University of Heidelberg / Heilbronn University
Medical Informatics

Master Thesis

Interactive application independent data processing using synthetic filesystems

---

Clas Rurik
176965
October 30, 2012

First examiner: Prof. Dr. Dirk Heuzeroth
Second examiner: Prof. Dr. Oliver Kalthoff

Declaration of Authorship

I hereby declare that this thesis has been written entirely as the result of my own work without the use of documents or aid other than those stated below.

This work has not been submitted, either in part or whole, for a degree or qualification at any other University

---

Clas Rurik
Heilbronn, October 30, 2012

Abstract

In this thesis a software system is proposed that provides transparent access to dynamically processed data using a synthetic filesystem for the data transfer as well as interaction with the processing pipeline.

Within this context the architecture for such a software solution has been designed and implemented. Using this implementation various profiling measurements have been acquired in order to evaluate the applicability in different data processing scenarios. Usability aspects, considering the interaction with the processing pipeline, have been examined as well.

The implemented software is able to generate the processing result on-the-fly without modification of the original input data. Access to the output data is provided by means of a common filesystem interface without the need of implementing yet another communication protocol.

Within the processing pipeline the data can be accessed and modified independently from the actual input and output encoding. Currently the data can be modified using a C/C++, GLSL or Java front end.

Profiling data has shown that the overhead induced by the filesystem is negligible for most usage patterns and is only critical for realtime processing with a high data throughput e. g. video processing at or above 30 frames per second where typically no file operations are involved.

≺There are three ways of doing something. ≻
≺The easy, the hard, and the third. ≻

Acknowledgments

My humble thanks go to the following people and institutions for their influence on the concept behind and the words in front of this thesis.

- My advisor Prof. Dr. Dirk Heuzeroth for his influence in making this document as easy to read as possible.
- The community for providing such a vast amount of frameworks and libraries to build on.
- Unix, BSD, MINIX, Plan 9 and Linux for the many great concepts and ideas that have been promoted over such a long time.

Contents

List of Figures

List of Tables

List of Listings

List of Measurements

List of URLs

Nomenclature

API	Application Programming Interface
CIFS	Common Internet File System
DICOM	Digital Imaging and Communication in Medicine
DSL	Domain Specific Language
FBO	Framebuffer Object
FUSE	Filesystem in Userspace
googletest	Google C++ Testing Framework
HTTP	Hypertext Transfer Protocol
IDE	Integrated development environment
ITK	Insight Segmentation and Registration Toolkit
JPEG	Joint Photographic Experts Group
NFS	Network File System
OpenCV	Open Source Computer Vision Library
PNG	Portable Network Graphics
RHEL	Red Hat Enterprise Linux
ROI	Region of interest
TIFF	Tagged Image File Format
TSC	Time Stamp Counter
VTK	(The) Visualization Toolkit

1 Introduction

1.1 Motivation

Data processing is an important part of clinical diagnostics and medical research. To begin with, data in this field can appear in different forms and dimensions. Starting with time-based biosignal measurements, up to 3D+t volumetric data sets acquired over a certain time period. In the context of this thesis, data primarily means 2D imaging data, commonly reconstructed from previously acquired CT or MRT data sets.

Processing in this case can happen in forms of pre- or post-processing steps. Pre-processing is an important part of reconstructing an actual image from raw measurement data. Post-processing can then digitally improve such images even further [Bankman, 2009]. Thereby supporting the process of acquiring some form of decision or knowledge from these images. In this context focus is on the latter.

Generally, few or several of these processing steps are combined in a processing pipeline. In order to improve and ease the interaction with such processing pipelines, several frameworks and graphical applications have been developed over time. Usually such frameworks provide modules with existing functionality that can be reused in order to achieve different effects. In general these toolkits also provide some form of custom programming interface which can be used to implemented and integrate new modules. Another aspect of these frameworks is to simplify the handling of different file formats and specifications. Usually some input and output classes exist that allow reading and writing of certain internal data structures.

By utilizing and integrating functionality of the previously described toolkits, graphical applications have been built to improve interaction with the end-user. These applications provide easily usable visual elements in order to interact with processing pipelines in order to achieve a specific objective. A schematic overview of such an architecture and the communication between the different layers can be seen in figure 1.

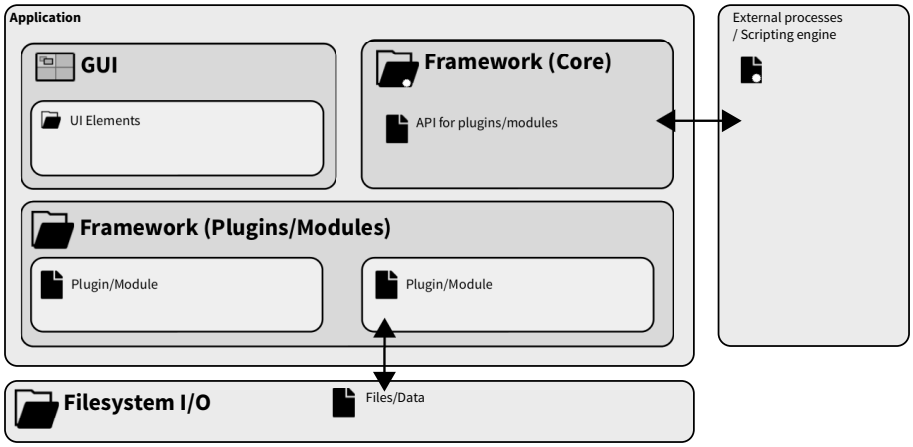


Figure 1: Conventional architecture for data processing pipelines and applications. Modules are bound to a specific framework by implementing a certain programming interface. These modules can be used programmatically by utilizing the given framework. Usually graphical applications interact with the user and provide a visual way of interacting with incorporated frameworks. Data is loaded directly from a persistent filesystem using standardized interfaces.

Quite some effort has gone into simplifying interaction with these processing pipelines and the development of custom processing algorithms.

However, combining lots of different frameworks and different programming languages is still a difficult task. So often the only feasible way of sharing data is by using a mutual storage and data format. Usually this involves the persistent storage of data in some image format within the filesystem. When implementing custom processing code it is usually only feasible to implement a single interface. This limits new modules to a certain framework, thereby causing a considerable amount of fragmentation.

1.2 Proposal

The major issue of the previously described architecture is the self-contained integration of functionality into individual applications. Due to this kind of integration, extensibility and access to the processing pipeline is only possible by implementing the specific interface used by the application. There have been attempts to solve this problem with the utilization of web services and ensemble languages. This way, the functionality is provided by a central service and invoked using a specific, mostly loose, interface. However, web-services also have the general problem that client applications need to implement a specific transfer interface or communication protocol in order to send or receive data. For instance WSDL and HTTP specifications are being used in this field.

There is also the DICOM standard which strives to cover the field of transferring and distributing medical images in a standardized way. However, processing of the image data is not considered in this context and generally only specialized medical applications implement this protocol. There have been approaches to combine the DICOM standard and workflow with web services, for instance by [Anzböck and Dustdar, 2005]. However, these approaches basically suffer from the same drawbacks as presented in the preceding paragraph.

This thesis proposes to move substantial parts of the processing code and algorithms from a framework with custom non-standard interfaces into the filesystem layer itself. This way the need for a custom module interface in order to access specific data is unnecessary. This functionality is comparable to general-purpose service oriented filesystems as described and proposed by [Hensbergen et al., 2009]. The evaluation there comes to the conclusion that RESTful HTTP interfaces, recently gaining attention by web developers, are "essentially a simplified web-instantiation of synthetic file system based service interfaces". Thereby, using the file based approach, similar services can "provide a unified language- and network-neutral interface to its users".

A schematic overview of the proposed architecture can be seen in figure 2. This architecture has been implemented in context of this thesis. The resulting filesystem has been named ***"GrapeFS"***.

The proposed approach has the benefit that any kind of application or automated processing application has direct access to the input, output and intermediate processing results without the need of implementing a specific interface. The application itself is basically reduced to a graphical user interface which interacts with the filesystem to transfer input data, specify processing code or variable parameters. Generally spoken, there is not even the need for a graphical application. For instance the whole computation process could be controlled and managed solely using shell scripts or scripting languages.

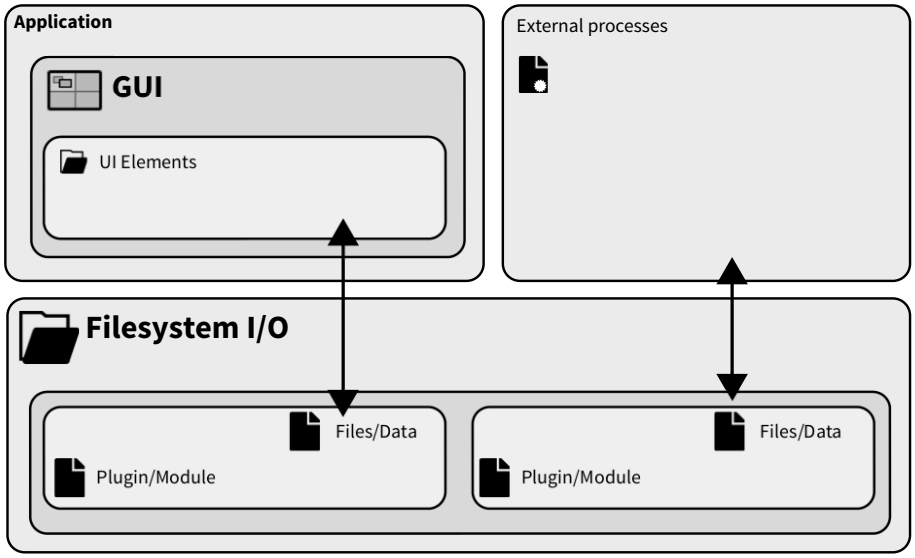


Figure 2: Proposed architecture. Processing algorithms and functionality is implemented in the filesystem layer itself, eliminating the need for a mutual framework API to transfer or access data.

Applications accessing this filesystem interface can be of any kind. An obvious example for this are research tools and applications for the reasoning process in clinical routine. However, even intermediate applications like HTTP servers would be able to access this interface, giving the end-user the ability to transparently access the data using a web browser or other access solutions.

In order to analyze the needed functionality and specifications for the evaluation of a suitable implementation, conventional medical data processing scenarios will be considered. This evaluation includes aspects of usability, as well as, benefits and drawbacks of the intermediate filesystem compared to the conventional approach of directly interacting with the processing functionality. The technical evaluation will include profiling measurements in order to evaluate the feasibility of performing aforementioned usage scenarios.

1.3 Scope and objectives

The proposed architecture uses the standardized filesystem interface and thus can be utilized by any kind of data processing application. This thesis concentrates on medical images and typical processing tasks in this field. Functional and non-functional requirements will be consolidated in detail within the analysis. Some explanations of common use cases will be given in the fundamentals.

The evaluation of the implementation provided in this thesis concentrates on characteristics with special attention on the following questions:

Possibility of implementing different scenarios

Is it possible to implement the required functionality using a filesystem interface? This question covers the technical possibility of implementing a certain functionality. In this context the most promising and efficient way of implementing a certain behavior is being evaluated, including a consolidation of possible disadvantages.

Comparison of benchmarking data between the filesystem and the conventional approach

Certain performance and benchmarking data from the conventional and the filesystem implementation will be collected, compared and evaluated. Possible characteristics for this are the filesystem overhead for the input, output and processing operations.

Possible improvements when using the filesystem abstraction

This question analyzes possible improvements when utilizing certain features that are exclusively available when using a filesystem interface. Such improvements could emerge due to a simplified or transparent interaction with the data when using a filesystem as intermediate interface.

1.4 Chapter overview

Chapter 2 - Fundamentals

Within the fundamentals the basic knowledge and technical foundations will be introduced. Additionally some of the essential processing scenarios and their context will be explained.

Chapter 3 - Analysis

The analysis will consolidate functional and non-functional requirements that an ideal implementation should have in order to comply with the initial proposal.

Chapter 4 - Related Work

This chapter will feature existing frameworks and applications and highlights some of their more important features and characteristics. These characteristics will be compared with the consolidated requirements from the previous chapter.

Chapter 5 - Implementation

The implementation introduces the interaction with the realized filesystem interface and how to achieve different effects using this interface. Additionally, technical details of some components and the role of these components in extending the available functionality will be explained.

Chapter 6 - Evaluation

In respect of the initial requirements, important profiling measurements will be presented and evaluated in this section. These profiling results have been acquired previously for both, the conventional and the filesystem implementation, in situations that are as comparable as possible. Additionally, this chapter will present a recall of evaluating the usability of the realized interface.

Chapter 7 - Conclusion

The conclusion will begin with a summary of the overall thesis and a final look at the acquired results. Ultimately within the outlook some of the most promising enhancements and possible extensions will be highlighted.

Appendix and attachments

The appendix contains some additional attachments like code snippets, build requirements and a comprehensive list of all test cases.

2 Fundamentals

2.1 Image processing

The different types of data and tasks in data processing significantly affect the required functionality of processing applications and services.

Section 2.1.1 starts with an overview of common image processing techniques. The effects of these techniques will be illustrated by addressing the common reasons for certain processing operations. This section will also point out why these reasons are not just avoidable during image acquisition. To simplify the context in a reasonable manner this section assumes grayscale images, based on the measured intensity during image acquisition.

Section 2.1.2 will outline some of the commonly used file formats in this environment. Important characteristics of different formats will be featured and where the use of these formats is reasonable.

The rest of this section will focus on important technical fundamentals regarding the filesystem interface in general. These explanations should be helpful in order to understand technical details of the proposed implementation and how the interaction with this interface is supposed to work.

2.1.1 Common processing tasks

Image visualization

Image visualization covers all operations that are used to improve the visibility and recognizability of certain components within the image. Typically these parts are certain anatomical structures of interest. The visualization can be done using a single image or by combining multiple images from an image series.

Based on the imaging modality or the physiology of certain body parts the interesting aspects may only occupy a certain grayscale range. Depending on the remaining image this intensity range could be relatively small. There are two common operations that can map the grayscale values from the original image to the visualized values. ***Windowing*** and ***histogram*** operations.

Intensity windowing

By applying a windowing operation, certain parts of the image can be masked. This is usually used to remove information that do not contain any additional knowledge. For instance the result of a windowing operation to remove bright bone structures is illustrated in figure 3. Windowing operations are quite effective when the important parts exclusively occupy a certain intensity range. Furthermore, this operation can be combined with histogram operations to maximize usage of the available output contrast [Pisano et al., 2000].

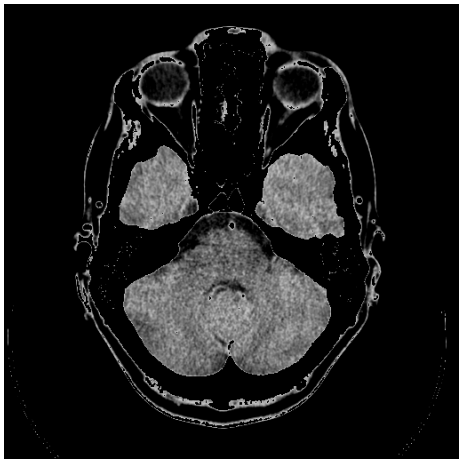


Figure 3: Example of an intensity based windowing operation. The intensity range from the bright bone structures has been cut off, only leaving larger parts of the darker tissue.

Histogram equalization

Histogram equalization is a global operation that strives to maximize usage of the displayable output contrast, thereby improving the visibility of structures with limited intensity variation. This is basically a stretch operation on the grayscale mapping that has two effects on the overall histogram. The first is that the available grayscale range will be utilized completely. Unused ranges, e.g. as they occur when applying an intensity window, will be removed completely. The second effect stretches the grayscale range based on their actual utilization within the image [Galatsanos et al., 2003].

An example of the influence of histogram equalization on the histogram distribution can be seen in figure 4.

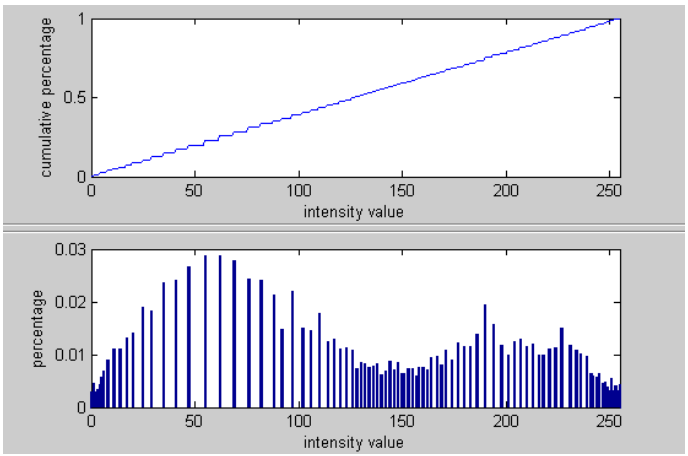


Figure 4: Example of linear histogram equalization. The more utilized intensity ranges take up a wider grayscale range. This is also indicated by the linear increase of the cumulative histogram. All available grayscale values are being utilized.

Volume visualization

The concept of visualizing volumes based on the measured intensity or segmented regions by itself is not a recent invention [W.Wallis and Miller, 1990]. However, in recent years the computation capabilities of hardware devices has increased significantly. This has created an actual possibility of using these techniques interactively in daily routine.

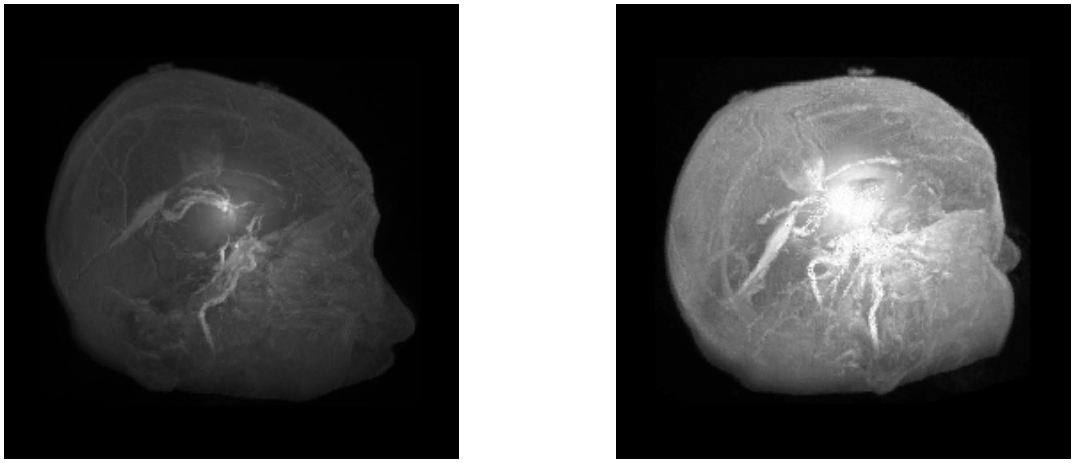


Figure 5: Examples for intensity-based visualizations using a maximum intensity projection. Rays are casted through the image series while taking the gathered intensities into account.

Ray casting and tracing are among the most commonly used methods for volumetric visualization. The increased computing potency generally allows the use of more advance techniques. The presentation of surfaces can be improved even further by considering the effects of reflection and refraction.

A volumetric visualization generated using the so-called "Maximum Intensity Projection" is shown in figure 5.

There are recent developments in increasing the speed of such visualization techniques, for example by using sphere tracing in combination with distance fields [Quilez, 2008].

Digital image enhancements

Noise reduction is a common pre-processing step in digital image enhancement. Generally the purpose is to reduce distortions within the image and provide an improved and clearer visibility of fine structures. This process is supposed to ease the distinction between normal and abnormal tissue or other pathological findings [Bankman, 2009].

There are two reasons why noise reduction is necessary as a standard method in image processing. These reasons cause different types of noise, therefore also requiring different noise reduction techniques.

The first kind of noise is naturally caused by the imaging procedure. This type of noise has characteristics similar to ***Gaussian white noise***. CT imaging measurements implies Poisson

noise, also known as shot noise. Noise introduced by MR imaging can be characterized as Rician noise [Gravel et al., 2004].

Lowering this kind of noise is possible. However, this usually implies longer imaging times. In case of CT imaging this comes down to higher or longer exposure and thereby higher radiation damage.

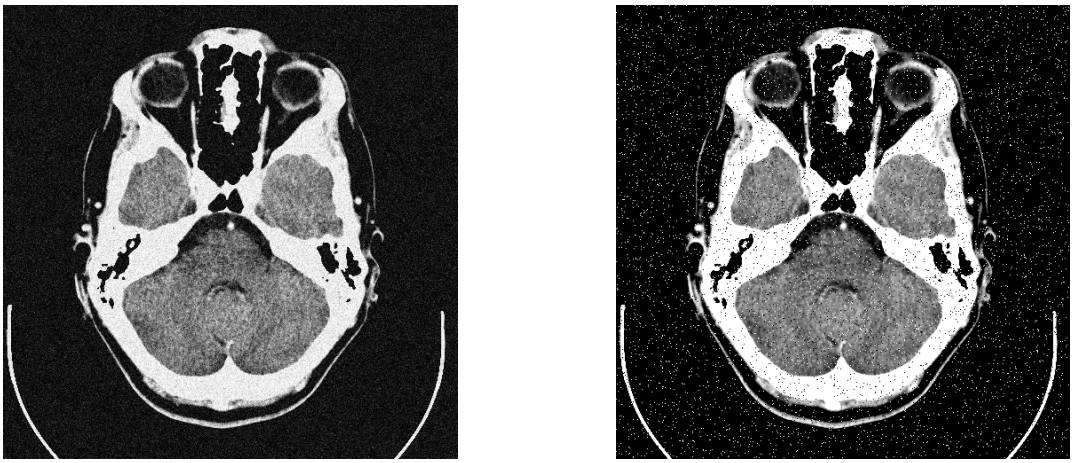


Figure 6: Examples for different types of noise. Left: Gaussian-like noise as commonly introduced by the imaging process. Right: Salt and pepper noise as caused by digital transfer errors or data loss.

The second kind of noise is characterized as ***Salt and pepper*** noise. The most prominent characteristics of this noise are very bright (white) or very dark (black) pixels. This type of noise is commonly caused by digital errors during image transfer, data loss during storage or faulty detector elements.

Examples for both types of noise can be seen in figure 6.

Methods

Generally digital image enhancement can be done in spatial or spatial frequency domain.

Processing in spatial domain directly interacts with the image pixels as seen in the normal image visualization. Modifications within this domain have a local effect within the image.

Processing in frequency domain interacts with the magnitude and phase components of each frequency within the image. Modifications in frequency domain have global effects within the actual image. Transformation into frequency domain can be done using the Fourier transformation and inverted the same way.

A simple way of handling Gaussian noise in spatial domain is by using a simple mean filter, results of this operation are far from perfect though. On the other side salt and pepper noise can be treated very effectively using order-statistics filtering like a media filter.

See figure 7 for two examples of different effects that can be achieved using these filter operations.

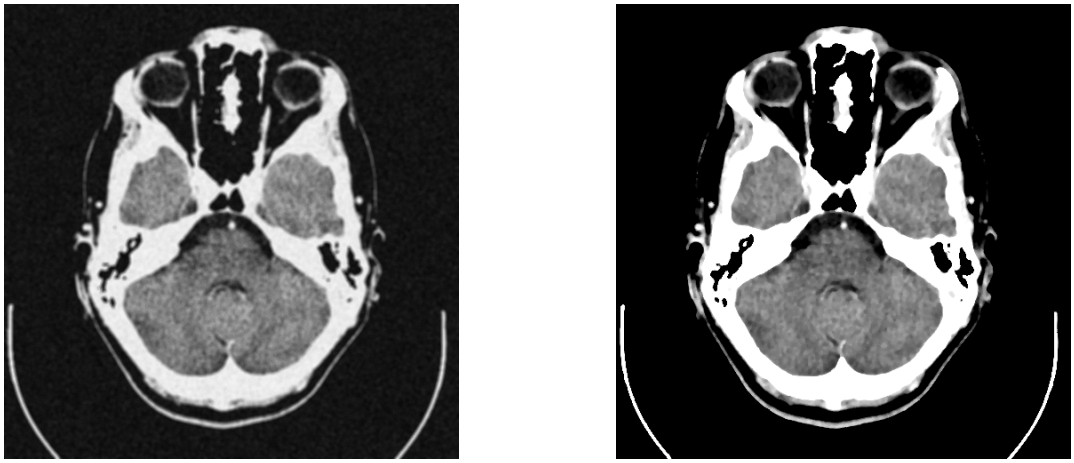


Figure 7: Examples for common noise reduction methods. Left: Reduction of Gaussian noise using a simple mean filter. The noise is reduced but the image becomes blurry. Right: Reduction of salt and pepper noise using a median filter. This noise reduction is very efficient and yields very good results.

Some types of noise are clearly visible in spatial frequency domain. Figure 8 illustrates the potential effect of high frequency noise in frequency domain. There is a possibility that these types of noise can be reduced in frequency domain without affecting the important image information excessively. As a standard method low-pass and high-pass filters can be performed quite efficiently in frequency domain.

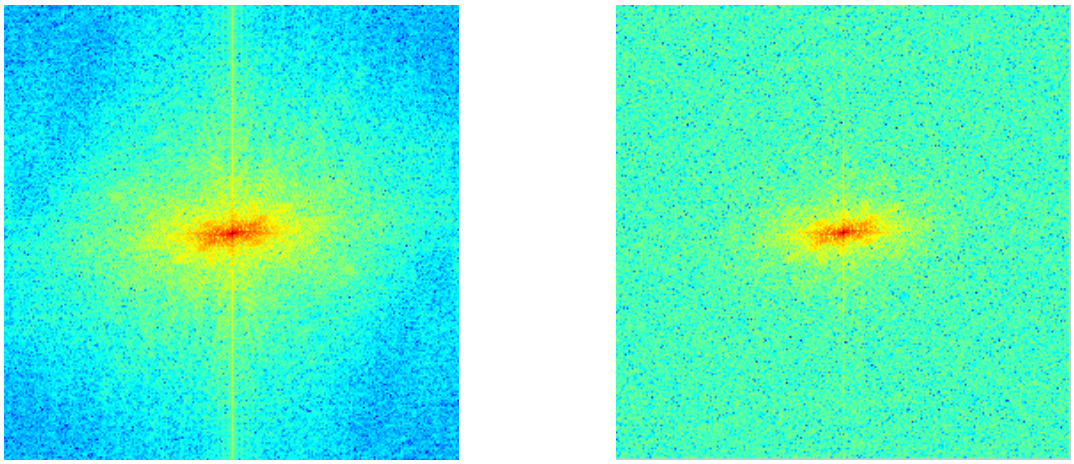


Figure 8: Example for the visibility of noise in spatial frequency domain. The ratio of high frequency parts is increased clearly.

Image segmentation

Segmentation is the process of separating relevant from irrelevant structures. Contrary to human capabilities this process can be quite challenging for algorithmic approaches, especially when functioning in an automated environment [Bankman, 2009]. The ultimate result is the contour or volume of a certain region (ROI). ROIs are commonly used for further analysis in the reasoning process, registration of multiple images or visualization.

Methods

The easiest way of segmenting regions is based on the absolute pixel value. These values can be analyzed globally or locally by using the similarity of multiple pixel values.

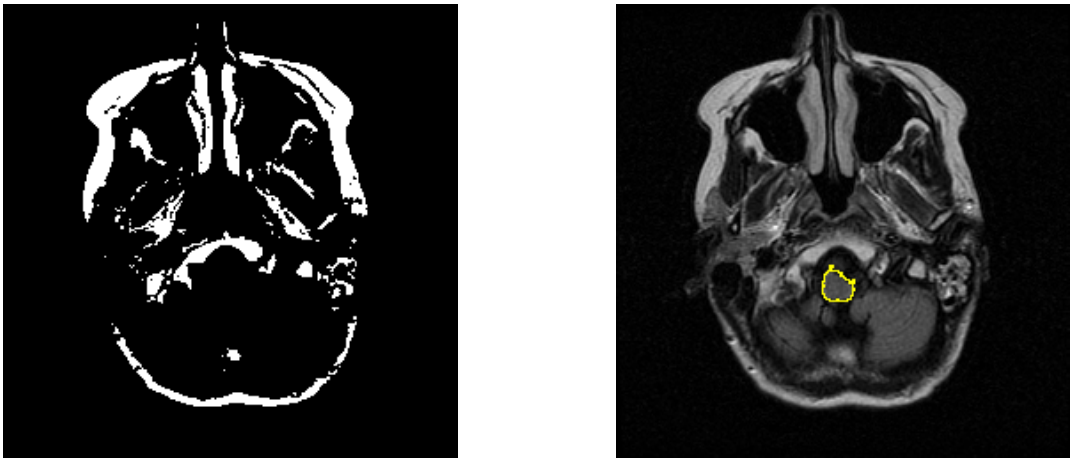


Figure 9: Examples of image segmentation. Left: Intensity-based binarization. Right: Region growing based on intensity and a single seed point. [1]

A potential way to globally segment structures with a very limited intensity range is by applying a binarization filter using a certain threshold. All pixels within this threshold range will be marked as foreground (while) and all remaining pixels will be assumed as background (black).

Local segmentation is possible using a region growing method. Regions with a higher variance of pixel values can be segmented by including pixels with a value similar to their neighbors. This method usually requires a seed point that is used as the initial start for the growing process.

Examples of these two basic methods can be seen in figure 9.

Image registration

Registration is the process of aligning identical structures and image regions of two or more images. There are two common reasons why this process in unavoidable [Bankman, 2009].

- Images have been acquired using the same modality but at different times.

[1] URL http://www.barre.nom.fr/medical/samples/ (Image source)

- Images have been acquired using different modalities and at different times.

Using images from different modalities poses a special challenge. Due to the differences in image acquisition even identical structures may result in completely different intensity distributions.

Methods

Registration of two images is generally done by optimizing a certain similarity measure. When comparing images from the same modality this measure can be simply based on mutual intensity values. When comparing images from different modalities this is not possible. In this case the measure for image similarity can be based on certain anatomical features that have been segmented and tagged identically in both images.

There are different type of registration. Rigid registration for example only considers the offset and rotation against each other. However, it is not unusual that image distortions need to be considered. In this case the registration need to find a more complex transformation that also allows other kinds of deformations.

2.1.2 Image files and formats

Imaging files and formats

Image files, formats and containers are always involved when transferring, accessing and storing medical data. Demand and complexity of the latter has increased particularly due to the requirement of storing medical images over a long time. Also collaborative efforts have raised the need to conveniently exchange and access medical data [Bankman, 2009].

Compressed file formats usually provide two different categories of image compression techniques. Lossless (reversible) and lossy (irreversible).

Lossless compression methods provide a lower compression ratio but ensure that every detail and original information within the image will be preserved. After compression and decompression the image will be fully identically.

Lossy compression can provide significant higher compression ratios. Unfortunately this implies the loss of smaller details within the image which can be a significant drawback, especially for medical reasoning. An exaggerated example of a possible loss of detail is shown in figure 10.

Most file formats are restricted to a specific set of compression techniques. Examples for this are the lossy JPEG format and the lossless PNG format. However, some container formats like TIFF or DICOM are able to utilize different compression methods. Therefore these containers can be used more flexibly by using lossy/lossless compression depending on the current needs and requirements [Graham et al., 2005].

Image compression

In medical diagnostics image compression can be problematic as fine structures and aberrations can be degraded by compression algorithms easily. Lossy compression methods are es-

pecially suitable in situations where the transfer rate is limited or the data size has a noticeable impact. This is of importance in remote and network scenarios. Therefore, possible uses for lossy compression methods can be in teleradiology [Erickson, 2002]. Image archiving in situations with limited storage space could be another possible scenario.

Figure 10: Dramatized example of irreversible image compression. Fine structures are completely unrecognizable due to the block artifacts. (JPEG compression with quality <10)

General-purpose image formats

General-purpose image formats are usually the only option when data is supposed to be accessed by non-medical applications. These applications typically do not support the DICOM format at all. General-purpose formats can also be considered as a valid option when exchanging images without a specific need of preserving patient or modality information.

Table 1 lists some of the more important general-purpose formats and characteristics according to [Wiggins et al., 2001].

Parameter	GIF89a	JPEG	TIFF	PNG
Maximum color depth	8-bit (256)	24-bit (millions)	48-bit (trillions)	48-bit (trillions)
Compression technique	Lossless	Lossy	Lossless (lossy)	Lossless
Interlacing	Yes (1D)	No	No	Yes (2D)
Gamma correction	No	No	Yes	Yes
Patent issues	Yes	No (yes)	Yes	No

Table 1: Characteristics of some general purpose image formats.

DICOM

The DICOM standard is more than a container format for storing images and acquisition information. Additionally DICOM specifies a well defined protocol for the exchange, storage and

printing of images between different devices. This standard is available online as a digital document [2] [Association, 2011].

```
0008,0008  Image Type: DERIVED\PRIMARY\RECON TOMO\EMISSION
0008,0022  Acquisition Date: 1993.11.24
0008,0023  Image Date: 1995.05.30
0008,0032  Acquisition Time: 09:53:00
0008,0033  Image Time: 11:14:39
0008,0060  Modality: NM
0008,0070  Manufacturer: ACME Products
0010,0010  Patient's Name: Anonymized
0018,0015  Body Part Examined: HEART
0018,0050  Slice Thickness: 6.231669
0028,0008  Number of Frames: 13
0028,0009  Frame Increment Pointer: T
0028,0010  Rows: 64
0028,0011  Columns: 64
```

Listing 1: Partial example of a DICOM header

However, in this context focus will be on DICOM as a file format for the storage and exchange of images. Some of the information about the patient and the image modality stored in a DICOM header can be extracted from listing 1.

2.2 Filesystems and the VFS

2.2.1 The Virtual Filesystem (VFS) interface

The overall idea of the proposed architecture is to utilize an interface for the processing service that is implemented and supported by every existing application. Fortunately such an interface exists and is widely used: The filesystem. Any application is able to access and modify data in the filesystem by using a standardized and mutually known interface. To be precise, applications use a specific API, for instance implemented by glibc in Linux. This API calls the VFS interface provided by the kernel. An illustration of this interaction is shown in figure 11. This VFS interface has certain semantics that can be assumed as valid for every filesystem implementation utilizing this interface [Galloway et al., 2009].

[2] URL ftp://medical.nema.org/medical/dicom/2011

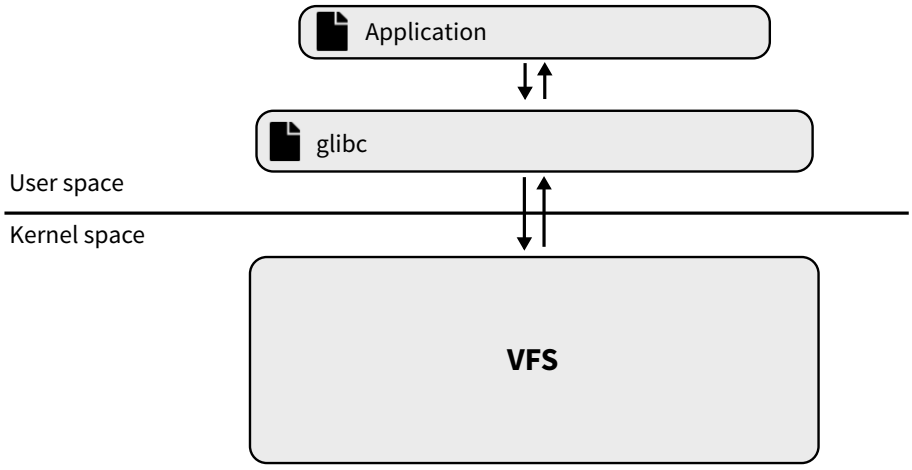


Figure 11: Interaction between applications in user-space and the VFS layer in kernel-space.

Most filesystems implementing this interface provide a passive look at data that is mostly static. But this is no necessity at all. So-called "pseudo" filesystems apply an active data model that provides a dynamic view on anything expressible as bits and bytes [Minnich, 2005].

2.2.2 Synthetic/Pseudo filesystems

The idea of ***pseudo*** or ***synthetic*** filesystems is to utilize the mutual filesystem interface to provide a view on an active data model. This basically means that not all data within the filesystem is user-provided or static. Instead the files within a synthetic filesystem provide an abstract view on arbitrary data structures or interfaces. This way data can even change over time without any user interaction at all.

Such functionality is not limited to user-space filesystems. For example this approach is used by the ***sys*** and ***proc*** filesystems in Linux [Mochel, 2005]. These filesystems provide an interface to access and modify kernel variables and information of running processes.

There are also several examples of such filesystems in user-space. For instance a filesystem that provides transparent access to a MySQL database [Ordulu, 2010] or XCPU2 which provides functionality to manage virtualized environments [Ionkov and Hensbergen,].

2.2.3 User-space filesystems

The usual approach of providing a VFS implementation is by directly implementing the interface in kernel-space. However, it is also possible to provide an implementation in user-space. Following this approach has advantages as well as disadvantages. The most prominent of these will be presented in the rest of this section.

Advantages

Features available in user-space

Use of existing user-space libraries and frameworks is a lot easier when done directly in user-space. Thereby, user-space filesystems can provide an elegant way of providing access to a variety of different processing functionality. Also access to GPU hardware is usually easier from user-space, as interfaces like OpenGL or CUDA are accessed through a user-space library.

Stability

The actual filesystem implementation is executed in user-space and only the filesystem part of the FUSE driver is hosted in kernel-space. Therefore, any faults or errors within the implementation have no severe effect on the kernel execution itself.

In contrast to this, memory faults and other undefined behavior in a kernel-space implementation can have serious consequences like system crashes or any possible kind of data loss/corruption.

Licensing

The Linux kernel itself is distributed using the GPLv2 license. This also applies to any module within the kernel tree, basically imposing extensive licensing constraints. When using a user-space filesystem, only the FUSE driver needs to comply with the GPL license. The user-space implementation itself can be distributed using arbitrary licenses.

Disadvantages

There is some general criticism about user-space filesystems mainly in the area of performance when compared to native in-kernel filesystems. [Torvalds, 2011] This issue will be considered when analyzing the individual throughput needs and evaluating profiling information from the filesystem implementation.

2.2.4 FUSE and 9P

A very common way of implementing a user-space filesystem this is by using the FUSE library. FUSE provides a proxy-like interface between the kernel and the filesystem implementation. This interface can be implemented in user-space and is passed through the mutual VFS in kernel-space. ***GrapeFS*** uses FUSE to provide its filesystem interface, the reasons for this will be explained later in the implementation (section 5.1).

Figure 12 illustrates an overview of the interaction between the VFS call in user-space and the ***GrapeFS*** implementation on the other side.

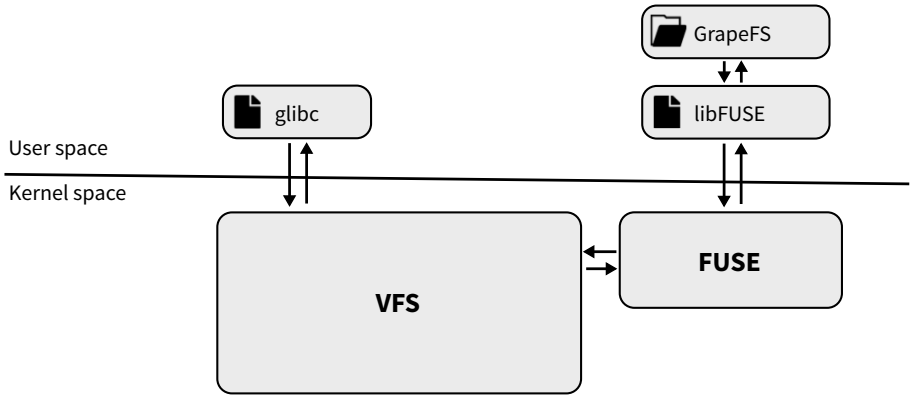


Figure 12: Interaction architecture between the kernel VFS and the FUSE library.

The FUSE concept has been adapted by other operating systems as well. ReFUSE is a port for BSD based [Kantee and Crooks, 2007] systems. Dokan provides a similar concept for Windows based systems [3]. A recent example for the Dokan driver is provided by the glassfs project [CodiceSoftware, 2012]. FUSE implementations are also available for Mac OS X. The most recent by the OSXFUSE project [4].

A frequently updated list of FUSE examples is maintained on the corresponding Wikipedia page [5]. A brief overview of common filesystems using the FUSE library is shown in table 2.

Name	Description
SSHFS [Hoskins, 2006]	Transparently access remote data using a SSH tunnel.
ZFS-FUSE	FUSE implementation for ZFS pools.
TrueCrypt [6]	Transparent encryption.
NTFS-3g	NTFS implementation in user-space

Table 2: Userspace filesystems based on FUSE.

There are other ways of providing a filesystem interface in user-space as well. For example by implementing the distributed resource protocol used in the Plan 9 operating system [Minnich, 2005]. These services can be mounted into the filesystem structure the same way as FUSE. A modern implementation of this protocol is available in Linux as well as the 9P2000 revision [Van et al., 2005]. For instance, using the 9P2000.L protocol, a filesystem passthrough has been developed with support for paravirtualization. This project, called VirtFS, has been built into the QEMU project [Jujjuri et al., 2000].

2.3 Relevant architectural patterns

Most important design patterns described here have their origin in the well known book "Design Patterns: Elements of Reusable Object-Oriented Software" or are similar to architectural patterns introduced there [Gamma et al., 1994]. Section 5.1.3 contains a description of the specific implementation in ***GrapeFS***.

[3] URL http://dokan-dev.net/en/
[4] URL http://osxfuse.github.com/
[5] URL http://en.wikipedia.org/wiki/Filesystem_in_Userspace#Examples
[6] Unclear if still using FUSE for XTS encryption mode. But used for older LRW and CBC modes.

Pipes and filters

The approach in many different frameworks like VTK and ITK is based on the pipeline pattern. The overall processing pipeline consists of several individual nodes. Each node can receive input data from other nodes while providing output data to connected nodes.

The first node can receive input data from sources outside the individual pipeline. The last output node provides the processing result to the caller. Nodes can have multiple input and output connections. Cycles are usually not allowed. Figure 13 illustrates the common utilization of this pattern and the communication between the different filters.

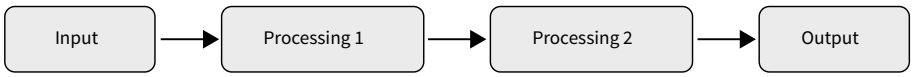


Figure 13: Pipes and filters design pattern.

This pattern is similar to the "Chain of Responsibility" pattern from "Design Patterns" [Gamma et al., 1994]. However, within the pipeline pattern every node is expected to handle the data. Data is not supposed to be passed through the pipeline and only processed by a single responsible node.

Composite pattern

The composite pattern describes how the structure of a child/parent pattern is seen by interacting external objects. Using this pattern the distinction between leaf and intermediate nodes within a tree-like structure is usually avoided.

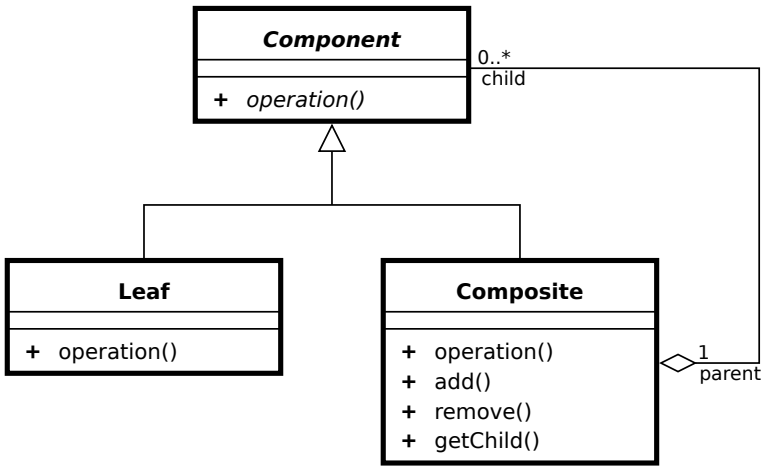


Figure 14: Composite design pattern. [7]

In order to simplify the interfaces within the tree structure, all classes are based on a single base class and therefore share a common interface. An example of such a class hierarchy in UML notation is shown in figure 14.

[7] URL http://en.wikipedia.org/wiki/File:Composite_UML_class_diagram_(fixed).svg

Decorator pattern

The decorator pattern is used to extend an existing class (structure) with additional functionality from outside the actual class structure.

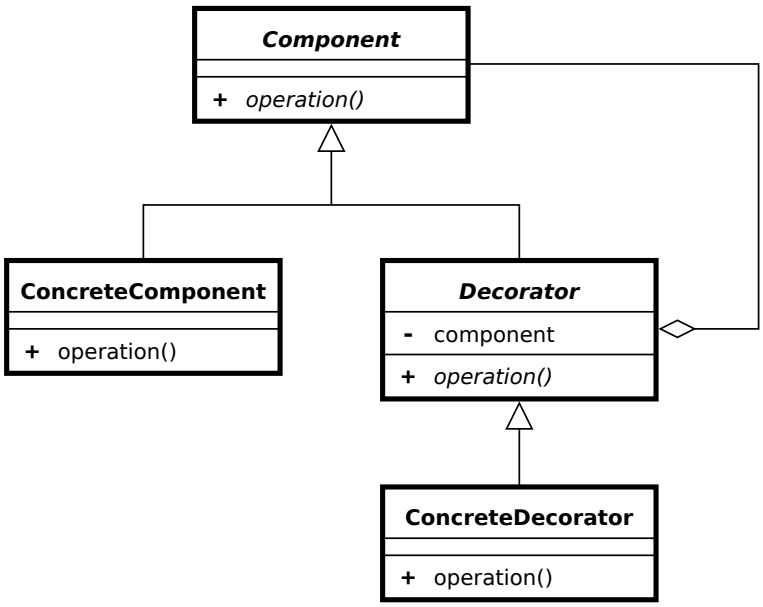


Figure 15: Decorator design pattern. [8]

In contrast to inheritance from existing classes, the decorator pattern allows to extend classes without interfering with the existing inheritance structure. This allows for additional flexibility when extending existing structures with additional, possibly optional, functionality. This also provides a way of using multiple decorators for different functionality without the need of solving conflicts between multiple decorators.

An example of the interaction within such a class hierarchy in UML notation can be seen in figure 15.

Double dispatch

Double dispatch is a special case of multiple dispatch that is used in polymorphic programming situations to dynamically map method calls to the correct argument type. This pattern is used in situations where the specific object type can not be deduced directly from the argument type and type lookup, like $\ll typeof \gg$, is to be avoided.

[8] URL http://en.wikipedia.org/wiki/File:Decorator_UML_class_diagram.svg

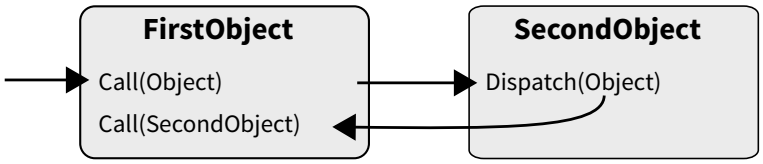


Figure 16: Double dispatch architecture overview.

Due to the unknown type, the first call dispatches to a generic method using *Object* as the argument type. This method calls the actual method on the target object. As the object itself knows the correct type the second call is dispatched using the correct argument type at runtime. An example of this call structure is illustrated in figure 16

3 Analysis

This chapter elaborates on the needed functional and non-functional requirements. These requirements will be evaluated with respect to typical usage scenarios and requirements. For instance, from a technical perspective this covers the need for different file formats and certain capabilities of data processing.

Afterwards, based on this functionality, different evaluation and success criteria will be specified. This includes profiling measurements which will be collected from the conventional and the filesystem implementation. These measurements will later be used for a comparison based on the success criteria defined in this chapter.

3.1 Functional requirements

Format transparency

The intermediate data processing is required to be independent of the actual input and output format of the data.

This imposes some restrictions on the internal data handling:

Whenever the user transfers data into the processing pipeline, the format of this data needs to be handled transparently. The encoded data needs to be translated into a common and processable data representation.

Within the processing pipeline the data is only accessed and processed using this common representation. Also, the preceding and subsequent pipeline elements must not have any influence on this behavior.

The aforementioned transparency also applies to the output data when being accessed by an external process. The output format needs to be definable independently from the processing operations or the initial input format.

Listing 2 lists some of the important file formats from [Wiggins et al., 2001].

- DICOM
- PNG
- JPEG
- TIFF

Listing 2: List of important file and container formats.

Access transparency

Access to intermediate processing results needs to be provided at any stage within the processing pipeline, not only to the final result. Of course, this applies to the data transfer between different processing nodes, as well as, access from external processes.

Processing

From a functional perspective, the processing pipeline must be at least capable of performing common image processing tasks. These tasks have been listed and described in the fundamentals section. The data processing should be implementable using conventional programming languages and methods. Reuse of functionality from existing frameworks is beneficial and provides a great way of extending the base functionality without excessive effort.

3.2 Non-functional requirements

Extensibility

The implementation needs to be as extensible as possible. This includes support for different file formats, programming languages and frameworks.

Interactivity

The processing operations and settings must be thoroughly controllable using the standard (filesystem) interface.

Either the input data format needs to be detected automatically, or the user must be able to specify the type of input data. The format of the output data needs to be specifiable as well.

The processing pipeline must not be in any way static or pre-defined. Definition of processing nodes and the algorithms performed by these nodes should be specifiable at runtime. If applicable, variable parameters should be accessible directly using the filesystem. Likewise this also applies to the connection and data transfer between different processing nodes.

3.3 Evaluation and success criteria

3.3.1 Functional requirements

Functional correctness of the implementation will be validated using a comprehensive set of automated test cases. These test cases should cover all mandatory requirements from the following paragraphs.

The following scenarios will be highlighted as minimal requirements:

Req. #1: Image data input and data processing

- Import of a single image file in a format as described in listing 2, page 23.
- Grayscale image data modification (Exemplary using noise reduction).
- Export of the processed data to different formats, independent of the initial input format.

Req. #2: Image combination

- Import of multiple images.
- Combining the input images in a processing operation (Exemplary by computing the correlation between two images).
- Exporting the resulting image to different image formats.

Req. #3: Visualization/Rendering

There is a definite need of supporting hardware accelerated visualization techniques. Interactive speed for the visualization of 3-dimensional imaging data has been achieved for some time using accelerated surface based rendering. Also especially when visualizing volumetric data it is necessary to utilize modern hardware. For instance, by utilizing GPU hardware to fully accelerate ray casting, as done by [Heng and Gu, 2005], it is possible to achieve interactive visualization of volumetric data. However, all of these methods require some kind of hardware acceleration. Techniques to achieve this may range from the old deprecated [9] fixed function OpenGL rendering to more recent techniques like GLSL based OpenGL rendering or CUDA/OpenCL acceleration.

- Import of a volumetric image series dataset.
- Hardware accelerated visualization with frame latencies applicable for interactive use.

Req. #4: Transparent access

Access to intermediate processing results at any stage within the processing pipeline must be possible using existing applications without specific adaptions or extensions.

3.3.2 Profiling comparison

Profiling and performance measurements will be acquired from the filesystem as well as the conventional implementation. The methodology of this process will be described later and should be as accurate as possible.

The interactivity and interaction with existing applications is tested and evaluated with manual hands-on tests.

The following situations will be focused when evaluating and comparing the runtime behavior and measurements:

Perf. #1: Initial input and output times

The times needed to perform the following actions are considered:

- Import of image data
- Data processing of a single image
- Export of an individual image in some file format

[9] URL http://www.opengl.org/wiki/Legacy_OpenGL

Perf. #2: Repeated computations using parametric arguments

- Dynamic modification of certain processing parameters
- Repeated data processing using data that is already imported into the processing pipeline

Perf. #3: Visualization throughput

Evaluation of the overhead that is induced by the following steps:

- Transfer of a rasterized visualization into system memory.
- Output of the data as image format.

4 Related Work

4.1 Frameworks and Toolkits

There are several frameworks that provide existing functionality, even exceeding the requirements as presented in chapter 3. Most of these frameworks provide interfaces to many different programming languages like C/C++ and Java. Even interpreted languages for scripting purposes, like Python, are supported by most of them.

The more important parts however, like transparent and interactive access, are usually not intended by design. Usually only a programmable interface is provided. Therefore, integration of different frameworks, as well as existing applications, needs to be done manually. In case of closed source applications this can even be difficult or not possible at all.

Nevertheless, current and future functionality provided by these frameworks can be reused in ***GrapeFS***.

4.1.1 VTK/ITK

The ***Visualization Toolkit*** (VTK) [10] is, as the name suggests, mainly focused on the visualization of data [Schroeder et al., 1996]. The VTK has support for compiled languages, like C++. Bindings for Java as well as interpreted languages for scripting exist. Using this scripting support, VTK is well suitable for rapid development [Caban et al., 2007].

Especially for the visualization of big datasets the architecture provides a streaming pipeline architecture [Kitware, 2012b].

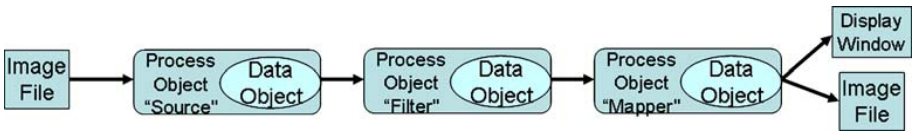


Figure 17: Overview of the data pipeline as used in VTK and ITK [Caban et al., 2007].

The ***Insight Segmentation and Registration Toolkit*** (ITK) [11] is an essentially image based toolkit [Ibáñez et al., 2005]. The ITK heavily relies on C++ templates but also provides support for Java bindings and Python scripting as well [Caban et al., 2007]. ITK algorithms are concentrated on segmentation of structures and image registration with focus on medical use [Kitware, 2012a]. Visualization is generally not accounted by the ITK.

The interaction of both toolkits is based upon a pipeline architecture. An example of how the interaction within such an architecture may look like is pictured in figure 17

VTK and ITK, alone or in combined form, are often used as integrated parts in rather monolithic applications like ParaView.

[10] URL http://www.vtk.org
[11] URL http://www.itk.org

✓ Req. #1, #2	Image processing functionality provided by ITK.
✓ Req. #3	Accelerated visualization provided by VTK.
✘ Req. #4	Only programmable access to data and pipeline using VTK or ITK API.
✓ Perf. #1, #2, #3	VTK and ITK designed for use in interactive scenarios.

4.1.2 OpenCV

The ***Open Source Computer Vision Library*** (OpenCV) is a C/C++ based framework with an extensive variety of algorithms from many different areas of expertise. Language bindings to Java and Python scripting support are provided.

Provided algorithms cover the fields of image processing, image transformation, pattern matching and object detection, contour finding, segmentation, (motion) tracking and camera interaction. Even algorithms for machine learning exist [Bradski and Kaehler, 2008].

✓ Req. #1, #2	All important image processing functionality provided by OpenCV.
✓ Req. #3	OpenCV seems to lack support for OpenGL accelerated visualization. However, combination with VTK is possible as well.
✘ Req. #4	Only programmable access to data and processing functionality using OpenCV API.
✓ Perf. #1, #2, #3	OpenCV designed for use in realtime scenarios.

4.2 Interfaces

Despite the aforementioned programmable interfaces and the proposed filesystem interface, there are other interfaces as well. Some of the more important of these interfaces will be presented in the following paragraphs.

4.2.1 Applications

Graphical user interfaces are the most common instrument used by desktop applications in order to provide control over existing functionality. These software solutions are usually self-contained. Extensibility with custom modules is usually only intended using a specific framework or scripting interface. With respect to the requirements of standardized access, graphical applications usually inherit the same problems as previously described for the frameworks. An overview of how the interface of such applications may look like is presented in figure 18.

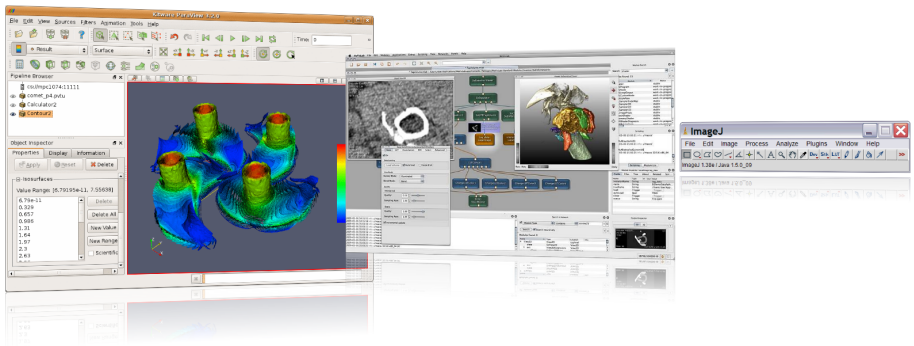


Figure 18: Overview of interactive applications based on different frameworks. From left to right: ParaView [12], MeVisLab [13], ImageJ [14]

ParaView [15] is an open-source application developed by Kitware Inc., Sandia National Laboratory and the Los Alamos National Laboratory. Its intended use is in the area of visualization and pre-processing of large data sets [Laboratory et al., 2012]. ParaView has the capability of utilizing a remote rendering interface using a client/server architecture [Par, 2011]. As the backing framework ParaView is mainly based upon the VTK while providing a graphical user interface using the cross-platform Qt framework.

MeVisLab [16] is a commercial closed-source application developed by Frauenhofer MEVIS and MeVis Medical Solutions AG. MeVisLab provides visualization capabilities using the OpenInventor 3D toolkit [17]. However, it also provides extensive image processing functionality, called ML. VTK and ITK integration is available as well [Koenih et al., 2006].

MeVisLab has an extensible plugin architecture that supports the development of custom plugins in C++ [Rexilius et al., 2006]. Writing scripts and macros in Python is supported as well [Heckel et al., 2009].

The graphical interface provides a direct and interactive way of initializing, connecting and configuring plugins.

ImageJ [18] is a Java based application and framework that can be used for the visualization and processing of imaging data. By default ImageJ has capabilities for different general-purpose image file formats. DICOM containers and image sequences are supported as well. ImageJ also provides basic tools for the manipulation of images. It also integrates algorithms for the segmentation and registration of structures [Abràmoff et al., 2004].

ImageJ itself can be extended with macros and plugins in the Java programming language [Ferreira and Rasband, 2012], therefore it is often used in teaching and prototyping of algorithms. It is also possible to embed and use ImageJ within other Java applications. Up to now it is

[12] URL http://ait.web.psi.ch/services/visualization/paraview.html
[13] URL http://www.mevislab.de/typo3temp/pics/add6b9b646.jpg
[14] URL http://rsbweb.nih.gov/ij/docs/install/images/imagej-window.gif
[15] URL http://www.paraview.org
[16] URL http://www.mevislab.de
[17] URL http://oss.sgi.com/projects/inventor/
[18] URL http://rsb.info.nih.gov/ij/

however not possible to utilize ImageJ as a pure processing toolkit without the graphical application layer. This is likely to improve with the development of a new upcoming major ImageJ revision [19]. Language support however seems to be restricted to Java furthermore.

✓ Req. #1, #2, #3	All required functionality is provided by common applications.
✘ Req. #4	Interaction limited to the user interface or programmatically by extensions using the specific API.
✓ Perf. #1, #2, #3	Applications commonly designed for use in interactive scenarios.

4.2.2 (Web-)Services

Web-Services are another approach on providing functionality to the user by utilizing proliferated network and web-based technologies [20].

The functionality and interactivity of such services is comparable to the proposed approach. However, the way of accessing data and functionality is very different by utilizing network technologies like HTTP and WSDL interfaces [Anzböck and Dustdar, 2005]. In comparison to the frameworks from section 4.1, web-service interfaces are usually more loose, however applications would still have to implement the capabilities of using the previously described network technologies.

✓ Req. #1, #2, #3	Processing functionality depends on the backend behind the web-service. The interface by itself does not limit this functionality.
✘ Req. #4	Access easier than pure application interfaces but not sufficient to comply with the requirements.
✘ Perf. #1, #2, #3	The reaction time induced by network latency and service requests is likely to exceed reasonable limits.

4.2.3 Userspace filesystems

There are few filesystem interfaces that really perform some kind of data processing by providing a virtual directory structure. The most related approach is likewise implemented by filesystems like YacuFS [21] or MP3FS [22]. These filesystems provide a transparent layer on top of another persistent filesystems by transparently converting the formats of the available files. However, this transparently layer is mostly static and configured at filesystem startup. Besides the format conversion no other data processing is done or considered by design.

[19] URL http://developer.imagej.net
[20] Examples: HTML5, CSS3 and JavaScript
[21] URL https://www.uitwisselplatform.nl/projects/yacufs (Not accessible anymore)
[22] Transparent access of FLAC files as MP3

✘ Req. #1, #2, #3	Only very specific data is supported. No processing or required extensibility is considered by design beneath file conversion.
✓ Req. #4	Access possible through the filesystem by any application.
☐ Perf. #1, #2, #3	Evaluation of the performance is not possible due to the missing functionality.

4.2.4 HDFS/MapReduce

An interesting approach is provided by the Apache Hadoop [23] project or more specifically the MapReduce architecture as promoted by Google [Dean and Ghemawat, 2004]. The specific goals of processing very large datasets in a distributed computing architecture however are different from the specified requirements. It is possible to access the stored HDFS data using a FUSE mount [Had, 2012] though. Such a distributed architecture could be an interesting option for future development.

✓ Req. #1, #2, #3	The necessary data processing functionality, similar to the frameworks (cf. 4.1), can be implemented using the MapReduce paradigm.
☐ Req. #4	Access to HDFS data is possible using FUSE. However, interaction with the data processing seems only possible by deploying Java code using the Hadoop/MapReduce API.
✘ Perf. #1, #2, #3	The distributed data storage/processing is not specifically designed for realtime access where every millisecond counts.

[23] URL http://hadoop.apache.org

5 Implementation

The proposed interface has been implemented and tested in various situations. This section contains details about the capabilities and the implementation, as well as, information on how the different requirements have been implemented.

Section 5.1 shortly explains the utilized methodology and the general environment of the implementation. The successive sections illustrate in which way the different requirements have been implemented and how this functionality can actually be used. The concluding chapters contain some of the more important details about the architecture and technical details.

The prototypical implementation has been named ***GrapeFS*** and thus will be called so when being mentioned.

5.1 Methodology

5.1.1 Programming language and filesystem interface

As mentioned, there are two ways of implementing a filesystem interface. Directly within the kernel or using a user-space implementation.

The latter has been chosen due to the need of various different frameworks (LLVM, DCMTK, FreeImage, OpenGL, ..) and hardware features. One of the first steps therefore is to evaluate a suitable programming language and, more importantly, an adequate library for the filesystem implementation. Regarding the initial requirements any user-space filesystem library should be able to provide the necessary functionality in order to realize the desired interface. To get a rough impression of the most promising choices, FUSE and 9P, initial testing has been done using the iozone benchmarking utility [24]. The results can be found in the appendix in section C.

The tests have been performed using a single thread. The first measurement has been done using a pure tmpfs mount in order to evaluate the upper limits that are set due to CPU or memory throughput limitations. The other two measurements have been performed using a simple passthrough filesystem, both using a FUSE and a 9P implementation. The FUSE interface produces better results in nearly all tests. The 9P throughput however would be generally high enough to be usable in this context. Most bottlenecks will probably be due to CPU limitations when executing the processing code.

The final decision has been made in favor of FUSE. Due to the potentially better performance and the more widespread use in existing projects this approach is more reasonable. Also support for some modern filesystem features will be necessary. This includes features like user access permissions, symbolic links and extended attributes. In contrast, regarding the 9P2000 protocol, these features would only be available using the latest Linux specific 9P2000.L extension [Garlick, 2011].

FUSE generally requires the implementation of a C interface. However, in respect of maintainability and compatibility to potential third-party libraries, the core application will be implemented in C++. Particularly, the implementation will make use of some more important features of the most recent C++11 standard [ISO, 2011]. Several C++11 features are being used to improve overall code structure and readability, for instance nullptr, overrides, type deduction and constructor delegation. Atomics are used to already ensure a certain amount of thread

[24] URL http://www.iozone.org

safety. Lambdas and std::function are used for a flexible implementation of the double dispatch pattern.

5.1.2 Goals and requirements

This thesis strives to answer a few major questions regarding the proposed architecture.

The most important step is to assess the technical possibility of the proposed implementation and outline potential problems. This implementation is therefore used to collect profiling information. These information are successively used to evaluate the possible use of this approach in a variety of different situation. The required situations have been described in the analysis (cf. section 3.3.2, page 25). The necessary measurement requirements will be determined in the evaluation and compared with the measurement data.

A secondary goal is to assess the flexibility and possibility of extending the proposed architecture and implementation for various other scenarios.

5.1.3 Implemented design patterns

This section contains a brief overview of the different patterns from the fundamentals (section 2.3, p. 18) and how these patterns are applied in ***GrapeFS***.

Pipes and filters — Overall structure of the filesystem. The folders represent the filters. The argument files, especially when used as symbolic links, represent the pipes that transfer the data.

Composite — Internal realization of the filesystem tree, especially when mapping the filesystem path to specific objects. For FUSE requests, the exact type of a filesystem object is of no importance.

Decorator — Used to attach different types of filesystem decorators to the internal filesystem implementation. This allows to replace the FUSE interface with other filesystem interfaces without modification of classes within the GrapeFS hierarchy.

Double dispatch — Utilized in dispatching the FUSE calls to the correct FUSE decorator implementation.

5.2 Design and user interaction

5.2.1 Startup and initialization

The ***GrapeFS*** filesystem can be mounted on any empty directory within the local filesystem tree, assuming write access is available. The output of this startup process, as shown in figure 19, contains some information about loaded plugins for language frontends and data formats.

Figure 19: Mounting GrapeFS. Any mount point can be chosen as a startup parameter. The "-s" option is mandatory to prevent multiple FUSE threads. Currently not all components of GrapeFS are thread-safe. The "big_writes" option significantly improves write throughput by reducing the number of write calls to user-space.

From this point on, the mounted directory will provide the processing functionality through the filesystem interface. By default the directory will be empty. A hidden ***.libs*** folder is available at any time within the mounted root directory by default and therefore poses an exception to this. This special directory will be explained later.

Folders and filters

Filters to process data are represented by folders within the ***GrapeFS*** filesystem. Folders can be created by the user anywhere within the filesystem as shown in figure 20. Nesting of several folders within each other is possible and can represent a meaningful structure. Folders can have arbitrary names specified by the user. However, similar to files, every processing folder needs to have a specific extension that specifies which kind of programming interface will be used to specify the actual processing code.

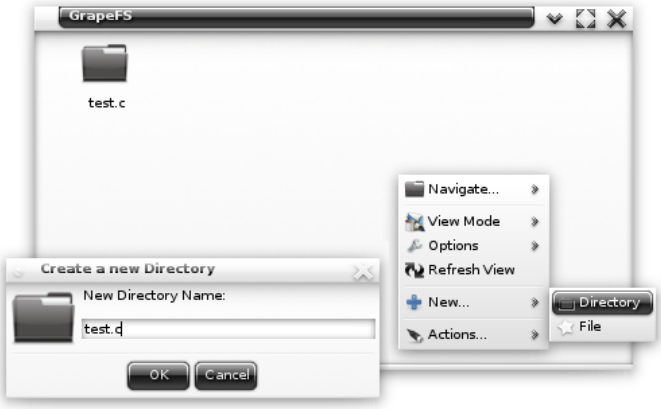


Figure 20: Folder creation in GrapeFS using a ".c" extension. This specifies that the processing code within this folder contains C syntax.

Depending on this extension the folder will be associated with a certain ***executor***. This association also affects the default content of the folder.

Within the ***.c*** folder a file called ***Kernel.c*** exists automatically. This file must be used to provide the processing source code to ***GrapeFS***. The code within this file is the only place where data modification can be done. This forces a minimal amount of structure within the filesystem. Figure 21 contains an example of a newly created folder.

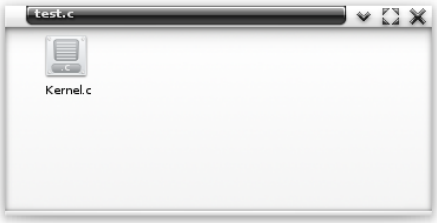


Figure 21: Every newly created folder automatically contains files that are necessary to specify the actual source code for the data processing. This file can not be deleted by the user.

Within these folders other special files and folders can exist as well. Files are generally used to transfer image data and arguments to the processing function. A special ***Output*** folder is used to provide user access to the processing result. This folder is visible as soon as any result data is available. These special files and folders, among other interaction details, will be examined more closely within the following paragraphs.

Processing code and result

The most direct and native way of specifying the filter kernel is by using C code. Any editor or character stream can be used to write this code to the ***Kernel.c*** file. ***GrapeFS*** specification dictates that the code needs to define a ***compute*** function. This function will be called by the filesystem to perform the data processing. For the data transfer between the filesystem and the processing code a lightweight structure, called ***gdata***, is used. This structure is declared in the ***GrapeFS.h*** header. The ***gdata*** structure solely contains the number of image dimensions/channels and the actual data as raw (unsigned) bytes.

After processing, the compute function is required to return such a structure as well. The returned value will be used as the processing result.

Figure 22 shows an example and the result of a simple "passthrough" filter. The result of this filter is accessible by the user using the output folder. An extended version of this filter is shown in figure 23 which iterates and inverts all data values.

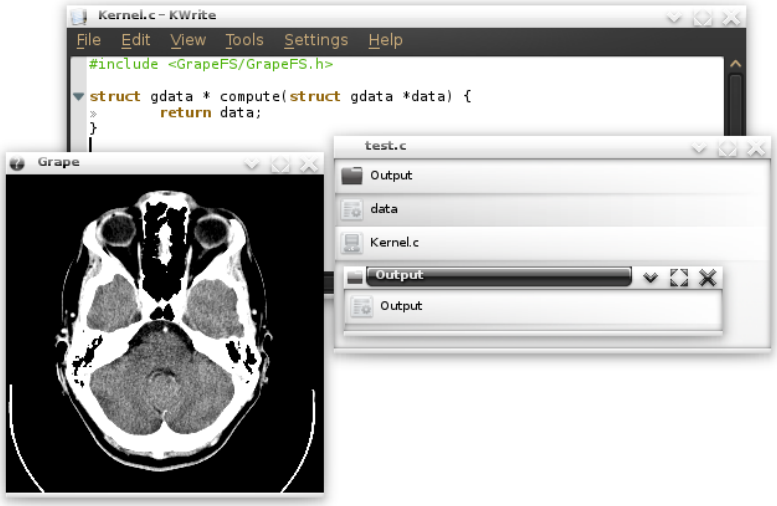


Figure 22: Example for the most simple processing code that can be passed to GrapeFS. The compute function receives a single image and just returns the data without any modification.

Within the output folder a file is provided that contains the raw result data. To visualize the raw data, a special imaging application has been used. Further explanations on file formats will follow later.

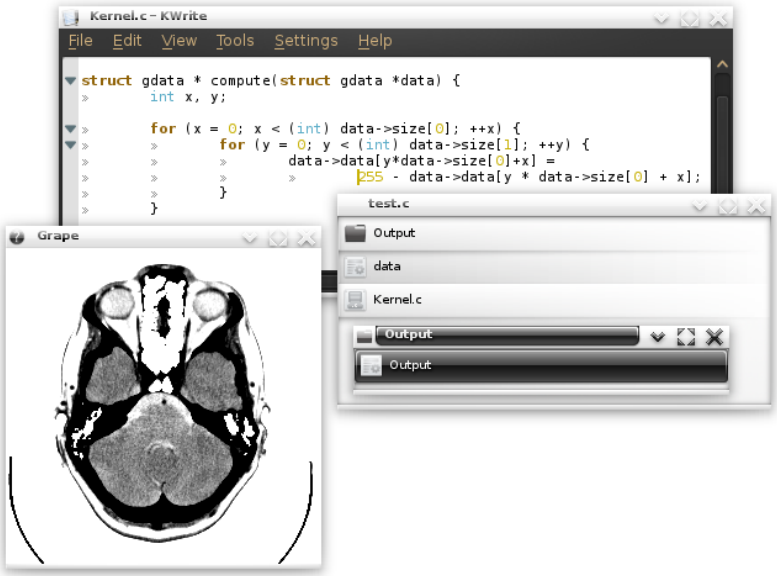


Figure 23: Example for a more advanced processing filter. This filter iterates over the width and height of the image and inverts all pixel values. The return statement is unchanged.

Any folder, or processing filter, can only provide a single result using the ***Output*** folder. The ***Output*** folder is mapped directly to the returned ***gdata*** structure.

Files and arguments

Up to now the processing code has only received imaging data as input arguments. Additionally, the same way of using files as a transfer method can be used to specify other arguments as well. These arguments can be additional images, but also other plain data like numeric values. From the user perspective, it is sufficient to simply declare these arguments in the function definition. The specified arguments will be automatically provided as files in the filesystem and passed to the data processing upon execution. An example of processing code that receives an additional argument and the influence on the folder structure is shown in figure 24.

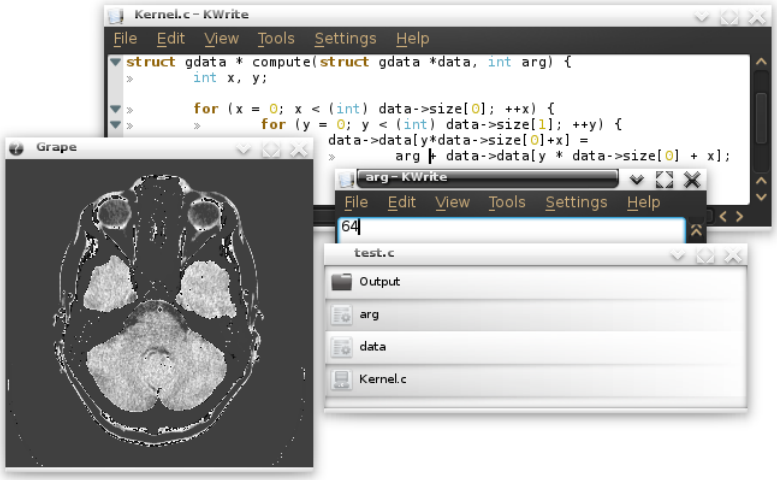


Figure 24: Example for numeric input arguments. The ***int*** argument is taken from the parser and made available as a file. The associated argument name is used as the file name.

Any text editor or script can be used to write values to these argument files. The output result will be updated as soon as an input argument has changed.

Data transfer

Writing data to files is not the only way of transferring data into the filesystem. Any file representing an argument can be replaced with a symbolic link to user-defined files. ***GrapeFS*** will read the content of these files and pass the content as argument values, just as if the content had been written to the file directly. An illustration of these two ways of transferring data is shown in figure 25.

This is not only useful to avoid unnecessary I/O overhead but is also the way to go when connecting processing folders with each other. To access the result of other filters the symbolic link can just point to the corresponding processing folder itself. An example of this is shown in listing 3. The processing result of this folder will then be used as the passed argument value, just as manually transferring the ***Output*** content to the file.

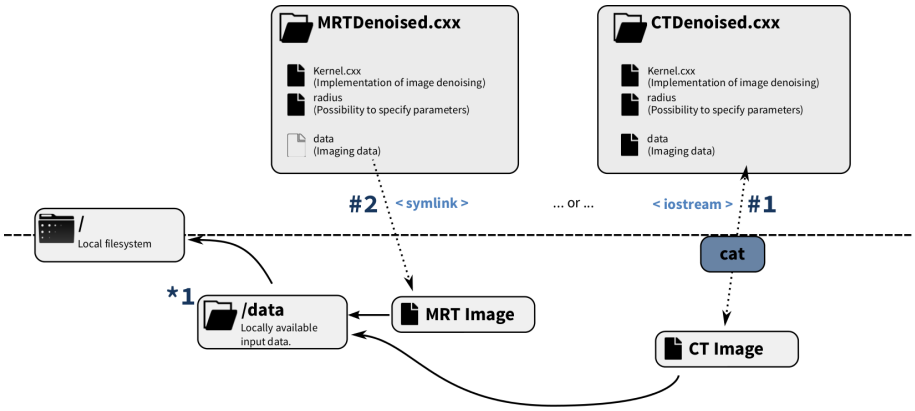


Figure 25: Options to transfer data to the GrapeFS filesystem. Data can be written to the argument file directly or by replacing the file with a symbolic link. These two ways are illustrated by ***#1*** and ***#2***. ****1*** represents locally available data. ***#1*** symbolizes the process of writing data directly to the ***data*** file. This data transfer must be managed by a process outside the filesystem, e.g. as pictured using the cat command. ***#2*** symbolizes the use of symbolic links. The file is removed and replaced with a symbolic link to the target data. The filesystem reads this data by itself without the need of an external process.

To pass data from one processing directory to another a directory listing would look like the following:

```
[spx@rapture-arch32 output.c]$ ls -lisa
total 0
 6 0 drwxr-xr-x 2 spx spx   0 Oct 12 10:03 .
 1 0 drwxr-xr-x 2 spx spx   0 Oct 12 10:03 ..
18 0 lrw-r--r-- 1 spx spx  20 Oct 12 10:05 data -> /tmp/GrapeFS/input.c
 7 0 ---------- 1 spx spx 277 Oct 12 10:03 Kernel.c
 9 0 drwxr-xr-x 2 spx spx   0 Oct 12 10:05 Output
```

Listing 3: Example listing to connect two processing folders

5.2.2 Types and file formats

Images are usually stored and transferred using some type of image format. This applies to the transfer of images into the filesystem as well as the access of results from the ***Output*** folder. To cope with this, ***GrapeFS*** is able to dynamically parse the format of input data or encode output results to different formats.

Files and folders can be associated with certain types and formats as well as a user-specified flag for these formats. For this ***GrapeFS*** utilizes the filesystem feature of "extended attributes" (xattr). These attributes usually have a namespace, a name and a value. The namespace for all ***GrapeFS*** attributes is ***grapefs***. The name to specify the format is ***encoding***. Therefore, a valid example to set the format for the input data or output result is shown in listing 4.

```
$ setfattr -n grapefs.encoding -v image Output
$ setfattr -n grapefs.encoding.flag -v 50 Output
```

Listing 4: Extended attributes example to set the output format

Further explanations for available formats and flags can be found in the implementation details in section 5.4.

When setting such attributes on an input file, the data will be converted immediately after releasing the write handle. The result of the format conversion is then passed to the processing filters in raw form. When set on the ***Output*** folder the behavior is a little different. Instead of providing the raw data as a single file, the ***Output*** folder will contain additional folders. One folder for every available format from the specified encoder. This folder structure and the content of the folders can be seen in figure 26. Figure 27 shows an example of the extended attributes in case of the previous setting.

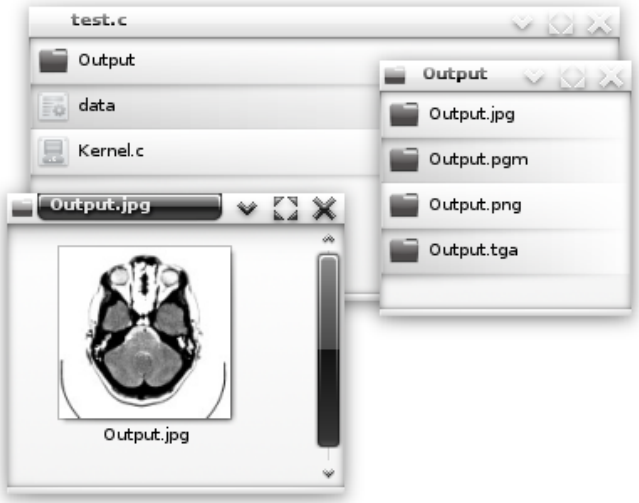


Figure 26: General-purpose image output. The output folder provides access to different available output formats using additional subfolders. Within these folders the data can be accessed directly using the appropriate format.

Figure 27: Exemplary output of extended attributes on the output folder. The grapefs.encoding attribute can be set to any value. The content of the output folder will dynamically adapt to formats available for the set attribute value.

The reason for these additional folders is easily understandable. File listings for the ***Output*** directory, for example as requested by file managers, also include information about the specific files. Among other information these include the file size. As soon as an external process

accesses the folder, the raw data needs to be encoded in order to provide a correct file size. Using additional subfolders this is only necessary when accessing the specific folder. This saves a meaningful amount of CPU time as the format encoding happens on-demand and not for all possible file formats.

5.2.3 Modifications and change notifications

Notifying applications that have accessed or are currently accessing output files upon change events has been a bit tricky. By default, applications would only be able to detect changes in the output directory using manual polling. However, the implemented solution of broadcasting change notifications is quite suitable and works reliably.

Upon startup of the ***GrapeFS*** filesystem, a subprocess is forked that communicates with the filesystem using conventional pipes. Every time the result of a filter kernel has changed this process is notified and issues a utime call, thereby triggering an inotify event within the kernel. This event can be received by user-space applications in order to react to changes in the output result. Figure 28 shows a scheme of the call sequence for such an event.

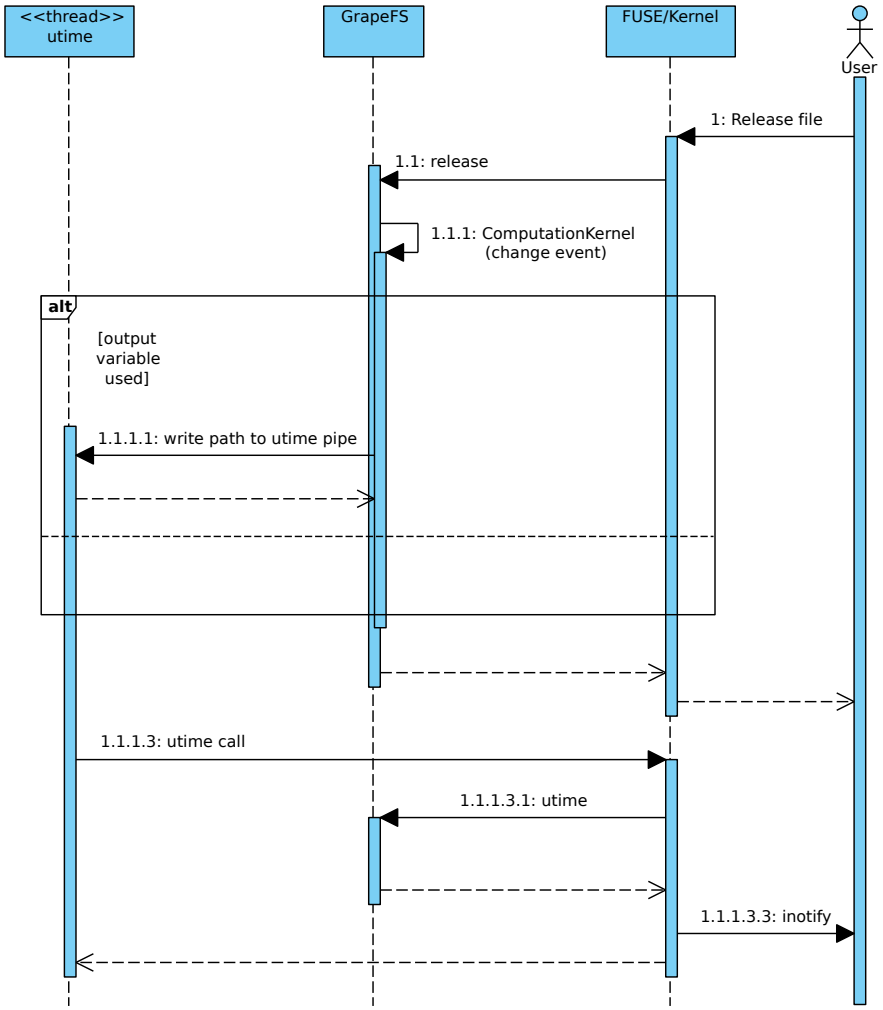


Figure 28: Sequence diagram that illustrates the interaction between the ***GrapeFS***, the ***utime*** process and the FUSE calls upon change events.

The utime call is only performed when the corresponding output folder has been accessed before. Otherwise the call itself would trigger unnecessary format conversions.

This slightly cumbersome event cascade is needed because of multiple problems.

1. utime call

inotify events are the up-to-date method, issued by the kernel, to notify observing processes of changes within folders or files. However, these events can only be send for events that are observable by the kernel itself. As changes in the processing results are usually triggered internally within the filesystem, there in no way for the kernel of knowing that the actual output has changed. Because of this the change event has to be triggered manually through the kernel VFS.

2. Forking

The current ***GrapeFS*** implementation has some parts that are generally not thread-safe and therefore the recommended behavior is to start the filesystem process using a single thread. The problem is that the previously mentioned trigger would issue another call to the filesystem while the issuing operation is still active. This provokes a reproducible deadlock within the filesystem. The intuitive way of solving this situation is by issuing the utime call in an independent process, letting the original filesystem call return.

5.3 Language frontends

ComputationKernel (API)

All available frontends for different programming languages inherit a mutual interface called ***ComputationKernel***. Currently these are the ***ClangKernel***, ***GLKernel*** and ***JNIKernel***. Explanations to all three will follow in the successive sections. Figure 29 shows an overview of this inheritance structure.

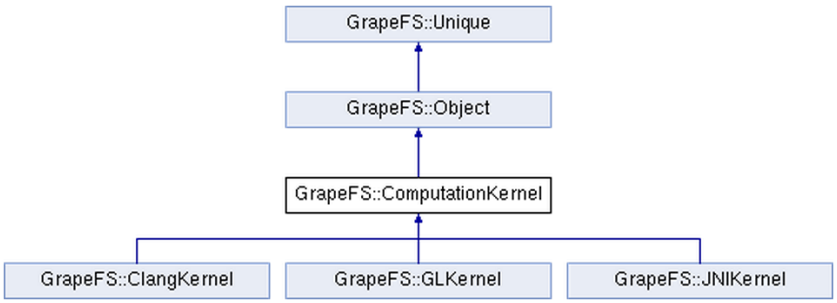


Figure 29: Class structure of ComputationKernels and the available implementations.

The interface implemented by the language frontends is rather lightweight and tries to make the process of implementing support for new languages as easy as possible. Basically only two distinct mechanisms are provided. The first one handles modifiable arguments discovered by the parser. The second handles execution of the data processing and returning the processing

result. The ComputationKernel interface must be implemented by any new language frontend. Figure 30 shows the important parts of the API.

Every ***ComputationKernel*** implementation is compiled to a shared library and loaded at runtime. Retrieving of suitable directory extensions for a certain implementation is done using a C interface. An example of an implementation for this interface can be found in the appendix in section F.

Public Member Functions

	ComputationKernel (const std::string &**name**, **Directory** *parent)
virtual	**~ComputationKernel** ()
virtual int	**setArgument** (const std::string &**name**, **Object** *node)
virtual int	**clearArgument** (const std::string &**name**)
virtual int	**executeAssembly** (**gfs_data_t** **dest, size_t **size)=0
virtual const std::vector < **ComputationArgument** > &	**arguments** () const

Figure 30: ComputationKernel API. ***executeAssembly*** is used to perform the actual data processing.

If the extension of newly created directories maps to a known ***ComputationKernel***, an instance of this implementation is created and associated with the folder object. This associated object is used to generate the data provided through the output directory. As the format handling for input and output data is done transparently, this object can assume to solely handle raw data.

As mentioned before, ***ComputationKernel*** provides two tasks which can be used in any specialized frontend implementation. These tasks are explained in the following two paragraphs.

Argument handling

The actual frontend implementation only needs to provide a list of available arguments that can be modified by the user. These are retrieved using the ***arguments*** method. The rest of the argument handling is completely done internally by the filesystem. The arguments are automatically exported as files into the filesystem. Also the handling, when the files have been replaced with symbolic links, is done without any manual work. The results of this automated handling are then passed to the frontend using ***setArgument*** in combination with the associated argument name. When performing the data processing, the frontend can simply query this argument mapping for the correct argument value.

Execution

Execution of the data processing is done with a single invocation to ***executeAssembly***. This method receives nested pointers to the destination and size of the processing result as arguments.

Before performing the previously described action, the processing directory first compares the modification times of the current processing code and the current output data. If the processing code and the argument mappings have not changed in the meantime, the modification times of both items will be identical. If this is the case, the current output data is still up-to-date and the data processing is not performed again. This has the benefit that no processing time is wasted and the processing is only triggered upon user request or when some change has to be expected.

5.3.1 C/C++

Instead of compiling and loading the executable processing code as a relocatable shared library a more flexible approach has been chosen. The C code is parsed out-of-process using the clang compiler and translated to LLVM bytecode using the common stdout stream. An illustration of this process is shown in figure 31.

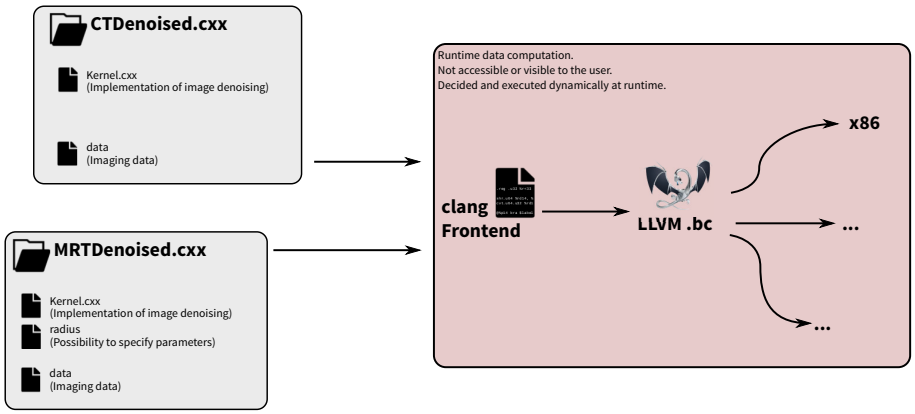


Figure 31: Schematic visualization of the C parsing and execution process. The processing code is parsed using the Clang compiler front-end and translated to LLVM code. The LLVM module is used to retrieve the arguments of the compute function. When executed, the arguments are dynamically passed to the LLVM execution engine. The compute function is executed using the LLVM JIT engine.

The LLVM bytecode is then loaded as an executable module and optimized using LLVM passes and the LLVM JIT compiler. In this context, no extensive investigations have been done on all available passes, so performance of the filter execution could possibly be improved at this point. However, this is not an actual problem. Also clang and LLVM as a compiler and execution environment is constantly improving, getting close to the optimizations provided by the GCC compiler. [25]

Resolving external libraries and symbols at runtime

The current ***GrapeFS*** implementation has an initial mechanism to load and resolve external symbols. Symbolic links to external libraries can be created in the special ***.lib*** directory in the filesystem root. The shared library this link points to is opened and imported globally. When loading external libraries using this mechanism all exported functions can be used within the C or C++ processing kernels.

5.3.2 Java

Processing code using Java can be specified just as C or C++ code. The ***GrapeFS*** executor is able to find the processing code as long as a static compute method is defined within a class called Kernel. An example of utilizing this processing frontend is shown in figure 32. Contrary to C/C++ and GLSL filters there is currently no parser support for Java code. Therefore, specification states that the input data is passed as a ***ByteBuffer*** to this method. This buffer can be modified

[25] URL http://openbenchmarking.org/s/clang

directly. However, thereby it is currently not possible to define additional arguments that will be passed through the filesystem interface.

When the Java source code changes, it will be dynamically compiled to Java bytecode and executed using JNI.

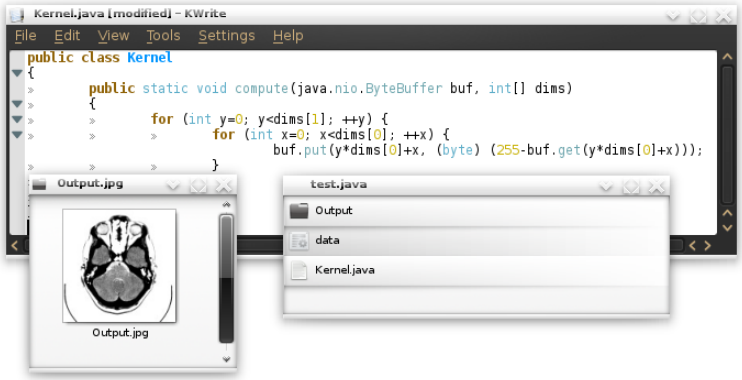


Figure 32: Example for data processing using the GrapeFS Java front-end. The processing folder has been created using the .java extension. Java code is written to the Kernel.java file. Within the Java code the data is handled using a ByteBuffer. As with every other processing front-end the processing result can be accessed using the output folder.

5.3.3 OpenGL/GLSL

The visualization support using accelerated OpenGL rendering needs some kind of offscreen rendering as the filesystem by itself does not have any window handle for direct rendering. There are two ways of achieving such a desired behavior. The pbuffer extension allows the creation of a valid OpenGL context without requiring a window handle at all. A different, and partially more compatible, approach is to use a framebuffer object (FBO) in combination with a hidden window. However, in both cases a running window system is needed.

The current implementation uses the pbuffer approach in combination with the GLX protocol which is the standard API for X11/Linux applications to use OpenGL acceleration. The same could be achieved using Windows WGL as well. However, pbuffers have been deprecated in Mac OS X 10.7, hence FBOs could be a viable option for future adaptions. A schematic of the overall architecture is shown in figure 33.

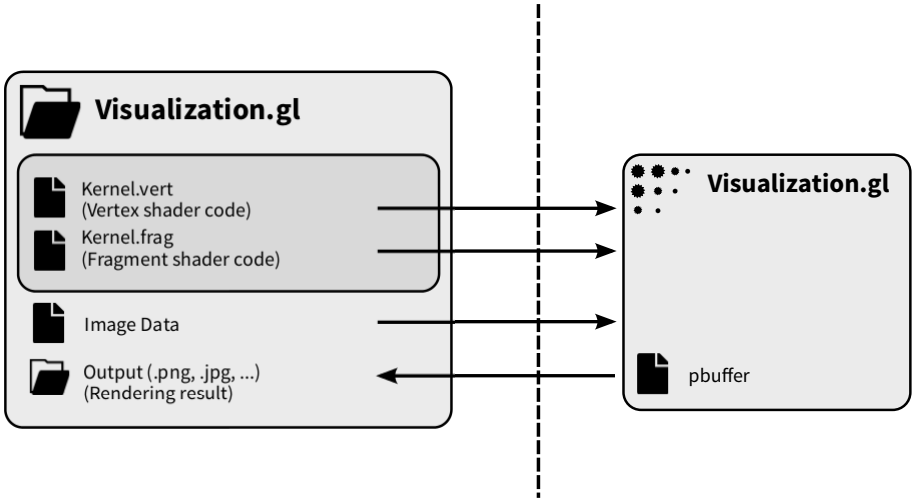


Figure 33: Architecture of the OpenGL visualization. The vertex and fragment shader are compiled within the OpenGL kernel and rendered to a pbuffer. After rendering this output is read into system memory and exported through the output folder. Naturally other processing code could use this output as well by using a symbolic link to the .gl folder.

The folder of OpenGL/GLSL kernels provides two files to specify the actual processing code. These files represent the common vertex and fragment shaders within the OpenGL processing pipeline. In most cases the more important part will be the fragment shader which eventually assigns the result to the image coordinates. Examples of two basic shaders for this purpose are shown in figure 34. Figure 35 shows the previously described folder structure and the processing result of these shaders.

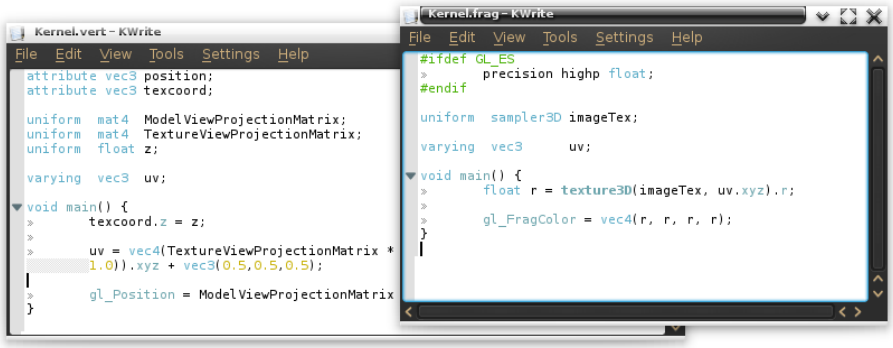


Figure 34: Simple GLSL shader for the image visualization using GLSL. The input data is an image series that is passed to the shader as a GLSL sampler3D texture. The name of this GLSL uniform is used as the associated file within the processing folder.

Figure 35: Example output visualization for the previous GLSL shaders. The image is accessed using the output folder.

The overall approach is similar to the remote rendering approach described in [Engel et al., 2000]. As described in figure 34 the input data is read as DICOM format while the result is accessed using general-purpose image formats. Figure 36 illustrates the interaction with extended attributes in this particular situation.

Figure 36: Example use of the extended attributes to specify different formats for the input and output data. The input data is read from a DICOM image and converted to a raw image series. The output folder is set to provide the visualization result using general-purpose image files.

5.4 File formats and flags

DataFormat (API)

New file formats can be added by implementing a single interface, called ***DataFormat***. This implementation is translated into a shared library which is loaded at runtime.

Every ***DataFormat*** has a name, a list of file extensions and two methods to read and write data. Figure 37 shows the important parts of this API

Public Member Functions

	DataFormat ()
virtual	**~DataFormat** ()
virtual std::string	**format** ()
virtual const char *	**extension** (int i)
virtual const char **	**extensions** ()
virtual size_t *	**parse** (unsigned char **dest, const unsigned char *data, const size_t &size)
virtual int	**dump** (const unsigned char *buf, **size_ptr** size, **VariableHandle::unique_ptr** handle, std::string **extension**)

Figure 37: API to implement formats in GrapeFS.

The name of the ***DataFormat*** is used for the ***grapefs.encoding*** attribute mapping. When this attribute is set on a file or output directory this name will be used to look up the correct format implementation.

General-purpose formats

A parser for general-purpose image formats has been implemented using the FreeImage library [26]. An example of utilizing this implementation to access the output data as an JPEG image is presented in figure 38.

The following list shows the most important file formats supported by this implementation. However, this is only a small part of all supported formats.

- BMP (reading, writing)
- GIF (reading, writing)
- JPEG/JIF (reading, writing)
- JPEG-2000 (reading, writing)
- PBM/PGM/PPM (reading, writing)
- PNG (reading, writing)
- TARGA (reading, writing)
- TIFF (reading, writing)

Listing 5: Partial list of supported general-purpose file formats using the FreeImage based parser.

To simplify the computation process in this context, all input data is converted to 8-bit grayscale values. However, this is an artificial limitation to initially simplify some aspects of the prototype.

The ***grapefs.encoding.flag*** attribute is used to pass additional encoding flags to the format writer. For instance this flag is used by the JPEG compression in order to determine the desired quality factor ranging from 1 to 100. The effect of using this flag to reduce the compression quality is shown in figure 39.

[26] URL http://freeimage.sourceforge.net

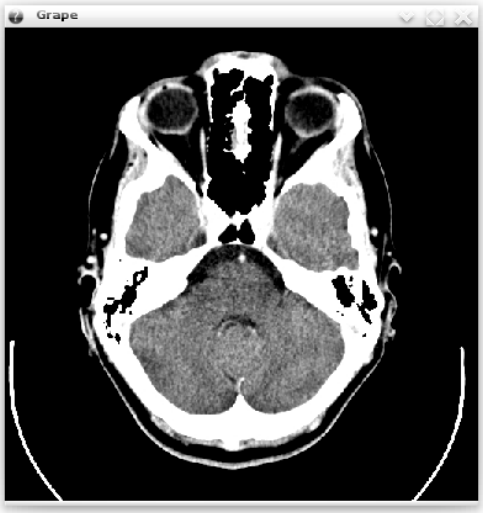


Figure 38: Use of extended attributes to specify the compression ratio for the JPEG format. The grapefs.encoding.flag attribute has been used to set the best possible quality (100).

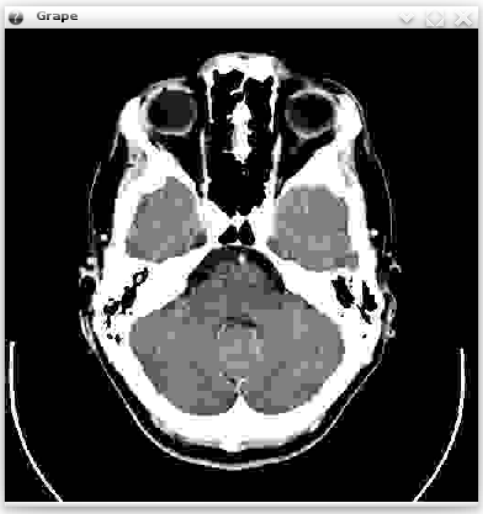


Figure 39: Example of lowering the output quality. The worst quality has been chosen by setting the grapefs.encoding.flag to 1. Severe degradation of fine structures are clearly visible.

DICOM

Due to the initial requirements, an additional format parser has been implemented. This parser has the capability of reading DICOM files using the DCMTK [27] library. In the process of evaluating the implementation the DICOM parser has been mainly used to read 3 dimensional image series as input data for the OpenGL based visualization.

[27] URL http://dicom.offis.de/dcmtk.php.en

5.5 Tools to improve user interaction

grapefs.dlopen

Synopsis: grapefs.dlopen MNTPOINT LIB

The ***grapefs.dlopen*** command is basically syntactic sugar around the manual linking of external libraries within the ***.libs*** directory. The symbolic link within the ***.libs*** directory is created automatically by specifying the mount point and external library.

```
1  ln -s "$2" "$1/.libs"
```

Listing 6: grapefs.dlopen code

grapefs.mkdir

Synopsis: grapefs.mkdir DIRECTORY TEMPLATE

The ***grapefs.mkdir*** command simplifies the creation of processing folders for a specific programming language and processing template. As with the common ***mkdir*** command, the first parameter specifies the path and name of the destination directory. The programming language is taken from the folder extension as usual. Additionally a code template is specified using the second parameter. ***GrapeFS*** ships with a few templates for every language. Assuming the specified template exists the code is automatically written to the adequate files containing the processing code. The user can then simply modify the data processing based on this template code. The shell code to achieve this effect is illustrated in listing 7.

```
1  if [ -d /etc/GrapeFS ]; then
2    tmpldir="/etc/GrapeFS"
3  elif [ -d /usr/local/etc/GrapeFS ]; then
4    tmpldir="/usr/local/etc/GrapeFS"
5  fi
6
7  if test "${tmpldir+set}" != set; then exit 1; fi
8
9  filename=$(basename "$1")
10 extension="${filename##*.}"
11 filename="${filename%.*}"
12
13 mkdir "$1"
14 if [ $? -ne 0 ]; then exit 1; fi
15
16 tmpldir="$tmpldir/Templates/$extension/$2"
17
18 if [ ! -d "$tmpldir" ]; then exit 1; fi
19
20 for f in `find "$tmpldir/" -type f`; do
21     fbase=`basename "$f"`
22
23     cat "$f" > "$1/$fbase"
24 done
25
26 data="$1/data"
27 output="$1/Output"
28
```

```
29  if [ -f "$data" ]; then
30    setfattr -n grapefs.encoding -v image "$data"
31  fi
32  if [ -d "$output" ]; then
33    setfattr -n grapefs.encoding -v image "$output"
34  fi
```

Listing 7: grapefs.mkdir code

Table 3 lists the templates that are currently available for direct creation.

Extension	Template	Description
c	passthrough	Simple input data passthrough
cxx	passthrough	Simple input data passthrough
cxx	opencv	Wrapping of the input image into an OpenCV structure
java	passthrough	Simple input data passthrough
gl	plain	Image series projection onto a plain
gl	circle	Example rendering of a circle gradient

Table 3: Available filter templates

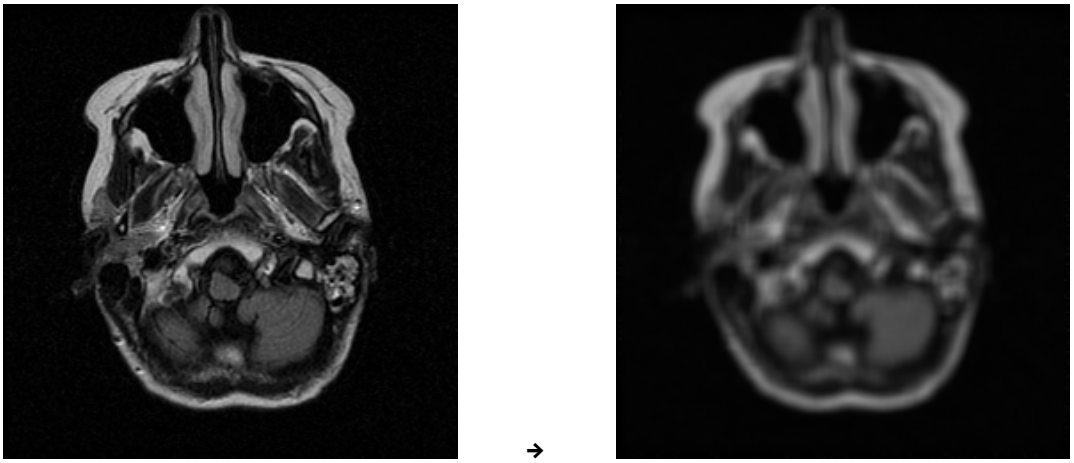
5.6 Implementation of the requirements

In order to acquire some hands-on experiences of the implemented functionality and the actual filesystem behavior, several example filters have been implemented and evaluated. Of course this also serves the purpose of showing that the prototype is able to actually fulfill the required functionality from section 3.3.1. The example code covers the C kernels, as well as, the OpenGL accelerated visualization.

Blur filter

To demonstrate the general handling of input data in combination with variable arguments a blur filter has been implemented. This filter receives a single input image as well as a numeric radius parameter and replaces every data value with the mean of all values within the given radius. To simplify the handling of the borders all values outside the actual data range are assumed as black. The effect of this filter is shown in figure 40.

This processing code implements the functionality required by "Req #1".



Original image data → Blur result data

Figure 40: Example results for Blur filtering (radius=3).

```
1  struct gdata * compute(struct gdata *data, int radius) {
2    if (data->dimensions != 3)
3      return 0;
4    if (data->size[2] != 1)
5      return 0;
6
7    unsigned char *result = (unsigned char *) malloc(data->size[0] * data->size[1]);
8
9    const unsigned int opSize = pow(radius*2+1, 2);
10
11   int x, y, tmp_x, tmp_y;
12   unsigned int tmp;
13
14   for (x = 0; x < data->size[0]; ++x) {
15     for (y = 0; y < data->size[1]; ++y) {
16       tmp = 0;
17
18       for (tmp_x = -radius; tmp_x <= radius; ++tmp_x) {
19         for (tmp_y = -radius; tmp_y <= radius; ++tmp_y) {
20           tmp += getValue(data, x+tmp_x, y+tmp_y);
21         }
22       }
23
24       result[y*data->size[0]+x] = tmp/opSize;
25     }
26   }
27
28   return gdata_new(
29     (size_t[]) {
30       data->size[0],
31       data->size[1],
32       1,
33       0
34     }, result);
35 }
```

Listing 8: Example code for blur filter

Image correlation

To process the combination of multiple images a spatial cross-correlation has been implemented. The first input data is used as the base image. The second input data is used as a movable correlation mask. This implementation, especially as being implemented in spatial domain, is far from being fast. However, it demonstrates the potential usage and combination of multiple input images. Figure 41 shows the file structure for the multiple input images. The result of this operation in presented in figure 42.

This processing code implements the functionality required by "Req #2".

Figure 41: File manager view of the ***GrapeFS*** folder and file structure.

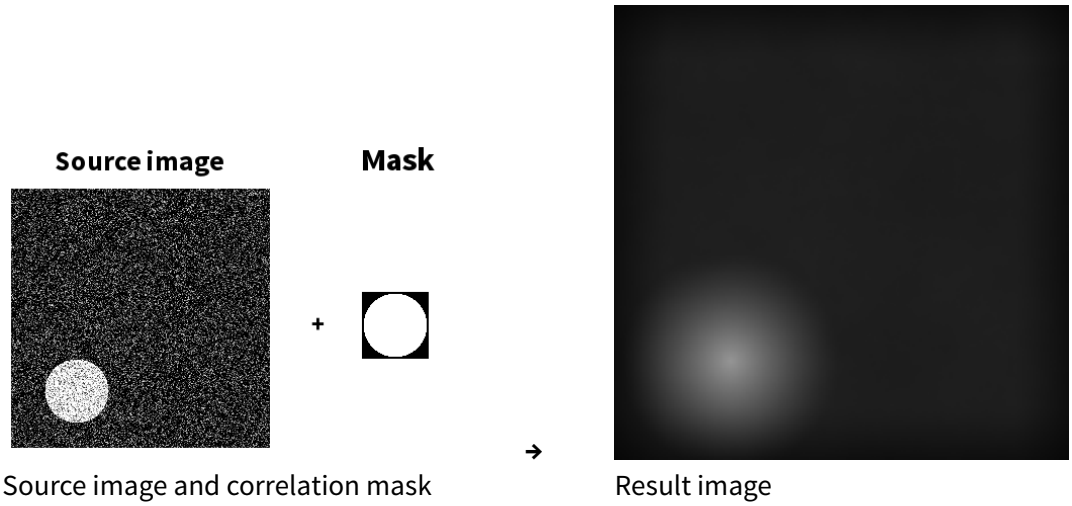


Figure 42: Example result for the combination of multiple images.

Visualization support

A simple shader code for the programmable OpenGL filter pipeline has been implemented in order to demonstrate the accelerated visualization of image series. This has been done mainly for the purpose of evaluating the throughput of the critical transfer paths in order to retrieving the rendering result. The result of this visualization can be seen in figure 43

This processing code implements the functionality required by "Req #3".

```
1  attribute vec3 position;
2  attribute vec3 texcoord;
3
4  uniform  mat4  ModelViewProjectionMatrix;
5  uniform  mat4  TextureViewProjectionMatrix;
6  uniform  float z;
7
8  varying  vec3  uv;
9
10 void main() {
11   texcoord.z = z;
12
13   uv = vec4(TextureViewProjectionMatrix * vec4(texcoord-vec3(0.5,0.5,0.5), 1.0)).xyz +
         vec3(0.5,0.5,0.5);
14
15   gl_Position = ModelViewProjectionMatrix * vec4(position, 1.0);
16 }
```

Listing 9: GLSL vertex shader used to transfer the xyz image series coordinates

```
1  uniform  sampler3D imageTex;
2
3  varying  vec3      uv;
4
5  void main() {
6    float r = texture3D(imageTex, uv.xyz).r;
7
8    gl_FragColor = vec4(r, r, r, r);
9  }
```

Listing 10: GLSL fragment shader used to map the image data

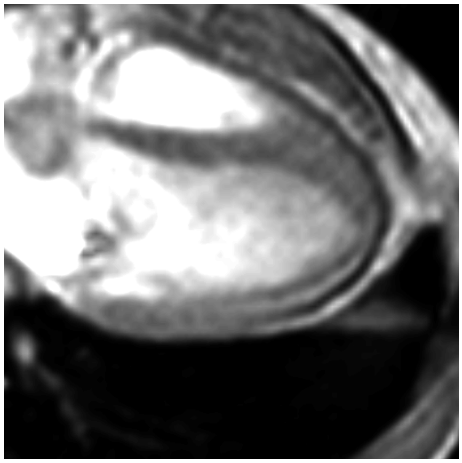


Figure 43: Example output using the preceding GLSL shaders.

6 Evaluation

The evaluation will begin with a comprehensive look at the experiences made during the actual use of the implementation. Positive experiences and advantages due to the implemented access interface will be highlighted. This includes expected capabilities which were now verifiable in actual real-world scenarios. However, this section will also contain possible weaknesses in usability that have been experienced in this context. These difficulties do in no way implicitly cause any conflicts with the necessary requirements but instead should give some hints on what could be possibly improved in future development.

The subsequent sections contain objective profiling measurements that have been acquired using the actual implementation. These measurements are timing results and processed event data providing internal measurements from the implementation itself. This allows for a reasonable comparison between the format handling and data processing between the filesystem implementation and a conventional implementation that directly interacts with the data.

6.1 Interface and user interaction

6.1.1 Overview

To set up a context for the experiences figure 44 summarizes the structure of the processing structure and dataflow between the different layers.

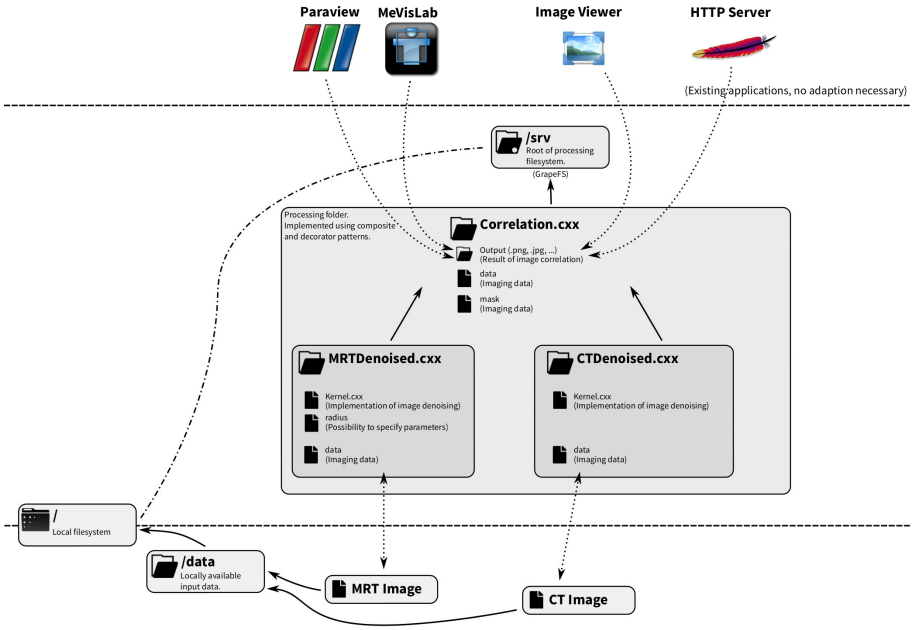


Figure 44: Overview of the overall ***GrapeFS*** filesystem structure. Top: Application layer; Center: Mounted GrapeFS filesystem; Bottom: Local filesystem/Persistent data

The layer at the top represents the user-space applications. These can be represented by any existing application run by the user. The application accesses the processing result using the provided general-purpose image format within the output folder. No adaption within the application itself is necessary in order to achieve this.

The middle layer represents the ***GrapeFS*** filesystem implementation. All files and folders in this layer are generated at runtime and represent a "virtual" structure. It is thereby possible for any application from the top layer to access the dynamically generated processing result using the provided filesystem interface.

The bottom layer represents the locally available data. This data contains the available images that are stored in some persistent format. These images are used as the input data for the processing tasks.

6.1.2 Advantages

The subjective experiences that have been made during the development and evaluation of the finished implementation were quite positive. The advantages that have been anticipated at the very beginning have shown to be possible and realizable by the implementation.

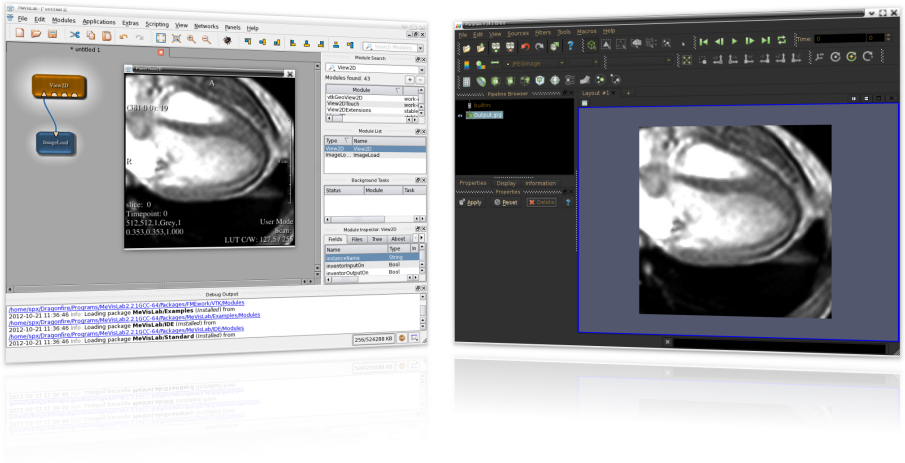


Figure 45: Real-world example using MeVisLab (left) and ParaView (right) to access the processing result in JPEG format. The image is transparently generated using the GLSL visualization from the implementation on page 54.

Interacting with the data and interactively performing the data processing was more fluent than expected. Due to the filesystem interface the processing results were accessible from any available application, from plain image viewing applications, up to specialized editing and processing applications. An example of accessing the processing result can be seen in figure 45. Interacting with the actual processing code and prototyping algorithms was very handy. Having access to view the results in realtime while changing the algorithms alleviates the prototyping of new algorithms while pursuing a desired result.

Even access from Windows applications and applications running on other machines was possible using a simple network share. This situation is shown in figure 46. No problems have been encountered with such a setup.

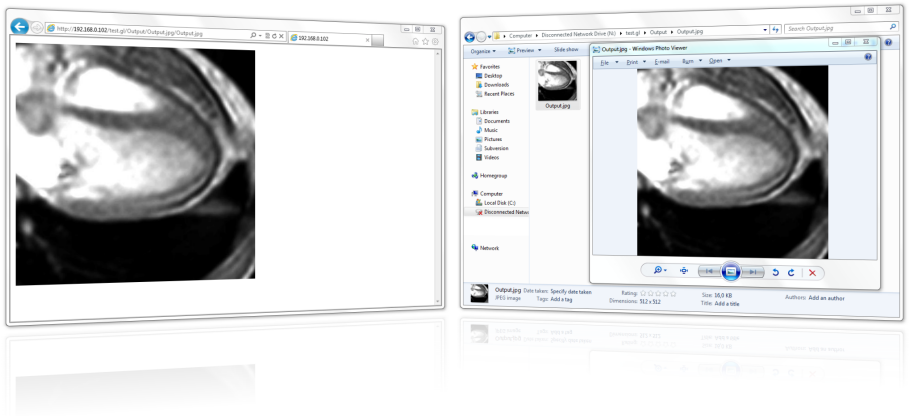


Figure 46: Real-world example using "Windows Internet Explorer" and "Windows Photo Viewer". The image is transparently generated using the GLSL visualization from the implementation on page 54. The "Internet Explorer" access is realized using a simple HTTP interface provided by an Apache server. The "Photo Viewer" access uses direct filesystem access with NFS.

From a technical viewpoint the whole processing pipeline has shown to be simpler to extend than planned. In many cases, extending the filesystem with new processing frontends only requires a few hundred lines of code, for instance, in case of the Java implementation. This is also likely to reveal additional advantages when pursuing other data processing methods like Halide [28] or Julia [29].

6.1.3 Usability issues

Despite the many positive aspects that have been described in the previous section, there were also a few negative aspects. These are no technical difficulties, but rather possible optimizations in the user interaction.

One uncomfortable aspect is the mechanism used to communicate feedback back to the user. Precisely, the problem is more to be found in the lack of such a mechanism. For example when the user specifies some processing code that is syntactically incorrect there is no natural way of signaling error messages back to the user through the filesystem interface. Also, currently there are no standard debugging capabilities that allows the user to gain additional insight into the individual execution steps when the data processing happens.

Additionally, a problem with existing applications has been noticed. Apparently many of these applications do not utilize the filesystem notification capabilities in order to be noticed of file changes. Many image viewing applications do not monitor inotify events. Some react with a noticeable delay because of detecting changes using polling instead. Few even require a manual refresh to actually see the result of changed processing code.

[28] URL http://halide-lang.org
[29] URL http://julialang.org

6.2 Measurement methodology

6.2.1 Gathering of measurement data

Types of profiling and timing data

For the purpose of gathering a reliable impression of the achievable filesystem performance and profiling characteristics, data has been collected from different types of situations.

The first and more general type is the observation as a black-box view of the overall performance characteristics. The aggregated time that is needed in order to complete a certain operation has been observed for important operations like accessing a certain file type or visualization result. In this context these results will not be broken down to different subroutines.

In order to gain a more specific view on the particular operations, a more comprehensive collection of measurement data has been acquired in immediate proximity of certain routines as well. For instance, by breaking down the time needed to access the result of a certain filtering operation to the basic operations: Generation of the corresponding LLVM module, execution of the LLVM JIT compiler and conversion of the data to the requested output format.

The data collected for the latter is stored and made available using a special ***".perf"*** directory, accessible in the filesystem root itself.

Measurement methods

The measurement has been done using the most accurate methods provided by the operating system. Timing results were acquired using the processor time stamp counter (***TSC***). Additionally, CPU usage and memory information has been collected using the Linux ***rusage*** mechanism.

To minimize the effect of the measurement on the runtime behavior, the profiling data is stored in a preallocated data structure. Only upon completion of a certain operation the data is stored in the global "perf" directory. Afterwards, the directory is refreshed with updated variables/files containing the newly acquired measurement results.

In order to not interfere with the usage in a real scenario or the black-box observations, the measurement can be toggled at compile time. This ensures that there is no performance loss due to unnecessary storage operations.

Exact TSC timings

Acquisition of the TSC from the processor must distinguish between x86 and x86_64 architectures. The code to retrieve the current TSC is shown in listing 11.

```
1  __inline__ uint64_t rdtsc(void) {
2    uint32_t lo, hi;
3    __asm__ __volatile__ (
4    "        xorl %%eax,%%eax \n"
5  #ifdef __x86_64__
6    "        push %%rbx\n"
7  #else
8    "        push %%ebx\n"
```

```
#endif
  "        cpuid\n"
#ifdef __x86_64__
  "        pop %%rbx\n"
#else
  "        pop %%ebx\n"
#endif
  :::"%rax", "%rcx", "%rdx");
  __asm__ __volatile__ ("rdtsc" : "=a" (lo), "=d" (hi));
  return (uint64_t) hi << 32 | lo;
}
```

Listing 11: Code to acquire TSC time stamps for the evaluation measurements

In order to minimize the influence of certain processor power saving features on the accuracy of measured timings, the system has been started without sleep states (Fixed C0 state) and frequency changes. Also usage of a tickless kernel has been disabled for this purpose.

```
Command line:
  processor.max_cstate=0
  nohz=off

Fast TSC calibration using PIT
Refined TSC clocksource calibration: 1994.999 MHz.
Switching to clocksource tsc
```

Listing 12: Kernel output to ensure a reliable TSC counter

Measuring the load and CPU usage

In addition to the timing measurements some process data has been collected that contains the cumulated CPU usage (both in user and system modes), as well as, the overall virtual memory usage. The interface of this data is presented in listing 13. It is however not certain if this data will give additional insight into certain questions.

```
SYNOPSIS
  int getrusage(int who, struct rusage *usage);

  RUSAGE_SELF: Return resource usage statistics for the calling process

DATA
  struct rusage {
    struct timeval ru_utime; /* user CPU time used */
    struct timeval ru_stime; /* system CPU time used */
    ...
  };
```

Listing 13: System specific load and system usage measurement method

6.2.2 Example measurements for the profiling data

To demonstrate and test the interpretation of the aforementioned measurement data the example code from listing 14 has been used to obtain some initial measurement data. This example code uses the ***GRAPEFS_PERF_ALL*** macro which is the essential way of storing the current

profiling data in memory. This include process information from */proc* as well as the current TSC value. The exact definitions of this and other available macros are available in the appendix in section F.

```
1   GRAPEFS_PERF_ALL;
2
3   sleep(2);
4
5   GRAPEFS_PERF_ALL;
6
7   timeval startTime;
8   timeval currentTime;
9   gettimeofday(&startTime, nullptr);
10  gettimeofday(&currentTime, nullptr);
11
12  while (((currentTime.tv_sec * 1000.0 + currentTime.tv_usec / 1000.0) -
13    (startTime.tv_sec * 1000.0 + startTime.tv_usec / 1000.0)) < 2000)
14  {
15    gettimeofday(&currentTime, nullptr);
16  }
17
18  GRAPEFS_PERF_ALL;
19
20  void *mem = malloc(10 * 1024 * 1024);
21  memset(mem, 1, 10 * 1024 * 1024);
22
23  GRAPEFS_PERF_ALL;
24
25  free(mem);
26
27  GRAPEFS_PERF_ALL;
```

Listing 14: Code to obtain exemplary profiling measurements

The five GRAPEFS_PERF_ALL operations lead to the following measurement data as shown in listing 15.

```
1   0   0  3624960 22382476233690
2   0   0  3624960 22386466499270 (after sleep)
3  97 102  3624960 22390456648080 (after while-spin)
4  97 102 14114816 22390477584990 (after malloc)
5  97 102  3624960 22390479195540 (after free)
```

Listing 15: Exemplary profiling measurements results

Operation: sleep

Naturally, the sleep operation has no effect on the CPU usage counters, both in user as well as system mode. Also the virtual memory size does not change due to this operation. However, there is a change of 3990265580 in the TSC timer. This concludes from the clock speed of 1994.999. In the time span of two seconds the TSC timer increases by an amount of $2 * 1000 * 1994999 = 3989998000$ ms.

Operation: spinning while-loop

The spinning while loop has another increase of 3990148810 in the TSC timer. Due to the spinning behavior of the loop, the CPU counters have additionally increased by an approximate value of 100. These CPU counters are measured in jiffies. In order to interpret these values, every profiling operation collects the value of ***SC_CLK_TCK*** which in this case is 100. When summing up the CPU time spent in user and system mode an expected average of 200 is observed when spinning for two seconds. As expected the memory measurement has not changed.

Operation: malloc/free

After acquiring and using $10 * 1024 * 1024$ bytes using the malloc operation the memory measurement has increased by expected 10489856 bytes. After releasing the memory with free the virtual memory counter drops to the exact previous value.

6.2.3 Subsequent data processing

To gain meaningful results for the profiling data, multiple iterations have been performed in two nested loops. The outer loop represents a complete restart of the processing operation, therefore avoiding caching of intermediate data or CPU instructions. For instance, within this loop the filesystem is unmounted completely and the input data and processing code is transferred into the filesystem without any prior knowledge.

The inner loop represents the processing operation with the possibility of some caching behavior within the filesystem subsystem or the CPU cache. Within this loop, the data processing is forcefully triggered without transferring the input data or processing code into the filesystem again.

6.3 Performance evaluation

In order to achieve mostly interactive processing speed and interaction, the overall benchmarking timings target a common goal of $\frac{1000}{60} \simeq 16.6$ ms or $\frac{1000}{30} \simeq 33.3$ ms per processed frame. Naturally, this only allows for a very lightweight processing of the data and mostly covers the data passthrough and format conversions. A more complex data processing approach is unlikely to achieve such low processing times per frame.

The analysis of the profiling results has been done using the R language. The R code used for the evaluation can be found in the appendix in section D.

To improve reproducibility of the acquired profiling data and benchmarking results, the evaluation has been performed using recent stable long-term 2.6.32 Red Hat Enterprise Linux (RHEL6) Kernel in a setup similar to [Padala et al., 2007].

6.3.1 Preliminary optimizations

After initial profiling and benchmarking results some preliminary optimizations have been performed in order to improve the significance and overall value of the final profiling results. These optimizations have initially shown to have a considerable impact with respect to the overall result.

Avoiding memory allocations/fragmentation

Having many and small memory operations can slow down the overall behavior noticeable and when done on the heap can also lead to memory fragmentation. Also memory reallocation caused by very many small write operations have shown to cause a considerable slowdown in throughput. The path mapping has been identified as one of the most often called methods. Therefore, heap allocations during the path mapping have been avoided.

Separating white-box and black-box observations

To get exact timing results for certain methods the already mentioned ***.perf*** structure has been established as a compile time option.

When enabled, these additional profiling instructions have at least a measurable impact in overall performance. Therefore, the observation of selected methods has been done separately from the overall benchmarking.

This way, when comparing the filesystem and the conventional approach, the profiling instructions do not interfere with the comparison results. Furthermore, this is no particular problem as the results for selected methods are examined individually.

Increasing throughput using big sector size (big_writes)

A useful capability of recent FUSE versions is the possibility of using considerable big block sizes, not only for read, but also for write operations. This avoids the negative overhead effects of many very small write operations. For the evaluation we have chosen on using a block size of 512 KiB.

6.3.2 Notes about the native and filesystem comparison

Ignoring input throughput/speed

The time needed to transfer the initial input data into the processing pipeline is not considered in the following comparisons. The problem with this particular operation is, that there is no standard value that could be assumed as "correct". Depending on the storage method or device, realistic transfer rates may range from a few 10 MiB/s up to several 100 MiB/s. However, this is less of a problem for comparing both approaches as both of these implementations would be affected by the exact same transfer throughput and latency for the initial data input.

There is a single exception to this. In section 6.3.7 the feasibility of using the filesystem to transparently compress image data on-demand has been evaluation. In this context the achievable speedup has been calculated depending on different transfer rates. Therefore, several common transfer rates have been considered, ranging from slow internet connection up to fast access using local storage.

Parser/Memory consistency between native and filesystem interfaces

In theory, the process of handling the image formats can be done differently for both approaches. For example, ***FreeImage*** offers a way of parsing the image based on a file path. For the filesys-

tem implementation this is not possible as the data in only available in memory and need to be read from there. To avoid variations in the evaluation due to different parsing methods in the ***FreeImage*** library, handling of the image format has been done identically for both approaches. Differences that would have been induced by implementation details in these format handling methods should not be considered for the evaluation of both interfaces.

6.3.3 Raw data throughput

For an impression of the overall throughput and performance of the realized implementation an iozone benchmark has been performed. This is the same benchmark as done for the initial filesystem library decision.

```
> perftest.grapefs.j1 <- read.table("Benchmarks/perftest.grapefs.j1.csv", sep=",")
> perftest.grapefs.j1 / 1024
```

```
                     x
Initial write   479.7358
Rewrite         326.8572
Read           1090.1213
Re-read        1081.2109
Reverse Read    931.8533
Stride read    1021.4606
Random read    1009.0621
Mixed workload  998.7510
Random write    337.7044
Pwrite          485.0719
Pread          1090.8105
Fwrite          290.8584
Fread          1083.7395
```

Measurement 1: Raw GrapeFS data throughput

These initial iozone results show that the ***GrapeFS*** implementation can thoroughly compete in terms of performance. There are no obvious input/output bottlenecks for certain operations. It should be noted that these results are not comparable with the results of the passthrough FUSE filesystem. This is due to the usage of the ***big_writes*** mount options, combined with a suitable block size. For the passthrough filesystem a common 4k block size has been used.

6.3.4 Format and data encoding

The initial paragraphs will take a separate look at the time that is needed for the transfer of the input and output of data. This basically covers the time needed to transfer existing data into the processing pipeline and retrieve the results from the processing pipeline using another application. To get a complete picture, two possible transfer methods have been taken into account. The parsing and encoding of formatted data, as well as, the processing of raw data. It is to be expected that the relative overhead induced by the additional filesystem is higher when processing raw data, especially when raw data is read and only passed through the filesystem without any modification.

Listing 16 presents a schematic of the processing loops that have been used to generate the measurement data used for the first paragraphs.

```
1  OUTER_LOOP
2
3     CREATE FILESYSTEM STRUCTURE, SET INPUT XATTR, WRITE KERNEL.C
4
5     INNER_LOOP
6        SETUP
7           COPY INPUT FILE / READ INPUT FILE + FREEIMAGE DECODE
8        SETUP
9
10       EXECUTE
11          STAT OUTPUT FILE / DIRECTLY EXECUTE C COMPILED FUNCTION
```

```
12        EXECUTE
13
14        TEARDOWN
15           SET OUTPUT XATTR, READ OUTPUT FILE / FREEIMAGE ENCODE
16        TEARDOWN
17     INNER_LOOP
18  OUTER_LOOP
```

Listing 16: Schematic presentation of measuring the black-box results

In the schematic above operations performed by the filesystem benchmarks have been separated by the direct handling using the "/" character. The left side is performed when utilizing the filesystem, the right side when directly handling the data.

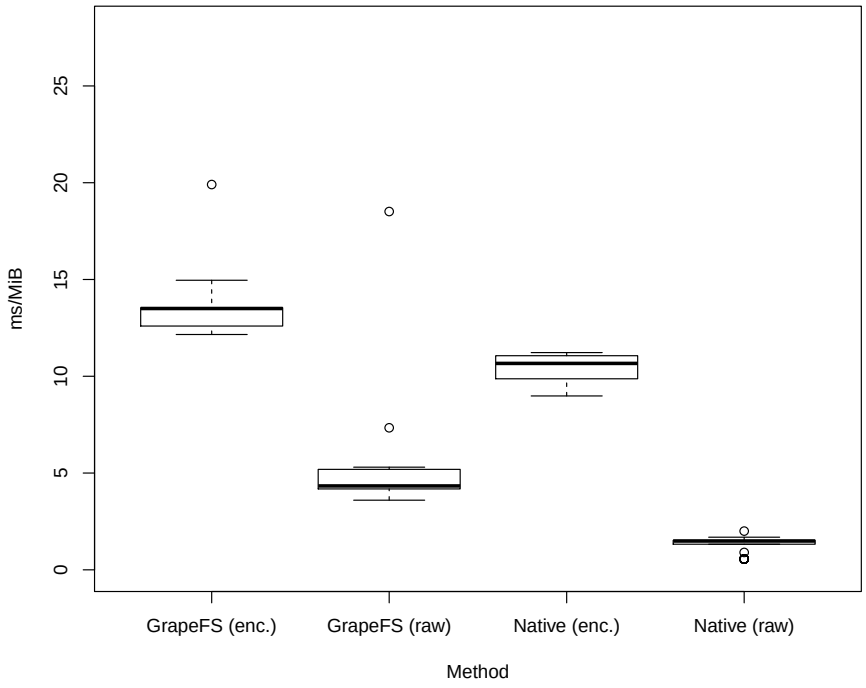
The FreeImage/xattr steps in the schematic are optional. When omitting the format conversion, the measurement result is identified as "raw" in the following paragraphs.

6.3.5 Data input (black-box)

This initial comparison highlights how severe the filesystem overhead of the ***GrapeFS*** implementation is compared to the direct data handling. This context focuses solely on the input of the data. Neither the processing, nor the output, have any influence within this context. This comparison is part of estimating the suitability of the proposed approach for different situations.

The following plot shows the time needed to transfer a single MiB of data into the processing pipeline. The results used for this comparison are the measured time span that has been used upon beginning until completion of the ***SETUP*** operation from the schematic in listing 16.

```
> boxplot(GrapeFS.Perf.Perf$throughput_setup, GrapeFS.Perf_Raw.Perf$throughput_setup,
            Perf.Passthrough.Perf$throughput_setup, Perf.Passthrough_Raw.Perf$throughput_setup,
            names=c('GrapeFS (enc.)', 'GrapeFS (raw)', 'Native (enc.)', 'Native (raw)'),
            xlab='Method', ylab='ms/MiB', ylim=c(0, 28))
```



Measurement 2: GrapeFS data input (encoded and raw)

The timing measurements in the preceding plot are separated into raw and encoded measurements. The "raw" data is measured for the direct transfer of raw data. This means that the comparison is solely based on the transfer overhead without any format conversion. The "encoded" data additionally performs a format conversion using the input data.

This initial profiling timings show that the difference in data input can be considered even smaller than expected.

```
> summary(GrapeFS.Perf.Perf$throughput_setup-Perf.Passthrough.Perf$throughput_setup)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.028   2.547   2.875   3.043   3.208  10.710
> summary(GrapeFS.Perf_Raw.Perf$throughput_setup-Perf.Passthrough_Raw.Perf$throughput_setup)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.652   2.837   3.048   8.305   3.851 256.600
```

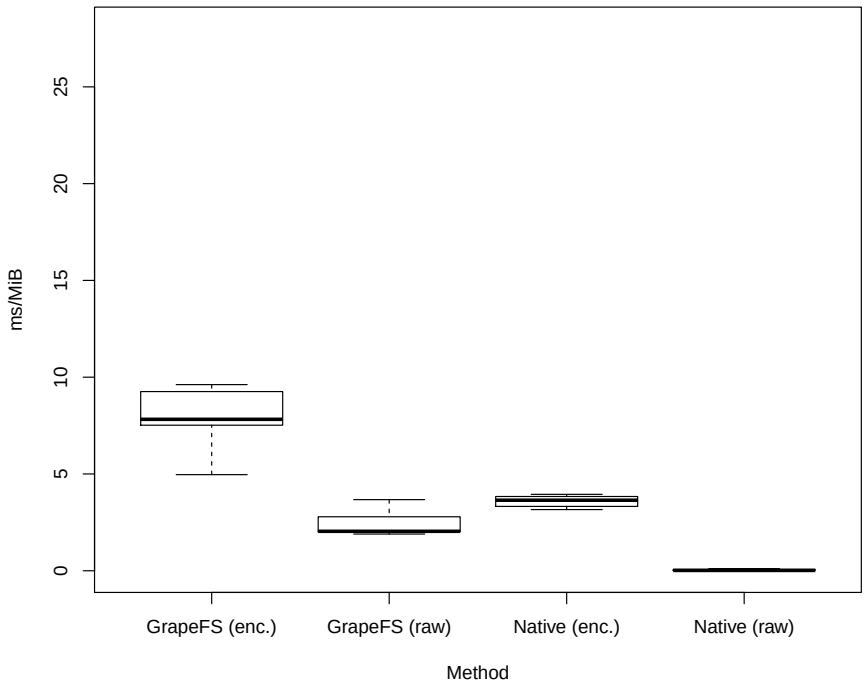
Per MiB of data approximately 3 ms of additional time is required in order to process the input data. The relative difference of this overhead is a bit higher when looking at the raw data without the format parsing. However, considering the aforementioned timing aims, the overhead of the data input for the ***GrapeFS*** implementation is rather negligible.

6.3.6 Result output (black-box)

Comparable with the input data, the following profiling data gives an initial overview of the necessary time to retrieve the result data from the ***GrapeFS*** implementation. The results used for this comparison are the measured time spans that has been used upon beginning of the ***TEARDOWN*** operation until completion.

The context of this operation is as described in the schematic in the initial paragraph. The reasons for this comparison are nearly identical to the previous one. In fact, both are used in combined form within the next comparison.

```
> boxplot(GrapeFS.Perf.Perf$throughput_teardown, GrapeFS.Perf_Raw.Perf$throughput_teardown,
            Perf.Passthrough.Perf$throughput_teardown, Perf.Passthrough_Raw.Perf$throughput_teardown,
            names=c('GrapeFS (enc.)', 'GrapeFS (raw)', 'Native (enc.)', 'Native (raw)'),
            xlab='Method', ylab='ms/MiB', ylim=c(0, 28))
```



Measurement 3: GrapeFS data output (encoded and raw)

Overall, the time needed for this kind of operation is a bit lower. This is to be expected as read operations are generally faster and easier to perform than write operations. This is also visible in the iozone results at the very beginning.

```
> summary(GrapeFS.Perf.Perf$throughput_teardown-Perf.Passthrough.Perf$throughput_teardown)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.224   4.216   4.550   4.685   5.525   5.727

> summary(GrapeFS.Perf_Raw.Perf$throughput_teardown-Perf.Passthrough_Raw.Perf$throughput_teardown)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.790   2.009   2.031   2.214   2.758   3.564
```

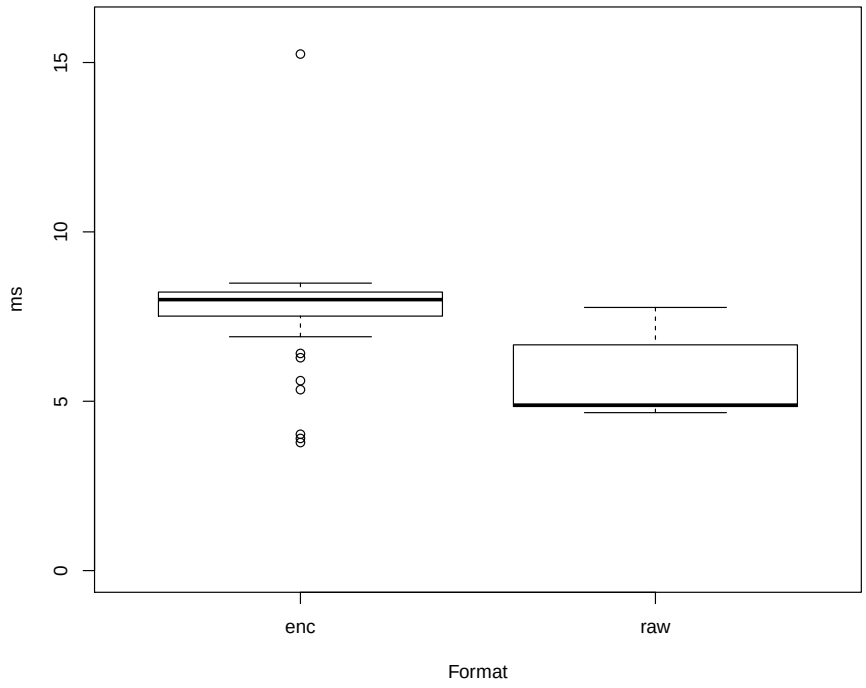
Depending on the necessity of an additional format encoding, the additional time needed when using the ***GrapeFS*** implementation is between 2 and 4 ms per MiB.

Overall, the relative overhead for the data output is higher than the data input. This is to be expected, as the ***teardown*** operation of the conventional implementation does not involve any filesystem operation at all. For both cases (encoded and raw) the data is already available in memory and can be processed further in-place. In comparison, when using the ***GrapeFS*** implementation, the data has to be retrieved from the processing result memory using VFS calls. This is different to the preceding input situation where both implementations need to use VFS calls in order to transfer the input data into memory.

6.3.7 Overall filesystem input and output overhead (black-box)

Based on the previous two paragraphs the data can be consolidated into the overall overhead (additional time in ms) induced by the ***GrapeFS*** implementation.

```
> boxplot(GrapeFS.Overhead, xlab='Format', ylab='ms', ylim=c(0,16))
```



Measurement 4: GrapeFS data throughput (input and output)

```
> summary(GrapeFS.Overhead)
      enc              raw
 Min.   : 3.781   Min.   :  4.663
 1st Qu.: 7.532   1st Qu.:  4.845
 Median : 8.000   Median :  4.886
 Mean   : 7.728   Mean   : 10.519
 3rd Qu.: 8.217   3rd Qu.:  6.661
 Max.   :15.250   Max.   :260.118
```

As expected from the first two paragraphs, the overhead of a complete data passthrough using ***GrapeFS*** can be estimated as about 5-8 ms. There is a small additional overhead of approximately 3 ms when accessing the data as general-purpose image format. This could be due to additional complexity in the object structure within the filesystem implementation when the file format needs to be handled.

The relative ratio of this overhead may vary among the raw and encoded scenarios. However, compared to the usual time needed for most processing operations, the additional time spent for the data transfer is considerably small. Common processing operations probably require a few 100 ms, up to several seconds.

Compression using image formats (white-box)

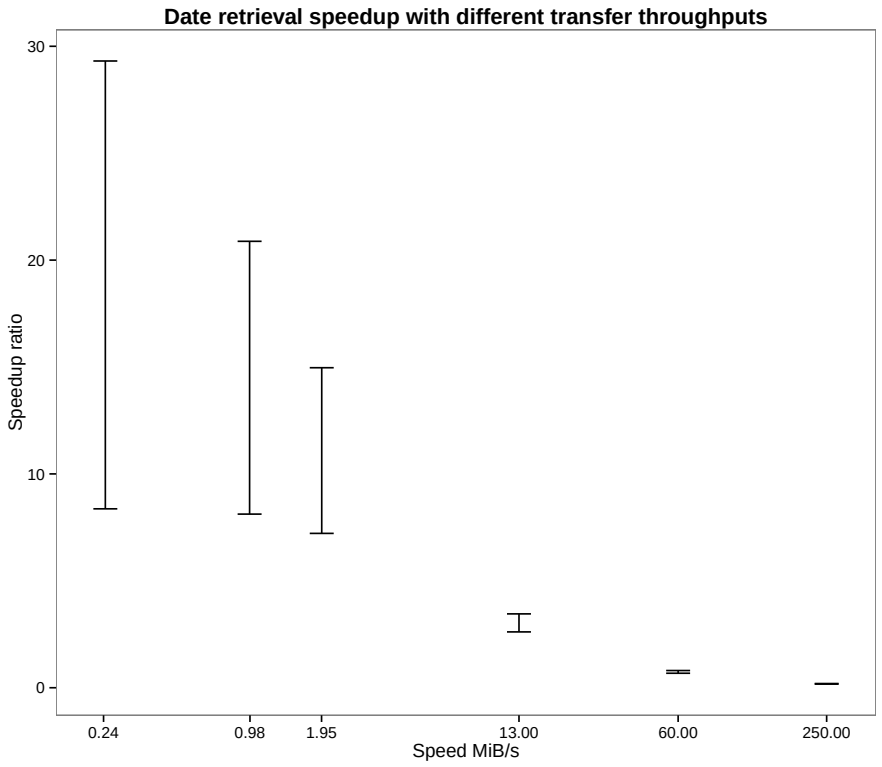
Within this paragraph we will estimate if the overhead induced by the ***GrapeFS*** interface and format handling can cope with the time that is being saved by dynamically compressing data.

For the following comparisons common throughput rates, beginning at a few 100 KiB/s up to multiple 100 MiB/s have been considered. The necessary time needed to transfer rather small imaging data with this throughput is used as the reference data. These rates will be compared with the necessary time to transfer the same, but compressed, image data in combination with the necessary time to transparently compress this data.

The necessary time to actual transfer the raw or compressed data has been calculated artificially and was not measured in a real test situation. Assuming a stable transfer connections, this should cause no noticeable difference, uncertainties from the compression measurements have been propagated to the speedup calculations. However, the necessary times for the compression have been measured in an actual test situation. The visualized range in the comparison indicates the variance in speedup by using different compression qualities (1-100).

```
> df <- data.frame(speed=df_comp_speed, mean=df_comp_mean, sd=df_comp_sd)
> limits <- aes(ymin=mean-sd, ymax=mean+sd)

> ggplot(df, aes(x=speed, y=mean)) +
            scale_x_log10(breaks=round(df_comp_speed, digits=2)) + theme_bw() +
            theme(plot.title=element_text(face="bold")) +
            geom_errorbar(limits, width=0.1) +
            theme(panel.grid.minor = element_blank(), panel.grid.major = element_blank()) +
            xlab("Speed MiB/s") + ylab("Speedup ratio") +
            ggtitle("Date retrieval speedup with different transfer throughputs")
```



Measurement 5: GrapeFS compression overhead

When assuming very high throughput for the data transfer (250 MiB/s) and small file sizes, the additional time that is needed for the format handling naturally slows down the overall transfer nearly by a factor of 5.

However, when assuming a slower connection for the data transfer, like a common 13 MiB/s (about 100 Mbit/s) network connection, the acquired timing data shows that the additional time needed for the data handling can easily cope with the slow transfer speed. Thereby, providing an overall speedup of 2 up to 2.5 for compression ratios with acceptable/high quality. This factor increases even more when assuming slower internet connections or a higher file size.

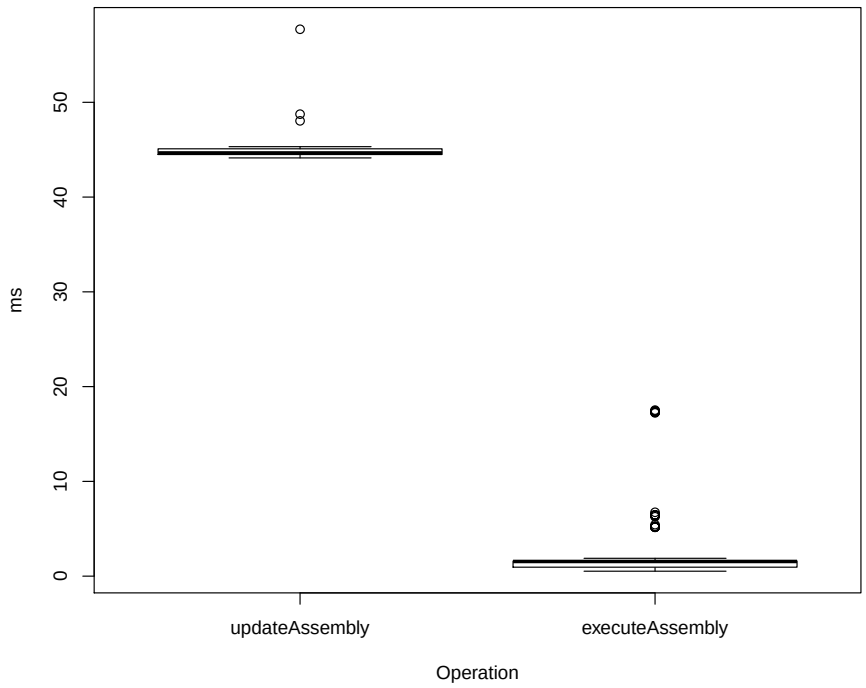
6.3.8 Filtering using JIT-compiled C kernels (white-box)

To get an impression of the design parameters of the data processing using C kernels the following benchmarking data has been acquired.

This paragraph visualizes the profiling results for the ***updateAssembly*** and ***executeAssembly*** operations. The ***updateAssembly*** operation is executed when the processing code has changes. This causes the code to be parsed and translated to LLVM code. Additionally, optimization passes are performed and the code is loaded into a JIT execution engine. The ***executeAssembly*** operation is executed when some parameter or input data has changed. This causes an updated of the processing result using the already available LLVM module. The pure execution should be a lot more lightweight than the complete translation of the source code.

The updateAssembly results are only shown for the sake of completeness. There is no knowledge of interest that can be deduced from these measurements within this context. The executeAssembly results can give some insight into the efficiency of the LLVM implementation used by the ***ClangKernel***. These information do not contain any statement about the filesystem approach in general but about this particular implementation.

```
> boxplot(updateAssembly, executeAssembly,
          names=c('updateAssembly', 'executeAssembly'),
          xlab='Operation', ylab='ms')
```



Measurement 6: GrapeFS profiling data for LLVM operations

```
> summary(updateAssembly)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  44.13   44.49   44.66   45.30   45.09   57.71
```

```
> summary(executeAssembly)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.5214  0.9448  1.5480  2.2540  1.5960 17.5200
```

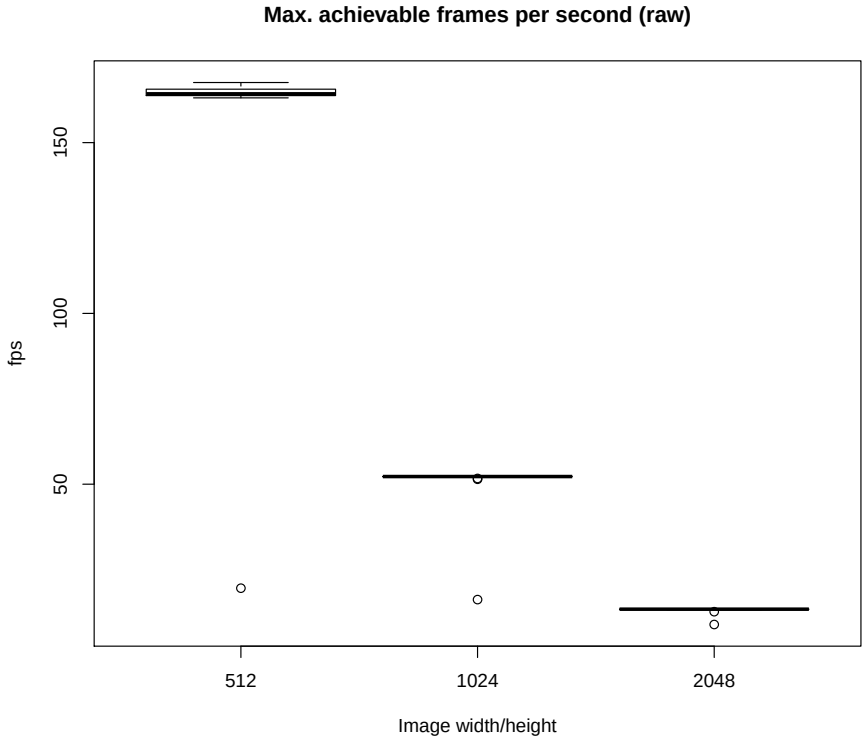
While the ***updateAssembly*** results are comparatively high, this only causes a delay of about 45 ms when the processing code changes. This delay is unique for any change and is therefore completely acceptable.

Also the execution of the LLVM-based processing only induces a delay of approximately 1.5 ms. This is probably higher than the time needed for a simple function call. However, compared to the processing times and the requirements in most situations, this is not likely to cause any troubles. Also this is an implementation detail of the ***GrapeFS*** implementation and not representative for this architecture.

6.3.9 Visualization/Filesystem throughput (black-box)

The following calculations estimate the number of frames per seconds that can be achieved when transferring data of varying size through the ***GrapeFS*** implementation. These estimations are deduced from the data used in the initial analysis of the filesystem overhead but broken down into different file sizes.

```
> boxplot(GrapeFS.Perf_Raw.Perf[!GrapeFS.Perf_Raw.Perf$data_width==256,]$fps ~
            GrapeFS.Perf_Raw.Perf[!GrapeFS.Perf_Raw.Perf$data_width==256,]$data_width,
            xlab='Image width/height', ylab='fps',
            main="Max. achievable frames per second (raw)")
```



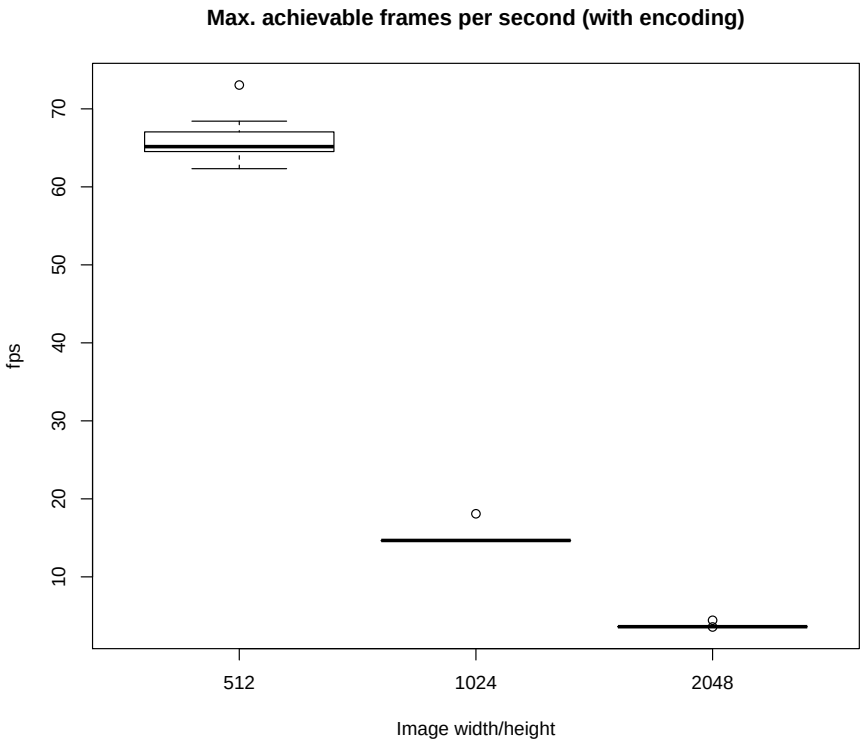
Measurement 7: GrapeFS filesystem throughput (Raw)

```
> summary(GrapeFS.Perf_Raw.Perf[GrapeFS.Perf_Raw.Perf$data_width==512,]$fps)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  19.55  163.80  164.30  155.70  165.40  167.60
> summary(GrapeFS.Perf_Raw.Perf[GrapeFS.Perf_Raw.Perf$data_width==1024,]$fps)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  16.22   52.20   52.24   49.94   52.30   52.45
> summary(GrapeFS.Perf_Raw.Perf[GrapeFS.Perf_Raw.Perf$data_width==2048,]$fps)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  8.924  13.350  13.400  13.080  13.440  13.460
```

Even when considering very big images (2048x2048), roughly 15 frames could be processed per second. This assumes that every image is streamed into the filesystem separately. Looking at common image sizes of 1024x1024 or 512x512 the achievable throughput of 50 or 155 frames

per second is even higher. However, these calculations assume that no format parsing or encoding is necessary.

```
> boxplot(GrapeFS.Perf.Perf[!GrapeFS.Perf.Perf$data_width==256,]$fps ~
            GrapeFS.Perf.Perf[!GrapeFS.Perf.Perf$data_width==256,]$data_width,
            xlab='Image width/height', ylab='fps',
            main="Max. achievable frames per second (with encoding)")
```



Measurement 8: GrapeFS filesystem throughput (Uncompressed TGA format)

```
> summary(GrapeFS.Perf.Perf[GrapeFS.Perf.Perf$data_width==512,]$fps)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  62.33   64.56   65.16   65.84   66.67   73.06
> summary(GrapeFS.Perf.Perf[GrapeFS.Perf.Perf$data_width==1024,]$fps)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  14.56   14.63   14.67   14.87   14.69   18.09
> summary(GrapeFS.Perf.Perf[GrapeFS.Perf.Perf$data_width==2048,]$fps)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  3.575   3.601   3.609   3.658   3.614   4.435
```

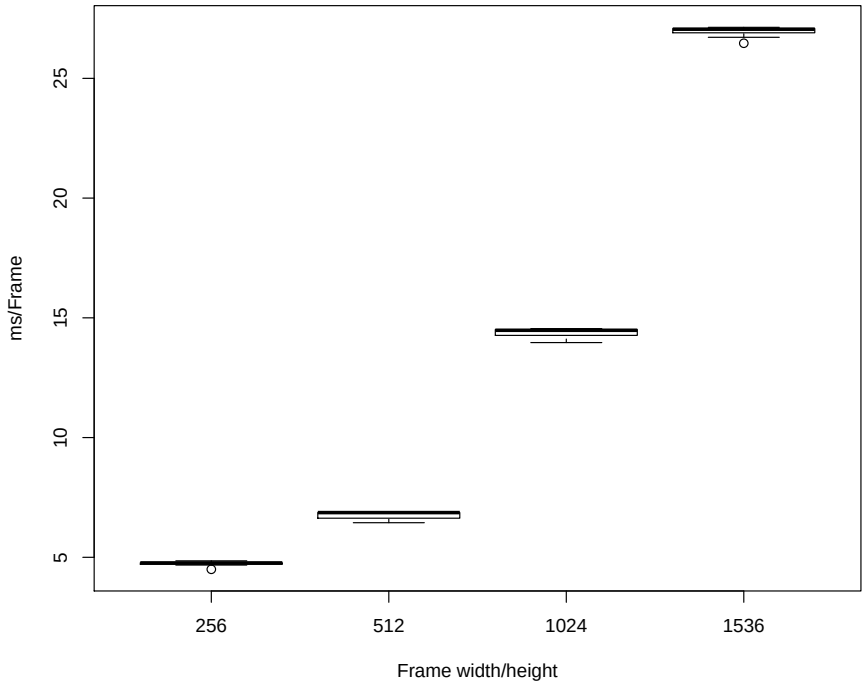
When taking the format conversion, for the input and output data, into account the situation has shifted a bit. However, for instance images with a dimension of 2048x2048 can still be processed at nearly 4 frames per second.

6.3.10 OpenGL/Filesystem throughput (white-box)

This paragraph presents the measurement information from the OpenGL visualization. Specifically, the time that has been used in order to transfer the visualization from the graphics hardware into system memory for further processing. This is the most interesting of these profiling information as it presents the limiting factor when using the GLSL implementation to visualize imaging data.

This process assumes that the input data only needs to be read once. Therefore, only the setup of the rendering and the data transfer into system memory is considered.

```
> boxplot(GrapeFS.Visualization[GrapeFS.Visualization$size>128,]$ms_raw ~
            GrapeFS.Visualization[GrapeFS.Visualization$size>128,]$size,
            xlab='Frame width/height', ylab='ms/Frame')
```



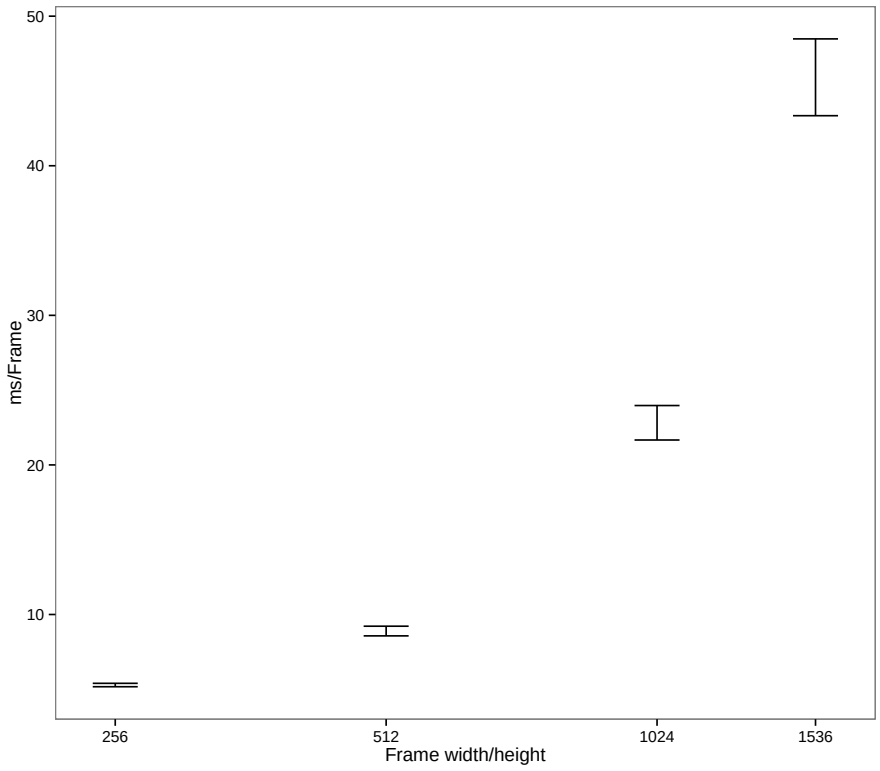
Measurement 9: GrapeFS OpenGL filesystem throughput (Raw)

```
> summary(GrapeFS.Visualization[GrapeFS.Visualization$size==1024,]$ms_raw)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  13.97   14.28   14.48   14.38   14.51   14.55
```

For an assumed visualization size of 1024x1024, the necessary time to provide the output data is approximately 15 ms per frame. This is correlates to approximately 66.6 frames per second. There will be some additional overhead to access the data through the raw output file. However, when assuming raw output data a possible rate of 60 frames per second using the GLSL-based visualization seems feasible.

```
> df <- data.frame(size=df_vis_sizes, mean=df_vis_mean, sd=df_vis_sd)
> limits <- aes(ymin=mean-sd, ymax=mean+sd)
> ggplot(df, aes(x=size, y=mean)) +
            scale_x_log10(breaks=df_vis_sizes) + theme_bw() +
            theme(plot.title=element_text(face="bold")) +
            geom_errorbar(limits, width=0.05) +
            theme(panel.grid.minor = element_blank(), panel.grid.major = element_blank()) +
            xlab("Frame width/height") + ylab("ms/Frame")
```



Measurement 10: GrapeFS OpenGL filesystem throughput (JPEG format)

```
> df_vis_mean[df_vis_sizes==1024] # Mean for size 1024
[1] 22.81735
> df_vis_sd[df_vis_sizes==1024] # Standard deviation for size 1024
[1] 1.152584
```

The above plot is an estimation of the frame times when every rendering result would be accessed using the JPEG format. The compression times in this context have not been measured in an actual situation. Instead a linear estimation of the times needed to compress 3 MiB has been used, which may introduce a smaller bias.

Taking this compression into account, it could still be possible to achieve roughly 43 frames per second for frame sizes of 1024x1024.

6.4 Functional correctness

When developing a product of higher complexity, at some point the interaction between different components may yield unexpected results or simply break at all. Especially for a product where a certain amount of functionality is not used on a regular basis this can cause a variety of serious problems. Up to the situation of measuring the behavior of a system that simply produces wrong output or no realistic output at all.

As the ***GrapeFS*** implementation was not able to mature for several years, the overall development has been based on a test-driven approach. This basically means that all functionality that is required by the preceding analysis, or the measurement process, is verified using one or more test cases.

Usually distinct functionality is verified by a trinity of test cases, described in the following paragraphs.

The first case verifies the functionality by making direct use of the implemented object structure. This can be basically broken down to the manual instantiation of a certain object structure, invocation of a certain operation and assertion of the acquired result.

The second test utilizes the behavior of the implemented filesystem/FUSE interface as if it would be used by an external process. Instead of directly instantiating the objects, the filesystem operations for the creation of folders and files, as well as, the read and write operations for files are used.

The third and last test performs a real world scenario. This means that the filesystem is mounted into the running operating system and utilized using the appropriate file operations. These actions are passed through the actual FUSE implementation. The actual implementation of the execution and assertion is done by making use of automated shell scripts.

These tests have been integrated into the build system using the CTest functionality of CMake. The tests have been implemented using the googletest framework.

The following section gives a brief overview of the particular functionality that is covered by the different test sets. A comprehensive listing of all tests and the result of a complete test run can be found in section B.

- GrapeTypesTests
- GrapeStructureTests
- GrapeComputationTests
- GrapeAttributeTests
- GrapeParserTests
- GrapeFUSEInterfaceTests
- GrapeFUSETests

Listing 17: List of all implemented test sets.

All test sets as a whole contain approximately 82 test cases.

6.4.1 Object structure and interaction

These tests focus on the internal data structure and interaction between objects.

GrapeTypesTests Tests the handling of variables and associated values with these variables. Also the read and write mechanisms for variable handles and the automatic conversion of values to integer and floating point types on write events is tested.

GrapeStructureTests Tests the overall parent and child structure when handling directories with child objects. This also includes the path mapping used to find child directories and variables.

GrapeComputationTests Tests the compilation and execution of computation kernels. C code, as well as OpenGL kernels, are being tested with different combinations and types of arguments and input data.

GrapeAttributeTests Tests the general handling of attributes on objects. This includes setting, retrieving and querying of attributes.

GrapeParserTests Tests the different format parsers by reading input files in different formats. If applicable, also the output of data formats is being tested.

6.4.2 FUSE interface and operation tests

These tests verify the actual filesystem interface.

GrapeFUSEInterfaceTests Tests the FUSE operations that are called by the FUSE library. These tests check the semantics of the FUSE implementation without the need of actually mounting the filesystem. In case of failures, these tests can give valuable information about the origin of an error, especially in conjunction with the FUSE tests described in the next paragraph, where the filesystem is actually mounted. Generally these tests build upon the functionality from the previous tests and ensure that the filesystem semantics work as expected.

GrapeFUSETests These tests actually mount the filesystem and perform different kinds of test cases atop the actual filesystem. All calls are passed through the usual filesystem interface and the FUSE libraries. Every scenario is implemented by a single shell script and run independently from the other tests. Basically these tests cover the same functionality as the pure FUSE interface tests while using the actual filesystem for the tests.

6.4.3 Runtime characteristics

Call analysis

With the utilization of kernel performance counters with OProfile, some of the most critical function calls have been consolidated. The necessary data has been generated during the run of several test scenarios. The OProfile results consist of four columns. The first column contains the absolute event count that has been observed by OProfile. These numbers need to be considered relatively to each other. Special focus is given to the second column. This column contains the relative share of a certain function within the given call trace. This is used to identify the reasons for the importance of certain code paths. The third column contains the location of the symbol which is of no importance here. The last column represents the currently active function. The indentation of the function illustrates the depth of the nested call. The function without indentation is focused. Functions above are the callers that have called this function. Functions underneath are functions that have been called.

The following listings show the call traces that have been observed as the most important paths due to the OProfile analysis.

Path mapping and attributes

```
1    1         0.0594  GrapeFS                 gfs_readlink
2    1         0.0594  GrapeFS                 gfs_mkdir
3    1         0.0594  GrapeFS                 gfs_opendir
4    6         0.3563  GrapeFS                 find_base
5    6         0.3563  GrapeFS                 gfs_access
6    16        0.9501  GrapeFS                 gfs_setxattr
7    17        1.0095  GrapeFS                 gfs_unlink
8    36        2.1378  GrapeFS                 gfs_truncate
9    45        2.6722  GrapeFS                 gfs_utime
10   76        4.5131  GrapeFS                 gfs_open
11   606      35.9857  GrapeFS                 gfs_getxattr
12   873      51.8409  GrapeFS                 gfs_getattr
13 103       6.1128  GrapeFS               GrapeFS::FUSE::mapPath
14   1176     69.0546  libGrapeFS_Core.so      GrapeFS::Directory::mapPath
15   245      14.3864  libstdc++.so.6.0.17     libstdc++.so.6.0.17
16   103       6.0482  GrapeFS                 GrapeFS::FUSE::mapPath [self]
17   53        3.1122  libc-2.16.so            __strlen_sse2_bsf
18   35        2.0552  libGrapeFS_Core.so      map<string, GrapeFS::Object*>::find
19   34        1.9965  libGrapeFS_Core.so      map<string, GrapeFS::Object*>::end
20   21        1.2331  libc-2.16.so            free
```

Listing 18: oprofile call analysis (mapPath)

Listing 18 shows the callers and callees around the mapPath function. This function, in the FUSE class, is called due to a variety of FUSE actions. However, the more important of the callers are the open and getattr/getxattr functions. Subsequently, the most time is spent in mapPath within the Directory class. This function does the actual path mapping. Therefore, in case the reaction time for the filesystem in general needs to be improved even further, this function should be a viable place for additional optimization.

Reading and writing

Beside the path mapping, the read and write operations are among the more important paths.

```
35        2.0772  GrapeFS                 gfs_write
  8726     98.2436  libc-2.16.so            __memcpy_ssse3
  82        0.9232  libGrapeFS_Core.so      GrapeFS::VariableHandle::write
  35        0.3941  GrapeFS                 gfs_write [self]
  17        0.1914  libGrapeFS_Core.so      unsigned long const& max<unsigned long>
  5         0.0563  libc-2.16.so            realloc
  1         0.0113  libGrapeFS_Core.so      GrapeFS::VariableHandle::updateValue
  1         0.0113  libc-2.16.so            malloc
```

Listing 19: oprofile call analysis (gfs_write)

```
16        0.9496  GrapeFS                 gfs_read
  25673    99.6507  libc-2.16.so            __memcpy_ssse3
  34        0.1320  libGrapeFS_Core.so      GrapeFS::VariableHandle::read const
  16        0.0621  GrapeFS                 gfs_read [self]
  14        0.0543  libGrapeFS_Core.so      unsigned long const& min<unsigned long>
  10        0.0388  libc-2.16.so            __memset_sse2
  6         0.0233  libGrapeFS_Core.so      __shared_ptr<char,  const
```

Listing 20: oprofile call analysis (gfs_read)

However, as shown in listings 19 and 20, most of the time is actually accounted for the memory operations. There are no obvious optimization possibilities, except for instance using the ***big_writes*** mount option to reduce call counts and perform larger memory operations.

7 Conclusion

7.1 Successes and problems

The experiences that have been collected during the design, implementation and especially the evaluation within this thesis are being concluded at this point.

The technical feasibility has shown to be more untroubling than expected at the very beginning of the design. The high flexibility of the implementation enables the use of several processing frontends and plenty of different file formats. The collected profiling data and hands-on experiences show clearly that the induced access latency and attainable throughput is unproblematic for most use cases. The only possible exception from this are high-throughput scenarios that naturally avoid filesystem calls by any means.

Even when considering the filesystem and format conversion overhead it is likely possible to achieve about 30 or 60 frames per second in lightweight processing situations. Also remote interaction was possible without any problems at all. Access using a HTTP connection with a common web browser or any Windows program using NFS shares was unproblematic.

The heterogeneous use of different programming languages and frameworks can work in a way that is viable for prototyping and all anticipated non-realtime scenarios. Regarding future possibilities the flexibility and extensibility of the implementation is able to be used for different types of data as well. For example, by adding support for further file formats or other frameworks, the functionality of processing folders and argument files could also be used to process 1D signal data.

✓ Req. #1, #2, #3	The required image processing functionality can be achieved either using custom C,C++ or Java code or by using the OpenCV framework. Hardware accelerated visualization is possible using GLSL shaders. General-purpose images, as well as, DICOM images can be read.
✓ Req. #4	Transparent access to all intermediate processing results is possible from any existing application using the filesystem interface. No further adaptions to these applications are necessary. Applications only need to support a general-purpose image format in order to read the output.
✓ Perf. #1, #2, #3	As shown, the latency induced by the filesystem interface and GrapeFS implementation is in the range of few milliseconds and therefore negligible for the significant interactive scenarios. The achievable throughput of several hundred MiB is more than enough for the considered scenarios.

Table 4: Conclusive evaluation of the required and the GrapeFS functionality.

As presented in table 4 all significant aspects from the initial requirements (sections 3.3.1 and 3.3.2, page 24) have been realized using the provided implementation.

7.2 Availability and licensing

Code repository:	https://github.com/crurik/GrapeFS
License:	2-clause BSD (Simplified/FreeBSD) license

```
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this
   list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.
```

Listing 21: GrapeFS license terms excerpt

7.3 Are we there yet? (Outlook)

While the prototype is already quite mature and usable there are several trends which could expose special potential. The following chapters describe a few of those fields that have been encountered during the research and development process.

7.3.1 User interaction and interfaces

While the interactivity for the prototyping of new algorithms and the overall interactivity was quite inspiring, one actual problem is related to this interaction. Most IDEs are geared towards usage in mostly static projects, involving a very limited set of programming languages. The task of implementing small processing kernels in various languages, while simultaneously incorporating different frameworks or operating on the raw data, can be a bit annoying when there is hardly any tool support at all.

Even though the direct interaction with the filesystem interface is mandatory, an interface that supports the user in directly interacting with the processing code of different folders would be able to accelerate the interaction with algorithms. Especially in a heterogeneous environment with many different programming languages it can be quite time consuming to use different tools to support the development in different languages. This could also be combined with management tools for input data or processing arguments. Implementation as a web interface could be feasible.

For instance, such an interface could display the folders as a flat structure directly next to each other. The processing code from every folder could be visible and modifiable directly, assisted by code highlighting and completion. When modifying any of this processing code the output would be visualized directly, including the impact on all intermediate results. Symbolic links could be visualized by connections within this flat structure. Available imaging data could be written to the input files directly from a repository of available data using drag and drop. Due to the filesystem interface, parallel access from other applications would still be possible.

7.3.2 Platform compatibility, multithreading and persistence

Window support

Due to the FUSE interface compatibility is currently limited to Linux and Mac OS X. The first step to support Windows platforms would be the implementation of the Dokan API. At least from the initial research Dokan seems to be the most promising way of achieving filesystems with user-space support for Windows. Additionally, some other dependencies would have to be solved, for instance GLX dependencies for the OpenGL support.

Multithreading support

The use of multithreading for a production ready filesystem would be desirable. The part that needs the most attention to achieve this is probably the OpenGL implementation which currently assumes a single active context within a single thread. Either the OpenGL context handling needs to be extended with support of being used in multiple threads, or the OpenGL operations need to be restricted to a specific thread when run in a multithreaded environment. Generally, the very same also applies to the JNI Java VM.

Persistence support

Another small issue that would improve actual use of ***GrapeFS*** is support for a persistent structure. In real-world scenarios it could be important to store the current structure and processing code for multiple reasons. Either to support progressive work on a more complex processing scenario, or to assist in distributing a specific filesystem state in collaborative situations.

The filesystem would need to accept a configuration file as startup parameter. This file would be used to store the filesystem structure as XML or JSON structure upon unmounting. When remounting the filesystem, the persistent file could be used to restore the identical filesystem structure.

7.3.3 Integration of language frontends, processing toolkits and DSLs

C, C++, Java and GLSL are sufficiently available programming front-ends to achieve some initial image processing functionality. However, there are other promising ways to gain additional functionality, even outperforming existing processing applications.

Halide

Halide is "a language for image processing and computational photography [...] designed to make it easier to write high-performance image processing code on modern machines." [30]. Halide is basically a domain specific language (DSL) that is embedded into C++ code and utilized by linking to a provided library. LLVM is used to target different platforms [Ragan-Kelley et al., 2012].

This could be used to significantly speed up and simplify the prototyping and structure of new or existing processing algorithms. The effort of embedding and supporting this in ***GrapeFS*** should be relatively small.

7.3.4 Heterogeneous computing and automated pipeline routing

In addition to adding functionality with programming languages, frameworks and libraries there are other ways of providing additional use for other existing hardware and execution models.

GPGPU and computing architectures

Recent research, especially in the field of the LLVM architecture, has ambitions to support translation of intermediate representations to a variety of platforms. This could be used to provide transparent support of utilizing completely different kinds of underlying hardware. For instance transparent utilization of GPU hardware [Rhodin, 2010].

In combination with the previous approach of using different hardware architectures it could also be interesting to implement a way of dynamically balancing these different target architectures. Different processing kernels could target different available hardware. At runtime the

[30] URL http://halide-lang.org

achievable throughput through the processing pipeline would be estimated for different target situations, selecting the most viable combination. However, it is not easy to estimate the advantage of this approach without further research.

A Appendix

References

[ISO, 2011] (2011). Iso/iec 14882:2011.

[Par, 2011] (2011). Setting up a paraview server. http://paraview.org/Wiki/Setting_up_a_ParaView_Server [Last access: 03.10.12].

[Had, 2012] (2012). Mounting hdfs. http://wiki.apache.org/hadoop/MountableHDFS [Last access: 03.10.12].

[Abràmoff et al., 2004] Abràmoff, D. M. D., Magalhães, D. P. J., and Ram, D. S. J. (2004). Image processing with imagej. *Biophotonics International*.

[Anzböck and Dustdar, 2005] Anzböck, R. and Dustdar, S. (2005). Modeling and implementing medical web services. *Data & Knowledge Engineering 55 (2005) 203–236*.

[Association, 2011] Association, N. E. M. (2011). Dicom specification (2011). ftp://medical.nema.org/medical/dicom/2011 [Last access: 08.10.12].

[Bankman, 2009] Bankman, I. H. (2009). *Handbook of Medical Image Processing and Analysis*. Academic Press (Elsevier).

[Bradski and Kaehler, 2008] Bradski, G. and Kaehler, A. (2008). *Learning OpenCV - Computer Vision with the OpenCV library*. O'Reilly.

[Caban et al., 2007] Caban, J. J., Joshi, A., , and Nagy, P. (2007). Rapid development of medical imaging tools with open-source libraries.

[CodiceSoftware, 2012] CodiceSoftware (2012). Plastic scm - glassfs: plastic made transparent. http://codicesoftware.blogspot.com/2012/07/glassfs-plastic-made-transparent.html [Last access: 01.10.12].

[Dean and Ghemawat, 2004] Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. Technical report, Google, Inc.

[Engel et al., 2000] Engel, K., Sommer, O., and Ertl, T. (2000). A framework for interactive hardware accelerated remote 3d-visualization. Technical report, University of Stuttgart, IfI, Visualization and Interactive Systems Group.

[Erickson, 2002] Erickson, B. J. (2002). Irreversible compression of medical images. *Journal of Digital Imaging*, Vol 15, No 1.

[Ferreira and Rasband, 2012] Ferreira, T. and Rasband, W. (2012). Imagej user guide (1.46r).

[Galatsanos et al., 2003] Galatsanos, N. P., Segall, C. A., and Katsaggelos, A. K. (2003). Digital image enhancement. *Encyclopedia of Optical Engineering*.

[Galloway et al., 2009] Galloway, A., Lüttgen, G., Mühlberg, J. T., and Siminiceanu, R. I. (2009). Model-checking the linux virtual file system. *10th International Conference*, VMCAI 2009.

[Gamma et al., 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[Garlick, 2011] Garlick, J. (2011). 9p2000.l protocol (diod). http://code.google.com/p/diod/wiki/protocol [Last access: 01.10.12].

[Graham et al., 2005] Graham, R., Perriss, R., and Scarsbrook, A. (2005). Dicom demystified: A review of digital file formats and their use in radiological practice. *Clinical Radiology*, 60, 1133–1140.

[Gravel et al., 2004] Gravel, P., Beaudoin, G., and Guise, J. A. D. (2004). A method and for modeling and noise in medical and images. *IEEE TRANSACTIONS ON MEDICAL IMAGING*, VOL. 23, NO. 10.

[Heckel et al., 2009] Heckel, F., Schwier, M., and Peitgen, H.-O. (2009). Object-oriented application development with mevislab and python.

[Heng and Gu, 2005] Heng, Y. and Gu, L. (2005). Gpu-based volume rendering for medical image visualization. *Proceedings of the 2005 IEEE, Engineering in Medicine and Biology 27th Annual Conference*.

[Hensbergen et al., 2009] Hensbergen, E. V., Evans, N., and Stanley-Marbell, P. (2009). Service oriented file systems.

[Hoskins, 2006] Hoskins, M. E. (2006). Sshfs: Super easy file access over ssh. *Linux Journal, Volume 2006 Issue 146*.

[Ibáñez et al., 2005] Ibáñez, L., Schroeder, W., Ng, L., Cates, J., and the Insight Software Consortium (2005). The itk software guide second edition. Technical report.

[Ionkov and Hensbergen,] Ionkov, L. and Hensbergen, E. Xcpu2 distributed seamless desktop extension.

[Jujjuri et al., 2000] Jujjuri, V., Hensbergen, E. V., and Liguori, A. (2000). Virtfs - a virtualization aware file system pass-through.

[Kantee and Crooks, 2007] Kantee, A. and Crooks, A. (2007). Refuse: Userspace fuse reimplementation using puffs.

[Kitware, 2012a] Kitware (2012a). Nlm's insight toolkit (itk) - technical summary. http://www.itk.org/ITK/project/technicalsummary.html [Last access: 01.10.12].

[Kitware, 2012b] Kitware (2012b). Vtk - technical overview. http://www.vtk.org/VTK/project/technical.html [Last access: 02.10.12].

[Koenih et al., 2006] Koenih, M., Spindler, W., Rexilius, J., Jomier, J., Link, F., and Peitgen, H. (2006). Embedding vtk and itk into a visual programming and rapid prototyping platform.

[Laboratory et al., 2012] Laboratory, S. N., Inc, K., and Laboratory, L. A. N. (2012). Paraview - features. http://www.paraview.org/paraview/project/features.html [Last access: 03.10.12].

[Minnich, 2005] Minnich, R. (2005). Why plan 9 is not dead yet and what we can learn from it. Technical report, Advanced Computing Lab, Los Alamos National Lab.

[Mochel, 2005] Mochel, P. (2005). The sysfs filesystem. *Linux Symposium*.

[Ordulu, 2010] Ordulu, N. (2010). A file system for accessing mysql tables as csv files. Master's thesis, Massachusetts Institute of Technology.

[Padala et al., 2007] Padala, P., Zhu, X., Wang, Z., Singhal, S., and Shin, K. G. (2007). Performance evaluation of virtualization technologies for server consolidation. *Enterprise Systems and Software Laboratory, HP Laboratories Palo Alto*.

[Pisano et al., 2000] Pisano, E. D., Cole, E. B., Hemminger, B. M., Yaffe, M. J., Aylward, S. R., Maidment, A. D. A., Johnston, R. E., Williams, M. B., Niklason, L. T., Conant, E. F., Fajardoand, L. L.,

Kopans, D. B., Brown, M. E., and Pizer, S. M. (2000). Image processing and algorithms for digital and mammography: A pictorial essay. *IMAGING & THERAPEUTIC TECHNOLOGY*.

[Quilez, 2008] Quilez, I. (2008). Rendering worlds with two triangles with raytracing on the gpu in 4096 bytes. *NVScene*.

[Ragan-Kelley et al., 2012] Ragan-Kelley, J., Adams, A., Paris, S., Levoy, M., Amarasinghe, S., and Durand, F. (2012). Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics*, 31.

[Rexilius et al., 2006] Rexilius, J., Kuhnigk, J.-M., Hahn, H. K., and Peitgen, H.-O. (2006). An application framework for rapid prototyping of clinically applicable software assistants.

[Rhodin, 2010] Rhodin, H. (2010). A ptx code generator for llvm. Technical report, Saarland University.

[Schroeder et al., 1996] Schroeder, W. J., Martin, K. M., and Lorensen, W. E. (1996). The design and implementation of an object-oriented toolkit for 3d graphics and visualization. Technical report.

[Torvalds, 2011] Torvalds, L. (2011). Linus torvalds on userspace filesystems and fuse. http://article.gmane.org/gmane.linux.kernel/1153038 [Last access: 08.10.12].

[Van et al., 2005] Van, E., Hensbergen, and Minnich, R. (2005). Grave robbers from outer space using 9p2000 under linux. (83).

[Wiggins et al., 2001] Wiggins, R. H., Davidson, H. C., Harnsberger, H. R., Lauman, J. R., and Goede, P. A. (2001). Image file formats: Past, present, and future. *RadioGraphics*, 21, 789-798.

[W.Wallis and Miller, 1990] W.Wallis, J. and Miller, T. R. (1990). Volume rendering in three-dimensional display of spect images. *The Journal of Nuclear Medicine*, Vol. 31.

B Test results (Functional correctness)

```
Running tests...
Test project /home/spx/GrapeFS-thesis
Start  1: GrapeTypesTests.VariableValueSet
1/87 Test  #1: GrapeTypesTests.VariableValueSet
  .. Passed    0.22 sec
Start  2: GrapeTypesTests.VariableValueSetGet
2/87 Test  #2: GrapeTypesTests.VariableValueSetGet
  .. Passed    0.19 sec
Start  3: GrapeTypesTests.VariableValueSetCOWGet
3/87 Test  #3: GrapeTypesTests.VariableValueSetCOWGet
  .. Passed    0.19 sec
Start  4: GrapeTypesTests.VariableHandlerChange
4/87 Test  #4: GrapeTypesTests.VariableHandlerChange
  .. Passed    0.19 sec
Start  5: GrapeTypesTests.TypedVariableBytePositive
5/87 Test  #5: GrapeTypesTests.TypedVariableBytePositive
  .. Passed    0.19 sec
Start  6: GrapeTypesTests.TypedVariableIntegerPositive
6/87 Test  #6: GrapeTypesTests.TypedVariableIntegerPositive
  .. Passed    0.19 sec
Start  7: GrapeTypesTests.TypedVariableIntegerNegative
7/87 Test  #7: GrapeTypesTests.TypedVariableIntegerNegative
  .. Passed    0.20 sec
Start  8: GrapeTypesTests.TypedVariableFloatingPositive
8/87 Test  #8: GrapeTypesTests.TypedVariableFloatingPositive
  .. Passed    0.19 sec
Start  9: GrapeTypesTests.TypedVariableFloatingNegative
9/87 Test  #9: GrapeTypesTests.TypedVariableFloatingNegative
  .. Passed    0.19 sec
Start 10: GrapeTypesTests
10/87 Test #10: GrapeTypesTests
  .. Passed    0.20 sec
Start 11: GrapeStructureTests.TestMappingNegative
11/87 Test #11: GrapeStructureTests.TestMappingNegative
  .. Passed    0.20 sec
Start 12: GrapeStructureTests.TestNodeChildrenEmpty
12/87 Test #12: GrapeStructureTests.TestNodeChildrenEmpty
  .. Passed    0.19 sec
Start 13: GrapeStructureTests.TestKernelChildren
13/87 Test #13: GrapeStructureTests.TestKernelChildren
  .. Passed    0.19 sec
Start 14: GrapeStructureTests.TestNodeAddChild
14/87 Test #14: GrapeStructureTests.TestNodeAddChild
  .. Passed    0.19 sec
Start 15: GrapeStructureTests.TestNodeAddRemoveChild
15/87 Test #15: GrapeStructureTests.TestNodeAddRemoveChild
  .. Passed    0.20 sec
Start 16: GrapeStructureTests.TestMappingNode
16/87 Test #16: GrapeStructureTests.TestMappingNode
  .. Passed    0.19 sec
Start 17: GrapeStructureTests.TestMappingVariable
17/87 Test #17: GrapeStructureTests.TestMappingVariable
  .. Passed    0.19 sec
Start 18: GrapeStructureTests.TestMappingNodeChild
18/87 Test #18: GrapeStructureTests.TestMappingNodeChild
  .. Passed    0.19 sec
Start 19: GrapeStructureTests
19/87 Test #19: GrapeStructureTests
  .. Passed    0.19 sec
```

```
Start 20: GrapeComputationTests.NodeComputationSource
20/87 Test #20: GrapeComputationTests.NodeComputationSource
  .. Passed    0.26 sec
Start 21: GrapeComputationTests.NodeComputationArgument
21/87 Test #21: GrapeComputationTests.NodeComputationArgument
  .. Passed    0.25 sec
Start 22: GrapeComputationTests.NodeComputationArguments
22/87 Test #22: GrapeComputationTests.NodeComputationArguments
  .. Passed    0.25 sec
Start 23: GrapeComputationTests.NodeComputationOutputRaw
23/87 Test #23: GrapeComputationTests.NodeComputationOutputRaw
  .. Passed    0.25 sec
Start 24: GrapeComputationTests.NodeComputationOutputImage
24/87 Test #24: GrapeComputationTests.NodeComputationOutputImage
  .. Passed    0.25 sec
Start 25: GrapeComputationTests.NodeComputationOpenGL
25/87 Test #25: GrapeComputationTests.NodeComputationOpenGL
  .. Passed    0.38 sec
Start 26: GrapeComputationTests
26/87 Test #26: GrapeComputationTests
  .. Passed    0.58 sec
Start 27: GrapeFUSEInterfaceTests.access_root
27/87 Test #27: GrapeFUSEInterfaceTests.access_root
  .. Passed    0.20 sec
Start 28: GrapeFUSEInterfaceTests.find_base
28/87 Test #28: GrapeFUSEInterfaceTests.find_base
  .. Passed    0.20 sec
Start 29: GrapeFUSEInterfaceTests.readdir_empty
29/87 Test #29: GrapeFUSEInterfaceTests.readdir_empty
  .. Passed    0.20 sec
Start 30: GrapeFUSEInterfaceTests.readdir_output_raw
30/87 Test #30: GrapeFUSEInterfaceTests.readdir_output_raw
  .. Passed    0.25 sec
Start 31: GrapeFUSEInterfaceTests.readdir_output_image
31/87 Test #31: GrapeFUSEInterfaceTests.readdir_output_image
  .. Passed    0.25 sec
Start 32: GrapeFUSEInterfaceTests.getattr_dir
32/87 Test #32: GrapeFUSEInterfaceTests.getattr_dir
  .. Passed    0.20 sec
Start 33: GrapeFUSEInterfaceTests.getattr_file
33/87 Test #33: GrapeFUSEInterfaceTests.getattr_file
  .. Passed    0.20 sec
Start 34: GrapeFUSEInterfaceTests.getattr_file_size
34/87 Test #34: GrapeFUSEInterfaceTests.getattr_file_size
  .. Passed    0.20 sec
Start 35: GrapeFUSEInterfaceTests.mkdir
35/87 Test #35: GrapeFUSEInterfaceTests.mkdir
  .. Passed    0.20 sec
Start 36: GrapeFUSEInterfaceTests.mkdir_rmdir
36/87 Test #36: GrapeFUSEInterfaceTests.mkdir_rmdir
  .. Passed    0.19 sec
Start 37: GrapeFUSEInterfaceTests.create
37/87 Test #37: GrapeFUSEInterfaceTests.create
  .. Passed    0.20 sec
Start 38: GrapeFUSEInterfaceTests.open_read_overflow
38/87 Test #38: GrapeFUSEInterfaceTests.open_read_overflow
  .. Passed    0.20 sec
Start 39: GrapeFUSEInterfaceTests.open_read_overflow_partial
39/87 Test #39: GrapeFUSEInterfaceTests.open_read_overflow_partial
  .. Passed    0.19 sec
Start 40: GrapeFUSEInterfaceTests.open_read_release
40/87 Test #40: GrapeFUSEInterfaceTests.open_read_release
  .. Passed    0.20 sec
```

```
Start 41: GrapeFUSEInterfaceTests.open_write_release
41/87 Test #41: GrapeFUSEInterfaceTests.open_write_release
  .. Passed    0.20 sec
Start 42: GrapeFUSEInterfaceTests.open_rw_release
42/87 Test #42: GrapeFUSEInterfaceTests.open_rw_release
  .. Passed    0.19 sec
Start 43: GrapeFUSEInterfaceTests.open_write_release_open_read_release
43/87 Test #43: GrapeFUSEInterfaceTests.open_write_release_open_read_release
  .. Passed    0.20 sec
Start 44: GrapeFUSEInterfaceTests.open_read_offset
44/87 Test #44: GrapeFUSEInterfaceTests.open_read_offset
  .. Passed    0.19 sec
Start 45: GrapeFUSEInterfaceTests.open_write_offset
45/87 Test #45: GrapeFUSEInterfaceTests.open_write_offset
  .. Passed    0.19 sec
Start 46: GrapeFUSEInterfaceTests.create_unlink
46/87 Test #46: GrapeFUSEInterfaceTests.create_unlink
  .. Passed    0.20 sec
Start 47: GrapeFUSEInterfaceTests.create_rename
47/87 Test #47: GrapeFUSEInterfaceTests.create_rename
  .. Passed    0.20 sec
Start 48: GrapeFUSEInterfaceTests.create_rename_change_parent
48/87 Test #48: GrapeFUSEInterfaceTests.create_rename_change_parent
  .. Passed    0.20 sec
Start 49: GrapeFUSEInterfaceTests.computation
49/87 Test #49: GrapeFUSEInterfaceTests.computation
  .. Passed    0.24 sec
Start 50: GrapeFUSEInterfaceTests.computation_opengl
50/87 Test #50: GrapeFUSEInterfaceTests.computation_opengl
  .. Passed    0.33 sec
Start 51: GrapeFUSEInterfaceTests.computation_readdir_variable
51/87 Test #51: GrapeFUSEInterfaceTests.computation_readdir_variable
  .. Passed    0.25 sec
Start 52: GrapeFUSEInterfaceTests.computation_variable
52/87 Test #52: GrapeFUSEInterfaceTests.computation_variable
  .. Passed    0.26 sec
Start 53: GrapeFUSEInterfaceTests.computation_variable_change
53/87 Test #53: GrapeFUSEInterfaceTests.computation_variable_change
  .. Passed    0.25 sec
Start 54: GrapeFUSEInterfaceTests.computation_variable_arguments
54/87 Test #54: GrapeFUSEInterfaceTests.computation_variable_arguments
  .. Passed    0.25 sec
Start 55: GrapeFUSEInterfaceTests.computation_symlink
55/87 Test #55: GrapeFUSEInterfaceTests.computation_symlink
  .. Passed    1.25 sec
Start 56: GrapeFUSEInterfaceTests.computation_child
56/87 Test #56: GrapeFUSEInterfaceTests.computation_child
  .. Passed    1.30 sec
Start 57: GrapeFUSEInterfaceTests.computation_child_opengl
57/87 Test #57: GrapeFUSEInterfaceTests.computation_child_opengl
  .. Passed    1.38 sec
Start 58: GrapeFUSEInterfaceTests.computation_child_change
58/87 Test #58: GrapeFUSEInterfaceTests.computation_child_change
  .. Passed    1.30 sec
Start 59: GrapeFUSEInterfaceTests.computation_parser_image
59/87 Test #59: GrapeFUSEInterfaceTests.computation_parser_image
  .. Passed    0.25 sec
Start 60: GrapeFUSEInterfaceTests.getxattr
60/87 Test #60: GrapeFUSEInterfaceTests.getxattr
  .. Passed    0.20 sec
Start 61: GrapeFUSEInterfaceTests.setxattr
61/87 Test #61: GrapeFUSEInterfaceTests.setxattr
  .. Passed    0.20 sec
```

```
Start 62: GrapeFUSEInterfaceTests.listxattr
62/87 Test #62: GrapeFUSEInterfaceTests.listxattr
  .. Passed    0.20 sec
Start 63: GrapeFUSEInterfaceTests.removexattr
63/87 Test #63: GrapeFUSEInterfaceTests.removexattr
  .. Passed    0.20 sec
Start 64: GrapeFUSEInterfaceTests
64/87 Test #64: GrapeFUSEInterfaceTests
  .. Passed    5.08 sec
Start 65: GrapeAttributeTests.set_get_attribute
65/87 Test #65: GrapeAttributeTests.set_get_attribute
  .. Passed    0.20 sec
Start 66: GrapeAttributeTests.list_attributes
66/87 Test #66: GrapeAttributeTests.list_attributes
  .. Passed    0.19 sec
Start 67: GrapeAttributeTests.remove_attribute
67/87 Test #67: GrapeAttributeTests.remove_attribute
  .. Passed    0.20 sec
Start 68: GrapeAttributeTests
68/87 Test #68: GrapeAttributeTests
  .. Passed    0.20 sec
Start 69: GrapeParserTests.image_pgm
69/87 Test #69: GrapeParserTests.image_pgm
  .. Passed    0.20 sec
Start 70: GrapeParserTests.object_attribute
70/87 Test #70: GrapeParserTests.object_attribute
  .. Passed    0.19 sec
Start 71: GrapeParserTests.object_attribute_change
71/87 Test #71: GrapeParserTests.object_attribute_change
  .. Passed    0.19 sec
Start 72: GrapeParserTests.object_attribute_toggle
72/87 Test #72: GrapeParserTests.object_attribute_toggle
  .. Passed    0.19 sec
Start 73: GrapeParserTests
73/87 Test #73: GrapeParserTests
  .. Passed    0.19 sec
Start 74: GrapeFUSETests.mount_umount
74/87 Test #74: GrapeFUSETests.mount_umount
  .. Passed    0.23 sec
Start 75: GrapeFUSETests.file
75/87 Test #75: GrapeFUSETests.file
  .. Passed    0.27 sec
Start 76: GrapeFUSETests.directory
76/87 Test #76: GrapeFUSETests.directory
  .. Passed    0.25 sec
Start 77: GrapeFUSETests.xattr
77/87 Test #77: GrapeFUSETests.xattr
  .. Passed    0.26 sec
Start 78: GrapeFUSETests.parser_image
78/87 Test #78: GrapeFUSETests.parser_image
  .. Passed    0.35 sec
Start 79: GrapeFUSETests.parser_image_compression
79/87 Test #79: GrapeFUSETests.parser_image_compression
  .. Passed    0.37 sec
Start 80: GrapeFUSETests.computation_source
80/87 Test #80: GrapeFUSETests.computation_source
  .. Passed    0.29 sec
Start 81: GrapeFUSETests.computation_opengl
81/87 Test #81: GrapeFUSETests.computation_opengl
  .. Passed    0.36 sec
Start 82: GrapeFUSETests.computation_arguments
82/87 Test #82: GrapeFUSETests.computation_arguments
  .. Passed    0.36 sec
```

```
Start 83: GrapeFUSETests.computation_symlink_absolute
83/87 Test #83: GrapeFUSETests.computation_symlink_absolute
  .. Passed    2.39 sec
Start 84: GrapeFUSETests.computation_symlink_relative
84/87 Test #84: GrapeFUSETests.computation_symlink_relative
  .. Passed    2.38 sec
Start 85: GrapeFUSETests.computation_child
85/87 Test #85: GrapeFUSETests.computation_child
  .. Passed    1.36 sec
Start 86: GrapeFUSETests.computation_child_opengl
86/87 Test #86: GrapeFUSETests.computation_child_opengl
  .. Passed    1.43 sec
Start 87: GrapeFUSETests
87/87 Test #87: GrapeFUSETests
  .. Passed   10.25 sec

100% tests passed, 0 tests failed out of 87

Total Test time (real) =  45.63 sec
```

Listing 22: CTest execution results

C Throughput measurements of selected filesystem interfaces

```
> perftest.tmpfs.j1 <- read.table("Benchmarks/perftest.tmpfs.j1.csv", sep=",")
> perftest.tmpfs.j1 / 1024
                        x
 Initial write   1064.056
 Rewrite         1060.333
 Read            1200.235
 Re-read         2700.640
 Reverse Read    1950.646
 Stride read     2833.145
 Random read     2449.362
 Mixed workload  1430.526
 Random write    1421.618
 Pwrite          3068.203
 Pread           1657.569
 Fwrite          3193.481
 Fread           1765.518
```

Measurement 11: Raw tmpfs throughput

```
> perftest.fuse.j1 <- read.table("Benchmarks/perftest.fuse.j1.csv", sep=",")
> perftest.fuse.j1 / 1024
                        x
 Initial write    130.2664
 Rewrite          469.8564
 Read             789.5089
 Re-read          884.1514
 Reverse Read     381.9043
 Stride read      788.0487
 Random read      353.8197
 Mixed workload   225.9207
 Random write     215.7021
 Pwrite           522.8767
 Pread            227.4655
 Fwrite          1078.9347
 Fread            226.6198
```

Measurement 12: Raw FUSE throughput

```
> perftest.9p.j1 <- read.table("Benchmarks/perftest.9p.j1.csv", sep=",")
> perftest.9p.j1 / 1024
                        x
 Initial write   112.17252
 Rewrite         107.74214
 Read             87.50192
 Re-read         126.34160
 Reverse Read    127.11728
 Stride read     112.77755
 Random read     141.14773
 Mixed workload  123.94226
 Random write    124.92919
 Pwrite          114.00226
 Pread           110.57054
 Fwrite          150.87754
 Fread           126.52247
```

Measurement 13: Raw 9p throughput

D R profiling/measurement processing code

The following code has been used in order to process the raw profiling data into the results used in the evaluation.

```
library(reshape)
library(ggplot2)
library(qpcR)

.wd <- getwd()

process_tsc <- function(x, var_begin, var_end) {
  return((x[var_end] - x[var_begin]) / 1994999);    # ms
}

process_mem <- function(x, var_begin, var_end) {
  return((x[var_end] - x[var_begin]) / 1024);       # KiB
}

process_time <- function(x, var_begin, var_end, sc_clk_tck) {
  return((x[var_end] - x[var_begin]) * 1000 / x[sc_clk_tck]); # ms
}

opplot <- function(prefixes, variables, operation, bpp) {
  res <- data.frame()

  for (i in 1:length(prefixes)) {
    res <- rbind(res, data.frame(operation=paste(prefixes[i], get(variables[i])[,'
        operation'], sep='.'), data=get(variables[i])[,paste('result', operation, 'tsc
        ', sep='_')]/(get(variables[i])$data_width*get(variables[i])$data_height*(bpp
        /8)/1024)))
  }

  boxplot(res$data*1024 ~ res$operation, xlab="Op.", ylab="ms/MiB", ylim=c(0, 100))
}

timeplot <- function(names, variables, operations, title) {
  tasks <- c()
  time_begin <- c()
  time_end <- c()

  for (i in 1:length(names)) {
    t <- get(variables[i])
    tpos <- 0

    for (op in operations) {
      tasks <- c(tasks, paste(names[i], op, sep='.'))

      time_begin <- c(time_begin, tpos)

      tpos <- tpos + mean(t[op][,1])
      time_end <- c(time_end, tpos)
    }
  }

  dfr <- data.frame(
    name        = factor(tasks, levels = tasks),
    start.time  = time_begin,
    end.time    = time_end
  )
  mdfr <- melt(dfr, measure.vars = c("start.time", "end.time"))
```

```
54    ggplot(mdfr, aes(value, name)) +
55      geom_line(size = 6) +
56      xlab("ms") + ylab("Op.") + ggtitle(title) +
57      theme_bw()
58  }
59
60  attrs <- c('cpu_sys', 'cpu_user', 'mem', 'tsc')
61
62  result_dirs <- c('~/GrapeFS-release/Testing/Evaluation/Results', '~/GrapeFS-eval-
        release/Testing/Evaluation/Results')
63
64  for (resdir in result_dirs) {
65    setwd(resdir)
66
67    eval_tests <- list.dirs()
68
69    detailEnv <- new.env(hash=TRUE)
70
71    for (i in 2:length(eval_tests)) {
72      files <- list.files(eval_tests[i], full.names=TRUE)
73
74      for (f in files) {
75        vname <- sub('/', '.', substr(f, 3, nchar(f)))
76
77        vobj <- read.table(f, header=T, quote="\"")
78
79        for (mem in colnames(vobj)) {
80          if (substr(mem, 0, 7) == 'before_') {
81            meth <- sub('_.*', '', substring(mem, 8))
82
83            for (attr in attrs) {
84              begin_var <- paste('before', meth, attr, sep='_')
85              end_var <- paste('after', meth, attr, sep='_')
86              result_var <- paste('result', meth, attr, sep='_')
87
88              if ((begin_var %in% colnames(vobj)) && (end_var %in% colnames(vobj))) {
89                if ((attr == 'cpu_sys') || (attr == 'cpu_user')) {
90                  vobj[result_var] <- process_time(vobj, begin_var, end_var, 'SC_CLK_TCK
                        ')
91                }
92                else if (attr == 'mem') {
93                  vobj[result_var] <- process_mem(vobj, begin_var, end_var)
94                }
95                else if (attr == 'tsc') {
96                  vobj[result_var] <- process_tsc(vobj, begin_var, end_var)
97
98                  parts <- unlist(strsplit(vname, '.', fixed=TRUE))
99
100                 if (is.null(detailEnv[[paste(parts[1], parts[2], sep='.')]])) {
101                   env_vobj <- c()
102                   env_vobj['name'] <- meth
103                   env_vobj['begin'] <- min(vobj[begin_var])
104                   env_vobj['end'] <- max(vobj[end_var])
105                 }
106
107                 detailEnv[[paste(parts[1], parts[2], sep='.')]]['name'] <- c()
108               }
109               else {
110                 print(paste('Unknown attr', attr))
111               }
112             }
113           }
114         }
```

```
115         }
116
117         assign(vname, vobj)
118       }
119     }
120
121     remove(i)
122     remove(f)
123     remove(files)
124     remove(begin_var)
125     remove(end_var)
126     remove(result_var)
127     remove(vobj)
128     remove(attr)
129     remove(mem)
130     remove(meth)
131     remove(vname)
132     remove(parts)
133     remove(env_vobj)
134     remove(eval_tests)
135   }
136
137   remove(resdir)
138   remove(attrs)
139
140   Perf.Passthrough.Perf$throughput_setup <- Perf.Passthrough.Perf$result_setup_tsc/(Perf
          .Passthrough.Perf$data_width*Perf.Passthrough.Perf$data_height*3)*1024*1024
141   Perf.Passthrough.Perf$throughput_execute <- Perf.Passthrough.Perf$result_execute_tsc/(
          Perf.Passthrough.Perf$data_width*Perf.Passthrough.Perf$data_height*3)*1024*1024
142   Perf.Passthrough.Perf$throughput_teardown <- Perf.Passthrough.Perf$result_teardown_tsc
          /(Perf.Passthrough.Perf$data_width*Perf.Passthrough.Perf$data_height*3)*1024*1024
143   Perf.Passthrough.Perf$throughput <- Perf.Passthrough.Perf$throughput_setup+Perf.
          Passthrough.Perf$throughput_teardown
144
145   Perf.Passthrough_Raw.Perf$throughput_setup <- Perf.Passthrough_Raw.
          Perf$result_setup_tsc/(Perf.Passthrough_Raw.Perf$data_width)*1024*1024
146   Perf.Passthrough_Raw.Perf$throughput_execute <- Perf.Passthrough_Raw.
          Perf$result_execute_tsc/(Perf.Passthrough_Raw.Perf$data_width)*1024*1024
147   Perf.Passthrough_Raw.Perf$throughput_teardown <- Perf.Passthrough_Raw.
          Perf$result_teardown_tsc/(Perf.Passthrough_Raw.Perf$data_width)*1024*1024
148   Perf.Passthrough_Raw.Perf$throughput <- Perf.Passthrough_Raw.Perf$throughput_setup+
          Perf.Passthrough_Raw.Perf$throughput_teardown
149
150   GrapeFS.Perf.Perf$throughput_setup <- GrapeFS.Perf.Perf$result_setup_tsc/(GrapeFS.Perf
          .Perf$data_width*GrapeFS.Perf.Perf$data_height*3)*1024*1024
151   GrapeFS.Perf.Perf$throughput_execute <- GrapeFS.Perf.Perf$result_execute_tsc/(GrapeFS.
          Perf.Perf$data_width*GrapeFS.Perf.Perf$data_height*3)*1024*1024
152   GrapeFS.Perf.Perf$throughput_teardown <- GrapeFS.Perf.Perf$result_teardown_tsc/(
          GrapeFS.Perf.Perf$data_width*GrapeFS.Perf.Perf$data_height*3)*1024*1024
153   GrapeFS.Perf.Perf$throughput <- GrapeFS.Perf.Perf$throughput_setup+GrapeFS.Perf.
          Perf$throughput_teardown
154
155   GrapeFS.Perf_Raw.Perf$throughput_setup <- GrapeFS.Perf_Raw.Perf$result_setup_tsc/(
          GrapeFS.Perf_Raw.Perf$data_width*GrapeFS.Perf_Raw.Perf$data_height*3)*1024*1024
156   GrapeFS.Perf_Raw.Perf$throughput_execute <- GrapeFS.Perf_Raw.Perf$result_execute_tsc/(
          GrapeFS.Perf_Raw.Perf$data_width*GrapeFS.Perf_Raw.Perf$data_height*3)*1024*1024
157   GrapeFS.Perf_Raw.Perf$throughput_teardown <- GrapeFS.Perf_Raw.Perf$result_teardown_tsc
          /(GrapeFS.Perf_Raw.Perf$data_width*GrapeFS.Perf_Raw.Perf$data_height*3)*1024*1024
158   GrapeFS.Perf_Raw.Perf$throughput <- GrapeFS.Perf_Raw.Perf$throughput_setup+GrapeFS.
          Perf_Raw.Perf$throughput_teardown
159
160   GrapeFS.Overhead <- data.frame(enc=GrapeFS.Perf.Perf$throughput-Perf.Passthrough.
          Perf$throughput, raw=GrapeFS.Perf_Raw.Perf$throughput-Perf.Passthrough_Raw.
```

```
      Perf$throughput)

df_comp_speed <- c(250, 60, 13, 16000/8/1024, 8000/8/1024, 2000/8/1024)

df_comp_mean <- c()
df_comp_sd <- c()

for (speed in df_comp_speed) {
  through <- GrapeFS.Perf.Perf$throughput_setup
  comp <- (GrapeFS.Compression.dump$result_size_tsc+GrapeFS.Compression.
      dump$result_convert_tsc+GrapeFS.Compression.dump$result_memory_tsc)+(GrapeFS.
      Compression.dump$output_size/1024/1024)/(speed/1000)

  throughput <- c(mean(through), sd(through))
  compression <- c(mean(comp), sd(comp))

  speed <- c(speed, 0)

  DF <- cbind(speed, throughput, compression)
  EXPR <- expression(((3/(speed/1000))/(3*throughput+compression)))

  res <- propagate(expr=EXPR, data=DF, type="stat", plot=FALSE)

  df_comp_mean <- c(df_comp_mean, res$summary[1,]$Prop)
  df_comp_sd <- c(df_comp_sd, res$summary[2,]$Prop)
}

GrapeFS.Perf.Perf$throughput_time <- GrapeFS.Perf.Perf$result_setup_tsc+GrapeFS.Perf.
    Perf$result_teardown_tsc
GrapeFS.Perf.Perf$fps = 1000/GrapeFS.Perf.Perf$throughput_time

GrapeFS.Perf_Raw.Perf$throughput_time <- GrapeFS.Perf_Raw.Perf$result_setup_tsc+
    GrapeFS.Perf_Raw.Perf$result_teardown_tsc
GrapeFS.Perf_Raw.Perf$fps = 1000/GrapeFS.Perf_Raw.Perf$throughput_time

executeAssembly <- rbind(GrapeFS.Compression.executeAssembly, GrapeFS.Conversion.
    executeAssembly, GrapeFS.Passthrough.executeAssembly, GrapeFS.Perf.executeAssembly
    , GrapeFS.Perf_Raw.executeAssembly, GrapeFS.Reencoding.executeAssembly)
executeAssembly <- executeAssembly$result_arguments_tsc +
    executeAssembly$result_execution_tsc

updateAssembly <- rbind(GrapeFS.Compression.updateAssembly, GrapeFS.Conversion.
    updateAssembly, GrapeFS.Passthrough.updateAssembly, GrapeFS.Perf.updateAssembly,
    GrapeFS.Perf_Raw.updateAssembly, GrapeFS.Reencoding.updateAssembly)
updateAssembly <- updateAssembly$result_parse_tsc + updateAssembly$result_bitcode_tsc
    + updateAssembly$result_jit_tsc + updateAssembly$result_arguments_tsc

GrapeFS.Visualization <- data.frame(ms_raw=GrapeFS.Visualization.
    executeAssembly$result_MVP_tsc+GrapeFS.Visualization.
    executeAssembly$result_readpixels_tsc,MiB_ps_raw=1000/((GrapeFS.Visualization.
    executeAssembly$result_MVP_tsc+GrapeFS.Visualization.
    executeAssembly$result_texturebind_tsc+GrapeFS.Visualization.
    executeAssembly$result_readpixels_tsc)/((GrapeFS.Visualization.
    executeAssembly$width*GrapeFS.Visualization.executeAssembly$height)/1024/1024)),
    MiB=((GrapeFS.Visualization.executeAssembly$width*GrapeFS.Visualization.
    executeAssembly$height)/1024/1024), size=GrapeFS.Visualization.
    executeAssembly$width)

df_vis_sizes <- c(256, 512, 1024, 1536)

df_vis_mean <- c()
df_vis_sd <- c()

```

```
for (size in df_vis_sizes) {
  tmp_executeAssembly <- GrapeFS.Visualization.executeAssembly[GrapeFS.Visualization.
      executeAssembly$width==size,]

  comp <- ((size*size)/1024/1024/3) * ((GrapeFS.Compression.dump$result_size_tsc+
      GrapeFS.Compression.dump$result_convert_tsc+GrapeFS.Compression.
      dump$result_memory_tsc))
  vis <- (tmp_executeAssembly$result_MVP_tsc+
      tmp_executeAssembly$result_texturebind_tsc+
      tmp_executeAssembly$result_readpixels_tsc)

  compression <- c(mean(comp), sd(comp))
  visualization <- c(mean(vis), sd(vis))

  DF <- cbind(visualization, compression)
  EXPR <- expression(visualization+compression)

  res <- propagate(expr=EXPR, data=DF, type="stat", plot=FALSE)

  df_vis_mean <- c(df_vis_mean, res$summary[1,]$Prop)
  df_vis_sd <- c(df_vis_sd, res$summary[2,]$Prop)
}

setwd(.wd)
```

Listing 23: R evaluation code

E Implementation details and instructions

E.1 Build instructions and external dependencies

E.2 Code metrics

```
cd ../Code/GrapeFS
sloccount .
```

```
SLOC Directory SLOC-by-Language (Sorted)
55739   GTest           cpp=42606,sh=10368,python=2765
2517    Testing         cpp=1739,sh=614,ansic=151,java=13
1890    top_dir         cpp=1890
1190    Kernel          cpp=1190
540     FUSE            cpp=540
480     Special         cpp=480
238     Parser          cpp=238
67      Include         ansic=67
66      Install         sh=27,cpp=18,java=12,ansic=9
19      Virtual         cpp=19
0       CMake           (none)

Totals grouped by language (dominant language first):
cpp:          48720 (77.65%)
sh:           11009 (17.55%)
python:        2765 (4.41%)
ansic:          227 (0.36%)
java:            25 (0.04%)

Total Physical Source Lines of Code (SLOC)                = 62,746
Development Effort Estimate, Person-Years (Person-Months) = 15.43 (185.22)
 (Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
Schedule Estimate, Years (Months)                         = 1.52 (18.18)
 (Basic COCOMO model, Months = 2.5 * (person-months**0.38))
Estimated Average Number of Developers (Effort/Schedule)  = 10.19
Total Estimated Cost to Develop                           = $ 2,085,004
 (average salary = $56,286/year, overhead = 2.40).
SLOCCount, Copyright (C) 2001-2004 David A. Wheeler
SLOCCount is Open Source Software/Free Software, licensed under the GNU GPL.
SLOCCount comes with ABSOLUTELY NO WARRANTY, and you are welcome to
redistribute it under certain conditions as specified by the GNU GPL license;
see the documentation for details.
Please credit this data as "generated using David A. Wheeler's 'SLOCCount'."
```

Listing 24: GrapeFS code metrics.

F Selected code extracts

```
const char *kExtensions[] = {
  ".c",
  ".cxx",
  nullptr
};

extern "C" void executor_init()
{
  llvm::InitializeNativeTarget();
  llvm::llvm_start_multithreaded();
}

extern "C" void executor_deinit()
{
}

extern "C" ComputationKernel * executor_create(Directory *parent)
{
  return new ClangKernel(parent);
}

extern "C" const char * executor_extension(int i)
{
  return kExtensions[i];
}

extern "C" const char ** executor_extensions()
{
  return kExtensions;
}

extern "C" void executor_destroy(ComputationKernel *object)
{
  if (object == nullptr)
    return;

  delete object;
}
```

Listing 25: Exported interface of ComputationKernel implementations

```
const char *pExtensions[] = {
  "jpg",
  "png",
  "pgm",
  "tga",
  nullptr
};

extern "C" DataFormat * create_parser()
{
  FreeImage_Initialise();

  return new DataParser;
}

extern "C" void destroy_parser(DataFormat *object)
```

```
17  {
18    delete object;
19
20    FreeImage_DeInitialise();
21  }
```

Listing 26: Exported interface of DataFormat implementations

```
1   [...]
2
3   typedef struct _PerformanceOperation
4   {
5     [...]
6
7     void FillProc()
8     {
9       struct proc_t pusage;
10      look_up_our_self(&pusage);
11
12      values.push_back(PerformanceValue(GrapeFS::ULongLong, pusage.stime));
13      values.push_back(PerformanceValue(GrapeFS::ULongLong, pusage.utime));
14      values.push_back(PerformanceValue(GrapeFS::UInt64, (uint64_t) pusage.vsize));
15    }
16
17    [...]
18  } PerformanceOperation;
19
20
21  #define GRAPEFS_PERF_DEF                GrapeFS::PerformanceDirectory *
        gfs_perfDirectory
22  #define GRAPEFS_PERF_INIT(x)            gfs_perfDirectory = new GrapeFS::
        PerformanceDirectory(x)
23
24  #define GRAPEFS_PERF_BEGIN              GrapeFS::PerformanceOperation __op(__func__,
        m_id)
25  #define GRAPEFS_PERF_CUSTOM(type, value) __op.values.push_back(GrapeFS::
        PerformanceValue(type, GrapeFS::PerformanceContent(value)))
26  #define GRAPEFS_PERF_CLOCK              __op.values.push_back(GrapeFS::
        PerformanceValue(GrapeFS::Clock, GrapeFS::PerformanceContent(clock())))
27  #define GRAPEFS_PERF_TSC                __op.values.push_back(GrapeFS::
        PerformanceValue(GrapeFS::ULongLong, GrapeFS::PerformanceContent(rdtsc())))
28  #define GRAPEFS_PERF_TIME               __op.values.push_back(GrapeFS::
        PerformanceValue(GrapeFS::Double, GrapeFS::PerformanceContent::FromTime()))
29  #define GRAPEFS_PERF_CPU_SYS            __op.values.push_back(GrapeFS::
        PerformanceValue(GrapeFS::Double, GrapeFS::PerformanceContent::FromSystemCPU()))
30  #define GRAPEFS_PERF_CPU_USER           __op.values.push_back(GrapeFS::
        PerformanceValue(GrapeFS::Double, GrapeFS::PerformanceContent::FromUserCPU()))
31  #define GRAPEFS_PERF_ALL                do { __op.FillProc(); GRAPEFS_PERF_TSC; }
        while (0)
```

Listing 27: Macro definitions to acquire the runtime profiling data

G Digital attachment

CD Attached

Contents

- GrapeFS project