

Universitatea “POLITEHNICA” București

Facultatea de Automatică și Calculatoare,

Catedra de Calculatoare



LUCRARE DE LICENȚĂ

Logrind: Analiza operațiilor I/O pentru
reducerea timpului de startup al
aplicațiilor

Conducători Științifici:

as. drd. ing. Răzvan Deaconescu
drd. ing. Octavian Purdilă

Autor:

Maruseac Mihai

București, 2011

University “POLITEHNICA” of Bucharest

Automatic Control and Computers Faculty,
Computer Science and Engineering Department



BACHELOR THESIS

Iogrind: I/O Operations Analysis For Improved Application Startup

Scientific Advisers:

as. drd. ing. Răzvan Deaconescu
drd. ing. Octavian Purdilă

Author:

Maruseac Mihai

Bucharest, 2011

I would like to thank Michael Meeks for the never ending support that he has given during the elaboration of this project. Although busy with many open source projects, he always found time to answer questions regarding this program, its purpose and its usage. We owe him for his endless patience, which we stress-tested so many times.

Also, I would like to thank Răzvan Deaconescu. He always challenged our interpretation of the project idea, raising new questions for Michael. This steady stream of challenges made this project possible.

Abstract

Application optimization is an important topic today, with a large part of the efforts being concentrated on startup time reduction. While the performance of CPU operations can be easily increased, any improvement in the I/O part is hard to measure because of the inherent mechanical timing jitter. The Iogrind project aims to provide a fast and deterministic framework which can be used to improve the I/O part of application startup. The infrastructure will work as an I/O profiler, allowing multiple useful scripts to work on the profiled data in order to ease the optimization process.

Keywords: I/O, optimization, SystemTap, profiling, kernel trace

Contents

Acknowledgements	i
Abstract	ii
1 Introduction	1
2 I/O optimization	3
2.1 I/O Improvement Strategies	6
2.1.1 Merging Files	6
2.1.2 Lazy Resources Loading	7
2.1.3 Ordering Reads	7
2.1.4 Changing Directory Structure	8
2.1.5 Libraries	9
2.1.6 Improving Kernel I/O Algorithms and Structures	11
3 State Of The Art	12
3.1 Iolog	13
3.2 Bootchart	14
3.3 Mortadelo	15
3.4 Icegrind	16
4 Iogrind	17
4.1 Iogrind on Valgrind	17
4.2 Tracing Iogrind	19
5 Trace Obtaining	22
5.1 SystemTap	22
5.2 Probing	23
5.3 Pagefault Handling	25
5.4 Glue Calls	26
5.5 Invoking	27
6 Tracing Results	29
7 Improvements and Future Plans	32
A Test Application	33
B Trace Generated by Iogrind	35

List of Figures

2.1	Histogram of startup times for multiple sequential runs of dpkg -l	3
2.2	Histogram of cold startup times for dpkg -l	4
2.3	Histograms of runs of dpkg -l sequential and preceded by a reboot, compared	5
2.4	Read ordering impact	8
2.5	Library layout and access	9
2.6	Library read ordering	10
3.1	Output of Bootchart while starting the GNOME environment	14
3.2	Mortadelo	15
4.1	Original architecture of Iogrind	17
4.2	Stack view of Iogrind	18
4.3	Address view of Iogrind	19
4.4	Scribble view of Iogrind	20
4.5	Architecture of Iogrind with SystemTap logging	21
5.1	Structures and probes used in Iogrind	27
5.2	Launching an application for tracing	28

List of Tables

1.1	Application startup times (in seconds)	1
6.1	Timing cold startup of application and trace (in seconds)	29
6.2	Timing warm startup of application and trace (in seconds)	29

List of Listings

3.1	strace output	12
3.2	iostat output	13
3.3	Iolog - parsed output	13
3.4	Iolog - seeks by process	13
3.5	Iolog - seeks by file	14
3.6	Icegrind - prelink command	16
5.1	SystemTap script to intercept all system calls	23
5.2	Intercepted system calls	23
5.3	SystemTap: begin probe	24
5.4	SystemTap: global variables	24
5.5	SystemTap: never probe	24
5.6	SystemTap: Intercepting read	24
5.7	SystemTap: pagefault handling	25
5.8	SystemTap: open	26
5.9	SystemTap: starting trace	28
6.1	Library I/O from ack-grep	30
6.2	I/O caused by interpreted code	30
6.3	I/O requested by programmer	30
A.1	Test Application	33
B.1	Complete Trace of Test Application	35

Notations and Abbreviations

AI – Artificial Intelligence
CLI – Command Line Interface
GIMP – GNU Image Manipulation Program
GTK – GIMP Toolkit
I/O – Input/Output operations
OS – Operating System
PID – Process ID
VFS – Virtual File System

Chapter 1

Introduction

Although computer resources are no longer scarce, the constant increase in user's expectations places application optimization as an important topic, even today. A great impact can be attained by concentrating on reducing the startup time of applications. There are a lot of common-used applications which takes a long period before displaying something on the screen, making users believe that something is wrong or they didn't start the program. For example, **Open Office** takes around 5 seconds to completely start and 2 seconds to display the loading screen. **DrRacket** takes 10 seconds to start. More startup time values are summarized in [Table 1.1](#) (all git commands were issued in the same repository). There is a significant time lost for the user and there is a demand to reduce it. This is a known fact to developers. Now, they are trying to improve this part of the application.

Table 1.1: Application startup times (in seconds)

Application	Cold startup time	Warm startup time
Audacious	2.50	0.76
Chromium	14.65	3.90
DrRacket	12.79	6.30
Firefox	12.47	4.13
Gitk	1.70	0.41
Gitg	3.37	2.00
Open Office	5.72	0.59
Pidgin	8.19	1.96
Skype	7.16	2.56
Tig	1.38	0.15
Totem	2.56	1.90
VLC	2.59	1.65
XChat	10.17	3.20

As can be seen from [Table 1.1](#), the delay is even greater in the case of cold startups, the first startup of the application after the system was booted up. This is because at that time, a vast number of libraries and other shared resources are not loaded because they were not needed before. The operating system knows to keep the shared resources in memory or in a cache, such that subsequent demands can be satisfied in a shorter time. Usually, cold startup times are at least 30% longer than subsequent startups. Thus, optimizing for cold startup will greatly affect all the perceived startup times.

Right now, the startup time is nicely divided into 80% I/O and 20% CPU (see [10]). Of these two parts, the later was extensively optimized over time. Profilers for CPU bound code are abundant and easy to use. However, the I/O part is an unsure territory but a very promising one. While CPU bound processes are easy to optimize, following certain patterns and looking for certain use cases, in the I/O land the situation is different. There are a lot of ideas to improve application response and testing them completely takes a lot of time. Also some of the testing results will have a high degree of uncertainty. This is the main reason why all efforts have been concentrated on the CPU part, although it is the I/O which will give the most remarkable improvements.

The **Iogrind** project aims to provide a fast and deterministic framework which can be used to improve the I/O part of application startup. In fact, using the **Iogrind** tool, one can not only optimize a single application with regard to the I/O, but also optimize the entire operating system if he/she knows that it would be mainly used for a certain range of applications, all of them analyzed using this tool. As a result, the normal user will have improved startup times for a single application and the operating system maintainers will be able to improve the performance of the entire system.

An important aspect of this tool is that it touches a lot of kernel internals. Thus, it is absolutely necessary to run it from a `root` account. However, this is needed only in the first part of the application usage, that is until the original application is traced (as will be seen in [chapter 4](#)). Afterward, one could use a normal user account to work with the trace, if he/she wants to.

The first part, mentioned above, uses **systemtap** (go to [3] for the homepage) scripting. It will only work on Linux kernels and only after compiling them according to the SystemTap instructions (given in the wiki page at [13]). However, after this part, there is no need to use SystemTap. The results can be analyzed and acted upon on any computer with any architecture as long as the generated module or the generated source code can be moved to it.

This application will not optimize another application. It will only offer some hints, working like a normal profiler. It won't be a simple I/O profiler though. The output of this project is another executable application, equivalent to the original one with regard to the I/O activities that both applications are doing. Having the source code of the obtained executable allows one to develop new utilities on top of this profiler. Also, optimizations to the reduced application can be done exactly in the same way to the original one.

Chapter 2

I/O optimization

As presented in [chapter 1](#), concentrating on optimizing application I/O will give remarkable improvements in startup times. Still, this topic is not so popular because there are a few aspects making this very hard to be done properly.

To start with, I/O has to work very closely with the hardware. However, rarely an application developer will need to be concerned with the underlying mechanical systems. This is why we have operating systems after all. Although the OS helps us to concentrate on the essential and forget about small details, it hides and makes hard to detect opportunities for optimizations. Those opportunities can be revealed only by using a tool to analyze the application, a tool just like **Iogrind**.

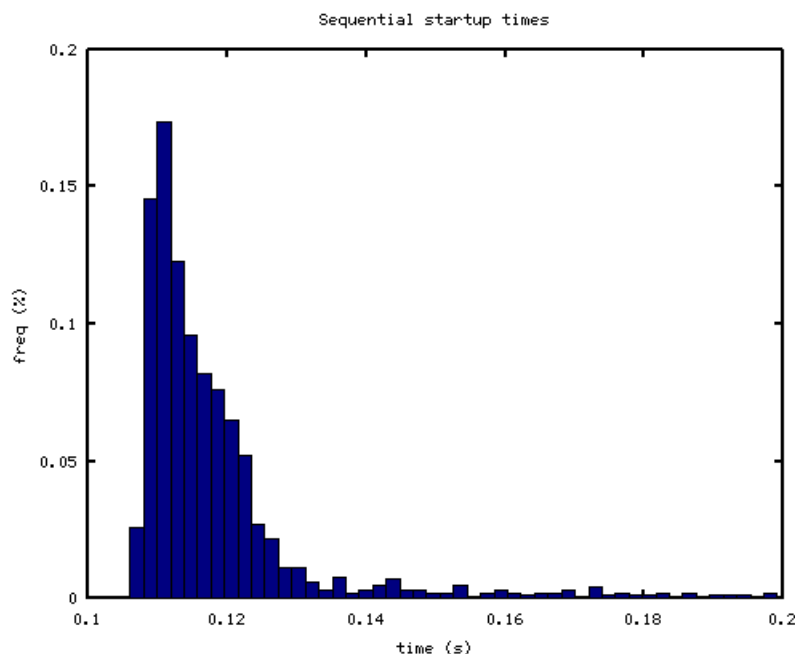


Figure 2.1: Histogram of startup times for multiple sequential runs of `dpkg -l`

In our case, many of the details of the underlying mechanical system of the hard disk are lost to the user-space developer. He/She can only see a jitter in runtime duration. For example, [Figure 2.1](#) shows a distribution of the startup times for subsequent runs of a single application

(in this case `dpkg -1`). The data follows a gamma distribution but the parameters are not relevant. The important fact is that the majority of values are within a 18% window centered around the median.

In order to see why this affects the optimization process, suppose someone obtains a theoretical 5% improvement in I/O code. To test this, he/she would have to run the application. If there is an approximate 10% jitter in timing, that improvement is not seen. Worse, it can turn out that instead of improving the application, the developer made it perform poorly, depending on the hardware's and the operating system's state.

Of course, one can run the application multiple times. Using statistical methods, he/she can then compute a realistic approximation of the application performance. This method has two disadvantages, though.

Firstly, some applications take a long time to start. Running them for several times in a row (maybe 100 or more) is a very time consuming process. It has to be repeated after each new change promising to add an improvement. This will increase development time, reducing time to market. Secondly, since it is routine work, there are many places where one can do a mistake. Using a script to run this experiment will work but it will still take a long time and the developer will take long breaks or will be forced to do multitasking working on different projects - thus doing frequent context switching from the task at hand. This will increase the probability to introduce a new bug.

Moreover, only the first run after system boot will be really relevant. That is because of the various caches that the operating system has in order to improve the overall performance. When running an application for the second time, almost all shared libraries are in memory, some of the configuration files are in I/O caches, etc. Of course, not all the data resides in memory but, depending on the frequency of the startups and the load of the system, all runs after the first one will seem to take a shorter time to start.

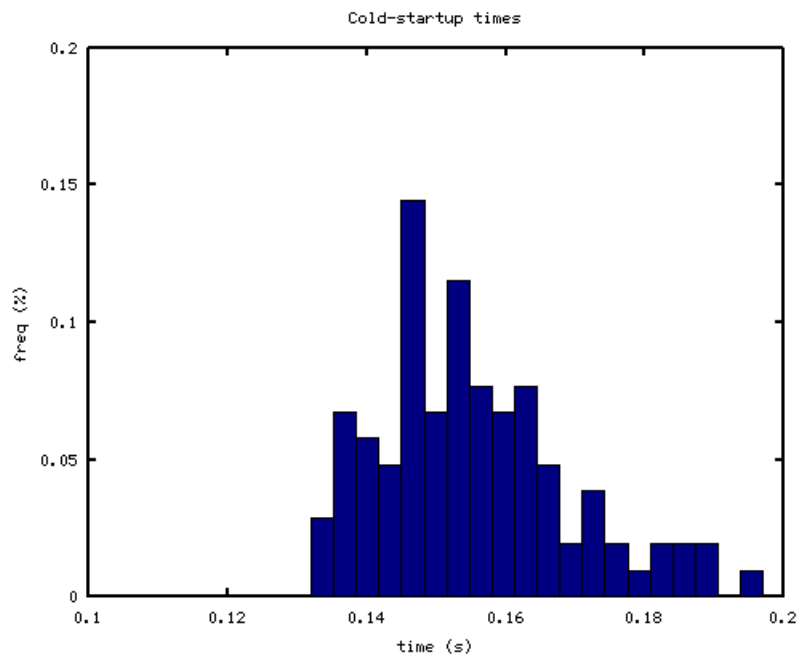


Figure 2.2: Histogram of cold startup times for `dpkg -1`

For example, running the same application (`dpkg -1`) each time after a new system boot gave different results, summarized in [Figure 2.2](#). Now, there is still a gamma distribution but with

lower kurtosis. More importantly, the high density window is almost 33% around the median, meaning that there is now a higher barrier for improvements to be noticed.

However, the most relevant aspect is the shift in the startup times. A comparative plot of both runs is given in [Figure 2.3](#). From there, one can see that testing by running multiple instances of the same application in a row will give different results than running it only after a reboot of the system. Depending on the program tested, this difference can be a significant one. Of course, optimizing for cold startup means optimizing for any startup. The reverse, however, is not always true.

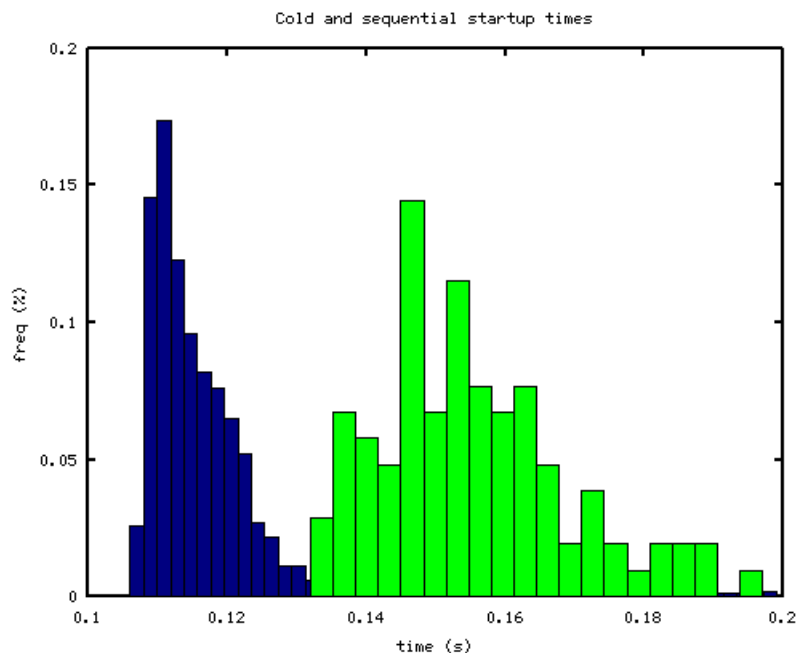


Figure 2.3: Histograms of runs of `dpkg -l` sequential and preceded by a reboot, compared

To solve the second problem, a complete OS restart is needed before each application is launched to be tested. Otherwise, data will be skewed and some optimization opportunities might be missed. But, doing a restart each time will make the first problem mentioned above even worse. Now the booting time (much longer than the application) is added to the total waiting time.

Also, if a complete restart is required there are also reaction times to be taken into account. The user will have to be careful to run the program immediately after the startup and to reboot the system as fast as possible, in order to minimize lost times.

Scripts cannot be used in this setup. One has to turn to virtualization in order to use automatic methods of testing, thus reducing the testing times. Still, the time spent testing is far greater than the time needed to start the application.

For example, to obtain the above plots, 100 runs of the same command (`dpkg -l`) were timed using a script and a VMware virtual machine. Each boot process took additional 45s, much more than the application's startup time.

It seems that the two problems cannot be solved simultaneously. Trying to solve the cold-startup problem has only increased the gravity of the other way too much over the payoff threshold. No one would want to spend half a day rerunning the same application only to apply statistical methods and afterward to determine if his change improved the performance of the application.

For a long time, the only way to optimize I/O was to use a theoretical model and some heuristics

and improve the application according to them. Something along the lines of *we assume that each inode reading takes 10ms*. Although it is a brute-force approximation, it works better than the throughout testing method presented in the previous paragraph. It comes with a cost: the model no longer reflects the exact reality. Therefore, some details are lost. Hence, if one wants faster testing times, he/she will have to use a less realistic model. If one wants to use a more realistic model, the testing will take longer; there should be a better solution.

Before presenting what has been done in this matter, we will enumerate several opportunities for optimization. All of them can be used to increase the performance of an application.

2.1 I/O Improvement Strategies

Looking at the actual I/O operations done while the application is starting, one can extract several strategies to use while optimizing. Combined with theoretical models, like the one mentioned above, applying any of the strategies presented here can lead to some optimizations but it is also prone to false positives. Sometimes, using only the theory does not work as intended.

Besides the most obvious ideas, like removing duplicated reads or trying to read only the relevant data – only information used or needed later in the application, there are even more important strategies which can be combined together.

In this chapter, we will present some of them. Of course, this paper cannot present all strategies that can be used or all heuristics. Looking at [4] and [6], we see that this topic contains many improvement ideas. However, lacking a good testing framework, those ideas are nothing more than simple suppositions. The topic is so new and has so many promises that that after finding a good method to quantify the improvements a lot of new ideas will spring.

2.1.1 Merging Files

Many applications need to read several configuration files before starting up. Some of the configurations refer to the GUI, some to network parameters, etc. Or, there is a global configuration file (somewhere in `/etc/default/...`) and a user configuration (in `/home/user/...`) which extends and overrides some of the settings in the initial file. All of them need to be read before completing the startup.

Since the files are spread on the hard disk, there is a least one seek of the reading head from one file's content to the other's. Moreover, the metadata associated with the paths to those files needs to be loaded in memory before touching them. All of this can be avoided if there is a single global configuration file, located in the same directory as the application. In this way, the metadata needed is minimized, thus the startup is improved.

For example, assume an application reading default configuration from `/etc/default/apprc` and user configuration from `/home/user/.apprc`. Suppose that the inode for `/home/user` is in inode cache. Thus, in the first case, we would have to look at 4 inodes. On the other hand, having just a single file in `/home/user` would need only one inode to be read. Hence, there can be an improvement by almost 25% with regard to this aspect. True, this is a very coarse-grained model, a more realistic one would also look at the length of the configuration files or will not assume facts about the inodes found in cache.

This idea has one drawback, though. The configuration files will be too big and way to hard to change. Sometimes it is better to have different configuration files for different purposes. Just like `bash` has a `.bashrc` file for basic configurations and a `.bash_aliases` file for the

aliases. One can use the first file and place aliases there but if they are in a big number, using a separate file is recommended.

Having multiple files can degrade application performance. However, having everything in one big files makes changing the configuration a very hard task to do. That is why it is better to handle configurations via a window dialog, a dedicated command or via a dedicated flag or menu entry.

The strategy of using a single configuration file can be implemented naively as follows: assume that the application uses only the configuration in `/etc/default/...` when it is started, after being installed. After starting it, we can copy this file to `/home/user`. If the user wants to change some configurations, we offer him a dialog window for doing this – it is safe not to let him know that all settings are in a single big file.

2.1.2 Lazy Resources Loading

Another optimization suggests to load only the minimum number of resources needed to start the application, reading the others in a background thread when they are needed or before. This can be improved by using AI techniques to predict the optimal ordering of readings such that all needed resources are in memory when they are required. However, this topic has an entire research domain dedicated to it and will not be expanded in the present paper.

For example, an application using custom icons won't need to load all of the icons from the startup. Only those displayed on the status bar are needed, all others can be loaded after the user sees that his program has started. More exactly, GTK has around 100 icons. All of them can be replaced by custom ones and the programmer is free to add more. Reading all 100 icons from the beginning is not a sensible thing to do. Usual application menus use less than 20 icons. If we read only the needed ones, a great improvement will follow.

This strategy is already used in games. No application from this domain will load all levels into memory. In fact, there will not be a single entire level loaded. Everything is split into pieces and they are loaded only on demand. Although this is done to increase the frames per second count, the idea can be used in other areas too. Also, the field of game development offers a lot of strategies which can be used to optimize the I/O part of any application.

The laziness has its cost too. If the resources are scattered along multiple files, using a lazy loading technique implies the fact that, at some time during the application runtime, there will be many open files. Depending on the operating system's settings, opening another file will fail, causing the entire application to fail.

Another cost of laziness is payed by the developer. He/She would have to be very careful when writing the code. No resource should leak, all file descriptors should be closed. Otherwise, problems similar to those caused by memory leaks will start to appear. Even heisenbugs may show, making the development process difficult and increasing time to market or count of bug reports.

To conclude with, this idea shouldn't be implemented for all applications. Sometimes there is a great cost implied. Even if it was tested by some developers (see [6]), only a reliable I/O profiling framework could say whether lazy loading is a good strategy to improve a specific application or not.

2.1.3 Ordering Reads

Both methods presented above will give little improvement at the cost of greater complexity. Used alone, they will not be really useful. However, by ordering the actual read operations

such that the total number of seeks needed to finish them is minimized provides a significant reduction in startup time.

For example, suppose an application has to read 3 data blocks from a file. Due to irrelevant reasons, the layout of the file is the one shown in Figure 2.4. Reading the blocks in the needed order will need two longer seeks than when the reads are done taking into consideration the layout of the file. In fact, not the layout of the file is relevant but the layout of the harddisk. The blocks in the image may represent different files and this discussion will still hold.

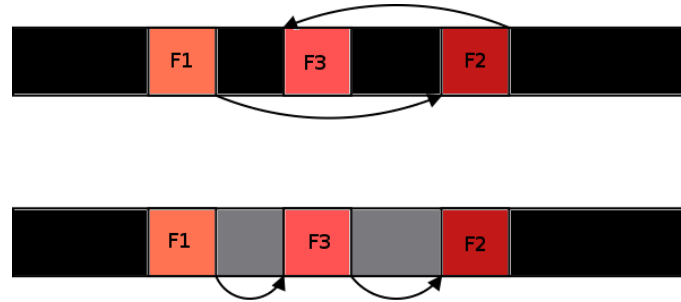


Figure 2.4: Read ordering impact

A further improvement would be to read the grayed areas too, if they are small. Maybe, the data found there will be requested at a later time. Even if it is not needed, there will be no seek from one file location to another. As a side effect, the application code would be simpler on the reading part.

Moreover, if readahead is active in the kernel, application behavior will greatly improve. Almost any time, the data requested to be read from the disk is already in memory because the kernel read it for us when we requested something placed before it.

However, usually, a developer will not pay too much attention to the read ordering strategy. In fact, in a team, each of the developers will read the needed data from a file. To make use of this idea an I/O profiler is needed, something which can provide as output a trace of reads from the filesystem.

2.1.4 Changing Directory Structure

As it was hinted above, the location of the files can have a huge implication in the performance of several applications. Changing the layout of the directory tree will prove useful in those cases.

For example, instead of reading a lot of files nested deep in a hierarchy of directories it is always better to read them from the same directory. It is one thing to have 100 files in a single directory and other to have the same number of files in 10 directories, each containing 5 subdirectories containing 2 files each. In the former case, only 100 inodes need to be touched, in the later 160 (one inode per each file, one inode per each directory and one inode for each subdirectory for a total of $10 + 10 * 5 + 10 * 5 * 2 = 160$ inodes).

There are several applications and use cases which could benefit from this change of layout. The most striking example is compiling a big project with many components and plugins, when each plugin is placed in its own directory. While compiling to get the object binaries is a simple task, the work of the linker becomes harder. It will have to look in many directories just to extract and read a single object file from each of them in order to construct a single executable in the end. If the building script moves all of the binaries to a common location before starting the linking phase, this process will end faster, reducing compilation time.

Still on the same topic, a Java application with many classes will work faster if the majority of them are in the same package than if there are many packages with very few classes distributed in each package.

However, having all files in a single directory has disadvantages too. It is harder to find a file in the directory, for example. It is hard both for the user and for the operating system. While the former will have to scroll a list of files to find what he needs, the operating system will have to read a dentry containing the list of files in that directory. Then, it will have to search in that list for the specific file before opening it. Thus, care has to be taken when using this strategy: while a too nested structure is not optimum, squashing it to a single directory with too many files can imply a bigger performance penalty.

To increase management capabilities and to help users navigate among the files, both structures could exist. The user will use the deep tree while the application will read from the shallow copy. To make sure that both of them are synchronized, each file will be hardlinked from one place to the other.

To conclude with, although this idea can help in some cases, it also comes with big costs. To determine if the benefits outweigh them, a good and reliable framework for I/O profiling need to be used.

2.1.5 Libraries

Going to the operating system's activity, a big part of the application startup time is spent loading libraries. Usually, there is a small number of them. Having a kernel thread or a specific process dedicated to preload them will provide some improvements. Several communities have tried to use this idea in order to improve one application or the other ([2], [10]).

The big gain is due to the fact that libraries are usually loaded on demand at the startup. Depending on the program flow, new code is loaded from disk into memory. This means that more pages have to be read, more disk I/O has to be done. Since the linker creating the library doesn't know in advance how it will be accessed by applications, in most cases, the needed pages are all over the library causing many page faults and multiple disk accesses with many seeks between them. Figure 2.5 shows this in a clear way. While the linker will produce a library with a layout almost similar to the one presented on the first frame, an usual application will only need a few objects found in the library. Thus, the access pattern would look like in the second frame. It is desirable to have an access pattern like the one in the last frame. However, this is very hard to create while the library is constructed.



Figure 2.5: Library layout and access

In Figure 2.5, we didn't show actual jumps from one address to the other. For a normal application there would be jumps to a lower address or a greater address than the current one. The entire access pattern will be a sequence of back-and-forward jumps. This will give a performance penalty because there will be a long number of seeks in different directions. Also, it will invalidate almost all I/O caches.

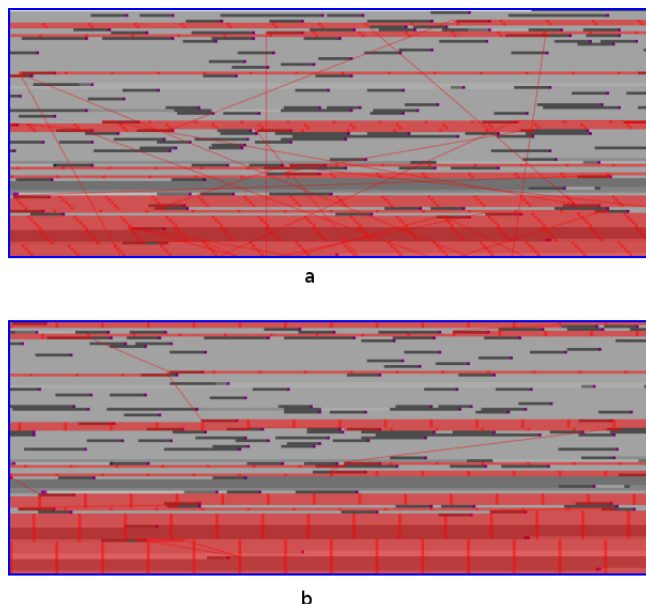


Figure 2.6: Library read ordering

The order in which each object from each library is read has the greatest impact on performance. Figure 2.6 shows this clearly: each red area represented in the figure signifies a memory page corresponding to a mapped library. The high intensity red dots represent page touches causing I/O. Each thin red line represents a seek. The first frame – labeled a – shows the usual access pattern of an application. The second frame shows an optimized access pattern. Reading in a linear way from each library minimizes the number of seeks. This implies that the time spent loading the application is reduced. Also, readahead can take advantage of the read pattern, providing more speedup.

For applications using a great number of libraries, this strategy will provide the best improvement. But, in order to do this, we have to know how an application will access a library, before compiling it. Since this is a hard task to do – if not impossible – we need something else. One idea would be to tie libraries to application: one would have a version of a library for an application and a distinct version for any other application. However, this is the worst idea possible since it is against the reason why we use shared libraries. Another idea – maybe harder to implement – would be to have libraries with dynamic layout in order to be able to reorder objects in library when the application is working with it. This needs artificial intelligence techniques and is a nice topic to study.

A reasonable idea would be to see the common access patterns and change library layout according to them. It won't achieve maximum performance for a single application but it will improve the average loading time of all applications using this library. It is the best that can be done.

However, in order to do this one would need to have a complete log of all library accesses taken from as many applications as possible. Even with them, finding the best ordering is not a simple task. In fact it is a problem similar to the Traveling Salesman Problem from Algorithmic

Theory. Testing each candidate solution will need to run all applications from which the trace was obtained, at least once – better results will be obtained if the applications are run multiple times averaging the durations until they start. However, doing this takes a long time. It is preferable to use something else instead of the entire application. A stripped version of it, equivalent with regards to the library being optimized, is a good alternative.

2.1.6 Improving Kernel I/O Algorithms and Structures

The last place where everything can be optimized is the kernel. In the case of I/O, there are a lot of opportunities.

New filesystems that can take advantage of file size and location can improve performance. It is one thing to read several small files in a filesystem dedicated to read them than to read them in a filesystem dedicated to databases. Google created its own filesystem for this reason. Several other companies are doing this at the moment.

New I/O planning algorithms can be developed. The usual elevator algorithm that is used can be optimized to take into account the type of files read. AI techniques can be employed to create better anticipatory algorithms. If any of the above ideas gains widespread acceptance, kernel algorithms will have to change too.

In fact, this domain is so abundant in ideas that entire chapters will need to be devoted to it. However, the important thing to note is that, until a good framework for quantifying the optimization process is created there would be no real improvement in this domain.

While optimizing for a single application is easier than optimizing library layout or kernel algorithms, the later cases will give more improvements to user space applications. However, they will require a more reliable and more sophisticated method of I/O profiling.

Chapter 3

State Of The Art

Since this field is important, there are a lot of programs and utilities dedicated to application optimization by increasing I/O performance. All of the tools provide a simple statistic of various aspects of the application, at different levels of detail.

Some applications, like **strace**, offer a detailed view of the system calls happening in the application. By filtering them and looking only at I/O related calls, one could get an idea of why the application is starting slowly. For example, here is a part of **strace**'s output, focusing on opening files.

```
1 1126702167.515387 open("/etc/ld.so.cache", O_RDONLY) = 3
2 1126702167.516001 open("/lib/tls/libc.so.6", O_RDONLY) = 3
3 [...]
```

Listing 3.1: **strace** output

However, this method hides a lot of details and some strategies won't work. Also, it cannot be used to quantify the results of a program transformation targeting I/O optimization. There is no timing information associated with it which can be used by a good profiler.

And also, there are some I/O operations happening without an equivalent system call. Two examples are swapping in some memory pages and reading from mapped files, using `mmap`. None of them is captured by **strace**, making it unsuitable for the majority of related tasks.

On the other end of the specter, there is **iostat**, an utility displaying I/O usage information. It requires certain optional configurations in the kernel in order to work. However, almost all widespread distributions use those configurations, so this tool is easy to install.

There are also several configurations not enabled by default in the kernel and not used by the distributions but not vital for the application. For example, if `CONFIG_TASK_DELAY_ACCT` is not enabled in the kernel, then the swap actions will not be presented. Also, the tool would not be able to compute the actual percentage of I/O operations.

When started with no arguments, **iostat** shows a tabular view with each task from the system and the speed of writes and reads to the disk. It can be used to show the amount of I/O that processes had done since **iostat** was started. But that's the only thing it can do. It doesn't show filesystem related details, thus, it cannot be used alone for optimizations. From the following output of **iostat** – started in batch mode and showing only processes doing I/O – it is easy to see that not many things can be concluded from it. The speed of the disk is useful if one wants to develop theoretical models like *an inode is read in x ms*. Aside from that, none of the information presented is useful for an I/O profiler.

```

1 Total DISK READ: 22.63 K/s | Total DISK WRITE: 1202.98 K/s
2   TID  PRIO  USER      DISK READ  DISK WRITE  COMMAND
3   2180 be/4 mihai      0.00 B/s   26.40 K/s   chromium-browser
4    341 be/4 root       0.00 B/s   49.02 K/s   [kjournald]
5    893 be/4 root       0.00 B/s  131.99 K/s   [kjournald]
6   1722 be/4 mihai     22.63 K/s  116.90 K/s   gconfd-2
7   5974 be/4 mihai      0.00 B/s   45.25 K/s   audacious

```

Listing 3.2: **iostat** output

While the previous two tools can be used to get a quick overview of the application, there are also tools able to help in I/O optimization. Those presented in the following pages are several of them, the list is longer and it is increasing with time because of the novelty of the topic.

3.1 Iolog

Iolog is a small kernel module developed by the GNOME team in order to analyze and improve GNOME startup time ([4]). It uses an ext3 patch to log all block reads to the kernel log. Thus, when the system is starting, all reads are logged and can be retrieved later using **tail**, **dmesg** or any other program able to work with log files.

In addition to this, a script can be used to parse the log and merge contiguous reads into a single entry. The result, would be something along the following lines:

```

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

```

Listing 3.3: Iolog - parsed output

The last columns show pages read. For example, the first line suggests that **gdm** reads 32KB (8 pages of 4KB each) of data from **/usr/local/gnome/sbin/gdm-binary**.

Other parsers of the same data can provide statistics about seek count. We can get information about seeks done by a process, for example:

```

1      122 nautilus
2      109 gnome-session.r
3       74 bonobo-activati
4       49 gnome-panel
5       46 gnome-settings-

```

Listing 3.4: Iolog - seeks by process

Thus, when the Gnome environment is started, the **nautilus** process does the greatest amount of seeks.

If we want to find the file that is read by using the maximum number of seeks, we can use another parsing script, giving the following results.

```

1      26 /usr/local/gnome/lib/libgtk-x11-2.0.so.0
2      10 /usr/local/gnome/lib/libxml2.so.2
3       8 /usr/lib/libstdc++.so.6.0.5
4       8 /usr/lib/libstdc++.so.5.0.7
5       7 /usr/local/gnome/lib/libpoppler.so.0

```

Listing 3.5: Iolog - seeks by file

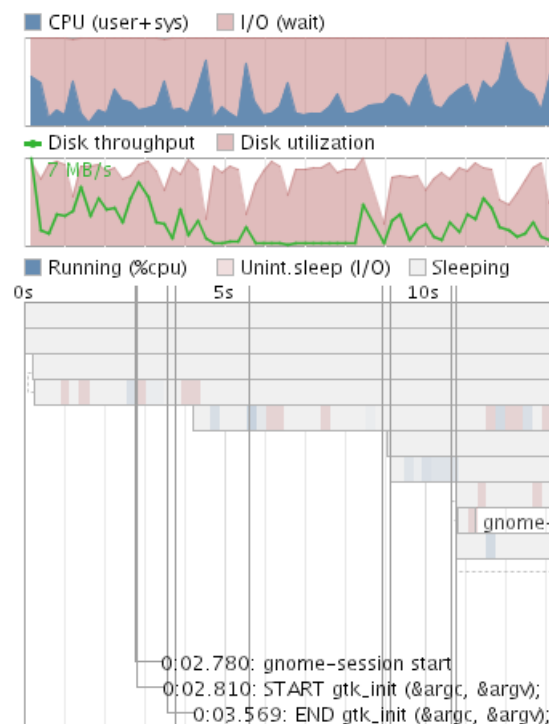
It seems that the GTK library is the least optimized. It was to be expected since this is the file from where all GUI resources are loaded. Also, there lies the code for the GTK functions and callbacks.

Using the **iolog** tool and the appropriate parsing script, one can discover every detail about what happens at the startup of GNOME. However, this is a costly solution since the kernel log is cluttered with information about all I/O operations.

3.2 Bootchart

Bootchart is a program developed by Ziga Mahkovec to monitor the boot process of a Linux system. It is composed from a shell script daemon and a java program.

The daemon monitors various system and process parameters at a regular interval, putting the collected data into a series of log files. The java program analyzes and works with those files, producing a chart showing the use of system resources by various processes over time. [Figure 3.1](#) shows a part of the output obtained while running **Bootchart** when starting the GNOME environment.

Figure 3.1: Output of **Bootchart** while starting the GNOME environment

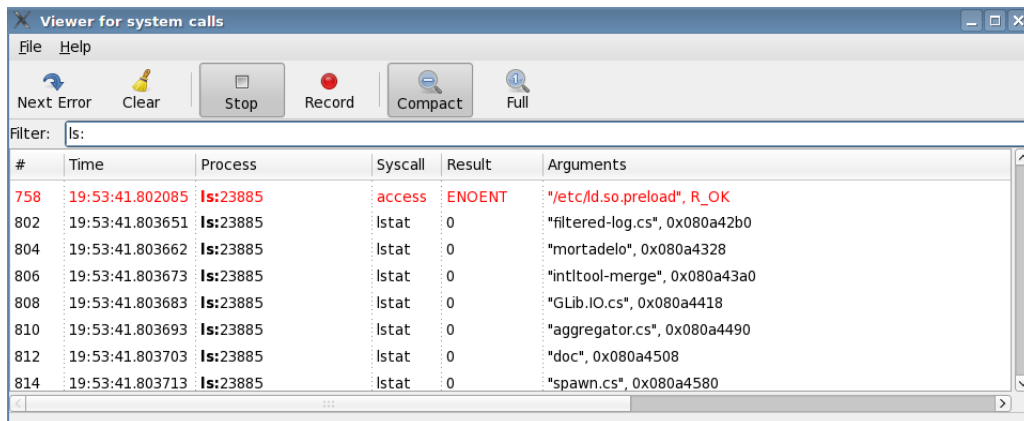
It shows disk and CPU utilization and displays the activity of each task. Thus, it is easy to see which thread does I/O at a specific moment in time. It also shows specific events – called milestones – from the application. To have them, the traced application must write strings of the form **xxx.yy <string>** to a file named **milestones.log**. This message is to be interpreted as follows: the **xxx.yy** part represents the system uptime, taken from **/proc/uptime**. The **<string>** part is a text string describing the event. So, an event is just a location in the source code.

Although very useful by itself, **Bootchart** cannot be used to monitor a single process, thus it is of little use when trying to optimize a single application. Also, as people from GNOME found ([4]), one will have to do a few modifications in order to use it for things other than the booting of a Linux system.

3.3 Mortadelo

Written by Federico Mena Quintero, **Mortadelo** is a system-wide version of **strace** with a graphical interface. Already, two of its main characteristics have been presented: system wide tracing and a GUI making easier to work with the collected data. It started as a simple clone of **Filemon**, a Windows utility equivalent to a system-wide **strace**.

Both of them are mainly used to detect why a program crashes, usually what file is missing, as can be seen from Figure 3.2. However, **Mortadelo** evolved and can also be used to application optimization, if one wants to do this. **Mortadelo** can be used to optimize an application because it has the ability to filter for only a subset of events or events touching a single file. In fact, filtering is done via regular expressions, making it a more powerful tool than **strace**.



The screenshot shows the 'Viewer for system calls' window. The 'Filter' field contains 'ls:'. The table below displays the captured system calls for the process 'ls' (PID 23885).

#	Time	Process	Syscall	Result	Arguments
758	19:53:41.802085	ls:23885	access	ENOENT	"etc/ld.so.preload", R_OK
802	19:53:41.803651	ls:23885	lstat	0	"filtered-log.cs", 0x080a42b0
804	19:53:41.803662	ls:23885	lstat	0	"mortadelo", 0x080a4328
806	19:53:41.803673	ls:23885	lstat	0	"intitool-merge", 0x080a43a0
808	19:53:41.803683	ls:23885	lstat	0	"GLib.IO.cs", 0x080a4418
810	19:53:41.803693	ls:23885	lstat	0	"aggregator.cs", 0x080a4490
812	19:53:41.803703	ls:23885	lstat	0	"doc", 0x080a4508
814	19:53:41.803713	ls:23885	lstat	0	"spawn.cs", 0x080a4580

Figure 3.2: **Mortadelo**

Nevertheless, it is still a **strace** equivalent. Although it solves some of the problems of the above mentioned tools – mainly the ability to filter and be used only for a single application –, the collected data is just the data coming from **strace**. Which means, not all of the relevant events are captured. All virtual memory events are lost, thus no I/O caused by memory mapped files can be logged.

This tool uses a SystemTap script to gather data making him a very close friend of **Iogrand**. It was featured in one edition of LWN ([5]). However, its repository ([7]) shows no activity since 2009.

3.4 Icegrind

Developed for optimizing **Firefox** (see [2]), **Icegrind** is a **Valgrind** plugin created to look at the difference that a good binary layout makes to the loading time (see section 2.1.5 for the exact reasons while library layout matters).

This program needs small changes in the way one compiles the application: instead of being compiled normally, two more flags, `--ffunction-sections -fdata-sections`, must be passed to **gcc**. Also, the binary should be created using a prelinker by invoking the following command while building:

```
1 prelink $LD_LIBRARY_PATH/firefox-bin $LD_LIBRARY_PATH/*.so
```

Listing 3.6: Icegrind - prelink command

Then, we need a description of interesting files obtained by using another utility written by the same author. After this, the application is run under **Icegrind** producing a log file for every memory mapped file with sections described as above. Combining these results, one could use the **gold** linker and tell him to reorder objects for best results.

While this tool was intended to a single purpose – reducing application startup time by re-ordering objects in library –, it is the closest to a real tool in the field of I/O optimization. It collects data from the running application and stores it for later improvement phases, just like any good profiler will have to do.

Because it works as a **Valgrind** plugin, this tool is also a close friend of **Iogrind**, coming closer to an I/O profiler. However, the most important thing about **Icegrind** is the fact that it gives hints to the linker in order to optimize the startup of the application.

Chapter 4

Iogrind

The **Iogrind** program is another tool for I/O optimization. It was originally created by Michael Meeks. The main purpose of it was to give an accurate description of all the I/O an application would perform on a cold startup.

4.1 Iogrind on Valgrind

The initial version of this tool was centered on **Valgrind** (see repository at [8] or the original project's page at [9]), just like the before mentioned **Icegrind**. However, these two tools are orthogonal in their activity. While one tries to give hints to the linker, the other tries to help the developer to figure out new methods in order to optimize the I/O part of an application or a set of them. The difference between these tools was presented in a clear way by Michael at various conferences (for example, two very enlightening talks are [11] and [12]).

Initially, it was used to trace the application, logging it into a temporary file. After this, using **ext2dump**, the filesystem is saved to an XML file. Both parts are needed by the third: the visualization application. The entire process is described by Figure 4.1. By combining the log with the layout of the filesystem, the GUI can show relevant details which can provide optimization hints.

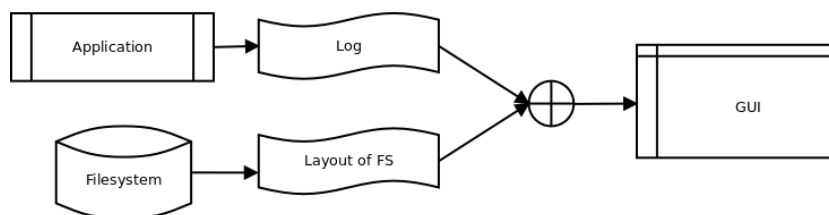
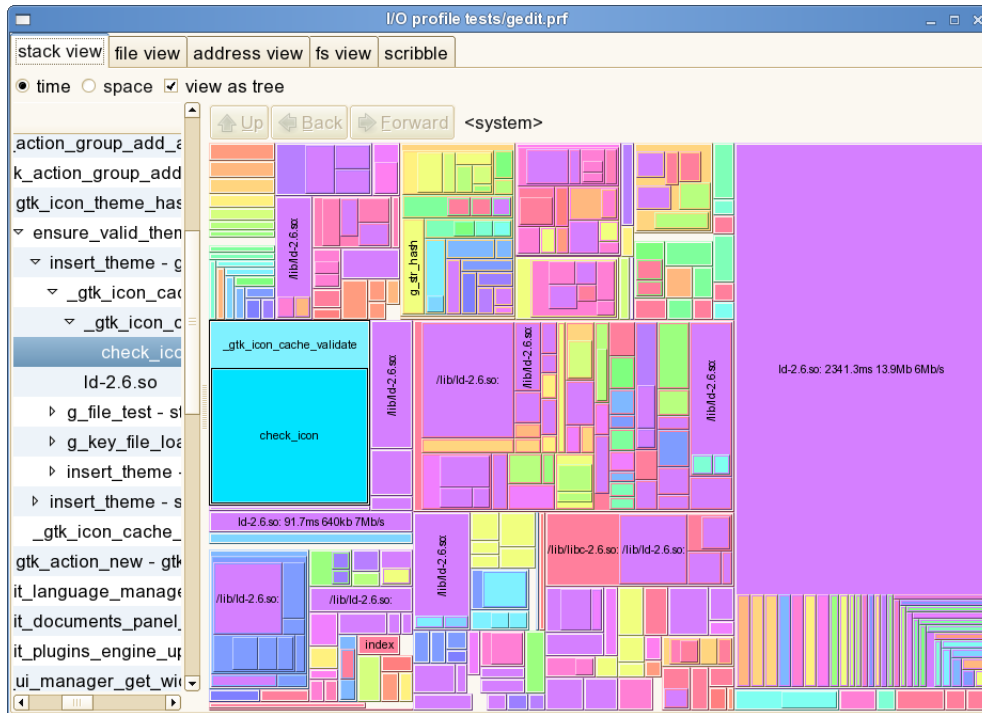


Figure 4.1: Original architecture of **Iogrind**

Iogrind had several visualization modes. To start with, it was possible to draw a histogram with stack frames in a tree view, just like in Figure 4.2. The area of each rectangle represents simulated time. This can be used to detect which files are loaded slower. Hence, it can be used as a hint: the largest area rectangle should be the first target to the optimization process.

In our case, the trace showed there was taken while starting **gedit**. Aside from the big rectangle that is **ld.so**, we see that the application is spending much time loading icons (**check_icon**).

Figure 4.2: Stack view of **Iogrind**

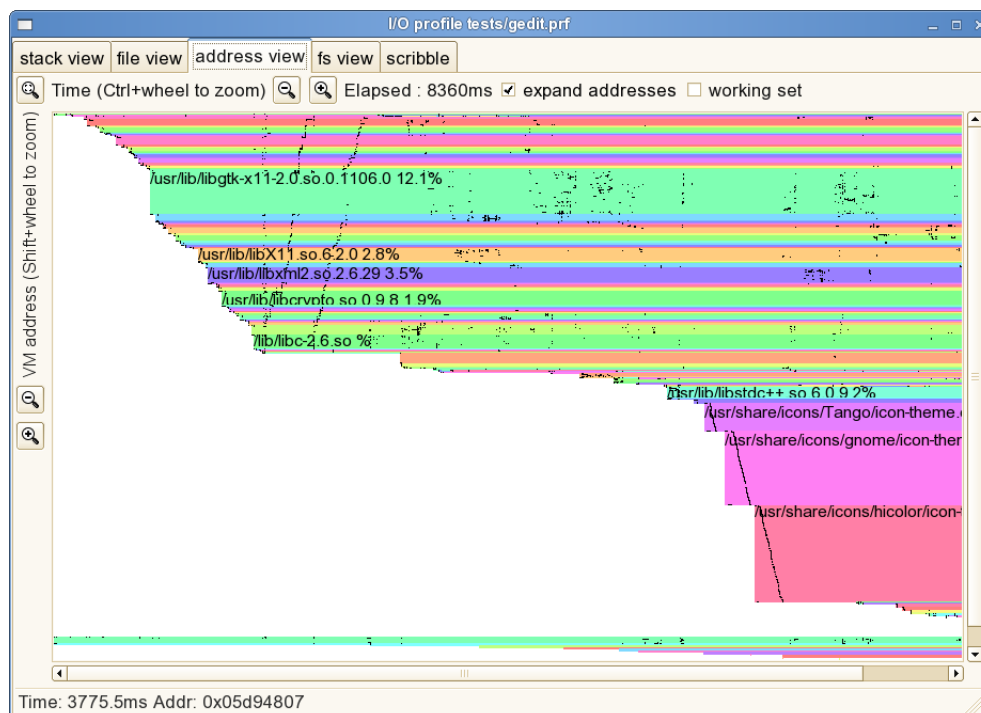
Thus, even from a simple view like the one in Figure 4.2, one can quickly determine what needs to be improved and why the application is taking so long to start. In our case, there seem to be too many icons loaded at once. Thinking at the strategies mentioned in Chapter 2, we see that we can benefit from lazy resource loading. Maybe, the layout of the images is what makes the application behave so poorly and this is what we have to work on. The developer will improve the application according to one of these ideas. Then, he/she will test again to see what happens.

Another view is the address view. Just like in Figure 4.3, it is possible to see the address space of the process and how it is accessed over time. Each black dot in the view represents a touch of a page, an action causing I/O. Using this, one can see how each library is accessed and decide which one to target for optimization. For example, `/usr/lib/libgtk-x11.so` seems to be accessed in a totally random way, making it a suitable target for optimization. On the other hand, `usr/share/icons/hicolor/icon.so` seems to be accessed in a linear way, thus it is safe to let that file unchanged.

It is possible to show a scribble view of the filesystem, just like in Figure 4.4. Each red line represents a seek on the hard disk, each small rectangle represents a file. Ideally, there should be as few red lines as possible, covering the smallest possible area. This is not the case for **gedit**, as it is seen in Figure 4.4. The developer optimizing this application will have to be careful with this aspect.

Thus, we see that from three views obtained from a single run of the application one can deduce enough facts about the tested code. Now, he/she can go and optimize it using the above facts as hints on what to try first. This project seems to be very useful, giving very relevant hints.

However, there were several problems with this architecture. It needs a simulation of the disk, and the one implemented is highly inaccurate. Because there are a lot of pieces missing – per-file-system heuristics for example – no write operation can be simulated. Moreover, threaded applications or applications forking other processes cannot be traced realistically, because of

Figure 4.3: Address view of **Iogrind**

the same reasons.

Using **Valgrind** provided a way to obtain a good trace of the application. Adding a simulation of the disk allowed this tool to be created. However, accesses to memory mapped files was not realistically simulated. This and the other problems mentioned above required a new version, written from a new perspective.

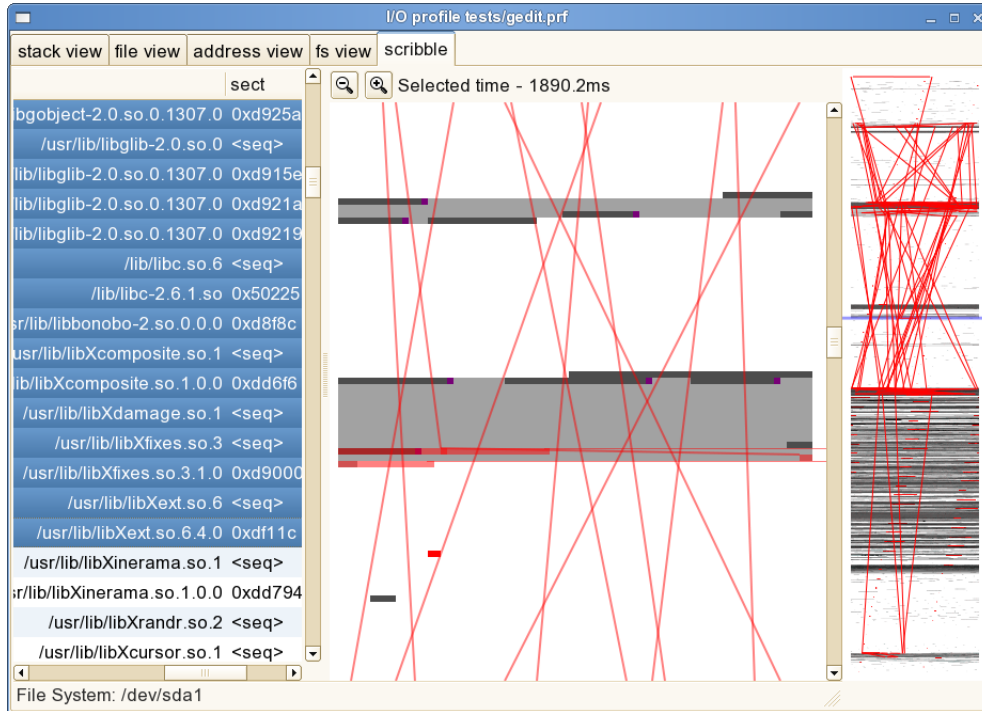
4.2 Tracing Iogrind

The current version – the one presented by this article – started from a simple idea needed to avoid inaccuracies in disk simulation. Basically, the layout of the filesystem is constructed using a set of system calls. Each application is another set of system calls. By tracing both sets, one can decide to use one of them in order to improve both activities.

Even if tracing the filesystem’s creation is not possible, having a reliable trace of the application makes it possible to understand what is happening there, what has an impact on performance, etc. Having this trace, one can optimize the application. Thus, the new version of Iogrind will have to take this into account.

Although presented at various conferences, the initial version didn’t gather enough interest from developers. The project stagnated and lost contact with **Valgrind**’s development. During this time, while thinking at the above mentioned problems, Michael decided that the **Valgrind** way is not the best alternative for this tool. Instead, something looking at the kernel events was desired. It is the only way in which a reliable trace can be obtained.

In order to obtain it, knowledge of Linux kernel is needed. And a way to instrument a running application, looking for events in kernel space, is the important piece of the **Iogrind**

Figure 4.4: Scribble view of **Iogrind**

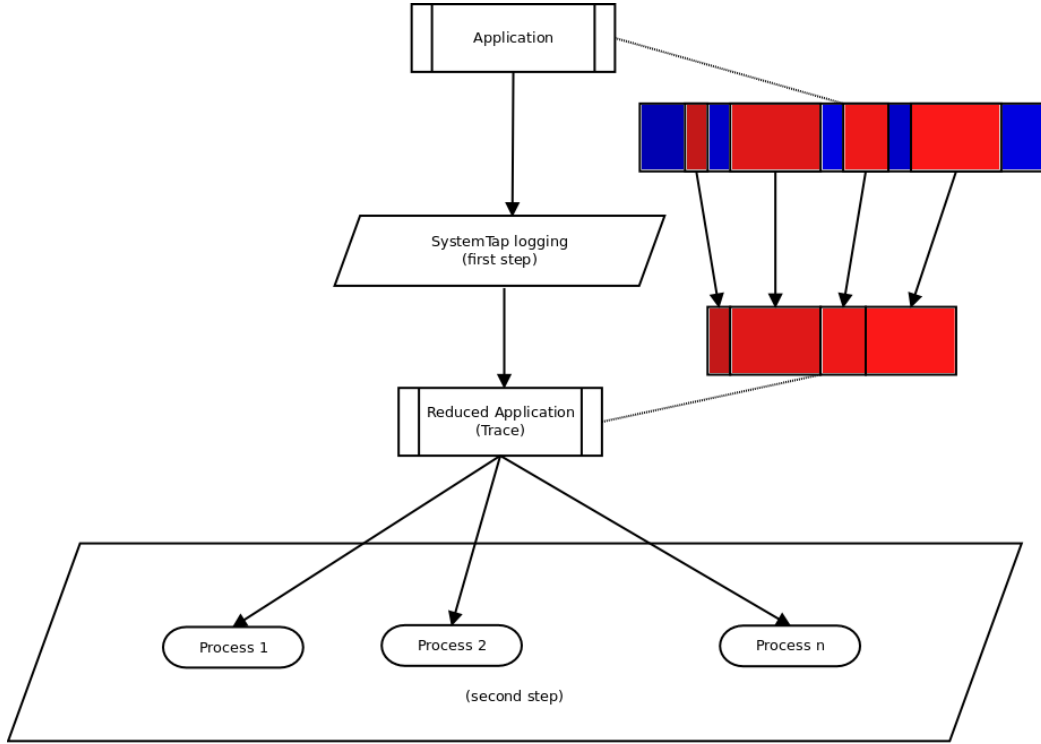
project. From all the possible alternatives, using **SystemTap** was considered to have the best architectural advantage.

The initial version allowed only playing with the obtained trace. However, if the trace can be turned into a valid C program, running it instead of running the application will have the same I/O behavior but will take a shorter amount of time. Thus, this was the main goal of the new version. The graphs obtained in the first version can also be obtained from this version's trace by writing the appropriate parsers.

The main aspect, however, is that the trace should allow someone to execute a reduced instance of the original application, stripped down to the I/O part and the relevant glue calls. The easiest way for this would be to obtain it as a source file in some programming language. We have chosen to offer a valid C program as the trace of the application. This way, if someone wants to use **Iogrind** only for benchmarking he/she will obtain – after a single run of the original application – everything he/she needs to get a deterministic and reproducible benchmark.

Moreover, since we don't want to be limited only to benchmarking, we would like to allow other tools to use this trace and offer various outputs to the user. This means that some metadata will have to be stored in the trace: information irrelevant to the trace when viewed as a source file but very important when viewed as a real trace. For example, we will store in comments the return value of almost all operations. If they are needed, no one would have to rerun the application or the trace to obtain it, a simple parsing will suffice.

The entire architecture is presented in Figure 4.5. We see that there are two phases very loosely connected. The first phase takes care of reducing an entire application to another application equivalent to the first from the I/O point of view but doing no CPU operations unless they are a prerequisite to the I/O ones. The second phase is not clearly delimited: it involves working with the reduced executable or with its source code; one can decide to parse that source code to obtain nice graphical representations of the original application; another one may decide to use the executable to do benchmarks for the optimization strategies he/she has used; or, one

Figure 4.5: Architecture of **Iogrind** with SystemTap logging

can try to optimize the reduced application. Taking into account the isomorphism between the original and the reduced executable, optimizing one of them will optimize the other too.

Breaking the program into two phases, trace obtaining and trace parsing, has one additional benefit. Since kernel events need to be intercepted, root privileges are needed to run the tracing part. Requiring them for the entire application runtime is a risky thing, reducing the requirement only to the trace obtaining step increases security a little. The user parsing the trace needs only the trace and the parsing script. No additional privileges or resources are needed.

Moreover, parsing the trace can be done on another machine if needed. This increases security: one can obtain it by running the application on a virtual machine and copying the resulted file to the local machine. A malevolent application will only break the virtual machine, leaving the user unharmed. As a side note, this can also be used to audit an application, to create an antivirus, etc. Of course, with several more enhancements but the core is already constructed.

Finally, having the trace as a C source file allows combining several programs into a single one in order to test a big workload of applications. The resulting program can then be used to benchmark kernel optimizations. For speed and safety reasons, this benchmark should be done using another virtualization method. The current architecture of the project makes it possible.

Chapter 5

Trace Obtaining

As suggested in the previous chapter, in order to obtain a good tool for I/O optimization, one will have to get a trace of kernel space events happening while an application or a set of applications is running.

Not all kernel events are important, though. We are interested in all I/O-related events. This means reads and writes to the disk. However, to obtain something human readable we will also need some glue calls. That means that opening files, mapping files in memory and other similar operations need to be intercepted. All of these events will have to be captured into the trace.

5.1 SystemTap

However, in order to obtain it we have two options: either the relevant places in the kernel are patched to log the relevant information or a logging framework is used. We have decided to take the second path as it was safer and allowed the application to stay up-to-date with the newer kernel versions.

From the entire suite of kernel logging frameworks, **SystemTap** was selected. The main reason for the selection was the fact that writing the probes for the traces is a very simple thing to do, thus the application can be extended very easily. The exact description of probes will be given in [section 5.2](#)

Basically, **SystemTap** was created to eliminate the need for the developer to go through the tedious task to recompile, install and reboot that is required when trying to collect data from kernel. It provides a simple command line interface and a simple scripting language for instrumenting a live system.

One can instrument a single program, a tree of processes, all processes started by a single command or all processes in the system. Even kernel threads with no equivalent in userspace can be traced.

Although it is used by a lot of applications (**Mortadelo** being one of them as presented in [section 3.3](#)), there are several religious wars started by kernel developers and **SystemTap** developers. This is because it is very similar to the **dtrace** tool found on Solaris with regard to the provided features, yet its internals are not as nicely laid out as its competitor's. For a comparison between these two tools, see [1].

In order to use **SystemTap**, one will have to write a simple script. For example, the following listing can be used to intercept all system calls.

```

1 probe syscall.*
2 {
3     printf("%s->_%s(%s)\n", thread_indent(1), probefunc(),
4         argstr)
5 }
6 probe syscall.*.return
7 {
8     printf("%s<-%s\n", thread_indent(-1), probefunc());
9 }

```

Listing 5.1: SystemTap script to intercept all system calls

Because `thread_indent` was used, each nested call will appear a little to the left of its parent. This function returns a string with enough spaces to do this. A portion of a sample run is given in the following listing

```

1      0 sshd(793): -> sys_write(3, ":\345\0051\376\2161\021W\327?\363\232\351\345"... , 16432)
2 13587 sshd(793): <- sys_write
3      0 sshd(793): -> sys_select(10, 0x220df3e0, 0x220df3d0, 0x0, NULL)
4 164 sshd(793): <- sys_select
5      0 sshd(793): -> sys_rt_sigprocmask(SIG_BLOCK, [SIGCHLD], 0xbfff848c, 8)
6 22 sshd(793): <- sys_rt_sigprocmask
7      0 sshd(793): -> sys_read(9, 0xbfff442c, 16384)
8 131 sshd(793): <- sys_read
9      0 sshd(793): -> sys_mremap(0xb751a000, 823296, 856064, MREMAP_MAYMOVE, 0xd1000)
10 28 sshd(793): <- sys_mremap
11      0 sshd(793): -> sys_write(3, "3u\255q;\254\006_\354\262\006\235\345d\320o3\"... , 16432)
12 5808 sshd(793): <- sys_write
13 0 sshd(793): -> sys_select(10, 0x220df3e0, 0x220df3d0, 0x0, NULL)
14 104 sshd(793): <- sys_select
15 0 sshd(793): -> sys_rt_sigprocmask(SIG_BLOCK, [SIGCHLD], 0xbfff848c, 8)
16 141 sshd(793): <- sys_rt_sigprocmask

```

Listing 5.2: Intercepted system calls

A complete description of **SystemTap** can be found on the project's wiki ([13]). From now on, we will concentrate on the features used while creating **Iogrind**.

5.2 Probing

Each interception is done by writing code into a SystemTap probe. Basically, it looks like a function from any normal programming language.

There are a variety of probes. One can install callbacks to be called after a certain amount of time has passed, when the SystemTap script starts or is finished, on events of the I/O block layer, on events from the VFS, on the system call interface, etc. If none of the previous is helpful, he/she can also install probes to functions or even line numbers. However, doing the latter make the script non-portable across kernel changes. Installing probes at function-level interface is portable as long as those function are from a public interface, otherwise they could be changed overnight and the entire script will fail.

Each of the probes can be attached to the entry point of the function or to the exit point. Attaching to the entry point gives access to all local variables declared in the function, attaching to the exit point gives access to the return value. Both methods allows one to access the original parameters when the function was called, even if they are changed inside it. If one needs access to a specific value of a variable which is changed in a function, he/she will have to use a probe linked to a specific instruction. This is the only place where that kind of probes can be useful.

The following listing will give an example of the `begin` probe. In **Iogrind**, this probe was used to build the prologue of the log, the constant part of the generated C source file. That means

generating a warning comment, generating the inclusion of headers and printing the beginning of the main function.

```

1 probe begin
2 {
3     buf_count = 0
4     printf("/*_Autogenerated:_DO_NOT_EDIT!_*/\n\n")
5     printf("#include<unistd.h>\n")
6     printf("#include<fcntl.h>\n")
7     printf("#include<sys/mman.h>\n")
8     printf("\nint_main()\n{\n\tchar_a;")
9 }
```

Listing 5.3: SystemTap: begin probe

Looking at the function, we see that a variable – `buf_count` – is declared there. It is be used as an unique index for the various buffers which will be presented in the output file. In order to be visible, it has to be declared global, using a syntax like the following listing.

```

1 global buf_count
```

Listing 5.4: SystemTap: global variables

Another probe is the `never` probe. It will never be executed but it is useful to initialize some global variables to a default value. The initialization can be done while running the script via the command line but a warning will be given if it is not done in the SystemTap source.

```

1 probe never
2 {
3     targetPID = 1
4 }
```

Listing 5.5: SystemTap: never probe

The following listing gives an example of a normal system call interception. In this example, we chose to intercept `read`.

```

1 probe syscall.read.return
2 {
3     # Only in traced process
4     if (targetPID != pid())
5         next
6
7     # Only if opened as file
8     if (!($fd in opened_files))
9         next
10
11     printf("\n\tchar_buf%d[%d];\n", buf_count, $count)
12     printf("\tread(%d,_buf%d,_d);_/*_s_*/\n", $fd, buf_count,
13         $count,
14         opened_files[$fd])
15     buf_count += 1
16 }
```

Listing 5.6: SystemTap: Intercepting read

When the probe is activated we check if the process entering it (obtained via `pid()`) is the same as the traced one (remembered in the global `targetPID` variable). If not, the probe is skipped entirely using `next`. Otherwise, we check if the file descriptor being read corresponds to a previously opened file. Since `fd` is the name of the argument corresponding to the file descriptor in the function implementing the `read` system call, we will retrieve it using `$fd`. We also have a global associative array of opened file descriptors – `opened_files` – and we check to see if the received descriptor is a key in that array.

If everything works as presented above, in the generated traces we will print two lines. One declaring a buffer – this is where `buf_count` variable is used – and one issuing the actual `read` system call. The second line has a comment in which the filename being read is indicated. The return value is not relevant since, in the trace, the actual data read is not needed. How the mapping is obtained will be explained in [section 5.4](#).

5.3 Pagefault Handling

Probing for system calls or specific functions is simple. It is easy to understand what happens in those cases. In fact, it will give results similar to a **strace**.

However, what almost all of the tools presented in [chapter 3](#) lacked was a way to handle pagefaults. A pagefault can happen because copy on write is used and the page needs to be copied or when the access is invalid. However, it can also happen when the page needs to be swapped in or there is a file mapped at that location in memory and data needs to be read from the file.

```

1 probe vm.pagefault.return
2 {
3     # Only in traced process
4     if (targetPID != pid())
5         next
6     # If there's no back-up file, skip.
7     if (!$vma->vm_file)
8         next
9     # Find the mmap which includes the address and is from the
       good file then leave
10    valid = 0
11    foreach (ix in mmap_starts)
12        if (mmap_starts[ix] <= $address && mmap_ends[ix] >
            $address) {
13            valid = 1
14            break
15        }
16    if (!valid)
17        next
18
19    printf("\ta_=(map%d_+_%d);_/*_s_*/\n", ix,
20        $address - mmap_starts[ix], files_mapped[ix])
21 }

```

Listing 5.7: SystemTap: pagefault handling

The later case is what makes pagefault handling an important part of any tool devised to help I/O optimization. If it is missing, a great part of the application behavior will be hidden. If it

is not implemented properly, the end result will be a useless trace, usually having more bogus data than real information.

In our implementation, we intercept all pagefaults. Since we are only interested in mapped files, we look around at the parameters and a little knowledge of the kernel helps us to distinguish between a pagefault happening because the page belongs to a mapped file or one produced for any other reason.

Even after finding that the page belongs to a mapped file, the work is not done yet. We have to detect the exact mapping of the file in order to reproduce it. Thus, we will check each map to see if the fault address belongs to the mapped area. If one is found, we extract its index and touch the page using it.

This scheme doesn't handle swapping yet, because doing this comes with a small problem. Usually, swapping depends on the entire OS context, making it almost impossible to reproduce the entire context exactly. We have decided to ignore this, therefore Iogrand does not implement swap tracing.

5.4 Glue Calls

As mentioned at the beginning of the chapter, several glue calls need to be intercepted if one wants to reproduce the entire I/O behavior. For example, we are interested in opening and closing of files in order to distinguish between writes to a socket or to a terminal and writes to a file (see [listing 5.6](#) for an example on how we make this distinction). The following listing shows how it is done in case of open:

```

1  # Intercept the opening of files and record their fd.
2  probe syscall.open.return
3  {
4      # Only in traced process
5      if (targetPID != pid())
6          next
7
8      # Only if succeeded
9      ret = $return
10     if (ret < 0)
11         next
12
13     fname = user_string2($filename, "<?>")
14     opened_files[ret] = fname
15     printf("\n\topen(\"%s\",_%d,_%d);_/*_%d*/\n", fname, $flags
16         , $mode, ret)

```

Listing 5.8: SystemTap: open

We take the return value – `$return` – and test to see if the opening succeeded. If not, since we are not interested in non-existent files we leave the probe. Otherwise, we read the filename using `user_string2` function, one of the many string functions that SystemTap offers. Then, we store that filename in the `opened_files` structure, where it will be searched later when I/O is done via the obtained file descriptor.

When looking at the virtual memory and pagefault handling, we see that we need several more structures. This time, since we can unmap only a portion of a previously allocated area, the

situation is not that simple, requiring several probes to be installed and several data structures, as shown in Figure 5.1.

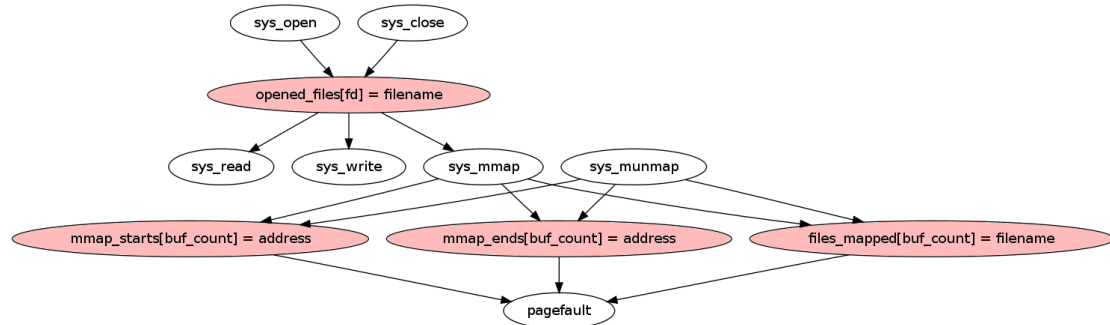


Figure 5.1: Structures and probes used in **Iogrind**

Each of the coloured nodes represent a dictionary data structure used to keep information from one probe to the other. The key for each dictionary is given in square brackets while the value is given after the equal sign.

Each arrow represents a read-write dependency. If an arrow leaves a probe and enters a dictionary, then that probe changes the dictionary by removing or adding at least a key. Otherwise, if the arrow leaves a data structure then that dictionary is used read-only by the probe receiving the arrow.

5.5 Invoking

Running the script is the actual implementation of the trace collecting step of Iogrind. It can be run with a single command: **stap script**. However, this has several disadvantages.

To start with, running everything from a single command will recreate and recompile a kernel module. The script is parsed, transformed to a C source file, compiled to a kernel module and, after this, the module is inserted into the kernel and the tracing begins. Usually, we don't want to wait at every trace collecting step the entire amount of time needed to recompile the module. This is easily solved by using a few more arguments, telling to stop after a certain step in the elaboration and to keep the module instead of deleting it - by default, SystemTap deletes all generated files after the tracing is stopped. Thus, one would use the following command line **stap script -p4 -m module**. After this, the module can be inserted at any time using **staprun module**.

We want to trace a single process. While SystemTap offers command-line options to enable filtering by PID, there are still several calls belonging to other processes which will be intercepted. Since they are not useful to the trace, each probe in the script should do the filtering by itself. This is why we have a global `targetPID` argument.

In order to initialize it to the needed value, we will define it when running **staprun**. However, running from the command line will not work this time. It is possible that the trace application run a little before the SystemTap module is inserted and the tracing begins. Thus, it is possible to lose a relevant portion of the startup.

To solve the above problem, a launcher program was created. It collects the program to be run and its arguments and it makes sure that when it is started the module is already inserted and set up to trace calls from the program. In order to do this, we use the following idea, as suggested by Figure 5.2.

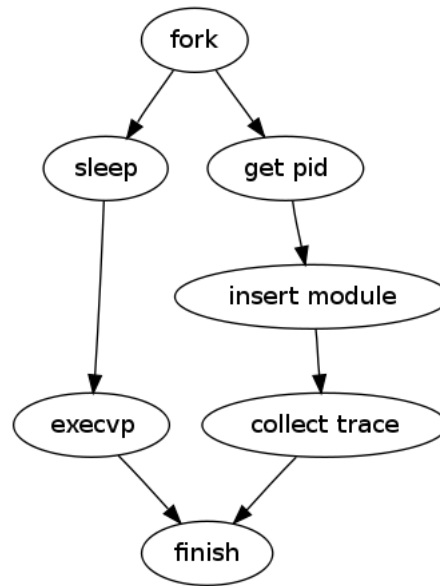


Figure 5.2: Launching an application for tracing

We do a `fork` inside the launcher. This gives the PID of the child process to the parent. After this, the parent can insert the module because it knows all necessary parameters. The child process cannot immediately start the traced program, we don't want to lose a part of the startup process. Instead, the child does a short sleep, enabling us to be sure that the tracing is set up. After this, the application is started by doing a `exec`. The following listing gives a stripped down version of the launcher program.

```

1 pid = fork();
2
3 if (pid == 0) {
4     /* wait before executing the real thing */
5     sleep(1);
6     execvp(argv[1], args);
7 }
8
9 sprintf(stapstr, "staprun_iogrind.ko_o_%s\"LOG_PREFIX\"_targetPID=%d"
10         , argv[1], pid);
11 system(stapstr);

```

Listing 5.9: SystemTap: starting trace

There is a problem with synchronization by sleeping. However, there is no proper way to set up the tracing. After the **staprun** command is issued, that process is blocked until the tracing stops. Thus, any synchronization mechanism must signal the insertion of the module before it is inserted. It seems that race conditions are unavoidable here. Testing suggested that the sleeping method works fine. Future improvements will change this method.

Chapter 6

Tracing Results

Several command line applications were tested using Iogrind. We limited ourselves to CLI because the tracing was done on a virtual machine and it was easier not to install a window manager.

Testing was done on a VMware virtual machine, running a 2.6.35 Linux kernel. Each application was traced consecutively. Because the tracing is independent of whether we have a cold startup or not there was no need to reboot the machine. Relevant aspects from the traces will be given below.

To compare the efficiency of the tracing idea, each traced application was tested. After each reboot, the application and the trace were run and the running time was collected. Results were averaged over 10 runs. The same procedure was applied for subsequent startups. In the end, all the data looked similar to the one in the following tables.

Table 6.1: Timing cold startup of application and trace (in seconds)

Name	Application	Trace
ack-grep	0.231	0.111
dpkg -l	0.198	0.066
grep	0.296	0.059
tig	0.913	0.027
vim	1.850	0.052

Table 6.2: Timing warm startup of application and trace (in seconds)

Name	Application	Trace
ack-grep	0.194	0.042
dpkg -l	0.156	0.034
grep	0.084	0.026
tig	0.384	0.014
vim	0.529	0.016

It is easy to see that running the trace took a much shorter time than running the entire application, for both tested scenarios. Looking at the times, we see that one developer can run

many traces in the same amount of time that will be needed to run the entire application. This can only lead to reduced development cycles.

Looking at the trace for **ack-grep** – for example –, we see that there are many kinds of I/O activity. To start with, we can see how the OS maps each library in order to launch the application. It is easy to understand the information about page touches. As an example, the following listing gives a part of the library-caused I/O trace.

```

1 char *map16;
2 map16 = mmap((void *)0x401fb000, 2097152, 1, 2, 3, 0); /* /usr/lib/locale/
   locale-archive, 0x401fb000 */
3 a = *(map16 + 16); /* /usr/lib/locale/locale-archive */
4 a = *(map16 + 6564); /* /usr/lib/locale/locale-archive */
5 a = *(map16 + 11101); /* /usr/lib/locale/locale-archive */
6 a = *(map16 + 19848); /* /usr/lib/locale/locale-archive */
7 char *map17;
8 map17 = mmap((void *)0x40021000, 4096, 1, 2, 3, 0); /* /usr/lib/locale/
   locale-archive, 0x40021000 */
9 close(3); /* /usr/lib/locale/locale-archive */
10 a = *(map16 + 103872); /* /usr/lib/locale/locale-archive */
11 a = *(map16 + 135944); /* /usr/lib/locale/locale-archive */
12 a = *(map16 + 272004); /* /usr/lib/locale/locale-archive */
13 a = *(map17 + 0); /* /usr/lib/locale/locale-archive */
14 a = *(map16 + 360208); /* /usr/lib/locale/locale-archive */
15 a = *(map16 + 1329588); /* /usr/lib/locale/locale-archive */
16 a = *(map11 + 145104); /* /lib/libc.so.6 */
17 a = *(map8 + 41008); /* /lib/libpthread.so.0 */

```

Listing 6.1: Library I/O from **ack-grep**

Another kind of I/O operation done by **ack-grep** is caused by the fact that its code is interpreted. Hence, many files containing common functions need to be loaded.

```

1 open("/usr/share/perl5/App/Ack.pm", 32768, 0); /* 4 */
2 char buf26[4096];
3 read(4, buf26, 4096); /* /usr/share/perl5/App/Ack.pm */
4 open("/usr/share/perl5/File/Next.pm", 32768, 0); /* 5 */
5 char buf27[4096];
6 read(5, buf27, 4096); /* /usr/share/perl5/File/Next.pm */
7 char buf28[4096];
8 read(5, buf28, 4096); /* /usr/share/perl5/File/Next.pm */
9 open("/usr/share/perl/5.10/File/Spec.pm", 32768, 0); /* 6 */
10 char buf29[4096];
11 read(6, buf29, 4096); /* /usr/share/perl/5.10/File/Spec.pm */

```

Listing 6.2: I/O caused by interpreted code

The last part of the I/O operations is represented by the real operations done by the application. The process launched by **ack-grep** needs to read all files in the current directory and its subdirectories. We can see a part of those reading in the following snippet from the same trace.

```

1 open("iogrind.stp", 32768, 0); /* 3 */
2 char buf92[4096];
3 read(3, buf92, 4096); /* iogrind.stp */
4 close(3); /* iogrind.stp */
5 open("iogrind.stp", 32768, 0); /* 3 */
6 char buf93[4096];
7 read(3, buf93, 4096); /* iogrind.stp */
8 close(3); /* iogrind.stp */

```

```
9  open("1-log.c", 32768, 0); /* 3 */
10 char buf94[4686];
11 read(3, buf94, 4686); /* 1-log.c */
12 char buf95[4096];
13 read(3, buf95, 4096); /* 1-log.c */
14 open("1-log.c", 32768, 0); /* 4 */
15 char buf96[4096];
16 read(4, buf96, 4096); /* 1-log.c */
17 close(4); /* 1-log.c */
```

Listing 6.3: I/O requested by programmer

The above snippet shows a bug within **ack-grep**. A file descriptor seems to be leaking or the same file is opened multiple times. Someone will have to look into this to see what is really happening.

The complete trace wasn't given here because it was very long. However, a simple program and its trace are appended to this article.

Chapter 7

Improvements and Future Plans

The application is far from being completed. There are still several things to do. All of them are planned for the future.

To start with, there are several system calls acting as a glue between I/O operations which have not been analyzed so far. For example, `dup` or `clone` are not treated. There are applications that need this to be implemented.

Some of the implemented probes cut a few corners to make this prototype viable. For example, the implementation of `munmap` probe ignores partial unmapping of memory pages. Depending on the application, results following this may be wrong.

Both of the above problems can be fixed in a week or two of working on the project. They were left out for now because each of the tested application didn't use them.

Mapping files in memory and generating pagefaults in the traced source code is buggy. A good starting point for improvements to **Iogrind** is to rewrite those parts.

Another improvement would be to create a better mechanism for synchronization between the process running the **staprun** and the traced application. As mentioned in [chapter 5](#), finding a best solution for this problem is still an open problem.

Next, the plan is to create several scripts for displaying nice graphical representations for the traced data. It is something simple to do, one will have just to parse the trace file and look for the relevant events.

After this is implemented, the application will need a few changes in order to incorporate it. Either several flags will be added or another launcher will be created which will call the relevant part of the application, just like **git** does. It is possible to create a plugin interface which will allow graphical output applications to be simple plugins for the Iogrind application.

I will release the application under an Open Source license. Any improvements and suggestions will be taken into consideration. The plan is to create a valid framework for I/O optimization as soon as possible. A reliable infrastructure which will help in this task is what all developers need.

Appendix A

Test Application

```
1 #define _GNU_SOURCE
2 #include <sys/mman.h>
3 #include <fcntl.h>
4
5 #include <stdio.h>
6 #include <stdlib.h>
7
8 int main()
9 {
10     char c;
11     FILE *f;
12     char *p;
13     int fd;
14
15     fd = open("iogrind.stp", O_RDWR);
16     read(fd, &c, 1);
17     printf("read:_%c\n", c);
18     close(fd);
19
20     close(42); /* will fail but should not be recorded */
21
22     /* both of the following will fail and should be skipped */
23     fd = open("no_such_file", O_RDWR);
24     close(fd);
25
26     fd = open("1.c", O_RDWR);
27     p = mmap(NULL, 512, PROT_READ | PROT_WRITE, MAP_SHARED, fd,
28             0);
29     printf("map:_%s\n", p);
30     p = mremap(p, 512, 1024, MREMAP_MAYMOVE);
31     munmap(p, 512);
32     close(fd);
33     close(fd); /* will fail, should be skipped */
34
35     f = fopen("1.c", "r");
36     fscanf(f, "%c", &c);
37     printf("scanf:_%c\n", c);
```

```
37         fclose(f);  
38  
39         return 0;  
40     }
```

Listing A.1: Test Application

Appendix B

Trace Generated by Iogrind

```
1  /* Autogenerated: DO NOT EDIT! */
2
3  #include <unistd.h>
4  #include <fcntl.h>
5  #include <sys/mman.h>
6
7  int main()
8  {
9      char a;
10     open("/etc/ld.so.cache", 0, 0); /* 3 */
11
12     char *map0;
13     map0 = mmap((void *)0x40021000, 18476, 1, 2, 3, 0); /* /etc/
        ld.so.cache, 0x40021000 */
14
15     close(3); /* /etc/ld.so.cache */
16     a = *(map0 + 0); /* /etc/ld.so.cache */
17     a = *(map0 + 5900); /* /etc/ld.so.cache */
18     a = *(map0 + 13713); /* /etc/ld.so.cache */
19     a = *(map0 + 17245); /* /etc/ld.so.cache */
20     a = *(map0 + 8468); /* /etc/ld.so.cache */
21
22     open("/lib/libc.so.6", 0, 0); /* 3 */
23
24     char buf1[512];
25     read(3, buf1, 512); /* /lib/libc.so.6 */
26
27     char *map2;
28     map2 = mmap((void *)0x40026000, 1427880, 5, 2050, 3, 0); /*
        /lib/libc.so.6, 0x40026000 */
29     a = *(map2 + 1415580); /* /lib/libc.so.6 */
30     a = *(map2 + 1412476); /* /lib/libc.so.6 */
31     a = *(map2 + 1412520); /* /lib/libc.so.6 */
32
33     close(3); /* /lib/libc.so.6 */
34     a = *(map2 + 440); /* /lib/libc.so.6 */
35     a = *(map2 + 75898); /* /lib/libc.so.6 */
```

```

36     a = *(map2 + 80868); /* /lib/libc.so.6 */
37     a = *(map2 + 1405400); /* /lib/libc.so.6 */
38     a = *(map2 + 81924); /* /lib/libc.so.6 */
39     a = *(map2 + 86020); /* /lib/libc.so.6 */
40     a = *(map2 + 90116); /* /lib/libc.so.6 */
41     a = *(map2 + 22460); /* /lib/libc.so.6 */
42     a = *(map2 + 73365); /* /lib/libc.so.6 */
43     a = *(map2 + 8160); /* /lib/libc.so.6 */
44     a = *(map2 + 40908); /* /lib/libc.so.6 */
45     a = *(map2 + 68255); /* /lib/libc.so.6 */
46     a = *(map2 + 12776); /* /lib/libc.so.6 */
47     a = *(map2 + 54015); /* /lib/libc.so.6 */
48     a = *(map2 + 11860); /* /lib/libc.so.6 */
49     a = *(map2 + 35580); /* /lib/libc.so.6 */
50     a = *(map2 + 50252); /* /lib/libc.so.6 */
51     a = *(map2 + 58883); /* /lib/libc.so.6 */
52     a = *(map2 + 28028); /* /lib/libc.so.6 */
53     a = *(map2 + 45372); /* /lib/libc.so.6 */
54     a = *(map2 + 19260); /* /lib/libc.so.6 */
55     a = *(map2 + 41948); /* /lib/libc.so.6 */
56     a = *(map2 + 61440); /* /lib/libc.so.6 */
57     a = *(map2 + 29532); /* /lib/libc.so.6 */
58
59     munmap((void *)0x40021000, 18476); /* /etc/ld.so.cache */
60     a = *(map2 + 853008); /* /lib/libc.so.6 */
61     a = *(map2 + 477920); /* /lib/libc.so.6 */
62     a = *(map2 + 441440); /* /lib/libc.so.6 */
63     a = *(map2 + 513824); /* /lib/libc.so.6 */
64     a = *(map2 + 94432); /* /lib/libc.so.6 */
65     a = *(map2 + 188224); /* /lib/libc.so.6 */
66     a = *(map2 + 1252592); /* /lib/libc.so.6 */
67     a = *(map2 + 1085040); /* /lib/libc.so.6 */
68     a = *(map2 + 194896); /* /lib/libc.so.6 */
69     a = *(map2 + 173808); /* /lib/libc.so.6 */
70     a = *(map2 + 784384); /* /lib/libc.so.6 */
71
72     open("iogrand.stp", 2, -1076688984); /* 3 */
73
74     char buf3[1];
75     read(3, buf3, 1); /* iogrand.stp */
76     a = *(map2 + 293536); /* /lib/libc.so.6 */
77     a = *(map2 + 250384); /* /lib/libc.so.6 */
78     a = *(map2 + 260496); /* /lib/libc.so.6 */
79     a = *(map2 + 488064); /* /lib/libc.so.6 */
80     a = *(map2 + 425536); /* /lib/libc.so.6 */
81     a = *(map2 + 428688); /* /lib/libc.so.6 */
82     a = *(map2 + 432992); /* /lib/libc.so.6 */
83     a = *(map2 + 378256); /* /lib/libc.so.6 */
84     a = *(map2 + 780768); /* /lib/libc.so.6 */
85     a = *(map2 + 838224); /* /lib/libc.so.6 */
86     a = *(map2 + 436240); /* /lib/libc.so.6 */
87     a = *(map2 + 1246371); /* /lib/libc.so.6 */
88     a = *(map2 + 270697); /* /lib/libc.so.6 */

```

```

89
90     close(3); /* iogrind.stp */
91     a = *(map2 + 1094582); /* /lib/libc.so.6 */
92
93     open("1.c", 2, 1); /* 3 */
94
95     char *map4;
96     map4 = mmap((void *)0x40022000, 512, 3, 1, 3, 0); /* 1.c, 0
97         x40022000 */
98     a = *(map2 + 254058); /* /lib/libc.so.6 */
99     a = *(map2 + 265001); /* /lib/libc.so.6 */
100    a = *(map4 + 0); /* 1.c */
101    a = *(map2 + 481824); /* /lib/libc.so.6 */
102    a = *(map2 + 856704); /* /lib/libc.so.6 */
103    /* TODO sys_mremap(0x40022000, 512, 1024, 1, 0x00000003) 0
104        x40022000 */
105
106    munmap((void *)0x40022000, 512); /* 1.c */
107
108    close(3); /* 1.c */
109    a = *(map2 + 381376); /* /lib/libc.so.6 */
110    a = *(map2 + 462544); /* /lib/libc.so.6 */
111    a = *(map2 + 463009); /* /lib/libc.so.6 */
112    a = *(map2 + 445472); /* /lib/libc.so.6 */
113    a = *(map2 + 823536); /* /lib/libc.so.6 */
114    a = *(map2 + 909888); /* /lib/libc.so.6 */
115    a = *(map2 + 452912); /* /lib/libc.so.6 */
116    a = *(map2 + 455353); /* /lib/libc.so.6 */
117    a = *(map2 + 468960); /* /lib/libc.so.6 */
118    a = *(map2 + 818768); /* /lib/libc.so.6 */
119
120    open("1.c", 0, 438); /* 3 */
121    a = *(map2 + 505280); /* /lib/libc.so.6 */
122    a = *(map2 + 1272834); /* /lib/libc.so.6 */
123    a = *(map2 + 319856); /* /lib/libc.so.6 */
124    a = *(map2 + 308336); /* /lib/libc.so.6 */
125    a = *(map2 + 1268575); /* /lib/libc.so.6 */
126    a = *(map2 + 295892); /* /lib/libc.so.6 */
127    a = *(map2 + 303707); /* /lib/libc.so.6 */
128    a = *(map2 + 301792); /* /lib/libc.so.6 */
129    a = *(map2 + 315084); /* /lib/libc.so.6 */
130
131    char buf5[4096];
132    read(3, buf5, 4096); /* 1.c */
133
134    close(3); /* 1.c */
135
136    munmap((void *)0x40022000, 4096); /* /lib/libc.so.6 */
137
138    return 0;
139 }

```

Listing B.1: Complete Trace of Test Application

Bibliography

- [1] Comparison between SystemTap and DTrace (<http://sources.redhat.com/systemtap/wiki/SystemtapDtraceComparison>).
- [2] Blog entry about Icegrind (<http://blog.mozilla.com/tglek/2010/04/07/icegrind-valgrind-plugin-for-optimizing-cold-startup/>).
- [3] SystemTap home page (<http://sources.redhat.com/systemtap/index.html>).
- [4] Analysis of GNOME startup (<http://people.gnome.org/~lcolitti/gnome-startup/analysis/>).
- [5] Article on LWN about Mortadelo (<http://lwn.net/Articles/271796/>).
- [6] OpenOffice performance analysis (http://wiki.services.openoffice.org/wiki/Performance/OOo31_LibrariesOnStartup).
- [7] Gitorious repository for Mortadelo (<http://gitorious.org/mortadelo>).
- [8] Iogrind Gitorious repository for the original version (<https://gitorious.org/iogrind>).
- [9] Iogrind site for the original version (<http://live.gnome.org/iogrind>).
- [10] OpenOffice study about reordering symbols in a library (http://wiki.services.openoffice.org/wiki/Performance/Reorder_Symbols_For_Libraries).
- [11] Iogrind talk by Michael Meeks at FOSDEM 2008 (<http://www.mefedia.com/entry/iogrind/13598699>).
- [12] Iogrind talk by Michael Meeks (<http://www.youtube.com/watch?v=rGFBvTFuQdY>).
- [13] SystemTap wiki start page (<http://sources.redhat.com/systemtap/wiki>).

Index

Bootchart, [14](#)

cold startup, [1](#), [4](#)

CONFIG_TASK_DELAY_ACCT, [12](#)

directory structure, [8](#)

ext2dump, [17](#)

glue calls, [26](#)

I/O traces, [19](#)

Icegrind, [15](#)

improvement strategies, [5](#), [6](#)

Iogrind, [2](#), [17](#)

Iogrind - address space view, [18](#)

Iogrind - architecture, [20](#)

Iogrind - executable trace, [20](#)

Iogrind - glue calls, [26](#), [32](#)

Iogrind - launching tracing step, [27](#)

Iogrind - pagefault handling, [25](#)

Iogrind - scribble view, [18](#)

Iogrind - stack frame view, [17](#)

Iogrind - SystemTap probes and dictionaries, [26](#)

Iogrind with SystemTap, [19](#)

Iolog, [13](#)

iotop, [12](#)

kernel algorithms, [11](#)

kernel structures, [11](#)

lazy loading, [7](#)

libgtk-x11.so, [18](#)

libraries, [9](#)

mapped files, [12](#)

merging files, [6](#)

Mortadelo, [15](#)

multiple configuration files, [6](#)

ordering reads, [7](#)

OS restart, [5](#)

pagefault, [25](#)

startup time, [1](#)

startup time components, [1](#)

startup timing, [4](#)

startup timing jitter, [4](#)

strace, [12](#)

swap I/O, [12](#)

SystemTap, [19](#), [22](#)

SystemTap - begin probe, [23](#)

SystemTap - never probe, [24](#)

SystemTap - pagefault probe example, [25](#)

SystemTap - system call probe example, [24](#)

SystemTap probing, [23](#)

theoretical model, [5](#)

thread_indent, [23](#)

Traveling Salesman Problem, [10](#)

Valgrind, [17](#)