OULU UNIVERSITY OF
APPLIED SCIENCES

Michal Srb

OPTIMIZATION OF FONTCONFIG LIBRARY

OPTIMIZATION OF FONTCONFIG LIBRARY

Michal Srb
Bachelor's Thesis
Spring 2017
Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Degree Programme in Information Technology, Internet Services

Author: Michal Srb
Title of the bachelor’s thesis: Optimization of Fontconfig Library
Supervisor: Teemu Korpela
Term and year of completion: Spring 2017 Number of pages: 39 + 1 appendix

Fontconfig is a library that manages a database of fonts on Linux systems. The aim of this Bachelor's thesis was to explore options for making it respond faster to application's queries.

The library was identified as a bottleneck during the startup of graphical applications. The typical usage of the library by applications was analyzed and a set of standalone benchmarks were created. The library was profiled to identify hot spots and multiple optimizations were applied to it.

It was determined that to achieve an optimal performance, a complete rewrite would be necessary. However, that could not be done while staying backward compatible. Nevertheless, the optimizations applied to the existing fontconfig yielded significant performance improvements, up to 98% speedups in benchmarks based on the real-world usage.

Keywords: fontconfig, optimization, benchmarking, profiling

CONTENTS

VOCABULARY

ABI:	Application binary interface
API:	Application programming interface
Callgrind:	Tool from valgrind suite for profiling programs on a sythetic CPU.
Disk IO:	Input/output operations performed on a disk.
Glob:	A wildcard character that substitues one or more characters in a filename.
Gperf:	A perfect hash function generator.
GUI:	Graphical user interface
Intrinsic:	A function whose implementation is handled by the compiler.
KCacheGrind:	KDE frontend for Callgrind/Cachegrind.

1 INTRODUCTION

Fontconfig is a library that provides information about fonts installed in the system to applications. It is used by most graphical applications on Linux.

Fontconfig is responsible for finding font files installed in the system, extracting information from them and caching the information for a fast lookup. Applications can retrieve this information or search for fonts matching the given criteria. Fontconfig has a flexible configuration that allows the user to set rules that affect the information given to applications (for example, to set fallback fonts or to overwrite some rendering properties).

In this thesis the most common usage of fontconfig by applications is explored and the performance of the current fontconfig implementation is evaluated. Possible optimizations are implemented and impact on performance is measured.

2 BACKGROUND

1.1 Motivation

The idea that the current fontconfig implementation may not be well optimized came from analyzing the CPU usage during the start of graphical applications on Linux. The motivation of the author was to improve the startup times of applications.

The most obvious suspect for slow startups would be disk IO. However, some applications were observed to start slowly even during the repeated starts with enough unused operational memory, which allows all accessed files to be cached. Moreover, it was observed that CPU is highly utilized during the start of an application, which suggests that the startup operation is CPU-bound.

A rough measurement was done on various graphical applications in order to determine which part of an application is responsible for the high CPU usage during startup. The applications were ran inside callgrind and were manually terminated as soon as their window appeared. Obviously, this measurement method is not reliable because it depends on the reaction time of a human tester and it measures the CPU usage not only during the startup, but also during the short runtime and shutdown. Nevertheless, the measurements have shown a clear culprit, a fontconfig library. A disproportionate amount of instructions executed during the measurement originated from it.

TABLE 1. Callgrind measurement of start and immediate shutdown of Kate.

Instruction count	Library
1,082,214,650	libfontconfig
934,421,487	libQt5Core
413,363,774	libc
174,812,595	libcrypto
166,361,614	ld
138,371,618	libQt5Gui

Table 1 shows first six libraries ordered by the amount of instructions executed during the startup test performed on a Kate application. Kate is a well known text editor from the KDE application suite and it was used as a representative of a lightweight GUI application. It is not surprising to see the Qt libraries among the top six because Kate is a Qt based application and Qt is doing all the heavy work. Similarly, it is not surprising to see there *libc*, which is the standard C library and *ld* which takes care of loading dynamic libraries. A bit surprising is the time spent in *libcrypto*, which is the OpenSSL library. It turned out that most of the time was spend running a FIPS self-test during the OpenSSL initialization. This may be a candidate for another optimization. Finally, the most expensive library, in terms of instructions executed, is *libfontconfig*. There is no obvious reason why finding a font in a font database should use more CPU than any other component of the application.

Other graphical applications were measured and the amount of instructions executed by the fontconfig library was similarly high in them. The examples include Firefox, Gimp, Konsole, LibreOffice Writer and Nautilus.

1.2 Fontconfig

> The original ideas for Fontconfig came about during the design and deployment of the Xft library. Xft was originally designed to connect fonts rasterized with the FreeType library with the X Render Extension to the X window system. As FreeType provided no font configuration or customization mechanisms, Xft included its own. Extending the problem of font configuration by creating yet another incompatible configuration file format. (1)

Fontconfig was originally developed by Keith Packard and was introduced in 2002. It is written in C. It is open source and it was released under the MIT license. It is commonly used in Linux distributions and in some Linux-based systems, such as MeeGo from Nokia, Tizen from Samsung and Chrome OS from Google. (2, 3)

It is important to note that fontconfig’s tasks are only to inform applications about the available fonts in the system and to perfom queries on them. In a way, fontconfig is a specialized database for fonts. Fontconfig does not render fonts nor does it enforce any font rendering method on applications. Fontconfig also does not have any dependency on the X server. Most applications do not use fontconfig library directly but through some GUI framework. Notable GUI frameworks that use fontconfig are: Qt, GTK, FLTK, and wxWidgets. (4)

1.2.1 Function

During initialization, the fontconfig library reads the system's and optionally user’s configuration files. Then it examines font directories and extracts information from the font files found in them. The font files are parsed using the FreeType library. The extracted information is stored in cache files inside the current user’s home directory so that the fonts do not need to be parsed every time.

Extracting the information takes a considerable time which would slow down the first application that attempts to use fontconfig after the font files have changed. To solve that, fontconfig provides an **fc-cache** tool, which can be used to generate cache files in advance. It can be used to generate system-wide cache files, which can be used by all users. Linux distributions typically invoke **fc-cache** from hooks in their packaging system after a font package has been installed or removed.

A font in fontconfig database is represented by a set of properties. Each property has a name, a type and a list of values. The properties can be split into two categories: The first category consists of well known properties, such as e.g. *family, style*, *size and width*. which are recognized by fontconfig and can be used for font matching and completition. The second category consists of any arbitrary properties that are not interpreted by fontconfig, but are passed to the application, which may use them for example to set up a font rasterizing engine. Figure 1 shows an example of the data stored about the Arial font.

```
file=arial.ttf
family=Arial
familylang=en
style=Normal,obyčejné,Standard,Κανονικά,Regular,Normaali ...
stylelang=ca,cs,de,el,en,fi,hu,it,nl,pl,ru,sk,sl,vi,eu
fullname=Arial
fullnamelang=en
slant=0
weight=80
width=100
foundry=Mono
index=0
outline=True
scalable=True
charset=20-7e a0-17f 18f 192 1a0-1a1 1af-1b0 1cd-1dc ...
lang=aa|af|ar|av|ay|az-az|be|bg|bi|bin|br|bs|bua|ca|ce ...
fontversion=184812
capability=otlayout:arab
fontformat=TrueType
decorative=False
postscriptname=ArialMT
color=False
symbol=False
```

FIGURE 1. Information about the Arial font extracted from fontconfig database. Retrieved using the fc-cat tool and edited, some properties were shortened for the display.

The application can retrieve the font information using three main querying functions:

- `FcFontMatch`: Retrieves a single font which is the best match for a given pattern.
- `FcFontSort`: Retrieves all fonts sorted by the closeness to a given pattern.
- `FcFontList`: Retrieves all fonts that fully match a given pattern.

All three querying functions take a *pattern* as a parameter. The *pattern* is a set of font properties and their values. The *pattern* and *font* have so similar structure that they are both internally represented by the same data type called `FcPattern`. All of those functions search in fonts installed in the system. A variant that searches in a set of fonts supplied by the caller exists for all of them.

These querying functions are described in detail in chapter 2.1.

1.2.2 Configuration

Fontconfig reads the configuration files from the system's and user's directories. The configuration files are in the XML format. The configuration contains basic settings, such as directories where the font and cache files are stored, and rules that affect the font matching process. These rules define modifications, which are applied to a *pattern* before it is used for matching, or to a *font* before it is given to an application, or to a *font* when it is read from a font file, if the *pattern* or *font* fulfil some condition.

```
<match target="pattern">
    <test name="family">
        <string>Courier</string>
    </test>
    <edit name="family" mode="append" binding="same">
        <string>Liberation Mono</string>
    </edit>
</match>
```

FIGURE 2. Example match from fontconfig configuration

Figure 2 shows a rule that targets *patterns*. It appends “Liberation Mono” to the list of values in the “family” property if it contains the “Courier” value. Therefore, whenever the application makes a query for a “Courier” font, the pattern is edited making “Liberation Mono” a second choice. Effectively, it means that “Liberation Mono” becomes a fallback font in case “Courier” is not installed.

```
<match target="font">
    <test name="lang" compare="contains">
        <string>ja</string>
    </test>
    <edit name="embeddedbitmap" mode="assign">
        <bool>true</bool>
    </edit>
</match>
```

FIGURE 3. Example match from fontconfig configuration

Figure 3 shows a rule that targets *fonts*. It sets the “embeddedbitmap” property to true if the font supports Japanesse language. This is one of the properties that is not interpreted by fontconfig, but is only passed to the application which may use it as a hint on how to render the font.

2 ANALYSIS

The source and profiler records of fontconfig were examined. The algorithms of main entry functions and hot spots were analyzed.

2.1 Main entry functions

2.1.1 FcFontMatch

`FcFontMatch` finds and returns the best matching font for a given pattern. It is actually a specialization of a more general `FcFontSetMatch` function, which searches in fonts supplied by the caller. `FcFontMatch` does not require the caller to provide a list of fonts but searches in all available fonts.

The search is done by iterating over all fonts and calculating a *score* which represents the distance of the font from the given pattern. Fonts and patterns are compared by comparing their individual properties. All well-known properties have a defined priority for comparison. The score is represented by a tuple of numbers $(s_1, s_2, s_3, \ldots)$, where each number s_i holds the result of comparison of a property with priority i. The scores are compared lexicographically and the font with the lowest *score* is returned.

The properties in both fonts and patterns can hold multiple values. When such properties are compared, all combinations of values are compared and the lowest score is used. Individual property values are compared using a special comparison function associated with the property (such as `FcCompareFamily` for *family* property or `FcCompareNumber` for *size* property).

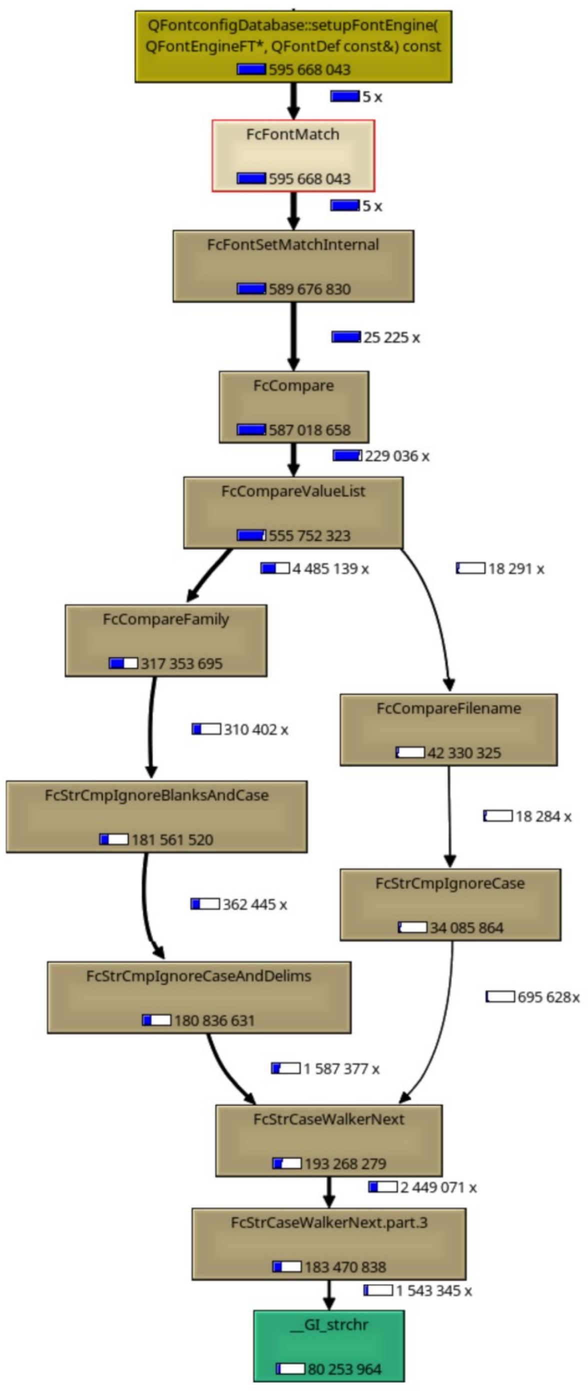


FIGURE 4. Diagram of expensive function calls originating from FcFontMatch.

In practice the properties in the pattern usually contain multiple values because fallback alternatives were added to them by the mechanism described in chapter 1.2.2 Configuration.

One can see that the number of operations performed depends on the amount of fonts in the system, the number of properties in the pattern and the number of values in every property. As there is only a constant amount of well-known properties, the number of properties involved in the comparison can be at most equal to that constant. In a big O notation the complexity is $O(n*m)$, where n is the amount of fonts in the system and m is the longest list of values in the pattern.

Figure 4 is a diagram showing which functions and how many times were called as a result of five calls of `FcFontMatch`. The test was ran on a machine with approximately 5,000 fonts installed (not an uncommon number when the TeX typesetting system is installed) and with a default fontconfig configuration, which extends the properties of every pattern with multiple fallback values. It can be seen that the initial five calls of `FcFontMatch` caused 25,225 comparisons of a font to a pattern, which caused 229,036 comparisons of their properties, which in turn caused 4,485,139 comparisons of family names and 18,291 comparisons of file names. The final property comparisons involved fairly expensive custom implementations of string comparators.

2.1.2 FcFontSort

`FcFontSort` returns a list of all fonts sorted by the closeness to a given pattern. It is actually a specialization of a more general `FcFontSetSort` function, which works on a set of fonts supplied by the caller. `FcFontSort` does not require the caller to provide a list of fonts, but works with all the available fonts.

`FcFontSort` starts by calculating the *score* representing the closeness to the given pattern for every font in a set. The score is calculated using the same algorithm as described in `FcFontMatch`. After that, the fonts are sorted using the standard C `qsort` function and a custom compare function that compares the font scores to each other.

`FcFontSort` does the same huge amount of font property comparisons as `FcFontMatch.` In addition, it spends some extra time for sorting the resulting set.

2.1.3 FcFontList

`FcFontList` returns a list of all fonts matching a given pattern. Unlike `FcFontMatch` and `FcFontSort`, which always return at least one font, `FcFontList` returns only fonts that perfectly match a given pattern. The fonts in the returned list contain only properties chosen by the caller and do not contain duplicates.

`FcFontList` starts by creating a hash set that serves for removing duplicities in the returned list. Then it iterates over all fonts and for each it tests whether it matches the given pattern. This test is different from the one used in `FcFontMatch` and `FcFontSort` and surprisingly it interprets some font properties slightly differently. For example, `FcFontMatch` ignores spaces when comparing *family* properties, but `FcFontList` does not, `FcFontMatch` interprets glob characters inside a *file* property, but `FcFontList` does not.

If a font passes the test, `FcFontList` adds it into the hash set unless it is already there. Only properties that will be returned to the caller are considered when testing and adding fonts to the hash set.

In the end, `FcFontList` pulls fonts from the hash set and creates a regular list, which is returned to the caller.

2.2 Disk IO

An analysis of file accesses performed by fontconfig shows that it is well designed in this regard. It uses its own cache files that aggregate information about installed fonts therefore it does not need to parse them if they have not changed. The cache file format is designed in such a way that the files can be read-only mapped into memory and used directly without any transformation. As every application, which uses fontconfig, maps these cache files into their memory in read-only mode, they can be shared among them, thus saving memory and removing the need to read them repeatedly from a disk.

3 OPTIMIZATIONS

3.1 Reimplementation consideration

The first considered course of action was to reimplement fontconfig from ground up. It would allow to completely redesign the internals of the library to use data structures and algorithms that would allow to match fonts effectively. It would also be possible to implement it in a language with higher level abstractions, such as C++ or Rust.

However, any reimplementation should keep the same API and ideally also the same ABI as the original fontconfig library to ease its adoption as a fontconfig replacement. It is unlikely that the current users of fontconfig would be willing to add an additional font database backend to their code even if it meant an improved performance.

A closer inspection of fontconfig's API showed that fontconfig currently exposes a wide range of functionality, including functions that are just loosely related to font matching. Examples include UTF-8, UTF-16 and UCS-4 conversion functions, functions for manipulating filenames, and functions for atomic file operations. A reimplementation would have to recreate all the fontconfig's functionality, including those functions. Considering that the current implementation works and it was tested by years of usage, it would be risky and take too much effort to rewrite everything. It is only the core of the library that needs performance improvements.

Additionally, the public API of fontconfig exposes an internal structure of some important data types, such as `FcValue` or `FcFontSet`, which forces any alternative implementation to implement them in the same way. That in turn limits algorithms that can work on top of them.

Upon considering the difficulties of a full reimplementation and the time frame of this thesis, it was decided to make incremental optimizations to the existing fontconfig instead.

3.2 Determining most common uses

In order to create optimizations that will improve real-world usages of the library, it was necessary to determine what kinds of patterns applications most commonly search.

The Linux dynamic loader can be configured using an `LD_PRELOAD` environmental variable to load a library before all others. This can be used to override functions from other libraries. (5) This trick was used to load a small custom library instead of fontconfig to all applications. The library records all calls of relevant fontconfig functions and their parameters to a file. It forwards the function calls to original functions from real fontconfig to keep the behavior same for the applications.

```
typedef FcPattern * (*OrigFcFontSetMatch)(FcConfig *config,
FcFontSet **sets, int nsets, FcPattern *p, FcResult *result);

FcPublic FcPattern *
FcFontSetMatch (FcConfig    *config,
                FcFontSet   **sets,
                int         nsets,
                FcPattern   *p,
                FcResult    *result)
{
    FcChar8* description = FcNameUnparse(p);
    preload_log("FcFontSetMatch: %s\n", description);
    free(description);

    OrigFcFontSetMatch orig =
        (OrigFcFontSetMatch) dlsym(RTLD_NEXT, "FcFontSetMatch");

    return orig(config, sets, nsets, p, result);
}
```

FIGURE 5. Snippet from library for recording fontconfig usage. It shows a function that intercepts calls to FcFontSetMatch.

This library was used to record fontconfig queries by various Linux graphical applications, especially testing applications using different GUI frameworks. These records were analyzed and some highlights are presented in Table 2.

TABLE 2. Highlights of typical fontconfig usage by various applications

Application	**Typical queries**	
	Function	**Pattern parameter**
KDE applications	FcFontList	empty
	FcFontMatch	has *file* and *family* and few other properties
	FcFontSort	has *family* property and few others
GTK applications	FcFontList	empty
	FcFontMatch	has *family* and many other properties
Chromium browser	FcFontMatch	has *family* property and few others
	FcFontSort	same as match
Firefox browser	FcFontSort	has *family* property and few otherss
	FcFontMatch	has *family* property and many others
Java Swing applications	FcFontMatch	has *family* property and few others

Many applications start with a query for all fonts using `FcFontList` with an empty pattern. All observed applications use fontconfig during the startup to retrieve fonts for their GUI. In case of fonts for GUI the fonts typically exist and are matched sucessfully. In case of KDE applications, the query even includes the *file* property, which points to the actual file of the font. Applications that display content, such as word processors or web browsers, typically query for additional fonts when the displayed content changes.

3.3 Benchmarks

With the knowledge of most commonly used queries, it was possible to create benchmarks that measure their performance. A Google's microbenchmark support library (6) was used. Table 3 shows the performance measurements of the fontconfig library before optimization. These numbers will be used as a baseline for measuring improvements by upcoming optimizations.

The fontconfig library was compiled using a clang version 3.8.0 (tags/RELEASE_380/final 262553) with `-O3` level of optimizations. The gcc compiler was also tested (versions gcc-4.8 and gcc-6.0) and while it produced code that worked correctly and the optimizations brought similar speedups, there were random fluctuations in perfomance. A change in a function occasionally caused a completely unrelated function to perform faster or slower. Clang was chosen over gcc to be used for micro-benchmarking the optimizations in this thesis as the code compiled with it had a more predictable perfomance.

Benchmarks were run on a x86_64 Linux system with Intel i7-3770K CPU using performance governon.

TABLE 3. Measurements before optimizations

Function	**Pattern**	**Time [ns]**	**σ [ns]**
Match	common_kde	16,979,822	6,619
Match	common_gnome	16,774,682	7,370
Match	common_chromium	16,674,855	9,558
Match	common_firefox	15,668,690	4,959
Match	existing_file	1,527,992	928
Match	non_existing_file	1,529,358	4,038
Match	existing_file_with_globs	1,824,564	993
Match	existing_family	383,132	351
Match	not_existing_family	377,222	633
Sort	common_kde_1	18,755,030	660
Sort	common_kde_2	12,424,092	4,343
Sort	common_firefox_1	18,721,599	26,949
Sort	common_firefox_2	19,694,770	15,660
Sort	common_firefox_3	2,127,628	10,978
List	empty_pattern	6,925,237	919
List	existing_file	1,557,534	120
List	not_existing_file	1,559,173	131,082
List	existing_file_with_globs	1,445,537	3,248
List	existing_family	258,622	5,717
List	not_existing_family	248,760	757

"*Match*", "*Sort*" and "*List*" are measuring the performance of `FcFontMatch`, `FcFontSort` and `FcFontList` functions respectively. The patterns with names "common_*something*" represent a pattern that has the same structure as a pattern observed in a real-world application. The patterns with descriptive names, such as an “empty_pattern” describe an artificial pattern which was created to measure performance of a specific input.

For more details about the benchmarks see Appendix 1.

3.4 Optimizations

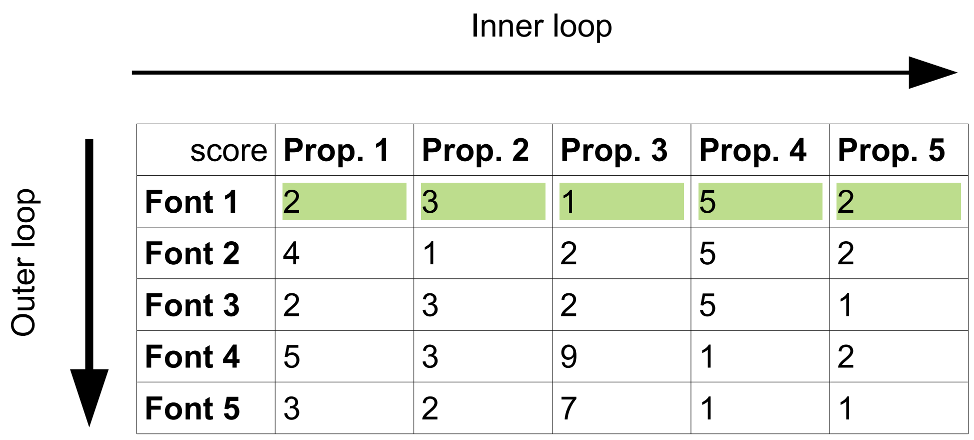
3.4.1 Rewriting FcFontMatch algorithm

Due to the algorithm described in chapter 2.1.1 FcFontMatch, the `FcFontMatch` function performs a large amount of property comparisons, some of which are very computationally expensive.

The analysis of most common queries showed that almost all patterns in `FcFontMatch` queries contain the *family* property and some also contain the *file* property. In almost all cases a font that matches precisely the query is found. This is because applications are mostly querying for fonts that will be used in their GUI and those fonts are known to exist in the system.

The *file* property has the highest priority of all well-known properties, which means that if a single font with a matching *file* property is found, it could be returned immediatelly because none of the remaining properties can make another font match more closely. Similarly the *family* property also has a fairly high priority and if a single match is found, there is no point in comparing properties with lower priorities. However, the original `FcFontMatch` algorithm compares all properties of all fonts.

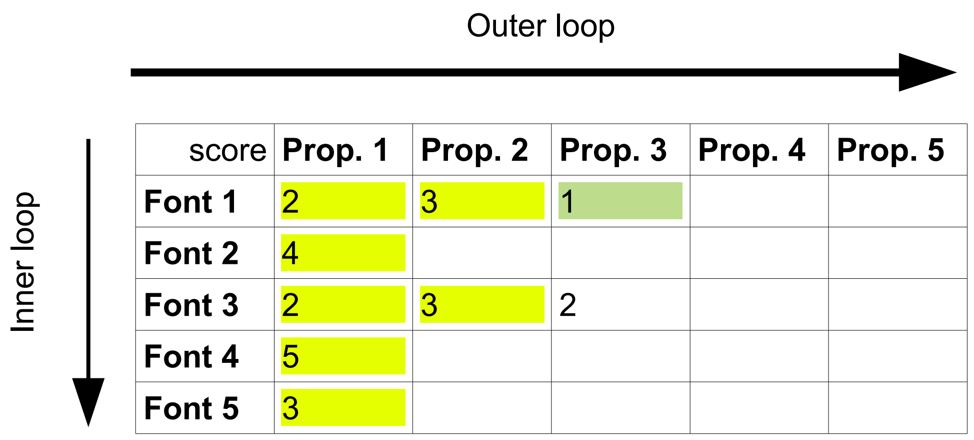
A new algorithm was designed that instead iterates over all well-known properties in the pattern from the highest to the lowest priority. For each property it reduces the set of candidates for the best matching font. Initially, the candidate set contains all fonts. The set is iterated and for every font in it only the current property is compared against the property in the pattern. The fonts that were closest match for the property are tracked. In the end if there was exactly one closest matching font, it is returned. If there were multiple fonts matching equally well, the candidate set is reduced to them and the search continues with the next property. There is always at least one best matching font if there is at least one font in the initial set.



score	Prop. 1	Prop. 2	Prop. 3	Prop. 4	Prop. 5
Font 1	🟩 2	🟩 3	🟩 1	🟩 5	🟩 2
Font 2	4	1	2	5	2
Font 3	2	3	2	5	1
Font 4	5	3	9	1	2
Font 5	3	2	7	1	1

FIGURE 6. Representation of the original FcFontMatch algorithm.

Figure 6 shows representation of the original `FcFontMatch` algorithm. First, all properties are evaluated for Font 1, then all properties for Font 2, etc. All properties of all fonts are evaluated before Font 1 (green) can be determined as the best match.



score	Prop. 1	Prop. 2	Prop. 3	Prop. 4	Prop. 5
Font 1	🟨 2	🟨 3	🟩 1		
Font 2	🟨 4				
Font 3	🟨 2	🟨 3	2		
Font 4	🟨 5				
Font 5	🟨 3				

FIGURE 7. Representation of the new FcFontMatch algorithm.

Figure 7 shows representation of the new `FcFontMatch` algorithm. The candidate set starts with all fonts, but is reduced to Font 1 and Font 3 after evaluating Property 1 (yellow). It stays the same after evaluating Property 2.

The search can stop after evaluating the candidate set against Property 3 because Font 1 (green) is known to be the best match.

This algorithm greatly reduces the amount of expensive property comparisons. However, while the old algorithm was able to find the best matching font in a single pass over all fonts, the new one needs multiple passes during which it keeps track of the candidate set. Thanks to the fact that the list of all fonts can be indexed, a bitset was chosen to represent the candidate set – the value of the *n*-th bit represents whether the *n*-th font is still a candidate for the best match. This way only one dynamic allocation of small size is required and removing fonts from the set is trivial.

TABLE 4. Speed comparison of the effects of FcFontMatch optimization.

Function	**Pattern**	**%**	**Before [ns]**	**After [ns]**	**σ [ns]**
Match	common_kde	-92%	16,979,822	1,400,777	6,619
Match	common_gnome	-11%	16,774,682	14,929,744	7,370
Match	common_chromium	-10%	16,674,855	15,084,903	9,558
Match	common_firefox	4%	15,668,690	15,031,823	4,959
Match	existing_file	-8%	1,527,992	1,409,579	928
Match	non_existing_file	-8%	1,529,358	1,405,771	4,038
Match	existing_file_with_globs	-7%	1,824,564	1,692,467	993
Match	existing_family	-15%	383,132	326,661	351
Match	not_existing_family	26%	377,222	474,051	633
Sort	common_kde_1	2%	18,755,030	19,163,212	660
Sort	common_kde_2	2%	12,424,092	12,623,249	4,343
Sort	common_firefox_1	2%	18,721,599	19,126,093	26,949
Sort	common_firefox_2	3%	19,694,770	20,201,083	15,660
Sort	common_firefox_3	3%	2,127,628	12,489,881	10,978
List	empty_pattern	2%	6,925,237	7,065,266	919
List	existing_file	2%	1,557,534	1,587,054	120
List	not_existing_file	1%	1,559,173	1,578,804	131,082
List	existing_file_with_globs	2%	1,445,537	1,479,631	3,248
List	existing_family	1%	258,622	261,303	5,717
List	not_existing_family	3%	248,760	256,553	757

Table 4 shows the speedups achieved using the described optimization. The *Match/common_kde* benchmark became much faster because it searches for a pattern with *file*, *family*, *pixelsize*, and *index* properties. Prior to the optimization, all of these properties had to be compared. After this optimization, the matching can stop immediatelly after a successful match of the *file* property.

Other *Match* benchmarks observed speedups, too. The benchmark *Match/not_existing_family* became slower because it had to compare the *family* property of all fonts, both before and after the optimization, but after it also had to keep track of the candidate set. It is however an artificial test and even with the slowdown, it is quite fast compared to others.

The *Sort* and *List* benchmarks did not receive any significant speedups or slowdowns as the optimization did not touch any code executed by them.

3.4.2 Value preprocessing

While the new FcFontMatch algorithm reduces the amount of property comparisons, some are still inevitable. In addition, it is not possible to do a similar optimization for FcFontSort. It does not search for a single best matching font, but orders all fonts by their closeness to the pattern. Therefore, it must compare all properties of all fonts to the given pattern.

Profiling the benchmarks showed that the *family* and the *file* properties have the most expensive comparison functions.

The *family* property is a UTF-8 encoded string and is compared using a custom string comparison function. This function ignores a character case (in a UTF-8 correct way) and skips spaces. For example, strings “Times New Roman” and “timesnewroman” are considered equal. This special behavior is the reason why the standard strcmp or strcasecmp can not be used, even if it is likely that they would be faster than the custom implementation.

The *file* property is also a string and is compared in a case-sensitive way first, followed by a case-insensitive comparison. A string that matches using the case-sensitive comparison is considered more similar than a string that matches only after ignoring a case. In addition, so called globs (“?” and “*” characters) can be present in the *file* property and serve as wildcards. The comparison function attempts to match the strings while replacing the wildcard characters with matching characters from the other string. For example, a string “times.ttf” will be matched by patterns “times.ttf”, “TIMES.TTF” and “times.*”, in decreasing order of closeness.

It is obvious that fontconfig does lot of redundant work during every comparison, such as converting UTF-8 characters to lowercase or glob matching. It would be beneficial if it could preprocess the cached font data into a format that would allow a faster comparison. The best option would be to introduce some kind of index into the font set, for example in a form of a trie. Unfortunately, the fontconfig API allows querying any arbitrary set of fonts supplied by the application, not only fonts from fontconfig cache. Therefore, it must be able to perform the queries even on font sets that were not preprocessed. This makes it difficult to fundamentally change the font set format.

As a basic optimization, a simple optional field with arbitrary preprocessed data was added to the font property data type. If it is present, it can speed up the property comparison, but if it is ommited the properties can be compared in the regular way. Fontconfig now preprocesses all the font properties when creating the cache files. When a querying function is called, the query pattern is preprocessed as well and all the comparison functions can access both the optional preprocessed value and the actual value.

Below is a summary of implemented preprocessing steps and improvements to the comparison functions:

- *family* property: Preprocessed by calculating the hash of the string after it was converted to lowercase and spaces were removed. The comparison function can quickly reject two strings if the hash is present but does not match. If the hash is missing or is equal, a full comparison is performed.
- *file* property: Preprocessed by calculating the hash of the string in lowercase. In addition, information on whether the string contains glob (“*”, “?”) characters is stored. The comparison function can skip both case-sensitive and case-insensitive comparisons if hashes are present but not matching. It can also skip a comparison using globs if no globs are present in the pattern (a common case).
- All other string properties: String is hashed and a comparison function quickly rejects two strings if hashes are not matching.

There is an opportunity for preprocessing other properties, such as *lang*.

TABLE 5. Speed comparison of the effects of value preprocessing.

Function	Pattern	%	Before [ns]	After [ns]	σ [ns]
Match	common_kde	-76%	1,400,777	332,228	376
Match	common_gnome	-37%	14,929,744	9,411,893	51,172
Match	common_chromium	-37%	15,084,903	9,551,819	69,481
Match	common_firefox	-37%	15,031,823	9,509,700	20,079
Match	existing_file	-79%	1,409,579	293,638	10
Match	non_existing_file	-79%	1,405,771	295,710	2,523
Match	existing_file_with_globs	-82%	1,692,467	299,461	2,824
Match	existing_family	-10%	326,661	292,995	993
Match	not_existing_family	-6%	474,051	446,242	2,616
Sort	common_kde_1	-28%	19,163,212	13,821,311	1,759
Sort	common_kde_2	-21%	12,623,249	10,021,203	117,270
Sort	common_firefox_1	-28%	19,126,093	13,817,795	68,154
Sort	common_firefox_2	-27%	20,201,083	14,776,120	82,414
Sort	common_firefox_3	-19%	12,489,881	10,136,953	95,042
List	empty_pattern	2%	7,065,266	7,231,435	45,431
List	existing_file	-1%	1,587,054	1,573,899	21,114
List	not_existing_file	1%	1,578,804	1,594,913	57,576
List	existing_file_with_globs	-1%	1,479,631	1,466,482	15,976
List	existing_family	-1%	261,303	259,634	4,249
List	not_existing_family	-4%	256,553	247,386	13,296

Table 5 shows the effects of value preprocessing on the benchmarks. The *Match* and *Sort* benchmarks benefited heavily as they perform many property comparisons. The times of *List* benchmarks did not change significantly because fonts are compared using a different method in `FcFontList` as is described in chapter 2.1.3 FcFontList.

3.4.3 Reducing heap allocations

Fontconfig defines a structure `FcStrSet`, which holds a set of strings, and `FcStrList`, which serves as an iterator for `FcStrSet`. These structures are used widely across fontconfig. The primary way of iterating over `FcStrSet` is by creating `FcStrList` using `FcStrListCreate` and destroying it after using `FcStrListDone`.

The creation function allocates `FcStrList` on heap, even that in most cases it is only needed locally and it is destroyed in the same scope. It could be easily stored on stack to reduce the amount of dynamic allocations.

To make that possible, the (de-)initialization of `FcStrList` was refactored out of the `-Create` and `-Done` functions into new `-Initialize` and `-Release` functions. All code using `FcStrList` was updated to store it on stack whenever possible.

None of the benchmarks have shown any significant speedup or slowdown. While there is not any improvement measured by the benchmarks, it is a good practice to remove dynamic allocations if possible. The allocator is shared with the rest of the application. Thus, reducing the amount of allocations and deallocations made by fontconfig may reduce heap fragmentation and have a positive effect on the rest of the application. (Error: Reference source not found)

3.4.4 Refactoring FcStrCaseWalkerNext

`FcStrCaseWalkerNext` is a function which walks a string and retrieves the next character converted to lowercase. It can optionally skip some specific characters (for example spaces). It is the building stone of fontconfig's string comparison and string hashing functions.

While the previous optimizations greatly reduced the amount of string comparisons and thus, the amount of invocations of this function, it is still present in many hot paths. Therefore, it was beneficial to fine tune it.

The main pain point of `FcStrCaseWalkerNext` was its ability to skip over characters. The function takes a string called `delims` containing all characters that should be skipped. When the function retrieves the next character, it checks whether it is present in the `delims` string using the `strchr` function. Currently, fontconfig used this function in three modes: not skipping over any characters, skipping over space and, in one rare case, skipping over space and a dash character. Therefore, the `delims` was at most a two-characters-long string.

As optimization, the `FcStrCaseWalkerNext` function was split to three separate functions, each optimized for its own purpose:

- `FcStrCaseWalkerNext`: Does not skip over any characters.
- `FcStrCaseWalkerNextSkipDelim`: Skips a single given character.
- `FcStrCaseWalkerNextSkipDelims:` Can skip an arbitrary amount of characters. However, it is expected that the list of delimiters is small (in practice, currently, at most two delimiters), thus calling a standard `strchr` function would be incredibly wasteful. Instead, the equivalent of `strchr` is open-coded inline.

This lead to a slight increase of code size and code duplication, which is a tradeoff for faster code. In case of very hot code paths, it is worth the performance improvement.

TABLE 6. Speed comparison of the effects of FcStrCaseWalkerNext optimization.

Function	**Pattern**	**%**	**Before [ns]**	**After [ns]**	**σ [ns]**
Match	common_kde	-6%	332,228	311,323	902
Match	common_gnome	-6%	9,411,893	8,802,655	2,714
Match	common_chromium	-7%	9,551,819	8,886,549	2,526
Match	common_firefox	-7%	9,509,700	8,844,730	1,070
Match	existing_file	1%	293,638	289,621	23
Match	non_existing_file	1%	295,710	293,542	389
Match	existing_file_with_globs	3%	299,461	291,578	992
Match	existing_family	3%	292,995	282,928	966
Match	not_existing_family	3%	446,242	430,772	318
Sort	common_kde_1	-6%	13,821,311	13,039,643	687
Sort	common_kde_2	-8%	10,021,203	9,196,111	6,319
Sort	common_firefox_1	-6%	13,817,795	13,015,027	2,444
Sort	common_firefox_2	-7%	14,776,120	13,750,692	6,708
Sort	common_firefox_3	-10%	10,136,953	9,145,413	18,151
List	empty_pattern	-10%	7,231,435	6,490,128	2,528
List	existing_file	-43%	1,573,899	896,997	13,842
List	not_existing_file	-44%	1,594,913	894,718	47,873

Function	Pattern	%	Before [ns]	After [ns]	σ [ns]
List	existing_file_with_globs	-42%	1,466,482	844,987	1,002
List	existing_family	-17%	259,634	215,111	172
List	not_existing_family	-17%	247,386	205,800	2,038

Table 6 shows the speedups after the described optimizations. The *List* benchmarks received the most significant speed increase as they do not use the preprocessed values and compare all string properties using their raw values, therefore invoke the `FcStrCaseWalker` function most often.

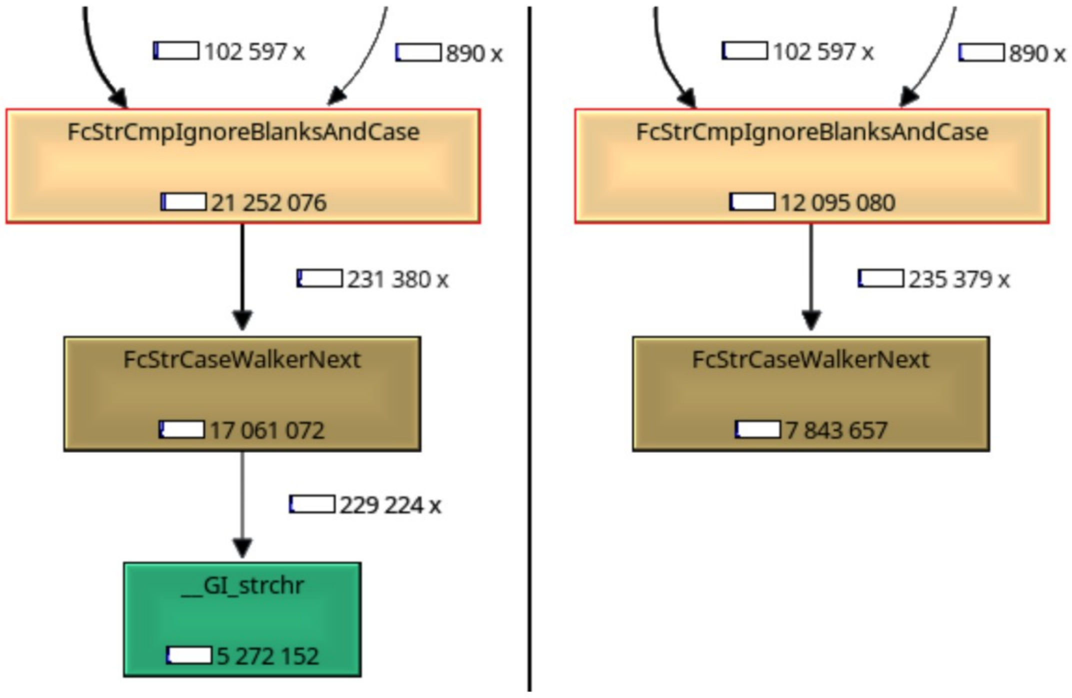


FIGURE 8. KCacheGrind visualizaton of costs of calling a string comparison function before (left) and after (right) the optimization.

3.4.5 Micro-optimization in FcCompareValueList

`FcCompareValueList` compares two lists of values by comparing every value from the first list with every value from the second list. Each comparison of two values gives a score representing their closeness and the smallest closeness is returned.

The function contains two nested cycles – an outer one iterating over one list and an inner one over the other list. The function is used in the way that the list iterated in the outer cycle originates from a pattern and usually contains tens of values, while the list iterated in the inner cycle originates from a font and usually contains a single value.

Switching the lists in the inner and outer cycles has no effect on the functionality, but it improves the performance as the cycle with more iterations becomes tigther. As `FcCompareValueList` is called many times per query, this micro-optimization has a visible effect on the overall performance.

TABLE 7. Speed comparison of the effects of FcCompareValueList optimization.

Function	**Pattern**	**%**	**Before [ns]**	**After [ns]**	**σ [ns]**
Match	common_kde	-1%	311,323	307,688	436
Match	common_gnome	-38%	8,802,655	5,420,507	4,584
Match	common_chromium	-38%	8,886,549	5,505,153	2,841
Match	common_firefox	-36%	8,844,730	5,626,977	16,454
Match	existing_file	-1%	289,621	285,435	20
Match	non_existing_file	-2%	293,542	287,079	456
Match	existing_file_with_globs	-1%	291,578	287,306	358
Match	existing_family	-1%	282,928	279,377	750
Match	not_existing_family	-1%	430,772	425,758	302
Sort	common_kde_1	-26%	13,039,643	9,670,799	385
Sort	common_kde_2	-21%	9,196,111	7,292,789	28,721
Sort	common_firefox_1	-26%	13,015,027	9,658,356	18,380
Sort	common_firefox_2	-25%	13,750,692	10,381,704	44,043
Sort	common_firefox_3	-18%	9,145,413	7,504,562	13,353
List	empty_pattern	-2%	6,490,128	6,378,110	2,463
List	existing_file	0%	896,997	897,239	595
List	not_existing_file	0%	894,718	893,086	57,168
List	existing_file_with_globs	0%	844,987	840,859	2,400
List	existing_family	-1%	215,111	213,489	1,969
List	not_existing_family	0%	205,800	205,551	110

Table 7 shows the speedups of the described optimization. The functions that invoke `FcCompareValueList` most often naturally benefited the most. The *List* benchmarks were not affected as they do not call the function at all.

3.4.6 Reduce FcObjectFromName call amount

Each font property in fontconfig has a name, which is a string, and an id, which is a number. A property id can be found from the given name using a function generated by gperf. Some functions, such as `FcFontList`, retrieve a `FcObjectSet` object, which holds a list of strings representing the requested properties of fonts to be returned.

`FcObjectSet` stores properties as strings and every place, which needs to work with the properties stored inside, must convert them from strings to ids. This causes the same property name to be looked up many times during a single query. Although the gperf generated function is quite fast, it would be faster if the conversion did not have to be done more than once.

Unfortunately, the internals of the `FcObjectSet` structure are exposed in the public API, therefore it is not possible to modify them without breaking compatibility.

As a compromise, a new structure called `FcObjectIdSet`, which holds object ids, was added. `FcObjectSet` serves as a public mutable facing representation of object set and is converted to a private immutable `FcObjectIdSet` in entry point functions. All remaining code was modified to use `FcObjectIdSet` and all redundant object name lookups were eliminated.

The benchmarks have shown only a small speed increase (-5%) for the *List/empty_pattern* benchmark. Other benchmarks were affected even less.

3.4.7 Hint branch predictor

Modern processors have a deep instruction pipeline. In order to keep the optimal perfomance, the processor tries to keep the pipeline filled at all times. If it encounters a conditional jump, it will predict whether it will be taken or not and speculatively execute the following instructions. If it determines that its prediction was incorrect, the results of the speculatively executed instructions are thrown away and the correct ones are executed instead. The time executing the thrown away instructions was wasted. It is therefore important to minimize the amount of mispredictions.

Most compilers have intrinsics that allow a programmer to specify whether a condition is likely or unlikely to be true. These intrinsics affect the structure of generated assembly code, which improves the instruction cache utilization and may hint the branch predictor, too. A commonly used name for such intrinsic is `__builtin_expect` and it is often wrapped in macros called `likely` and `unlikely`. (8)

Such intrinsics are widely used in the Linux kernel code and in some performance-sensitive user space code. However, they should be used sparsely and with care as a badly estimated probability of a condition may hurt the perfomance instead of improving it. (9)

Intrinsics are compiler specific and may cause portability issues. Luckily, the `__builtin_expect` intrinsic can be replaced with an empty macro without affecting the code functionality on compilers which do not support it.

As an experimental optimization, `__builtin_expect` was added to few hot paths in `FcCompareFamily`, `FcCompareValueList`, and `FcStrCaseWalkerNext` functions.

TABLE 8. Speed comparison of the effects of adding __builtin_expect.

Function	Pattern	%	Before [ns]	After [ns]	σ [ns]
Match	common_kde	-1%	308,819	305,621	1,179
Match	common_gnome	-3%	5,558,880	5,390,364	12,425
Match	common_chromium	-2%	5,603,347	5,466,933	18,512
Match	common_firefox	-3%	5,616,794	5,437,409	18,152
Match	existing_file	1%	288,278	290,082	552
Match	non_existing_file	1%	289,415	291,189	812
Match	existing_file_with_globs	0%	289,198	290,071	392
Match	existing_family	0%	280,991	279,957	479
Match	not_existing_family	1%	427,278	430,985	1,695
Sort	common_kde_1	0%	9,614,146	9,600,108	26,576
Sort	common_kde_2	0%	7,252,703	7,227,746	33,155
Sort	common_firefox_1	0%	9,637,490	9,645,399	15,793
Sort	common_firefox_2	0%	10,343,469	10,295,519	23,886
Sort	common_firefox_3	1%	7,460,484	7,500,646	80,453

Function	Pattern	%	Before [ns]	After [ns]	σ [ns]
List	empty_pattern	-2%	6,046,645	5,904,154	104,021
List	existing_file	-9%	915,556	835,606	2,010
List	not_existing_file	-9%	913,972	834,267	3,022
List	existing_file_with_globs	-8%	860,183	794,640	1,165
List	existing_family	0%	211,736	211,330	184
List	not_existing_family	-1%	205,281	203,930	159

Table 8 shows that the benchmarks that call the affected functions the most improved the performance as expected.

The effect of the optimization done in the `FcCompareFamily` function can be seen in the generated assembly code. As shown in Figure 9, the function starts by checking whether the preprocessed value contains the expected hash and by comparing the hashes of the two values together. If the hashes differ, the value `1.0` is returned, meaning that the strings are different. If the hashes are missing or equal, the strings are compared fully.

```
static long
FcCompareFamily (FcValue *v1, FcPrepValue *p1, FcValue *v2,
FcPrepValue *p2)
{
    if (p1->type == FcPrepStrHashIgnoreBlanksAndCase &&
        p2->type == FcPrepStrHashIgnoreBlanksAndCase))
    {
          // If hashes are not matching, return fast
          if (p1->str_hash != p2->str_hash)
              return 1.0;
    }

    /* rely on the guarantee in FcPatternObjectAddWithBinding that
     * families are always FcTypeString. */
    const FcChar8* v1_string = FcValueString(v1);
    const FcChar8* v2_string = FcValueString(v2);

    if (FcToLower(*v1_string) != FcToLower(*v2_string) &&
          *v1_string != ' ' && *v2_string != ' ')
       return 1.0;

    return FcStrCmpIgnoreBlanksAndCase (v1_string, v2_string) != 0;
}
```

FIGURE 9. The FcCompareFamily function prior to optimization.

```
1f5c0:  mov    %rdi,%rax
1f5c3:  cmpl   $0x3,(%rsi)      ; First comparison in first if
1f5c6:  jne    1f5d9            ; First comparison in first if
1f5c8:  cmpl   $0x3,(%rcx)      ; Second comparison in first if
1f5cb:  jne    1f5d9            ; Second comparison in first if
1f5cd:  mov    0x4(%rsi),%esi   ; Second if
1f5d0:  cmp    0x4(%rcx),%esi   ; Second if
1f5d3:  jne    1f669            ; Second if

1f5d9-1f668: ; Omitted many instructions doing the full comparison

1f669:  movsd  0xd70f(%rip),%xmm0  ; Return 1.0
1f671:  retq                       ; Return 1.0
```

FIGURE 10. Assembly of FcCompareFamily function prior optimization.

The corresponding assembly code is in Figure 10. The instructions on `1f5c3`-`1f5cb` correspond to the first `if`. The `p1->type` is tested and if it is not equal to `FcPrepStrHashIgnoreBlanksAndCase` (constant 0x3), the execution jumps to the full comparison, otherwise it continues to the next instruction. Then the same comparison is done for `p2->type`. The instructions on `1f5cb`-`1f5d3` correspond to the second `if`. The `p1->str_hash` and `p2->str_hash` are compared and if they are not equal, the execution jumps all the way down to the end of the function where it returns the value `1.0`. If they were equal, the execution continues down to the full comparison.

In the common case both of the hashes will be present and will not be equal. Therefore, the most common execution path will be going directly from the instruction `1f5c0` to `1f5d3`, then jumping to `1f669` and then returning from the function on the following instruction. It can be seen that the most common execution path is divided in half by rarely executed code. This means that more memory has to be accessed while executing the function, possibly slowing the execution down due to a less effective cache utilization.

Figure 11 shows the function after adding the `likely` macro.

```
static long
FcCompareFamily (FcValue *v1, FcPrepValue *p1, FcValue *v2,
FcPrepValue *p2)
{
    if (likely (p1->type == FcPrepStrHashIgnoreBlanksAndCase &&
                p2->type == FcPrepStrHashIgnoreBlanksAndCase)))
    {
          // If hashes are not matching, return fast
          if (likely (p1->str_hash != p2->str_hash))
              return 1.0;
    }

    // Rest of the function remains the same
}
```

FIGURE 11. The FcCompareFamily function after optimization.

The corresponding assembly is in Figure 12. It is almost the same as before the optimization, but the most common execution path is now in one coherent block. The code of the full comparison does not need to be loaded in the common case.

```
1f5c0:  mov    %rdi,%rax
1f5c3:  cmpl   $0x3,(%rsi)          ; First comparison in first if
1f5c6:  jne    1f5de                ; First comparison in first if
1f5c8:  cmpl   $0x3,(%rcx)          ; Second comparison in first if
1f5cb:  jne    1f5de                ; Second comparison in first if
1f5cd:  mov    0x4(%rsi),%esi       ; Second if
1f5d0:  cmp    0x4(%rcx),%esi       ; Second if
1f5d3:  je     1f5de                ; Second if
1f5d5:  movsd  0xd70f(%rip),%xmm0 ; Return 1.0
1f5dd:  retq                        ; Return 1.0

1f5de-1f664: ; Omitted many instructions doing the full comparison
```

FIGURE 12. Assembly of FcCompareFamily function after optimization.

3.5 Total speedup

TABLE 9. Speed comparison before and after all optimizations.

Function	Pattern	%	Before [ns]	After [ns]	σ [ns]
Match	common_kde	-98%	16,979,822	305,621	1,612
Match	common_gnome	-68%	16,774,682	5,390,364	2,031
Match	common_chromium	-67%	16,674,855	5,466,933	3,678
Match	common_firefox	-65%	15,668,690	5,437,409	4,090
Match	existing_file	-81%	1,527,992	290,082	7
Match	non_existing_file	-81%	1,529,358	291,189	2,009
Match	existing_file_with_globs	-84%	1,824,564	290,071	834
Match	existing_family	-27%	383,132	279,957	1,183
Match	not_existing_family	14%	377,222	430,985	776
Sort	common_kde_1	-49%	18,755,030	9,600,108	477
Sort	common_kde_2	-42%	12,424,092	7,227,746	6,285
Sort	common_firefox_1	-48%	18,721,599	9,645,399	5,156
Sort	common_firefox_2	-48%	19,694,770	10,295,519	33,127
Sort	common_firefox_3	-38%	2,127,628	7,500,646	29,890
List	empty_pattern	-15%	6,925,237	5,904,154	6,341
List	existing_file	-46%	1,557,534	835,606	207
List	not_existing_file	-46%	1,559,173	834,267	72,968
List	existing_file_with_globs	-45%	1,445,537	794,640	297
List	existing_family	-18%	258,622	211,330	461
List	not_existing_family	-18%	248,760	203,930	431

Table 9 shows the total speedups of the benchmarks before and after all optimizations. With the exception of *Match/not_existing_family*, all benchmarks received significant speedups. *Match/not_existing_family* is an artificial benchmark and even with the 14% slowdown, it is still very fast compared to others.

3.6 Speedups in real-world scenarios

A method described in chapter 3.2 Determining most common uses was used to collect the fontconfig queries done by various Linux graphical applications. These queries were replayed and benchmarked with fontconfig before and after the optimizations. Table 10 shows the speedup of the measured real-world actions in absolute numbers.

TABLE 10. Speedups of real-life scenarios

Action	**Queries**	**Speedup [ms]**
Starting Kate (KDE text editor)	5x Match, 1x Sort, 1x List	92.82
Starting Konsole (KDE terminal emulator)	3x Match, 2x Sort, 1x List	67.90
Starting Dolphin (KDE file browser)	2x Match, 1x Sort, 1x List	42.80
Starting Kmail (KDE email client)	6x Match, 9x Sort, 1x List	161.03
Starting KDevelop (KDE IDE)	14x Match, 2x Sort, 1x List	251.31
Starting VLC (media player)	3x Match, 1x Sort, 1x List	59.47
Starting Nautilus (Gnome file browser)	4x Match	45.54
Starting Octave	6x Match, 1x List	69.33
Starting NetBeans (Java IDE)	3x Match	34.15
Starting LibreOffice Writer	32x Match, 1x List	365.32
Opening document in LibreOffice Writter	24x Match	273.22
Opening font menu in LibreOffice Writter	15x Match	170.76
Starting GIMP (image editor)	14x Match, 1x List	160.40
Starting Inkscape (vector editor)	23x Match, 2x List	262.86
Starting Chrome	20x Match, 14x Sort, 1x List	346.67
Opening google.com in Chrome	3x Match, 3x Sort	59.43
Opening facebook.com in Chrome	19x Match, 24x Sort	418.53
Starting Firefox	6x Match, 11x Sort	160.99
Opening google.com in Firefox	3x Sort	25.28

The total startup time of selected applications was measured to get an idea of the significance of these speedups. In contrast with a single function, which has a clearly defined beginning and end, the startup of a graphical application is less clearly defined. The start can be easily defined as the moment when the application's process was started. However, the end needs to be recognized by the user because there is no signal given by the application when it is ready. One measureable moment is when the application registers its main window in the window manager. However, many applications display their window early but take an additional time before they are fully initialized and useable by the user.

To get a proper idea of the timing from the point of view of the user, the measurement was done by recording the computer screen with a camera. The recorded video was used to measure the time from the application command submission to the moment when the application window was fully rendered.

Table 11 shows the measured times together with the improvement. This experiment was done with the original version of fontconfig.

TABLE 11. Startup times of selected applications with original fontconfig

Application	**Startup time [ms]**	**Speedup [ms]**	**%**
Gimp	1,900	160	8%
VLC	400	59	15%
Dolphin	433	43	10%

4 CONCLUSION

After working with fontconfig code, it became apparent that it was not designed with performance in mind. It is possible that the original author did not anticipate the amount of fonts or the complexity of rules it is going to be used on. In addition, the public API exposed too many internal details, which made it difficult to make improvements without breaking the backwards compatibility.

Nevetherless, it was possible to achieve significant speedups just by optimizing the existing code while keeping the same data structures and nearly the same algorithms. Although not all of the optimizations brought speedups noticeable by the end user, they will improve the feeling of responsiveness of the system and the will also save energy.

A full reimplementation could bring further improvements for the cost of breaking the backward compatibility. If such a step is taken, it would be a good idea to clean up the public API and remove the functionality that is not commonly used or does not belong to the scope of a font database. A new reimplementation could be a simple wrapper around some embedded database, such as SQLite.

REFERENCES

1. Packard, K. 2002. Font Configuration and Customization for Open Source Systems. Date of retrieval 03.04.2017 http://www.phoronix.com/scan.php?page=news_item&px=Apple-No-More-PowerVR

2. triskelios@gmail.com. 2014. ChromeOS needs supported method to install new fonts. Date of retrieval 03.04.2017 https://bugs.chromium.org/p/chromium/issues/detail?id=320364#c4

3. Tizen Project. 2012. API Reference. Date of retrieval 03.04.2017 https://developer.tizen.org/development/api-references/native-application?redirect=/dev-guide/2.3.0/org.tizen.native.mobile.apireference/group__OPENSRC__FONTCONFIG__FRAMEWORK.html

4. Packard, K. 2016. Fontconfig. Date of retrieval 03.04.2017 https://www.freedesktop.org/wiki/Software/fontconfig

5. Kroah-Hartman, G. 2004. Modifying a Dynamic Library Without Changing the Source Code. Date of retrieval 03.04.2017 https://www.linuxjournal.com/article/7795

6. Google Inc. Benchmark Support Library. Date of retrieval 03.04.2017 https://github.com/google/benchmark

7. Walls, C. 2017. Dynamic Memory Allocation and Fragmentation in C and C++. Date of retrieval 03.04.2017 https://www.design-reuse.com/articles/25090/dynamic-memory-allocation-fragmentation-c.html

8. Drepper, U. 2007. What Every Programmer Should Know About Memory. Date of retrieval 03.04.2017 https://people.freebsd.org/~lstewart/articles/cpumemory.pdf

9. Kerrisk, M. 2012. How much do __builtin_expect(), likely(), and unlikely() improve performance?. Date of retrieval 03.04.2017 http://blog.man7.org/2012/10/how-much-do-builtinexpect-likely-and.html

```
#include <benchmark/benchmark.h>
#include <fontconfig/fontconfig.h>

void FontMatch(benchmark::State& state, bool apply_substitutions,
const char* pattern_name)
{
    FcPattern *pattern = FcNameParse((FcChar8*) pattern_name);

    if (apply_substitutions) {
        FcConfigSubstitute(nullptr, pattern, FcMatchPattern);
        FcDefaultSubstitute(pattern);
    }

    while (state.KeepRunning()) {
        FcResult result;
        FcPattern *found = FcFontMatch(nullptr, pattern, &result);
        FcPatternDestroy(found);
    }

    FcPatternDestroy(pattern);
}

/* Benchmarks based on real application usage */
BENCHMARK_CAPTURE(FontMatch, common_kde_1, true,
"Noto Sans:pixelsize=13:file=/usr/share/fonts/truetype/NotoSans-
Regular.ttf:index=0");
BENCHMARK_CAPTURE(FontMatch, common_gnome_1, true,
“Tahoma-
9:slant=0:weight=80:width=100:pixelsize=12:verticallayout=False:dpi
=96:lang=c:prgname=gedit");
BENCHMARK_CAPTURE(FontMatch, common_chromium_1, true,
"Noto Sans-
9.99976:slant=0:weight=80:width=100:pixelsize=13.333:verticallayout
=False:dpi=96:lang=en-us:prgname=chromium");
BENCHMARK_CAPTURE(FontMatch, common_firefox_1, true,
":pixelsize=13");

/* Artificial benchmarks */
BENCHMARK_CAPTURE(FontMatch, empty_pattern, false, "");

BENCHMARK_CAPTURE(FontMatch, existing_file, false,
":file=/usr/share/fonts/truetype/NotoSans-Regular.ttf");
BENCHMARK_CAPTURE(FontMatch, not_existing_file, false,
":file=/usr/share/fonts/truetype/DoesNotExist.ttf");
BENCHMARK_CAPTURE(FontMatch, existing_file_with_globs, false,
":file=/usr/share/fonts/*/NotoSans-Regular.ttf");

BENCHMARK_CAPTURE(FontMatch, existing_family, false, "Noto Sans");
BENCHMARK_CAPTURE(FontMatch, not_existing_family, false, "Does Not
Exist");


void FontSort(benchmark::State& state, bool apply_substitutions,
const char* pattern_name)
{
    FcPattern *pattern = FcNameParse((FcChar8*) pattern_name);

    if (apply_substitutions) {
        FcConfigSubstitute(nullptr, pattern, FcMatchPattern);
        FcDefaultSubstitute(pattern);
    }

    while (state.KeepRunning()) {
        FcResult result;
        FcFontSet *set = FcFontSort(nullptr, pattern, FcFalse,
nullptr, &result);
        FcFontSetDestroy(set);
    }

    FcPatternDestroy(pattern);
}


/* Benchmarks based on real application usage */

// Observed at least in kate, dolphin, ark, konsole, kcalc
BENCHMARK_CAPTURE(FontSort, common_kde_1, true,
"Noto Sans:slant=0");
BENCHMARK_CAPTURE(FontSort, common_kde_2, true,
"Liberation Mono,monospace:slant=0");

// Observed in Firefox
BENCHMARK_CAPTURE(FontSort, common_firefox_1, true,
"-moz-default:scalable=True:lang=en-us");
BENCHMARK_CAPTURE(FontSort, common_firefox_2, true,
"sans-serif:scalable=True:lang=en-us");
BENCHMARK_CAPTURE(FontSort, common_firefox_3, true,
"monospace:scalable=True:lang=en-us");

/* Artificial benchmarks */

BENCHMARK_CAPTURE(FontSort, empty_pattern, false, "");


void FontList(benchmark::State& state, bool apply_substitutions,
const char* pattern_name)
{
    FcPattern *pattern = FcNameParse((FcChar8*) pattern_name);

    if (apply_substitutions) {
        FcConfigSubstitute(nullptr, pattern, FcMatchPattern);
        FcDefaultSubstitute(pattern);
    }

    FcObjectSet *object_set = FcObjectSetBuild (FC_FAMILY,
FC_STYLE, FC_FILE, (char *) 0);

    while (state.KeepRunning()) {
        FcFontSet *set = FcFontList(nullptr, pattern, object_set);
        FcFontSetDestroy(set);
    }

    FcPatternDestroy(pattern);
}

/* Benchmarks based on real application usage */
```

```
BENCHMARK_CAPTURE(FontList, empty_pattern, false, "");

/* Artificial benchmarks */
BENCHMARK_CAPTURE(FontList, existing_file, false,
":file=/usr/share/fonts/truetype/NotoSans-Regular.ttf");
BENCHMARK_CAPTURE(FontList, not_existing_file, false,
":file=/usr/share/fonts/truetype/DoesNotExist.ttf");
BENCHMARK_CAPTURE(FontList, existing_file_with_globs, false,
":file=/usr/share/fonts/*/NotoSans-Regular.ttf");

BENCHMARK_CAPTURE(FontList, existing_family, false, "Noto Sans");
BENCHMARK_CAPTURE(FontList, not_existing_family, false, "Does Not
Exist");

int main(int argc, char** argv)
{
    FcInit();

    benchmark::Initialize(&argc, argv);
    benchmark::RunSpecifiedBenchmarks();

    FcFini();
}
```