FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Case Study with a High-Level Configuration API

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Klemens Böswirth
Registration Number 01429367

to the Faculty of Informatics
at the TU Wien
Advisor: Dr.techn. Dipl.-Ing. Markus Raab

Vienna, 6th October, 2019

Klemens Böswirth | Markus Raab

Erklärung zur Verfassung der Arbeit

Klemens Böswirth

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 6. Oktober 2019

Klemens Böswirth

Abstract

Elektra [10] is a framework for configuration management. It stores configuration values in a hierarchical manner using unique names for each value. Recently a new high-level *Application Programming Interface* (API) was added to make developing with Elektra easier and safer.

LCDproc [8] is a suite of applications used to display various information, primarily system information like CPU load and memory usage, on LCD devices. It is built as a server/client system to allow remote monitoring. The whole code base of LCDproc is quite old and uses a very limited custom parser to load INI files as its configuration framework.

In this thesis, we took this opportunity and created new Elektra-based versions of LCDproc. We will use these to compare the existing low-level API against the new high-level API and to evaluate the high-level API in general. In addition, we will investigate the implementation of the high-level API and see how it achieves the additional safety it promises.

From a strictly objective standpoint, our results indicate that Elektra's high-level API comes with an expected overhead compared to the low-level API. In many cases, this overhead is small to non-existent in terms of run-time, while the memory overhead is needed for the specifications and better validation. We will also show that the Elektra-based code is in fact easier to understand and maintain, in the sense that using Elektra results in shorter code with less branching.

Contents

CHAPTER 1

Introduction

1.1 Background

1.1.1 Elektra

Today there are many configuration file formats. Most of the time specialized parsers are required for all of these formats. Elektra tries to solve the problem of choosing one format by providing a single consistent API for different formats.

The basic principle behind Elektra is a global key-value database, the *key database* (KDB). Each configuration value stored in the database corresponds to a key-value pair. We call the key part of this pair, i.e. the name under which the value can be found in the KDB, its *key name*, while the value itself is referred to as the *key value*. The term *key*, meanwhile, will be used to refer to the key-value pair as a whole.

The key names are similar to UNIX file paths in their structure. Like in UNIX paths, a slash (`/`) is used to separate parts creating a kind of hierarchy. In Elektra there is, however, no difference between "files" and "directories." A single key may have a value and be the parent of other keys at the same time.

We say key A is the parent of key B, if their key names are the same, except that B has exactly one additional part. If B has more than one additional part, we say it is below A.

Each key has an arbitrary number of *meta keys* with corresponding *meta values*.

Since it would be exceedingly inconvenient to have the configuration for all programs using Elektra in a single file, it is possible to split the KDB into multiple files. This is done through so called *mountpoints*. There is again a parallel to UNIX filesystems. In UNIX, you can mount one filesystem into another, by telling the operating system that everything under a certain path is part of the other filesystem. Similarly, you can instruct Elektra to use a different file below a certain key.

The actual translation of files into a *key set* (a part of the KDB), is done by *storage plugins*. Elektra knows many types of plugins, apart from storage plugins the most notable are *validation plugins*. Validation plugins are used to verify that a key set conforms to a predefined specification.

To discuss how Elektra's specifications work, we first need to discuss another feature: namespaces. In Elektra keys are grouped into namespaces. These namespaces are `system`, `user`, `dir` and the special `spec` namespace. Every key name must either have one of these namespaces as its first part, or the first part must be empty — i.e. it starts with a slash (`/`). Keys that do not have a namespace are called *cascading keys*. These are resolved to one of the namespaces (except `spec`), when we search for them in a key set.

All this might sound confusing, but serves an important purpose. The `system` namespace is used for keys that are the same across the whole system. The files that store these are kept in a system-wide unique directory (e.g. `/etc/kdb`). However, some users of the system might want a different configuration. For that purpose the `user` namespace exists. Its files are stored inside the current users home directory. This means two different users might receive two different values, when reading the key `user/key`. In rare situations, it might be useful for a single user to have different configurations for a single program. To that end, the `dir` namespace can be used. It corresponds to files inside the `.dir` subdirectory of the current working directory.

Last but not least, we have the `spec` namespace. It is exclusively used for specifications, i.e. its keys describe expected properties of the keys in the other namespaces. This description is realized through meta keys. The verification of this specification is then done by the validation plugins.

1.1.2 LCDproc

LCDproc is a "client/server suite for controlling a wide variety of LCD devices" [8]. It consists of three main types of components: a server, one or more drivers and one or more clients.

Server

The server is the main component of LCDproc. It facilitates the communication between all other parts. The server is connected to the LCD device(s), which it controls through drivers. There may be multiple drivers active at once, but all LCD devices will always display the same information.

The server supports two kinds of information to display:

Screens are displayed by default. The server rotates between the available screens, but the user can also manually switch screen. A screen can contain different widgets like text, lines or icons.

Menus are accessed via the menu key. When the user opens a menu, they can browse through the hierarchy and select an item. What happens, when an item is selected, is determined by the client that created the menu. There are also interactive menu items like checkboxes and sliders.

Drivers

Drivers are the part of LCDproc that actually interacts with LCD devices. Each driver is implemented as a shared object loaded at runtime.

Drivers expose a well-defined API, through which the server tells a driver what it should display. Conversely, the API is also used to report information about the LCD device back to the server. Some drivers also support user input (e.g. via a built-in keypad), which can be used to control what is displayed.

Clients

Clients connected to the server provide the actual information that is displayed. The communication between server and clients happens over TCP, so they may be run on separate devices. This allows one server connected to a display to show information from multiple client devices.

There are three core clients inside the LCDproc repository [8]:

`lcdproc` displays system information like CPU load, memory usage, disk activity or the current time. Its name comes from the fact that a lot of this information is taken from the `/proc` filesystem on Linux.

`lcdexec` adds a menu to the server. Menu items are either submenus or commands, which correspond to a shell command that is executed on the client. Commands can also have parameters, the values of which are passed to the shell command as environment variables.

`lcdvc` displays the output of a virtual console. This client is rather esoteric. We only included it in this thesis as an example of a client with a very small configuration.

1.2 Motivation

LCDproc's current configuration loading code, is rather unsophisticated and as such, quite limited in what it can do. This limits the ways in which the configuration can be designed and used. It also means that using the configuration safely inside one of the applications requires a lot of, sometimes very repetitive, code. There is also no tooling for configuration management, so all configurations have to be designed in a way that makes fully manual editing easy.

We just developed a new high-level API for Elektra and wanted to test it in a real-world project. LCDproc was ideal for this:

- It is not too big, yet it has a varied set of configuration, including some rather rare things like the hierarchical menu structure in `lcdexec`. This allows us to test the more advanced features of the high-level API.
- The fact that almost all configuration values in LCDproc require some sort of validation, makes it a prime candidate for Elektra's specifications, on which the high-level API heavily relies.
- Elektra's support for many different file formats, will allow users to choose their preferred format and remove the influence that a fixed format has on configuration design.

Therefore, we decided to do a case-study and evaluate Elektra's high-level API by updating LCDproc.

1.3 Goal of the Thesis

The goal of this thesis will be to evaluate the new high-level API by using it to replace the current configuration access code of LCDproc.

The main questions we want to answer by updating LCDproc are:

RQ 1 *Which guarantees does the high-level API provide to the programmer?*

RQ 2 *What impact has using the high-level API on the performance of LCDproc?*

RQ 3 *How does the performance of the new high-level API compare to the existing low-level API?*

1.4 Methodology

To answer RQ 1, we will explain parts of the inner workings of the high-level API. We will then point out the guarantees made by the API and why they hold.

Answering RQ 2 and RQ 3 requires a more involved methodology. We will create two new versions of LCDproc: one with the old low-level API and one with the new high-level API. These two versions and the original version of LCDproc will then be compared according to a few select metrics:

1. **Number of source code lines**
   A good measure of the complexity of the source code, is the number of lines it

contains. The less code lines there are, the less effort is required to maintain the code base. A tool specialized in measuring source code size will be used. This way, we can distinguish between comments and code lines and get an estimate calculation for code complexity.

2. **Size of the compiled binaries**
   LCDproc is also used in embedded systems. These systems often only have limited resources. This is why we will also compare the size of the resulting binary files.

3. **Compile time**
   Since the high-level API relies heavily on a code-generator, we expect that the compile time will be impacted. To determine the impact of the code-generator call, as well as compiling the generated code, we will measure the time required by the compilation process.

4. **Runtime performance, specifically start-up time**
   Another important aspect of performance is of course runtime performance. In the case of LCDproc, this means the time required to start the application. Since the whole configuration is loaded at start-up, we will only change that part of the code. Therefore, measuring this start-up time will be sufficient and save us from having to devise a series of well-defined client-server interactions.

   We will create modified versions of the LCDproc applications that never execute the main loop. This will allow us to measure the time required for a single (non-interactive) program call and compare it between versions.

5. **Memory usage**
   Just as important as runtime performance, is memory usage. Especially on embedded systems, system memory might be precious.

   We will again use the modified versions, that don't execute the main loop. We will then attach a debugger to the application and stop it at representative points in its execution. Each time we stop the application, we will use the information the Linux kernel provides in `/proc/[pid]/status` to determine the current and peak memory usage.

CHAPTER 2

The new High-Level API

In this chapter we will give an overview of the new high-level API for Elektra. We will explain why the new API is needed and what its benefits are. Furthermore, we will expand on the goals we set ourselves for new API and how we have achieved them. In the end, this will also give an answer to RQ 1.

2.1 Motivation and Goals

Elektra's original API is very low-level. It is based on loading parts of the KDB into a key set. From this key set, individual keys can be retrieved via their full name. Each key can either contain binary or string data.

While this API is easy enough to understand and use, it was clearly designed for easy use in Elektra's extensive plugin system. Using this API within an application can become cumbersome quite quickly.

Among others, these are some reasons why a new API was created:

- The full name of a key has to be specified each time we want to extract it from a key set. At the same time, each application should use a unique prefix for its keys to avoid conflicts. This prefix could be specified once and omitted afterwards.
- The name of a key is given as a string, but strings are prone to typos and cannot be validated during compile time.
- There is no built-in support for types. Even values of primitive types like `int` have to be converted manually.
- Handling of arrays can be complicated. This is demonstrated by the existence of multiple helper functions for array access.

- Error handling is complicated and unwieldy, since almost every function in the API has multiple error cases.

To mitigate these problems in the original API, we decided that the new API should achieve certain goals and fulfil certain guarantees.

- The whole API should be *type-safe*.
- The API shall be based around configuration values, not entire key sets.
- The API shall use a specification containing the expected types and default values for all known configuration values.
- Getter functions, i.e. the functions used to retrieve a configuration value, should not be able to fail under normal circumstances.
- Setter functions shall store each configuration value immediately and not rely on a later call to a commit-type function. This also means that setter functions may fail and therefore need error handling.
- Error handling shall be mandatory. The API shall take steps to force users to explicitly handle or ignore errors. Implicitly ignoring errors, e.g. by not checking a return value, shall not be possible.

2.2 Implementation

This section will give a very brief introduction into the implementation of the new high-level API. The purpose is to present the reader with the information needed to understand the later sections and to show, how we achieved our goals for the API. More detailed explanations of our implementation can be found in the GitHub repository [4] and on the homepage [5].

The new API consists of three basic parts:

1. The initialization and clean-up functions `elektraOpen` and `elektraClose`.
2. Getter and setter functions for each type. Both getters and setters also exist as array variants, which are used to read/write array elements.
3. Various functions for configuration and error handling. Most notably among these is `elektraErrorReset`, used to reset an error variable.

Type safety is achieved by having one getter and setter function for each type. These functions then convert the value of the requested key to the requested type, but only if the key has the expected `type` metadata.

To ensure all keys which are part of the specification can be accessed, even if the user did not explicitly set them, the initialization function `elektraOpen` is supplied with a set of default values.

This means that — if the API is used correctly — there is only one case left, where getter functions may fail: If the requested `key` is not part of the specification and therefore does not have a default value. Solving this problem proved too difficult for a pure C API, so we decided to utilize a code generator.

Like stated above, strings are not a good representation of key names. Function names, however, are validated by the compiler (or the linker). Thus, by creating one function for each key in the specification, we can ensure a compile-time error for unexpected keys.

To generate such functions our code generator reads the specification of an application and produces a `static inline` function for each configuration value. These functions then call the plain C API, thereby hiding the strings which identify keys from the user and removing the potential for typos.

The code generator also makes some advanced features possible. Among these are safely converting a set of string values into C `enum` values and reading whole C `structs` with a single function call. It even allows us to read recursive `structs` in a single call.

In contrast, making error handling mandatory was quite simple. All functions that can return errors check that there is no existing error before doing any work. If an error was already set beforehand and it was not handled by the user, i.e. `elektraErrorReset` has not been called, we call the fatal error handler. This fatal error handler exits the process immediately. It is also used, if a getter unexpectedly fails because the API was used incorrectly. By exiting the process, it ensured that no error can be ignored by accident.

CHAPTER 3

LCDproc Implementations

Now that we have introduced the new high-level API, we will show how it can be used. We will give a brief outline of the three implementations of LCDproc, which will be evaluated and compared in the next chapter.

3.1 LCDproc 0.5

The original LCDproc implementation (*LCDproc 0.5*) uses configuration files in the INI format. These files are accessed through a very simple framework.

When one of the LCDproc applications is executed, it calls the initialization function of this framework. A basic hand-written parser then loads the whole file into a linked list of sections, each of which consists of a linked list of key-value-pairs.

Later on, when the application needs to access a configuration value, it calls one of the getter functions. The getter functions iterate over the list of sections until they find the correct one and then iterate over the key-value pairs in that section to find the requested one. This results in a theoretical runtime of $\mathcal{O}(n+m)$, where n is the number of sections and m is the number of keys in the requested section.

The framework only supports the types `string`, `int`, `double`, `boolean` and `tristate` (a `boolean` with a configurable third value). This results in many parts of LCDproc abusing `int` or `string` values in place of proper `enums`.

The framework also has no support for validation. If there is any kind of restriction — apart from the type — on a configuration value, the requesting part of LCDproc has to implement that on its own. This results in a lot of duplicate and very crude validation code.

3.2 LCDproc ELL

To allow us to analyse the impact of using the new high-level API compared to the existing Elektra API, we created a version of LCDproc that uses the existing low-level API. We call it *LCDproc ELL* (LCDproc with Elektra's Low-Level API).

In Elektra, we use a specification to define which configuration keys LCDproc expects and what values are accepted for these. The specification also defines which plugins should be used, when loading the configuration files. For each of the four LCDproc applications `LCDd`, `lcdproc`, `lcdexec` and `lcdvc` we created such a specification.

Since Elektra's low-level API doesn't provide a very nice interface for applications, we decided to re-use the framework from the original implementation. The hand-written INI parser was replaced with Elektra.

Additional changes to the source-code were only made as far as required. This also implies that there is some duplicate validation logic in this implementation, since we did not remove most of the old validation code. However, because most of these validations are rather simplistic, so the impact on our benchmarks should be minimal.

The application that has undergone the most changes in this version is `lcdexec`. The way it represented its menu structure in INI files is not compatible with Elektra's hierarchical KDB. To make the comparison fairer, we decided to use a structure that could be used with the high-level API with only minor modifications. This meant using flat arrays cross-referencing each other, instead of hierarchies that more directly mimic the menu structure.

3.3 LCDproc EHL

The second new implementation *LCDproc EHL* (LCDproc with Elektra's High-Level API) uses as many of the high-level API's features as possible. This meant slightly modifying the specification. Nevertheless, this will not significantly impact the benchmarks, because almost all the modifications solely affect the code-generator, not the running application.

Considering that the high-level API is designed to be used by applications, it would be redundant to create a framework on top of it. So we discarded the old implementation's framework and instead work directly with Elektra. This of course resulted in a lot more and sometimes much deeper modifications to the code, than in the low-level implementation. While it might seem, that this makes the comparison unfair, the opposite is the case. The low-level API is so badly suited for use in applications, that it is almost certain that developers would insert a layer between it and the actual application.

Using the high-level API and its code-generated features, also allowed us to remove the code used to parse command-line options and instead rely on Elektra's inbuilt features for that.

CHAPTER 4

Evaluation

In this chapter we will explore the results of the benchmarks described in section 1.4. We will compare the implementations described in the previous chapter and thereby answer RQ 2 and RQ 3.

All benchmarks where done with the following setup:

Processor	AMD Ryzen 7 3700X 8-Core Processor
System Memory	32 GB
Operating System	Ubuntu 18.04.3 LTS
Elektra Version	git commit `ca1dfd17`
LCDproc 0.5 Version	git commit `58738a8f` and `fe38bb2a`
LCDproc ELL Version	git commit `e5782200` and `2d86b170`
LCDproc EHL Version	git commit `3e2b9d3f` and `314d63d8`

The detailed results of our benchmarks and the scripts we used to run them, can be found online in our benchmark repository [2].

4.1 Number of Source Code Lines

First we will look at the size of the source code in the different versions. To perform this benchmark, we used `scc` [3], a tool specialized in measuring the size of source code files.

As shown in Table 4.1, the code of LCDproc is mostly composed of C and C Header files. Other than that, there are mostly `autoconf` and `automake` configuration files and various pieces of documentation. The only files we are interested in, are the C and C Header files since we didn't significantly modify anything else.

We also used `scc` to calculate a code complexity estimate. This estimate is based on counting branch and loop statements.

Language	Files	Lines	Blanks	Comments	Code	Complexity
C	171	84006	11760	19388	52858	10840
C Header	165	13182	1647	3315	8220	100
Autoconf	15	3981	687	1312	1982	67
Markdown	8	5255	710	0	4545	0
Perl	6	1800	242	551	1007	70
Shell	6	305	38	37	230	17
CSS	2	39	6	0	33	0
Makefile	2	55	16	7	32	0
Plain Text	2	558	154	0	404	0
License	1	339	53	0	286	0
Patch	1	131	5	0	126	0
YAML	1	10	3	1	6	0
m4	1	1225	63	51	1111	0
Total	381	110886	15384	24662	70840	11094
Estimated Cost to Develop $2,368,033						
Estimated Schedule Effort 21.289141 months						
Estimated People Required 13.176022						

Table 4.1: Source code statistics for LCDproc 0.5 (as produced by `scc`)

Language	Files	Lines	Blanks	Comments	Code	Complexity
C	171	83853	11799	19193	52861	10760
C Header	165	13131	1649	3252	8230	100
Autoconf	15	3981	687	1312	1982	67
Markdown	9	5546	752	0	4794	0
Shell	7	364	38	37	289	17
Perl	6	1800	242	551	1007	70
CSS	2	39	6	0	33	0
Makefile	2	55	16	7	32	0
Plain Text	2	558	154	0	404	0
License	1	339	53	0	286	0
Patch	1	131	5	0	126	0
Python	1	193	44	1	148	60
YAML	1	10	3	1	6	0
m4	1	1225	63	51	1111	0
Total	384	111225	15511	24405	71309	11074
Estimated Cost to Develop $2,384,497						
Estimated Schedule Effort 21.345267 months						
Estimated People Required 13.232745						

Table 4.2: Source code statistics for LCDproc ELL (as produced by `scc`)

In Table 4.2 we see, that using Elektra's low-level API doesn't affect the source code size or complexity much. This is mostly because we created a framework on top of Elektra. The API of this framework is very similar to the configuration parser framework of LCDproc 0.5, so the code didn't actually change very much.

In contrast, moving to the high-level API constitutes a much bigger change. This is reflected in the source code statistics of LCDproc EHL shown in Table 4.3. The code size went down by 2000 lines, which is about 5%, and complexity decreased significantly.

Language	Files	Lines	Blanks	Comments	Code	Complexity
C	167	79962	11378	18316	50268	9761
C Header	163	13090	1675	3151	8264	92
Autoconf	16	4037	692	1313	2032	74
Markdown	10	5459	753	0	4706	0
Perl	6	1800	242	551	1007	70
Shell	6	305	38	37	230	17
gitignore	5	36	0	0	36	0
CSS	2	39	6	0	33	0
Makefile	2	55	16	7	32	0
Plain Text	2	558	154	0	404	0
License	1	339	53	0	286	0
Patch	1	131	5	0	126	0
Python	1	249	25	1	223	38
YAML	1	10	3	1	6	0
m4	1	1225	63	51	1111	0
Total	384	107295	15103	23428	68764	10052
Estimated Cost to Develop $2,295,221						
Estimated Schedule Effort 21.037983 months						
Estimated People Required 12.923349						

Table 4.3: Source code statistics for LCDproc EHL (as produced by `scc`)

While 5% seems like a small change at first, we have to keep in mind that the configuration loading code is only a small part of LCDproc. Most of the code is dedicated to the interactions between client and server, as well as between the drivers and the hardware.

The numbers shown in Table 4.3 also include all drivers, even though most have not been updated. This flaw in the measurements is balanced to some degree, by the fact that using Elektra's high-level API requires the use of separate specification files. These specification files, in the case of LCDproc EHL provided in the INI format, are not supported by `scc` and as such do not show up in its output.

For a better overview, we summarized the important differences between LCDproc 0.5 and LCDproc EHL in Table 4.4.

Language	Files	Lines	Code	Complexity
C	−4	−4044	−2590	−1079
C Header	−2	−92	+44	−8
Total	−6	−4136	−2546	−1087

Table 4.4: Source code changes between LCDproc 0.5 and LCDproc EHL

4.2 Size of the Compiled Binaries

Next, we will compare the size of the binary files produced by compiling the different implementations of LCDproc.

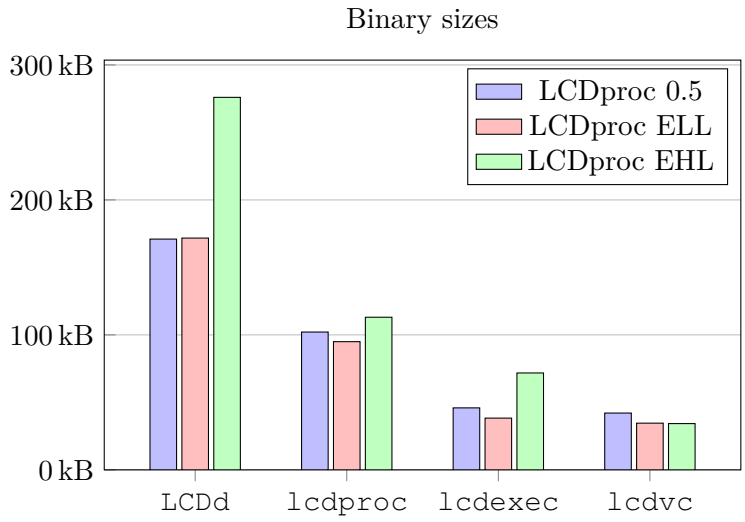

Figure 4.1: Comparison of the binary size of the LCDproc executables

In Figure 4.1, we see the sizes of the executable files of LCDproc. When moving from LCDproc 0.5 to LCDproc ELL, the binaries get slightly smaller (`LCDd` remained almost unchanged). This is to be expected, since the configuration file parsing code was removed and replaced by an external library (Elektra).

However, LCDproc EHL shows a clear increase in size for all applications except `lcdvc`. This can be explained by the different numbers of configuration settings in the applications. While `LCDd` has a lot of them (for technical reasons those belonging to drivers contribute to its count too), `lcdvc` has a very small number of settings.

From section 2.2, we remember: The code-generator produces accessor methods for each of these configuration settings. Because of compiler optimizations, many of the accessor methods generated for the configuration settings, do not affect binary size much. Nevertheless, using some advanced features, particularly those related to `structs`, does have an effect on binary size. This is because those features produce additional code, which is not optimized as aggressively with standard compiler settings.

All applications other than `lcdvc` use these advanced features to some extent and therefore show increases in binary size.

The fact that binary size is mostly unaffected if no advanced features are used, is also apparent when looking at the sizes of the shared library files of the drivers. In Figure 4.2 we compare all the drivers we chose to update. Everything is as expected (drivers do not use any of the advanced features), except for the `linux_input` driver.

The `linux_input` driver increased in size by 62%. We did not find any reasons, why this would be the case. As far as we know, this increase is not directly caused by Elektra-related changes. The likely cause of this increase is some compiler optimization that

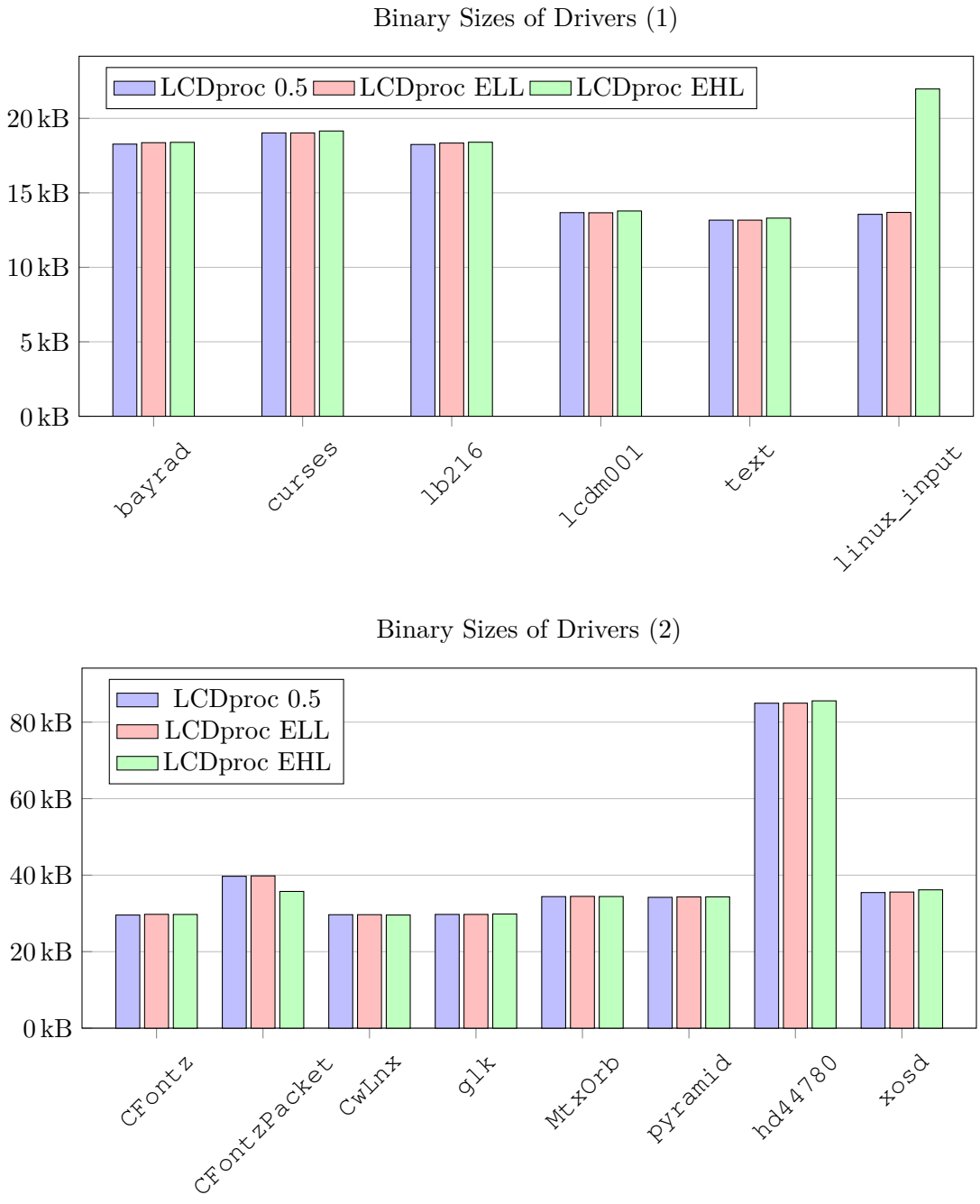

Figure 4.2: Comparison of the binary size of some drivers

was or was not applied as an indirect result of our changes. We came to this conclusion, because even parts of the code we did not change at all result in a lot more assembly code in LCDproc EHL compared to the other versions. LCDproc's release builds (which were used for these benchmarks), are optimized for speed, which means the compiler may sacrifice binary size as a result.

4.3 Compile Time

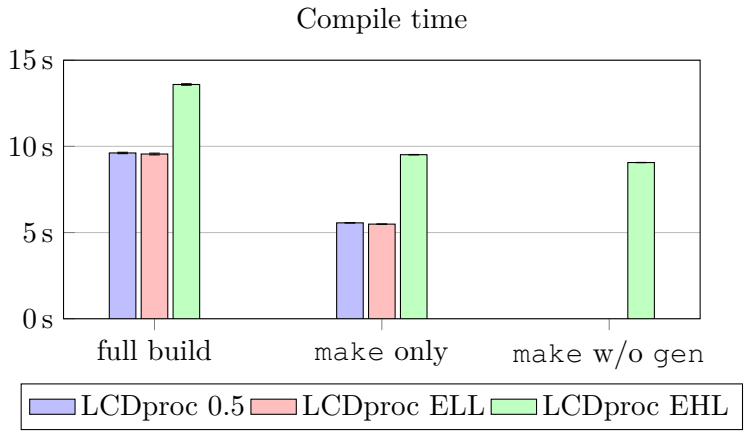

Figure 4.3: Comparison of the compile-time of a full build and after calling `make clean`. For LCDproc EHL compile-time excluding the code-generator call is also shown.

Since we are using a code-generator in LCDproc EHL, we were interested in compile time. Especially, what the impact of running the code-generator would be.

The times were measured with `hyperfine` [9], a handy tool for running benchmarks. `hyperfine` automatically runs a given command multiple times and records the run time of each execution. It also calculates the mean and standard deviation of the runtimes as well as some other statistical figures. The actual number of runs varies to make sure the measurements are accurate, but we set a minimum of 10 runs per command.

We did two tests for all configurations:

- **full build**: The "full build" runs started with a clean git repository.
- **`make` only**: The "`make` only" ones where executed after `make clean` and therefore did not run `autoconf`'s `configure` script.

In Figure 4.3, we see the results of our measurements. A clear increase in compile time can be observed for LCDproc EHL. By comparing the "full build" with "`make` only" we can also see, that only the `make` step of compiling, but not the `autoconf` step was affected by our changes.

To find out, how much of the increase stems from the call to `kdb gen`, we repeated the "`make` only" test for LCDproc EHL, with a modified `make clean` that does not remove the output files of the code-generator. As it turns out, the call to `kdb gen` is comparatively cheap, it only takes around 0.50 s. The other 3.50 s increase, is the time it takes to compile the generated C Files.

4.4 Start-up Time

While the compile time is interesting to developers of LCDproc, the more important figure is the run time performance. Since using Elektra only affects the start-up performance of LCDproc, we focused our benchmarks on this part. We again used `hyperfine` to measure the execution times of our modified binaries (see section 1.4). In addition to modifying the executables, we also had to add an artificial delay to the execution of `LCDd` and `lcdexec`. Without the delay, `hyperfine` would not have been able to produce reliable results.

To get an idea how configuration size affects the run time, we tested all four applications with three sizes of configuration:

- A **minimal** configuration containing just enough settings, that the application can start. For the server this mean setting a single driver, for `lcdexec` creating a menu with a single command. `lcdproc` and `lcdvc` do not require any configuration. They were run purely with default values.
- A **small** configuration with a handful of settings.
- A **big** configuration. The big configuration was taken from the example configuration files included in the LCDproc 0.5 source code and adapted for the other versions.

In addition, we also ran `LCDd` and `lcdproc` in their minimal configurations with additional command line arguments. Since LCDproc EHL uses Elektra's command line parsing functionality and the other versions use the standard `getopts`, we were interested in the impact of this change. The other two clients do not support a lot of command line options, and the ones that are supported are difficult to use in benchmarks, so we did not test them.

The results of all these runs are shown in Figure 4.4, grouped by application. We have adjusted the axes for `LCDd` and `lcdexec` to account for the artificial delays. The error bars show the standard deviation, of each benchmark.

We found that the impact of using Elektra is quite different across the LCDproc applications.

The server `LCDd` has a very short start-up time, so the effects of using Elektra are most obvious there. When switching from LCDproc 0.5 to LCDproc ELL, we see a 23 ms

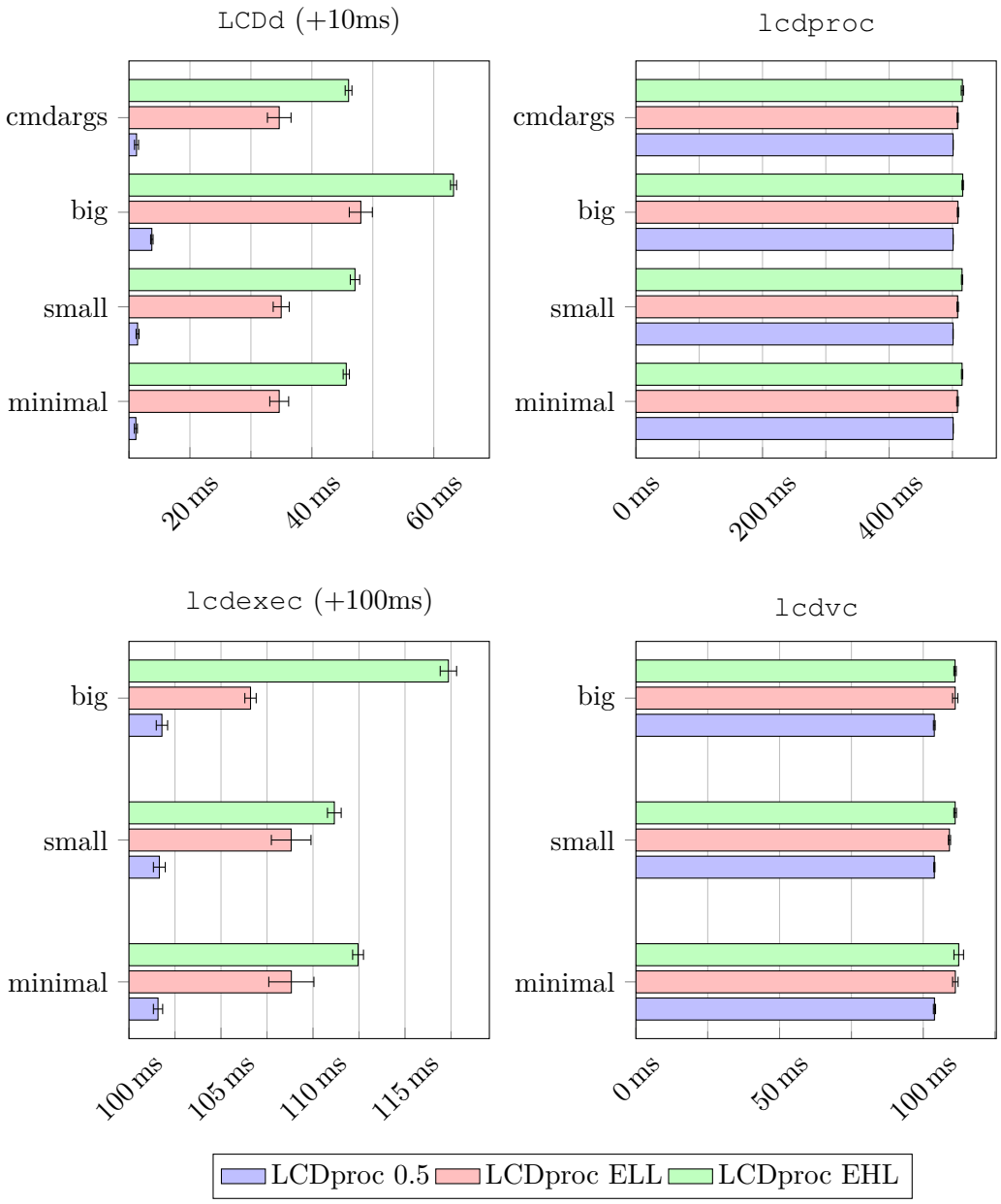

Figure 4.4: Comparison of start-up times. The times of `LCDd` and `lcdexec` are artificially increased via `sleep`, because they could not be reliably measured otherwise.

increase in run time, except for the big configuration, where the increase is 35 ms. The big configuration of `LCDd` is much bigger than the small one, so the bigger difference makes a lot of sense, the program has to process a lot more keys.

However, in LCDproc ELL, the difference between small and big configurations is much bigger than in LCDproc 0.5. This is a clear disadvantage of using Elektra. Elektra always fully processes all keys, while the less sophisticated framework in LCDproc 0.5 simply reads the whole configuration into a linked list and only processes settings, when the application actually loads them.

The move from LCDproc ELL to LCDproc EHL further increases the server start-up time by 10 ms to 15 ms. LCDproc ELL was executed in the simplest possible setup, this means that none of Elektra's validation plugins where configured and therefore did not contribute to the execution time. Therefore, we expected the minor increase in run-time.

For `lcdvc` it seems the impact is universally the same, about 7 ms, regardless of whether we use the low-level or high-level API and of the configuration size. `lcdvc` uses much fewer plugins and has much fewer configuration settings than `LCDd`. It seems `lcdvc`'s specification is so simple that the overhead of running the plugins is balanced out by the framework on top of Elektra used in LCDproc ELL.

In `lcdproc`, we see the same 7 ms difference between LCDproc 0.5 and LCDproc ELL. This time however, using the high-level API adds another 7 ms to the run time. A part of this likely comes from `lcdproc`'s use of the high-level API's `struct`-reading feature. The other part is that `lcdproc` has more configuration settings and therefore, although it does not use more plugins than `lcdvc`, the plugins have to process more keys.

Lastly, we take a look at `lcdexec`. Its results raise a lot of questions. The big configuration in LCDproc ELL is quicker than the small one, a similar phenomenon occurs for the minimal and small configurations in LCDproc EHL and the standard deviation of LCDproc ELL's minimal and small runs is unusually high. It is likely that these results are not very reliable, probably because of `lcdexec`'s very short execution time (less than 2 ms in LCDproc 0.5).

The only thing we are sure is a reliable result here, is that using Elektra's high-level API creates an overhead that is particularly obvious in big configurations. `lcdexec` uses recursive `structs` which are read automatically via the high-level API. This mechanism is very powerful, but has certain limitations compared to manually implementing the recursive reading. The relevant limitation in this case is that `lcdexec` needs a unique ID for each of its menus, commands and parameters. The high-level API has no way to provide such an ID, so we have to do some post-processing in LCDproc EHL that is not required in the other versions. This additional step is likely, why we see another overhead proportional to the configuration size in addition to the one cause by Elektra's handling of keys explained above.

Both `LCDd` and `lcdproc` also demonstrate, that Elektra's command-line parsing has no significant impact on runtime performance.

4.5 Memory Usage

Just as adding Elektra will affect the start-up time of LCDproc, it will affect its memory usage. Properly measuring the memory usage of a program over time is quite hard to do. There are many tools that can track `malloc` calls, but Elektra's cache directly uses `mmap`. These `mmap` calls would not be counted by most tools. Some tools can also track the low-level `mmap` and `brk` calls instead of `malloc`. However, that is also not a good solution, since it will produce unrealistically high results. The results will be much higher than expected, because they measure virtual address space instead of actual physical memory usage.

To get an idea of Elektra's impact anyway, we decided on a very simple solution. We attached a debugger to the applications and stopped them at certain location. Each time an application was stopped, we recorded the `VmRSS` and `VmHWM` values reported by the Linux kernel. `VmRSS` and `VmHWM` are the current and peak physical memory usage respectively.

A pseudo-code description of the points at which all applications where stopped is shown in Listing 4.1.

We graphed the resulting values in Figure 4.5. The graphs only show a rough overview of memory usage over time, obtained by putting peak values in between current values, if they exceeded the last peak. The interesting parts of the diagrams are the rough outline of the graphs, their peaks as well as their end points. Peak values are interesting, because they show how much memory is needed to properly run LCDproc and the rough outlines show us, when this peak is reached. The end points, meanwhile, give an estimate of how much memory the application occupies continuously over its lifetime (if it were to actually execute the main loop).

We found that all of LCDproc's applications have a base memory usage of about 2 MB in LCDproc 0.5. In LCDproc ELL this is increased to around 4.50 MB. In LCDproc EHL the base requirement differs between applications, `LCDd` and `lcdproc` need 6.50 MB, while `lcdexec` and `lcdvc` only need 6 MB and 5 MB respectively.

The impact of bigger configurations can be observed in `LCDd`. All three versions show an increase of 2 MB when switching from the small configuration to the big one. LCDproc ELL also has an increase of 500 kB when switching from minimal to small, which is not present in LCDproc EHL. This is because LCDproc EHL loads the specification and creates keys with default values for all drivers, regardless of whether they are in use or not. This is a known limitation of Elektra's specifications.

Because we did not configure the specification in LCDproc ELL, no keys with default values are created and no memory usage impact is observed. This also accounts for some of the difference between LCDproc ELL and LCDproc EHL.

All applications have their peak memory usage while processing the configuration. This was expected, since validating, transforming and otherwise processing the configuration

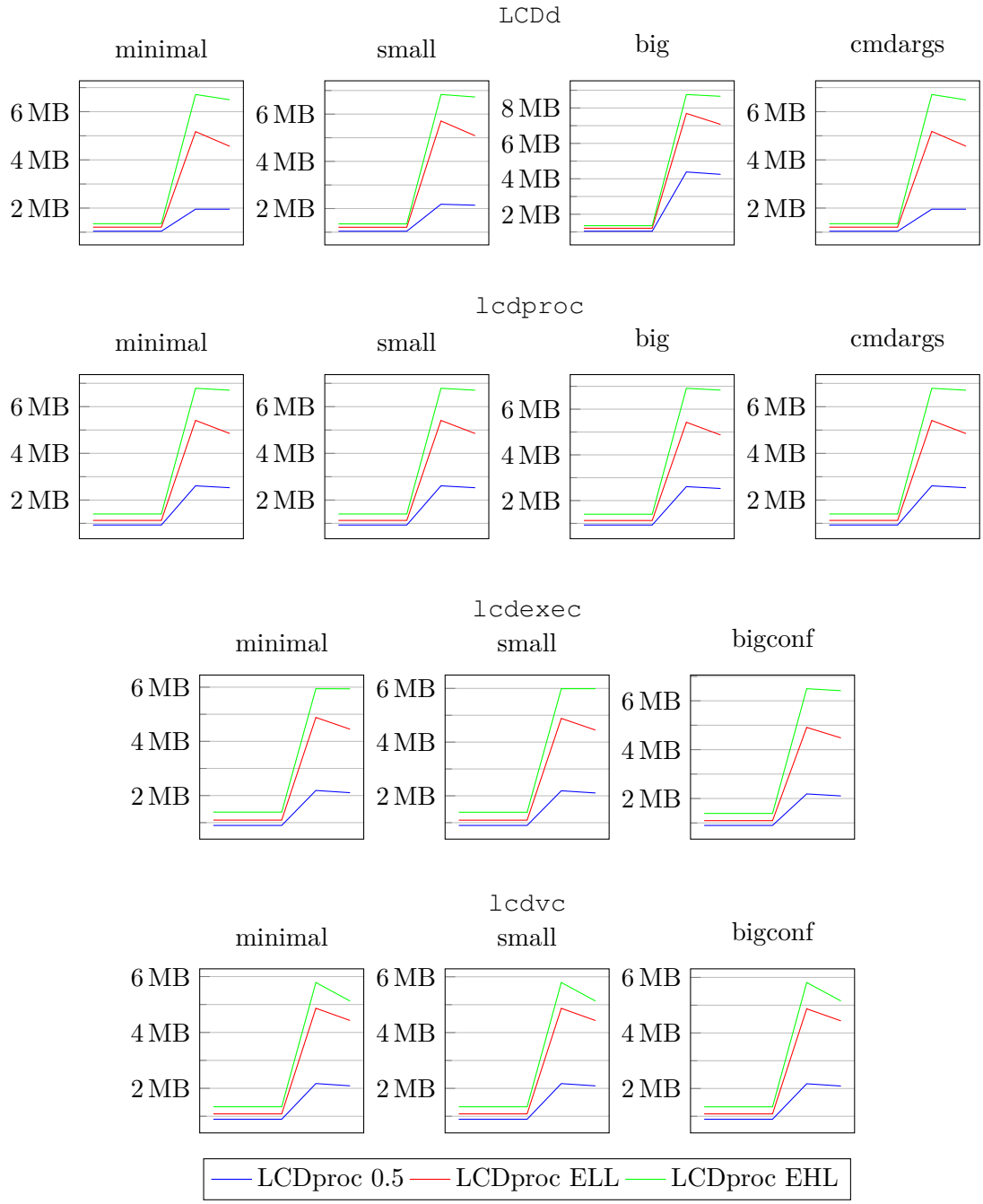

Figure 4.5: Comparison of the memory usage of the LCDproc applications using configurations of varying size.

Listing 4.1: Pseudo-code description of points at which LCDproc applications where stopped to record memory usage.

```
int main (int argc, char ** argv) {
    __record_memory_usage ();
    /* ... */
    __record_memory_usage ();
    loadConfiguration ();
    /* ... */
    __record_memory_usage ();
    mainloop ();
    cleanup ();
}
```

will require additional memory that can be discarded once we enter the main loop. The peak is more or less pronounced depending on the implementation, because all implementations handle their configuration access differently.

LCDproc 0.5's configuration data structure has very little overhead, so its peak will never be very high. Elektra's data structures do have quite a lot of overhead, so we clearly see, when some of it is discarded. This is most noticeable in `lcdvc`, which discards all Elektra structures before entering the main loop. The others only discard them right at the very end of the program's lifetime, which is not recorded in our diagrams.

Lastly, we can use the results of `LCDd` and `lcdproc` to see, that Elektra's command-line parsing has no substantial effect on memory usage as well.

CHAPTER 5

Related and Future Work

5.1 Related Work

5.1.1 Specification-based Code-generation

In "A Model-Driven Method for Fast Building Consistent Web Services from OpenAPI-Compatible Models" [13] Sferruzza et al. show an approach for validating the consistency of web services specifications. To show how this can be used in practical applications, they also implemented a prototype code generator that creates a fully functional web services from an extended OpenAPI specification. This code-generator is significantly more advanced than those commonly used in the industry [1], which only create skeleton code. The resulting code generator is in some sense similar to the one implemented for Elektra. Both of them do generate completely standalone code, which can be used by other parts of the program, without the need to fill in a skeleton first.

The paper "From Open API to Semantic Specifications and Code Adapters" [12] also focuses on generating code from an OpenAPI specification. However, unlike Sferruzza et al. the paper's goal is not to produce a fully functional web-service, but to create a more sophisticated semantic specification from the purely syntactic OpenAPI specification. In addition, their process is also just semi-automatic, in contrast to the fully automatic generator implemented by Sferruzza et al.. The authors then use the generated semantic specification together with the original syntactic OpenAPI specification to generate adapter code for the various requests the described web service accepts.

In his book "Code Generation in Action" [7] Jack Herrington gives a broad overview of different types of code generation. After giving a basic introduction into what code generators are and how they work, examples for many code generators are given. This ranges from simple things like generating code describing user interfaces and documentation to very complex topics like generating web service layers and even business logic. While this work clearly does not focus on research, it shows what can be

done by code generation and provides a nice introduction into how one might go about creating new code generators.

5.1.2 Refactoring

The book "Refactoring: improving the design of existing code" [6] is a very often cited and very extensive guide to refactoring. The author's goal is to show how refactoring can be done correctly and efficiently. In addition, the book also covers, why refactoring is important and what the benefits of regular refactoring are. A comprehensive list of signs of bad code as well as better ways of organizing code and data are presented as well.

5.2 Future Work

While we think that the creation of the high-level API makes Elektra fully production-ready, we cannot deny that there is still a lot that can be improved.

Firstly, not everyone likes programming in C and especially GUI applications are unlikely to be written in C. Elektra's low-level API already has bindings in many other languages including C++, Java, Python, Go and Rust. To allow developers using these programming languages to enjoy the benefits of the high-level API, we will need to extend those bindings to include the high-level API. In some cases, it might also make more sense to create new high-level APIs based on the existing low-level bindings. This way, we would not be limited by the underlying C high-level API.

In addition to these bindings, the code-generator should be extended to support different programming languages. The likely candidate for the first such extension is C++. It is directly compatible with C, which makes creating a high-level API binding easy, and its support of namespaces, classes and object-oriented programming in general would allow a much nicer generated API. For example, a class could be generated for each key of the specification. The namespace and name of the class would reflect the key name and the methods of the class could be used to get, set and otherwise manipulate the key value.

There are also many ways, in which Elektra's specification language can be enhanced. For one, the `specload` plugin, which is intended to allow users to safely change the specification, is still very limited. At the moment it mostly allows changing the `description` and other comment-style meta keys. For instance, it may be allowed to restrict the `type` of a key from `long` to `short` in many cases. However, there are also cases where this is dangerous (e.g. when the application also writes back to the KDB), which is why it is not allowed yet. We will have to put additional thought into which changes are safe under which circumstances. Most likely, there will have to be a way to allow and disallow such changes in the specification by hand.

The specification language could also be made more powerful. One idea for doing this is the introduction of so-called "contextual values" [11]. These are values, that are taken from the environment (or context) in which an application is executed. For example this

could be the current username, the hostname of the current machine, or the value of another key in the KDB. Crucially, these values are then used as part of key names and not as the value of some key. The creation of such a system was already attempted with the old Python-based code-generator [11], but this was based on a rather complicated template-based C++ setup and therefore could not be used from C.

The `spec` plugin is used to copy the metadata of the specification keys onto the keys containing the actual values, so that validation plugins will validate them. This plugin has some known limitations that should be improved upon.

At the moment, the plugin does support what is called "globbing". Through globbing, a single template key can be used as the specification for a whole set of keys. Currently, the only fully supported way of globbing is array-based. This means, that a template for all array elements of an array is used. Sometimes, however, using arrays is cumbersome. For instance, in `lcdexec` all menus are contained in a single array and referenced by their index. It would make the configuration much more user-friendly, if menus could be referenced via a name instead of an index. This is sadly not possible right now and might not ever be possible, with the current specification setup.

Lastly, we already knew that the `spec` plugin has some performance problems. Many of these are inherent to the way the plugin works, but a profiler run on `LCDd` revealed, that there are still possibilities for optimization Elektra's core. In all of Elektra, it is very common that keys are allocated temporarily. Our results (see Figure 5.1), show that about 40% of the start-up time of `LCDd` are spent in the key-creation function `keyNew`. This function in turn spends most of its time setting the key name via `elektraKeySetName`, a seemingly simple function that is also called from many other places. Therefore, this function is a prime candidate for future optimizations.

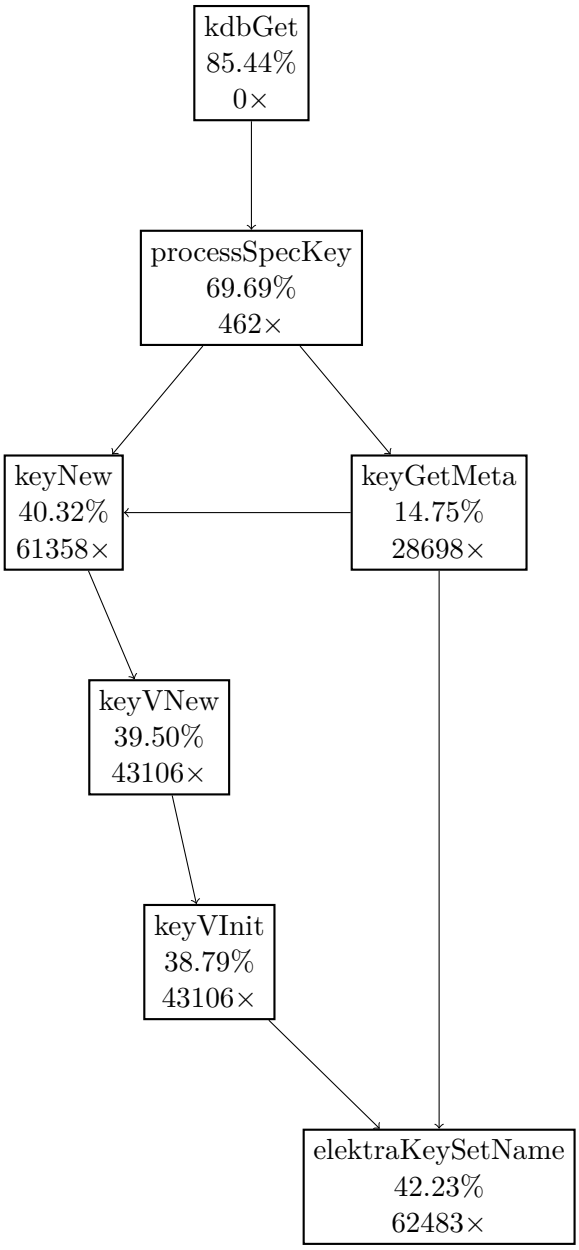

Figure 5.1: Extract of the call-tree produced by running `LCDd` with profiler `callgrind`. The full tree can be found in our benchmark repository [2]. The percent values indicate the fraction of processing time that was spent in the function, while the absolute values denote the number of times the function was executed. Not all code paths are shown, so sometimes children may be called more often than all their (shown) parents have been.

CHAPTER 6

Conclusion

We started with a version of LCDproc that used a custom, old and unsophisticated configuration loading framework. Looking at the implementation, we also expected it to be slow — at least that is what the theoretical runtime would suggest —, this turned out to not be true. The implementation of LCDproc 0.5 is faster, both in terms of start-up and of compile time, and more memory efficient. This would suggest that using Elektra instead, would not be a good decision.

However, we feel that the benefits of Elektra outweigh the performance penalties, which in the worst case were still substantially below 100 ms. We gained the safety and ease of use of a robust specification that replaces many lines of code. Code that is very hard to test properly. Using Elektra also enables us to use all of its powerful tooling and change the file format of our configurations whenever we want, without any changes to the code.

The 2 MB to 5 MB memory overhead of Elektra, is a big relative increase — +100% and more — but should be manageable for any modern system. Binary size also increased by less than 100 kB in some cases, but code size, which in our opinion is much more important nowadays, went down more than 2000 lines. Not only that, but the readability of the remaining code is also greatly improved compared to the version without Elektra.

Compared to the low-level API, the high-level API only has a minor overhead. At the same time it is, however, a lot easier to use. There is less potential for error and less manual setup is required.

The high-level API also provides very helpful guarantees. Most important among these are the infallible getters and the compile-time checking of key names.

In conclusion, we think that its new high-level API makes using Elektra much easier and at the same time safer as well. Elektra itself provides many benefits compared to simpler custom configuration loading frameworks, but its performance could still be improved.

List of Figures

List of Tables

List of Code Fragments

List of Terms

API v, 1, 3–5, 7–9, 11, 12, 14, 15, 21, 26, 29, *see* Application Programming Interface

Application Programming Interface In the context of this thesis: The set of functions exposed by some part of a program or library that other parts or different programs and libraries use to interact with it.

Cascading key A key with no explicit namespace. Its namespace is implied by the context in which it is used and resolved during the lookup in a key set. 2

Elektra "Elektra serves as a universal and secure framework to access configuration parameters in a global, hierarchical key database." [4] v, 1, 2, 4, 7, 12, 14–16, 19, 21, 22, 24, 26, 27, 29, 37, 38

KDB 1, 2, 7, 12, 26, 27, 37, 38, *see* Key database

Key The smallest unit in Elektra. It is short for key-value pair and represents a single configuration value. 1, 2, 7–9, 21, 22, 26, 27, 37, 38

Key database The hierarchical database, distributed across a number of different files across a system, in which Elektra stores all configuration values.

Key name The name of a key that uniquely identifies it inside the KDB. 1, 2, 26, 27, 29, 38

Key set An ordered collection of keys. Most of the time this represents a part of the KDB. 2, 7, 8, 37

Key value The actual configuration values stored in a key in the KDB. 1, 26

LCDproc "LCDproc is a client/server suite including drivers for all kinds of nifty LCD devices. The server works with different display sizes and supports several serial devices. [...] Various clients are available that display things like CPU load, system load, memory usage, uptime, and a lot more." [8] v, 2–5, 11–13, 15, 16, 18, 19, 22, 29, 38

LCDproc 0.5 The original LCDproc version using a hand-written configuration parser instead of Elektra. 11, 14–16, 19, 21, 22, 24, 29

LCDproc EHL Our new implementation of LCDproc that uses Elektra's new high-level API. 12, 14–16, 18, 19, 21, 22

LCDproc ELL Our test implementation of LCDproc that uses Elektra's low-level API. 12, 16, 19, 21, 22

Meta key A key that does not exist on its own inside the KDB, but instead is attached to another key and describes additional properties of this other key. The key name of a meta key uniquely identifies it among all meta keys of the key it is attached to. 1, 2, 26, 38

Meta value The value of a meta key. 1

Mountpoint Apart from the fixed set of default files, additional files can be used to store a part of the KDB. All keys in such a file are below a single key. This key is called a mountpoint. 1

Storage plugin A type of plugin for Elektra that is used to load and store configuration files. 2

Type safety Type safety is a measure describing whether a language or specification can detect type errors. A language or specification is said to be (fully) type-safe, if it detects all possible errors. In the context of Elektra this applies to the relation between the type of a key and native C types. A type error is caused by the attempt to convert a key to an incompatible native C type. 8

Type-safe 8, *see* Type safety

Validation plugin A type of plugin for Elektra that is used to do some kind of validation on the keys in the KDB. This can happen either after loading or before storing a configuration file. What kind of and how the validation should be done is defined via meta keys. 2, 21

Bibliography

[1] OpenAPI Initiative (OAI). *OpenAPI Generator · Generate clients, servers, and documentation from OpenAPI 2.0/3.x documents*. URL: https://openapi-generator.tech/ (visited on Apr. 11, 2019).

[2] Klemens Böswirth. *Elektra LCDproc Benchmarks*. URL: https://github.com/ElektraInitiative/rawdata/tree/master/lcdproc-benchmarks.

[3] Ben Boyter. *scc: Sloc, Cloc and Code*. URL: https://github.com/boyter/scc (visited on Oct. 4, 2019).

[4] Elektra Initiative. *Elektra*. URL: https://github.com/ElektraInitiative/libelektra (visited on Oct. 4, 2019).

[5] Elektra Initiative. *ElektraInitiative*. URL: https://libelektra.org (visited on Oct. 4, 2019).

[6] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[7] Jack Herrington. *Code generation in action*. Manning Publications Co., 2003.

[8] LCDproc. *LCDproc*. URL: https://github.com/lcdproc/lcdproc (visited on Oct. 4, 2019).

[9] David Peter. *hyperfine: A command-line benchmarking tool*. URL: https://github.com/sharkdp/hyperfine (visited on Oct. 4, 2019).

[10] Markus Raab. „Context-aware configuration“. PhD thesis. TU Vienna, Dec. 2017. URL: http://book.libelektra.org.

[11] Markus Raab and Franz Puntigam. „Program Execution Environments As Contextual Values“. In: *Proceedings of 6th International Workshop on Context-Oriented Programming*. COP'14. Uppsala, Sweden: ACM, 2014, 8:1–8:6. URL: http://doi.acm.org/10.1145/2637066.2637074.

[12] Simon Schwichtenberg, Christian Gerth, and Gregor Engels. „From open API to semantic specifications and code adapters“. In: *2017 IEEE International Conference on Web Services (ICWS)*. IEEE. 2017, pp. 484–491.

[13] David Sferruzza et al. „A Model-Driven Method for Fast Building Consistent Web Services from OpenAPI-Compatible Models“. In: *International Conference on Model-Driven Engineering and Software Development*. Springer. 2018, pp. 9–33.