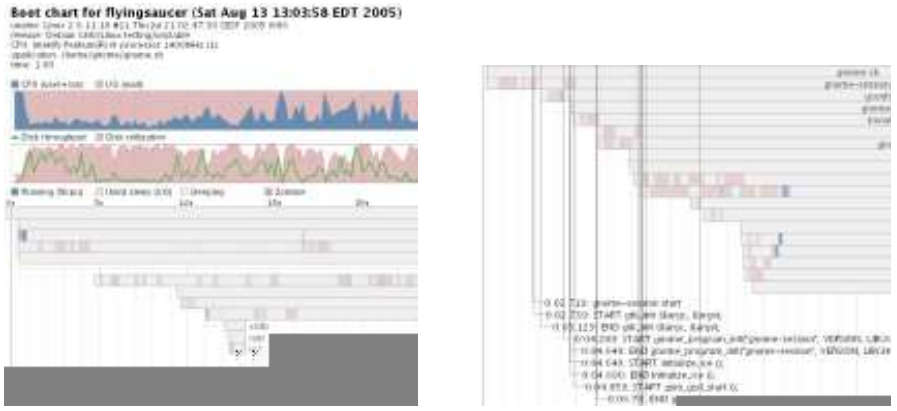
By far the most expensive I/O operation is a disk head seek. At the time of this writing, fast desktop hard disks run at 7200 RPM and have a transfer rate of about 30-70 MB/s, but a random access time of about 10-15 ms, which equates to 60-100 read operations per second. Therefore, while sequential reads might obtain close to the maximum transfer rate, a worst-case random read on a 4k-block filesystem would read at most 100 4k blocks per second, or 400kB/s, which is two orders of magnitude slower. Hard disks used in laptops computers usually run at 5400 RPM or even 4200 RPM and are even slower. This suggests that disk seeks play an important role in startup time and that being able to measure seeks is essential to identify which processes contribute the most to startup time. In the rest of this paper, we develop methodologies and tools to measure startup time and identify performance problems in I/O bound applications. Although we apply these methods to GNOME, they are not GNOME-specific and can be applied to analyse the startup of any I/O bound application.

We are not aware of much other work done on I/O profiling of open-source applications. A similar problem is tackled by Hubert [2], who takes a complementary approach, focusing not on tuning application behaviour but on optimising kernel behaviour for application use. Other work on improving open-source application startup time has been carried out in the `preload` [3] project, which, however, does not attempt to analyse the causes of performance problems but aims to speed up login time by using readahead during system boot.

2.1 Benchmarking methods

Initially, benchmarks were run by starting the test version of GNOME on a virtual X display (`Xvfb`) while the system was already running another X server and the distribution-provided version of GNOME. Since the distribution version and the test version are in different directories and share no files, this approximates the behaviour of a cold boot, unless data from a previous benchmark run is still in the cache. Therefore, the caches were empirically flushed between runs by reading large files into memory; this approximation allowed the most obvious performance problems to be identified and removed without having to reboot the system every time.

However, the results obtained using this method are too noisy to allow the measurement of small differences in startup time; furthermore, they do not reflect the effect of applications that may already be in cache during normal startup but not during benchmarking (e.g. `xrdb`, which is started both by the X server startup scripts and by GNOME, and effectively gives no measurable performance penalty when run by GNOME if GNOME is started immediately after X). Furthermore, it is less realistic than benchmarking from a clean boot because of the effects of other processes that are running on the system. Therefore, later benchmarks were performed by rebooting into single user mode and running a custom system startup script that loaded `gdm`, the



(a) (b)

Figure 1: Example bootchart output: (a) part of a chart; (b) milestones

GNOME display manager, which was configured for autologin. Unfortunately, even when rebooting between runs, consecutive benchmarks showed that startup time was slightly higher the first time a given configuration was benchmarked than on successive trials. This may be have been due to the the hard disk's internal cache; to obtain consistent results, every measurement was performed twice and the first result discarded.

2.2 Hardware configuration

All tests were conducted on a Dell Latitude D400 notebook PC with 512 MB RAM and a Toshiba MK3021GAS 30GB, 4200 RPM hard disk. The system was running Debian Unstable GNU/Linux with a custom 2.6.12.5 kernel and the X.org version 6.8.2 X server. The version of GNOME code analysed was downloaded from the GNOME CVS repository on 20 August 2005, shortly before the 2.12.0 release, and compiled with `jhbuild`. To avoid disturbing the distribution-provided GNOME installation, the test GNOME version was installed under `/usr/local/gnome`. For the purpose of this work, startup time is defined as the time between when gnome-session is launched and when disk activity stops with an otherwise idle system and no user intervention.

3 Tools used

Measurements were performed using a combination of existing tools and ad-hoc techniques. A brief overview of these follows.

3.1 Bootchart

Bootchart [4] is a program originally written to monitor the Linux boot process. It consists of (i) a shell script daemon that monitors various system and process parameters at

regular intervals and collects the output in a series of log files, and (ii) a Java program run after monitoring that analyzes and processes the log files and produces a chart that shows the use of system resources by various processes over time. An example output is shown in Figure 3.1. In order to monitor GNOME startup a few modifications to bootchart were made:

Non-root user support Bootchart normally stores its logs under `/var/log` and its configuration files under `/etc`, neither of which are accessible to a non-root user. It was modified to allow configurable log file and configuration file paths.

Better single-process support Bootchart uses a series of heuristics to collapse processes, to merge multiple processes into one, and to eliminate from the chart processes which are not ”interesting”. This does not work well for GNOME startup. Bootchart does have a ”single process tree” mode in which it attempts to monitor only the processes spawned by a single command, but it cannot track processes spawned by a process which has lost its parent, which is the case for virtually all of the programs launched during GNOME startup.

Milestone support Bootchart lacks a mechanism for marking on the graph the time at which a particular event occurred. However, such a mechanism is essential for monitoring specific applications and for finding out which phases in application startup take the most time. Therefore, support for “milestones” was added. The application monitored may write strings of the form:

```
xxx.yy <string>
```

to a file named `milestones.log` in the bootchart temporary directory, where `xxx.yy` is the system uptime taken from the `/proc/uptime` special file and `<string>` is a text string. Bootchart will then display them on the chart. Displaying the milestones on the bootchart can then be achieved simply by modifying the GNOME binaries to write a status line to the milestone log when some significant point in startup is reached.

3.2 Sysprof

Sysprof [5] is a Linux system profiler that uses a kernel module to profile the entire system as it is running. Sysprof was designed to profile CPU and not I/O usage, but the author has also developed a kernel patch to profile block reads on ext3 filesystems. This has the advantage of being able to monitor all I/O that actually occurs on the disk, rather than simply logging I/O requests or `read()` calls. Sysprof is a GUI application, so to monitor the GNOME startup process a console application, `sysprof-text`, was written. `sysprof-text` non-interactively collects profile data to a file that can then be opened and analysed in the Sysprof GUI. An example of Sysprof’s output is in Figure 4.

3.3 IOlog

Iolog is a small kernel module developed for this project which uses the sysprof ext3 patch to log all ext3 block reads to the kernel log. Every block read from the filesystem

```
(gdm/2982): /usr/local/gnome/sbin/gdm-binary 0-7
(gdm-binary/2982): /usr/local/gnome/lib/libgtk-x11-2.0.so.0 0-7
(gdm-binary/2982): /usr/local/gnome/lib/libgtk-x11-2.0.so.0 687-718
(gdm-binary/2982): /usr/local/gnome/lib/libgtk-x11-2.0.so.0 653-684
(gdm-binary/2982): /usr/local/gnome/lib/libgtk-x11-2.0.so.0 34-65
(gdm-binary/2982): /usr/local/gnome/lib/libgtk-x11-2.0.so.0 8-33
(gdm-binary/2982): /usr/local/gnome/lib/libgtk-x11-2.0.so.0 515-546
```

Figure 2: Example output of `analyzereads.py`

```
$ ./topseeks-byprocess.sh io.log.gnomeonly 5
    122 nautilus
    109 gnome-session.r
     74 bonobo-activati
     49 gnome-panel
     46 gnome-settings-
$ ./topseeks-byfile.sh io.log.gnomeonly 5
     26 /usr/local/gnome/lib/libgtk-x11-2.0.so.0
     10 /usr/local/gnome/lib/libxml2.so.2
      8 /usr/lib/libstdc++.so.6.0.5
      8 /usr/lib/libstdc++.so.5.0.7
      7 /usr/local/gnome/lib/libpoppler.so.0
```

Figure 3: Output of `topseeks-byfile.sh` and `topseeks-byprocess.sh`

will result in a line such as:

```
READ: 1125508642.466082 (metacity/377) 116 /usr/local/gnome/bin/metacity
```

where `1125508642.466082` is a timestamp, `metacity/377` is the name and PID of the process performing the read, `116` is the offset into the file in 4k blocks, and `/usr/local/gnome/bin/metacity` is the file being read.

Simple scripts were then written to parse the logs. A script written in Python, `analyzereads.py`, was used to parse I/O logs and coalesce contiguous read operations. An example of its output can be seen in Figure 3.3, which shows the `gdm` process reading the first 32k of the `gdm-binary` executable and then `gdm-binary` itself performing 6 non-contiguous reads on `libgtk-x11-2.0.so.0`. This makes it possible to distinguish contiguous I/O operations, which do not have a large impact on performance (unless the filesystem is fragmented), from non-contiguous I/O operations, which involve disk seeks and thus have a more significant impact on performance.

Two shell scripts, `topseeks-byfile.sh` and `topseeks-byprocess.sh`, wrap `analyzereads.py` and were used to produce lists of the files and of the processes that are responsible for the most non-contiguous reads. For an example of their output, see Figure 3. It is easy to see, for example, that the process causing the most seeks is `nautilus` and the file that is read in the largest number of non-contiguous reads is the GTK library.

3.4 Strace

`Strace` is a program that traces the system calls made by another process. It can be very useful to determine, which files a process is opening, if a process is blocking on a socket, if it is timing out on a DNS lookup, etc. Strace is very simple to use. For example, the command: `strace -e trace=open cat file` will print the all the `open()` system calls executed by the command cat file, with timestamps, to standard error. The output is similar to the following:

```
open("/etc/ld.so.cache", O_RDONLY) = 3
open("/lib/tls/libc.so.6", O_RDONLY) = 3
open("/usr/lib/locale/locale-archive", O_RDONLY|O_LARGEFILE) = 3
open("file", O_RDONLY|O_LARGEFILE) = -1 ENOENT (No such file or directory)
[...]
```

The `-p pid` option requests `strace` to attach to the existing process having PID `pid`, while the `-f` option follows forks and outputs the system calls executed by child processes as well.

4 Startup bottlenecks

In this section, we present performance problems in the login process that were found during benchmarking, discuss their impact, and show the improvement in startup time that could be gained by eliminating them, either by reporting the effect of proof-of-concept modifications made or by estimating their contribution.

4.1 Reading GConf defaults

GConf is the GNOME system for storing user preferences. It is one of the top causes of I/O during GNOME startup: from a bootchart of an unmodified installation it can be seen that `gconfd-2` performs almost constant I/O at many points in startup. Simply running `strace gconfd-2` showed it opening hundreds of files in subdirectories of `/usr/local/gnome/etc/gconf/gconf.xml.defaults`. This is because the GConf database is made up of hundreds of files (one per key) in directory trees that mirror the configuration database hierarchy. Fortunately, GConf already contains code to read whole configuration directory hierarchies from a flat file and a tool, `gconf-merge-tree`, can be used to merge a configuration directory hierarchy into such a file. Merging all the files under `gconf.xml.defaults` into one file in this way yields an approximately 7-second reduction in startup time.

Further improvement is possible: as shown in the bootcharts in Figure 5, when the files are merged GConf is I/O bound for about 3 seconds and then is CPU bound for about 4 seconds. This suggests that the flat file is being read into memory in its entirety and then processed all together. Therefore, a patch was written to process the file incrementally while reading it. As can be seen in Figure 5, when the files are also read incrementally, `gconf-2` performs I/O and uses the CPU simultaneously. This yields a further 2 second reduction in startup time

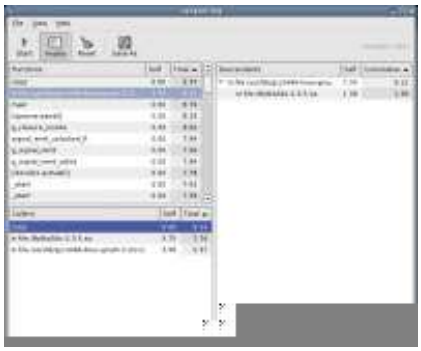


Figure 4: Example Sysprof output

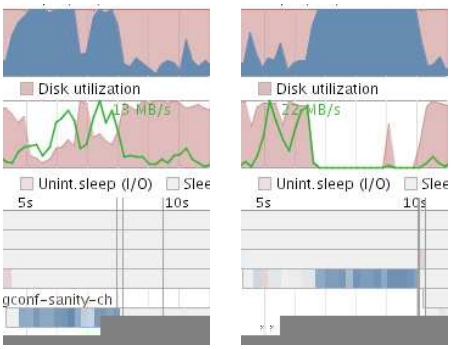


Figure 5: Effect of incremental parsing of the gconf database. Left: parse after reading; right: parse while reading

Still more reductions are possible by removing redundant translations of GConf key descriptions. The GConf database contains the description of configuration keys in many languages, only one of which is likely to be in use by a given user. When all non-English descriptions were removed from the flat file using a simple XSLT file, a further two-second reduction in startup time was achieved.

These problems, with a proof-of-concept solutions, were brought to the attention of the gconf developers, who addressed the issue. In GNOME 2.14 the GConf database is merged and translations are split into one file per language by default.

4.2 Loading the C compiler to process X resource files

`xrdb` is a program to manipulate X resource settings, which are user preferences for certain non-GNOME applications. Is it called twice, once by the display manager startup scripts to load user-specified X resources, and then by the gnome settings daemon in order to match the colors of non-GNOME applications to the GNOME colour theme. Since X resource files may contain C-style preprocessor statements, `xrdb` calls the system C preprocessor to process them. Unfortunately, since the GNU preprocessor is integrated in the C compiler, this requires loading the entire C compiler into memory (about 4 MB) at login time. Patching `xrdb` to use a smaller C preprocessor implementation such as `mcpp` [6] results in a 1.5 second reduction in startup time. This issue was brought up with the maintainers of `xrdb`, but the suggested solution (detecting a lighter C preprocessor at `xrdb` runtime) did not gain consensus.

4.3 Loading CORBA servers

The Bonobo Activation server is responsible for the keeping track of CORBA objects and servers within GNOME. At startup, it enumerates all CORBA servers, each of which is described by a `.server` XML file; on the test system had 58 such files. This causes a substantial number of non-contiguous reads during startup: even on an

otherwise idle system, reading the files takes more than 1.6 seconds.

Since the server files do not change often, an obvious optimisation would be to maintain a cache of all the server files in a single file, which would be rebuilt every time a file was modified or a new file was added. The impact of such a change was not evaluated, but we can assume it would save over 1 second of startup time, since if the files are all concatenated to a single file and the caches are flushed, reading the file requires less than 0.25 seconds.

4.4 Loading the screensaver

The screensaver is loaded by the gnome settings daemon very early in the startup process. However, it is not needed immediately after login, and loading it competes for resources with other startup tasks which are more important. Therefore, the screensaver should be loaded only after more important startup tasks have been performed. To evaluate the effect of lazy-loading the screensaver, it was wrapped by a simple shell script that loaded it after a fixed 25-second delay. This yielded a 1 second reduction in startup time. Since loading the screensaver causes a non-trivial amount of I/O, care must be taken to load it when the system is otherwise idle. This issue was addressed by the gnome settings daemon developers and in GNOME 2.14 the screensaver is loaded after a fixed time delay.

4.5 Opening the main menu

According to our definition of startup time in Section 2, the time it takes to open the main menu (or "foot menu") is not part of startup time. However, since the main menu takes several seconds to open and is one of the things that a user is most likely to do just after startup, we investigated the causes of its sluggish behaviour.

To determine what happens when the main menu is opened, caches were flushed and the `gnome-panel` process was terminated. Once the panel process was automatically reloaded by the system, `strace` was used to attach to it and the menu button was clicked. Analysis of the `strace` output showed that opening the main menu took about 4 seconds and required opening no fewer than 372 files, most of which (285) were desktop shortcuts. This suggests that caching desktop shortcuts in a contiguous file would greatly help improve the response time of the main menu.

5 The impact of dynamic library loading

Analysis of the I/O logs shows that most of the I/O during GNOME startup is caused by loading dynamic libraries, which are not read sequentially because they are loaded with `mmap()` and page faulted into memory as needed. Indeed, once other sources of non-contiguous I/O such as reading GConf defaults are removed, libraries account for about 65% of non-contiguous reads.

This is not something which is easy to address within GNOME itself. Efforts to consolidate libraries such as Project Ridley, which plans to merge a number of smaller libraries into GTK+, would not help: since large libraries are already being

Library name	# requests	Library name	# requests
libgtk-x11-2.0.so.0	32	libgdk-x11-2.0.so.0	6
libxml2.so.2	10	libORBit-2.so.0	5
libpoppler.so.0	7	libgstreamer-0.8.so.1	5
libnautilus-private.so.2	7	libecal-1.2.so.3	5
libgnomeui-2.so.0	7	libdbus-1.so.1	5
libglib-2.0.so.0	7	libbonoboui-2.so.0	5
libeel-2.so.2	7	libbonobo-2.so.0	5

Table 1: Libraries read in five or more non-contiguous read operations during startup

read sparsely using `mmap()`, consolidating many small libraries into one large file does not significantly reduce the number of disk seeks. Preloading the libraries with readahead does help, since in this case the library files are read contiguously. Although this increases memory usage, it is a worthwile tradeoff on a desktop machine.

To evaluate the impact of loading libraries contiguously at runtime, the GTK+ library, which is the library responsible for the most seeks, was preloaded simply by executing the command `cat libgtk-x11-2.0.so.0 > /dev/null` early in the startup process. This had the effect of loading the library into the buffer cache using fast sequential reads, so when the dynamic linker performed random reads at runtime no seeks were necessary. This reduced startup time by approximately 0.5 seconds; to determine if it was advantageous to preload other libraries too, all the libraries which were read in 5 or more non-contiguous reads during the whole startup process (shown in Table 5) were preloaded in the same way. This yielded a further 1-second reduction in startup time.

These results suggested that that it would be convienient to modify the linker in order to be able to configure it at compile-time or run-time to read whole libraries contiguously. Therefore, a one-line patch to the glibc dynamic linker was written in order to load every dynamic library sequentially into memory as soon as it is mapped. The patch reduces GNOME startup time by approximately 3 seconds, a more than 10% improvement, and shows similar improvements in the startup of other large applications such as the Firefox browser and the OpenOffice.org office suite.

This change was suggested to the glibc developers in a message to the glibc mailing list [7], but was rejected on the grounds that the dynamic linker is not an appropriate place for such an optimisation. However, discussion with other members of the community suggest that at least one high-profile Linux distribution modifies the dynamic linker in such a way.

6 Summary of proposed improvements

All the performance problems revealed by our analysis were reported to the developers of the packages concerned, in many cases with proof-of concept modifications showing the performance benefits. Table 2 shows a list of the improvements suggested.

As can be seen, many of the proposed changes have been implemented in the current version of the packages concerned. Others, especially those with non-GNOME

Proposed change	Gain	Status	Reference
Merge GConf defaults	7	FIXED	GNOME bug #316672
Separate GConf translations	2	FIXED	GNOME bug #316672
GConf incremental reading	2	FIXED	GNOME bug #316673
Use mcpp instead of cpp in xrdb	1.5	rejected	freedesktop.org bug #4325
Cache bonobo-activation files	1 (?)	—	—
Lazy-load screensaver	1	FIXED	GNOME bug #316674
Preload dynamic libraries in linker	3	rejected	libc-alpha mailing list

Table 2: State of the improvements suggested

software, were dismissed by the developers as being the wrong approach. This may be due to the fact that those projects have a broader focus than just on single-user desktop performance.

Figure 6 provides bootcharts of GNOME startup time before and after applying the suggested improvements. As can be seen, there is a substantial improvement in startup time. Ignoring the block on the left of the graph, which corresponds to X server initialisation, login time is 27 seconds without modifications and 16 seconds with modifications, a 40% decrease. (Note that these results are not directly comparable to those in Table 2 because they refer to a subsequent installation of GNOME.)

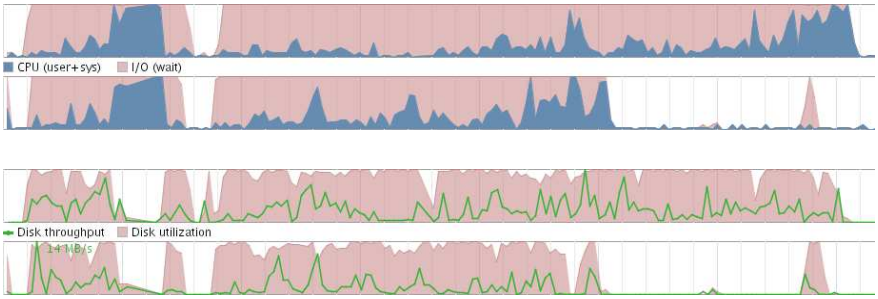


Figure 6: Login time before and after applying the suggested improvements. Above: CPU usage. Below: disk usage.

7 Conclusions and future work

We have performed a detailed analysis of the startup of the GNOME desktop environment, identifying performance problems, evaluating their impact, and suggesting possible solutions. Our results show that startup is I/O bound and dominated by disk seeks. We propose proof-of-concept modifications which provide a 40% reduction in startup time; these have been suggested to the developers and many have been implemented in the current release of the respective applications. Finally, we find that trivial changes to the dynamic linker can provide a 10% boost in the startup time of large applications.

Although we studied GNOME, our methodologies are not GNOME-specific and can be used to instrument any open-source program.

There is much work that can still be done in this area. As regards GNOME itself, the contributions of other components such as the panel, nautilus, and desktop applets could be analysed. On a more general note, logical next steps are investigating the effect of stripped libraries and of optimising for size at compile time: in both cases, the resulting libraries are smaller and thus might cause fewer non-contiguous I/O operations. The analysis could be further refined, for example by generating a graph of disk seeks over time by each process. More methodological contributions would be the development of a more generic I/O measurement structure, perhaps using the kprobes [8] facility of the Linux kernel. This could then be used by anyone – including the developers of the applications themselves – to analyse startup time in a very simple manner.

Acknowledgements

The author would like to thank Owen Taylor for mentoring this work and for providing support in every area. Thanks go to Ziga Mahkovec for writing Bootchart and to Søren Sandmann and the many other helpful people in `#fedora-desktop`. This work was supported by the Google Summer of Code programme.

References

[1] The GNOME Project. GNU object model environment (GNOME). `http://www.gnome.org/`.

[2] Bert Hubert. On faster application startup times: Cache stuffing, seek profiling, adaptive preloading. In *Proc. Linux Symposium*, July 2005.

[3] Behdad Esfahbod. Preload – an intelligent readahead agent. http://sourceforge.net/projects/preload.

[4] Ziga Mahkovec. Bootchart. `http://www.bootchart.org/`.

[5] Søren Sandmann. Sysprof – a system-wide Linux profiler. `http://www.daimi.au.dk/ sandmann/sysprof/`.

[6] Kiyoshi Matsui. Mcpp – a portable C preprocessor with validation suite. `http://www.m17n.org/mcpp/index_eng.html`.

[7] Lorenzo Colitti. Using readahead in `ld.so` for 10% startup gain in big apps. glibc mailing list thread, September 2005. `http://sources.redhat.com/ml/libc-alpha/2005-09/msg00054.html`.

[8] Prasanna Panchamukhi. Kernel debugging with kprobes. `http://www-128.ibm.com/developerworks/linux/library/l-kprobes.html`, August 2004.