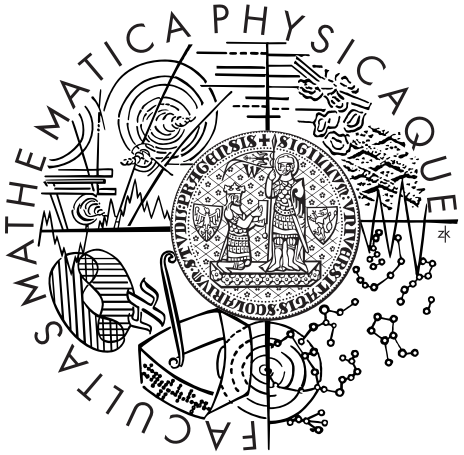
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Vladimír Matěna

Qt HDD benchmark

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Lukáš Marek

Study programme: Informatics

Specialization: Programming

Prague 2011

Thanks to:

Supervisor of this thesis who helped me with application development, especially the benchmarking code.

My grandfather who did the language correction of this text.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In..... date..... signature

Název práce: Qt HDD benchmark

Autor: Vladimír Matěna

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Lukáš Marek, Katedra distribuovaných a spolehlivých systémů

Abstrakt: Tato práce se zabývá měřením výkonnosti zařízení pro ukládání dat. Její účel je poskytnout program pro měření výkonnosti takových zařízení s grafickým rozhraním v prostředí Linuxu. Grafické rozhraní se snadno používá a zobrazuje výsledky měření v reálném čase. Navíc bylo několik běžných zařízení otestováno tímto programem. Zajímavé výsledky byly krátce popsány stejně jako faktory, které je ovlivnily. Obecné faktory ovlivňující výsledky v prostředí Linuxu byly také popsány.

Klíčová slova: srovnávání, pevný disk, grafický

Title: Qt HDD benchmark

Author: Vladimír Matěna

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Lukáš Marek, Department of Distributed and Dependable Systems

Abstract: This thesis deals with data storage device benchmarking. Its purpose is to provide storage device benchmarking application with graphical user interface for Linux environment. The graphical interface is simple to use and displays results in real-time. Moreover a few common device benchmarking was done with application developed. The interesting results were briefly described as well as factors influencing them. Also general factors impacting results measured within the Linux environment were described.

Keywords: benchmarking, data storage, graphical

Contents

Introduction

Problem definition

The purpose of this thesis is to provide easy to use hard drive benchmark with a Graphical user interface (GUI) providing real-time feedback. The benchmarks should be easy to use and should produce reproducible results. Currently there is a lack of such tools for Linux desktop, so Linux desktop was selected as target environment. Although Linux is primary target, some multi-platform readiness is desired. Therefore Qt graphical framework is intended to be used. Moreover, this thesis should provide description of both hardware and software factors impacting hard drive performance on Linux platform. This part focuses on caching layers and their possible impact on measured results.

The program should provide basic functions for benchmarking both raw drive performance and filesystem performance. Raw hard drive performance benchmarks should cover seek time and raw transfer speed. Filesystem benchmarks should focus on reading and writing single file and manipulating huge structures of files and directories. All operations done should be safe for data stored on device. Hence benchmarking raw device write is not intended.

Hard drive performance is bottleneck to desktop systems. Because of this operating system tries to improve storage performance by caching and not syncing changes to device. It is desired to limit impact of such optimization provided by operating system and not device itself as much as possible.

Graphical user interface should be easy to use and should provide real-time visualization of running benchmark. Qt framework does not provide direct support for dynamically changing graphs, nor any of its extensions provides this feature. Therefore implementation requires building such functionality on the top of more general resources provided by Qt.

What was done and how

A software called HDDTest was implemented using Qt graphical framework. Four raw device benchmarks and three filesystem benchmarks were implemented. Raw device benchmarks contain: device seek time, continuous read transfer speed, blocked read transfer speed and continuous read by block transfer speed testing. Filesystem benchmarks count: single file read and write transfer speed, directory structure build/read/destroy time testing. Detailed benchmark description is located in Project specification section. The whole device access code was encapsulated in a single class, so it can be altered easily in the case of porting code to another platform. In spite of this, only Linux environment is supported. Results measured can be stored to a file in Extensible Markup Language (XML) format and loaded later. A simple system for displaying dynamic graphs was implemented on the top of the QGraphicsView object.

Graphical interface was built on the Qt framework. Real-time updating graphs were implemented as wrapper around QGraphicsView object provided by Qt. Three graph types were implemented: line graph, bar graph and dot graph. All graph support real-time rescaling and updating. Updates are performed by continuous pooling of data structures filled by benchmark running in separate thread. Reference result data loading was implemented, so graph composed of two sets of results or one set of results and running benchmark data can be shown. Export to image function was implemented in order to simplify result publication.

When software was finished a small set of devices was tested. Devices tested contain laptop hard drive, desktop hard drive, Secure Digital (SD) card and Universal Serial Bus (USB) flash drive. Some of results were included in this thesis to provide example of what HDDTest can do. Some interesting results were included with a brief description.

Document structure

Problem analysis

Deals with detailed software description, description of its features and comparison with similar projects. It also covers changes made to specification reflecting results provided by development version of the software.

Documentation

Contains user and programmer documentation. User documentation describes all program functions and contains instructions how to use it. Programmer documentation contains description of program structure with remarks on constructs used and possible alternative solutions. It also includes all information needed for code extension.

Results

Deals with results measured on various devices and possible impact of caching on this results as well as caching levels in general.

1. Problem analysis

1.1 Project specification

This section describes goals set before actual implementation started and changes made to initial goals when first development builds were tested. For reasons behind this decisions taken see Detailed problem definition section.

The task is to develop an application which will provide user friendly benchmarks of Hard disk drive (HDD). Target platform for this application is GNU/Linux. Graphical environment of the application will be created using Qt libraries. The application will provide both raw hardware performance benchmarks and filesystem benchmarks. It should be easy to control the application and the benchmarks should be non-destructive to data stored on device. The measured result should be visualized in real-time.

There are several applications offering similar functions as this, but especially those coming from GNU/Linux environment are too sophisticated and complicated for desktop user. Most of them are command-line based providing no or very little callback to user. Those applications provide too many options which discourage less advanced users and disable results taken with different settings from being compared. This project aims at filling the gap and providing user friendly application with minimum settings.

Seek benchmark

Benchmarks speed of seeking given position on HDD. The benchmark consists of 1000 timed seeks to pseudo-random positions on drive. Positions are generated with stable seed to ensure the same behaviour every time benchmark is run on the same device. One byte is read from every position to ensure driver really accessed data at selected position. Initial intend was to show minimal maximal and average access times but later it came clear that it would not fit the graphical interface. Therefore, only average access time is explicitly displayed as minimum and maximum values are easy to guess from graph. The benchmark contains a dot graph showing seek length on horizontal axis and seek time on vertical axis. This graph should be updated as new seeks are being performed.

Read - random benchmark

This benchmark reads blocks of different sizes from random position on drive. Block sizes are powers of two from 512 bytes to 1 megabyte. Initial intention was to read 100 megabytes using each block size, but doing so takes too long. Reading 100 megabytes split into small blocks makes drive to perform a lot of seeks. That caused this benchmark to take several hours on average laptop drive. Finally, reading 100 blocks of each size was chosen. Benchmark run time was reduced to a few minutes and results seem to be the same. The results are shown in bar graph where every bar represents one block size. Every bar shows average transfer speed and read progress for given block size. To save space in GUI this benchmark was renamed to "Random".

Read - continuous benchmark

Purpose of this benchmark is to measure read transfer speed. Transfer speed is shown in graph where horizontal axis is drive position and vertical axis is transfer speed. Read operation is divided into blocks. Block should be large enough not to lower the transfer speed. Benchmark should read 4 gigabytes. Block size was intended to be 100 megabytes, but later tests proved that even 1 megabyte blocks have little effect on transfer speed. So 4 megabyte blocks were chosen as they provide smoother graphs than 100 megabyte blocks. This benchmark was renamed to "Continuous" in order to save some space in GUI.

Block benchmark

This benchmark was added to specification later. It is similar to read - random benchmark. Benchmark reads blocks of different sizes from drive but without seeking. Block sizes are powers of two from 512 bytes to 1 megabyte. 100 megabytes are read using each block size. Results are shown in bar graph where every bar represents one block size. Every bar shows average transfer speed and read progress for given block size. Purpose of this benchmark is to show how splitting read operation into small blocks affects performance.

R/W file benchmark

This benchmark measures read and write file transfer. It creates a file and writes one gigabyte into it. Then whole file is read again. Results are shown in graph where horizontal axis represents file position and vertical axis transfer speed.

File hierarchy benchmark

This is filesystem benchmark focusing on working with directory structure. It creates directory structure containing 1000 directories and 1000 files. Structure is pseudo-random but every time the same. Results are displayed by bar graph containing two bars. The first shows build time and the second shows time needed to delete the structure. The whole structure should be placed every time in the same place in filesystem to ensure result reproducibility. This benchmark was renamed to "Structure" in order to save space in GUI.

Small files benchmark

Benchmarks working with small files within large directory structure. First structure of 1000 directories is created. Then 1000 files are created within this structure. Files have sizes of 1 kilobyte to 10 kilobytes. Then all files are read in pseudo-random order. Finally, the whole structure is deleted. Bar graph is displaying time for these operations: Directory structure build, file write, file read, structure delete.

Info

Application should provide way how to get information about hardware being tested and software used. This information should be stored with tested data for future comparison.

1.2 Detailed problem definition

This section describes what is goal of this thesis in detail. The solutions chosen and reasons for this solutions are discussed. The problem is divided into two parts: HDD benchmark and Factors impacting measured results.

1.2.1 Benchmarking

The main goal of this thesis is to provide graphical HDD benchmark for Linux desktop. The benchmark software should be simple enough to be useful for an average user. It should provide minimum settings and results should be presented via graphical interface in real-time.

Benchmarks

should definitely contain random access and continuous read speed as these two are basic drive parameters. Even when modern drives contain technologies that make those two parameters a bit less important, they remain to be a measure by which many people judge drive performance. Continuous read speed gives estimation of overall drive transfer speed. Random access shows how the drive can deal with transfers consisting of a lot of small blocks. Unfortunately write versions of this two benchmarks cannot be included due to the fact that writing directly to drive would harm data on it. Even when it is possible backup data being overwritten, it is too risky especially in case of power failure. Moreover, standard rotating platters drives usually have equal read and write performance. This is unfortunately not true for Solid state drive (SSD). Those have big blocks that have to be erased before they are written. And the situation is also getting complicated by controllers doing wear levelling. These features make their write performance unpredictable and hard to guess from read performance. In addition, two more raw device benchmarks covering advanced drive behaviour should be present. These benchmarks should show how the drive behaviour changes depending on block size used for operation. Both benchmarks read blocks of several sizes from device. First benchmark called "Random" reads blocks of given size from random positions on device. The second one, called "Block", reads blocks of given size one by one without seeking between reads. The Block benchmark usually shows results similar to read transfer speed even when blocks sizes are as small as 512 bytes. The results of this benchmark are intended to be compared with the results of "Random" benchmark. The results of the "Random" benchmark of flash memory based devices show that some block sizes causes drive to behave much faster than others. Drives based on rotating platters show results depending on seek time needed to access block and continuous read speed by which the block is read.

Also benchmarks testing filesystem should be contained. Read and write file transfer speed should be benchmarked as they determine, how fast will the large file transfers to/from the device be. File access speed depends on device access speed and filesystem, especially on its type, setting and fragmentation. Results of this test are hard to reproduce as file can every time be divided into different blocks due to free space fragmentation. Even when comparison of this results

between drives is limited this benchmark shows transfer speed of daily tasks as file copying between devices or large file reading and writing.

Another thing that can be tested in filesystems is how they behave when dealing with a large directory structure. These benchmarks simulate tasks such as starting applications, copying or compressing directory structures and all other tasks that depend on reading a lot of small files. Two benchmarks should be provided. The first for working with directory structure and the second for working with small files. The first one called "Structure" consists of creating directory structure and deleting it. It shows how fast the filesystem can create and delete directories and files. The second benchmark called "Small Files" also creates directory structure, but then it writes small files into it. When files are written, it reads them and finally deletes whole structure. These benchmarks seem to be pretty sensitive on caching as many filesystems perform cached reads of small files and keep directory structure in memory instead of syncing it to device. Even when minimizing those effects in not every time possible those benchmarks can give insight in how different filesystems or devices handle small files and large directory structures.

The whole benchmarking process should not harm data on device as users are expected to run them on daily use systems. Benchmarks should not provide many settings. This allows user interface to be simple and results to be simply comparable. It also requires benchmarks to be designed well to fit large, small, slow, fast, rotating platters based and solid state drives. Benchmarks should take reasonably long on wide range of devices. New results data should be gathered fast enough to be displayed smoothly by graphic interface. For some benchmarks it is a compromise between getting smooth data and getting precise data. The initial plan was to use bigger block sizes for some continuous reading tests, but later was discovered that performance penalty of using smaller blocks is not so significant. Moreover, using smaller block improved user experience a lot. Changes done to benchmarks are described in Project specification section.

User interface

should provide easy ways for an average user to run benchmarks. It is important not to give user too many options. Nevertheless, some controls for basic functions are needed. User should be allowed to pick device to test or saved results to display. Moreover, it should be possible to select secondary saved results that will be displayed as reference to primary results. It was decided that these two selections will be implemented by comboboxes distinguished by colour. The users should also be allowed to select a benchmark. This was implemented by dividing window into two parts. The upper part of the widow consisting of two comboboxes and button for saving results are common to all benchmarks. The rest of the window is organized by tabs with headers shown on the left side of the window. Each tab contains benchmark results, progress, start/stop button and button for exporting graph as image.

The results should be displayed by graphs. The graph should be updated when benchmark is running. When reference results are loaded graph should display both result sets merged and distinguished by colour. Graph updates should be frequent enough not to let user think the program has hung. It also requires benchmarks to provide new data that can be displayed. The desired effect was

achieved by pooling benchmark results 10 times per second. This refresh rate seems to be enough to provide responsive graphs that worth user attention when test is running. Results for current device and reference results were distinguished by colours used in comboboxes. Where more colours are needed the primary colour is mixed with another one. The resulting colour scheme is described by legend in order to simplify results interpretation.

One of the most important things about GUI was choosing graphical framework that would fit project needs. As implementation for GNU/Linux was required the choice was between GTK and Qt. Both frameworks provide rich offer of functions to satisfy basic needs of this project. From the functional point of view the functions simplifying graph implementation were the most important. Unfortunately both frameworks do not provide direct implementation of graphs needed by this application. Therefore it was needed to implement graph on the top of something more general. Qt provides a slightly better universal graphic drawing system then GTK. Moreover, Qt is truly multi-platform so future porting to another platform would be much easier then with GTK. Evaluating this benefits Qt was chosen. Later using K Desktop Environment (KDE) libraries that base on the Qt was evaluated, but using KDE would not simplify implementation a lot and porting to other platforms would get complicated.

Device access

is another problem that needs to be resolved. The application should be able to enumerate devices and gain access needed by benchmarks to them. The Linux system contains a lot of block devices capable of being benchmarked, but just few are real hardware. Showing all block devices could make user confused. Therefore it was decided to limit list of devices to those found in "/dev/disk/by-path/". This path contains symbolic links to devices as they are connected to the system. Using targets of these links effectively limits list of all block devices in "/dev" to those being really connected to system and filtering out virtual ones. This list should be suitable for most users. If somebody wants to benchmark some exotic device not being listed the GUI still allows to type custom path to device.

Device access is realized by opening block device like a regular file. This method is sufficient for all operations needed by benchmarks.

The biggest challenge of device access is to limit influence of optimizations done by operating system. Those optimizations are caching device content and lazy writes. The first optimization causes repeated tests to be much faster and the second makes directory structure changes instant. In the final implementation effect of caches is limited by dropping caches before benchmark or its part. And by giving advice to operating system not to cache data being written. Lazy writes are limited by calling "sync" after every filesystem operation and counting sync call time to this operation. This countermeasures are not effective every time. Some filesystems like New Technology File System (NTFS) seem to do some caching that cannot be disabled in this way. Another problem is that dropping caches requires privileges that user running benchmarks does not have to have. In this case warning is shown to inform user that results can be affected by caching.

Another approach would be to remount filesystem with sync option and open raw device in sync file mode. Remounting filesystem with sync seems to work but it would interfere with the system a lot. Therefore remounting with sync

option was denied in order no to get user in trouble while running benchmark on root filesystem. Opening device with O_SYNC and O_DIRECT flags could limit cache effect a lot. But this method causes some operations with device to fail. Unfortunately, those operations are required by benchmarks.

1.2.2 Factors impacting results

Ideal benchmark would measure directly performance of the device without influence of other systems. Unfortunately, real benchmark can only try to limit such influence. Results can be influenced by caches and buffers provided by operating system. This influence is limited by dropping caches, but some filesystems seem to ignore such commands. It is hard to judge whether throughput is limited by device itself or the rest of the system is becoming a bottleneck. The GUI can take too much Central processing unit (CPU) power and cause benchmark to run slower. Moreover, another process in the system can consume resources needed by benchmark to achieve optimum results. Such processes can even use device being benchmarked and cause benchmark to report degraded performance. Many of this problems cannot be avoided or cannot be avoided without unacceptable modifications of the system. Even with this problems on a reasonable fast system a benchmark of non-system drive usually reports accurate results. When things go wrong it can be detected from pikes on resulting graphs.

1.3 Comparison with existing implementations

This section briefly describes two hard drive benchmarking tools. Both of them run on Windows platform. These two were selected among others as their features are similar to the software developed as part of this thesis.

HD Tach

HD Tach is software similar to the subject of this thesis. It also provides basic benchmarks as sequential read speed and writes, random access, interface burst and CPU usage. The results are presented in graph, but the graph is not displayed as the results are being picked. The software first performs all tests and show their progress and then a graph with the results is displayed. The usage is simple as there are just a few options besides drive selection. [5]

HD Tune

HD Tune is also more feature rich than subject of this thesis. It has a lot of features containing transfer speed and random access benchmarks with the same real-time result presentation as done by HDDTest which is subject of this thesis. It also has many features HDDTest does not have as error scanning, Self-Monitoring, Analysis and Reporting Technology (S.M.A.R.T) status reading and secure erase. [6]

2. Documentation

2.1 User documentation

This section describes HDDTest user interface, common tasks and some extended usage examples.

2.1.1 Starting application

HDDTest require Qt libraries to work. It should work with any recent Qt release, but the application is tested with version 4.7.3.

HDDTest application expects symbolic links to devices to be in "/dev/disk/by-path". If the system does not contain this folder or it does not contain such links the user will be unable to pick device to be tested easily from combobox, but he can still benchmark device by typing the whole path.

The application needs to access the devices being benchmarked. On a usual Linux system users are not allowed to access devices directly. Even when the benchmark starts it may happen that it will not work correctly. Results measured without full access to device may be incorrect. In order to eliminate these problems run the application with administrator privileges. This can be achieved by several ways. To do this *sudo* or *kdesu* needs to be installed. It is possible to run HDDTest with elevated privileges with one of these commands.

Listing 2.1: Running HDDTest with sudo

```
# sudo /path/to/HDDTest
```

or

Listing 2.2: Running HDDTest with kdesu

```
# kdesu /path/to/HDDTest
```

When the application is run without sufficient privileges two things may happen. The benchmark can stop with error as device cannot be accessed or a warning will be shown as caches cannot be disabled without sufficient privileges.

2.1.2 User interface

User interface tries to be straightforward. Its central area contains selected benchmark (5). There are tabs with benchmark selection on the left of the screen (4). Device selection combobox is placed top left (1). Reference device combobox is placed top right (2). This is where overlay results for comparison are selected. A button for saving all results for current device is placed in the top left corner (3).

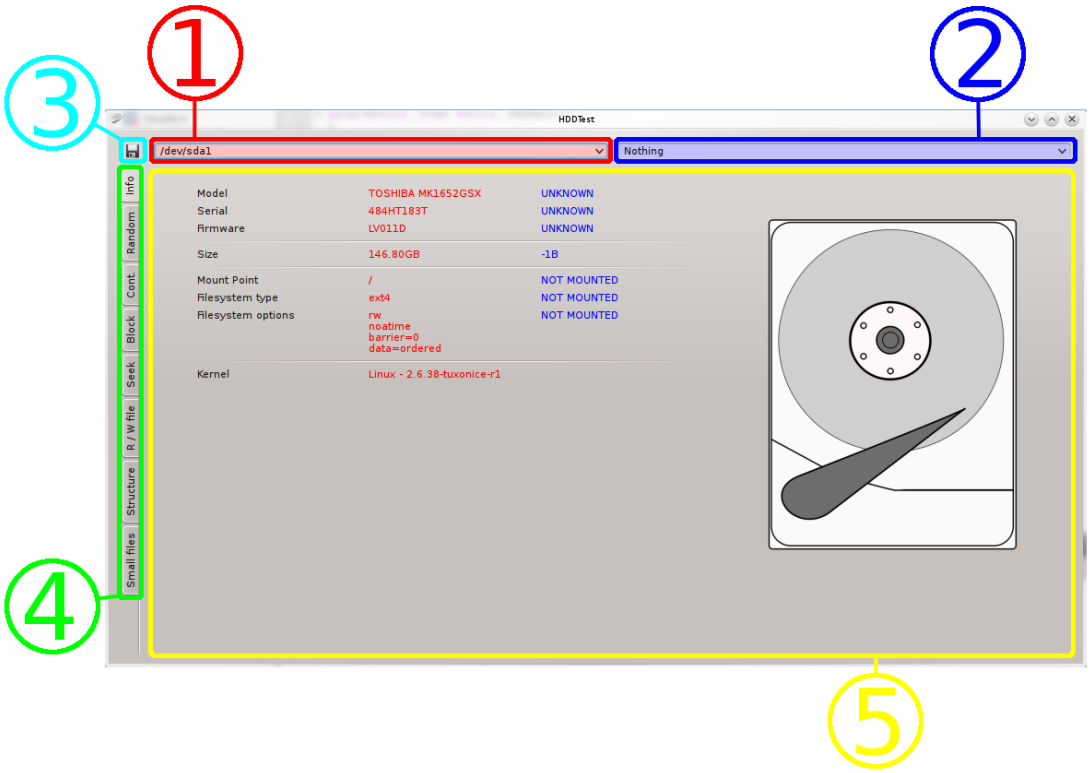


Figure 2.1: Application graphical interface

The top left combobox allow to pick device to be tested as well as saved results from current directory. When the desired device is not listed the user can type direct path to such a device. The path written manually in the combobox needs to be confirmed by pressing enter. Moreover, a file open dialog can be opened by corresponding option in combobox to choose saved results from filesystem. The top right combobox is like the left one but it works only with saved results. The both sets of results measured or loaded are displayed at the same time distinguished by colour. Current results or results loaded by left combobox are red or red-green when more colours were needed.

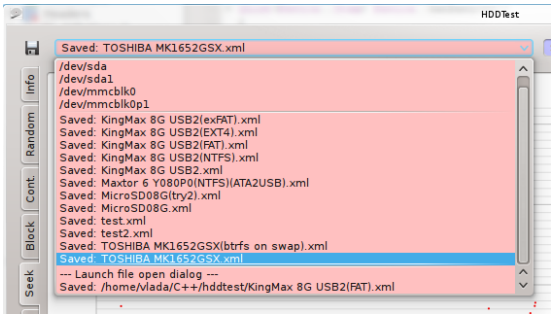


Figure 2.2: Device selection in detail

Results loaded from right combobox are blue or blue-green when more colours were needed. The reference results can be changed even while the benchmark is in progress.

The button placed in the top left corner with diskette icon is used to save current results measured (those displayed in red). This button opens file save dialog. The results are saved to file in the XML based format. Saved results can be viewed by HDDTest itself or processed by any XML processing software or script.

The benchmarks can be accessed by tabs on the left side of the window. The first tab shows information about selected device. The information tab is followed by four raw device benchmarks. The filesystem used does not affect the results of raw device benchmarks.

The "Random" benchmark is supposed to show how the device handles small blocks. The benchmark called "Cont" reads large blocks sequentially and displays sequential transfer speed. The "Block" benchmark reads small blocks sequentially. Usually device performs equally on every block size in "Block" but in some situations it can happen that some sizes are preferred. The last raw benchmark is "Seek". It benchmarks access speed. The raw benchmarks are followed by filesystem benchmarks.

The filesystem used does matter a lot in this benchmarks. First one is "R/W file". It writes and reads large blocks sequentially to and from file resulting in transfer speed graph. The last two benchmarks operate with structure of small files. Different operations as creating, reading, writing and deleting of files and directories are performed pseudo randomly.

Technical details of all benchmarks such as sizes and counts of blocks can be displayed with info button placed bottom right.

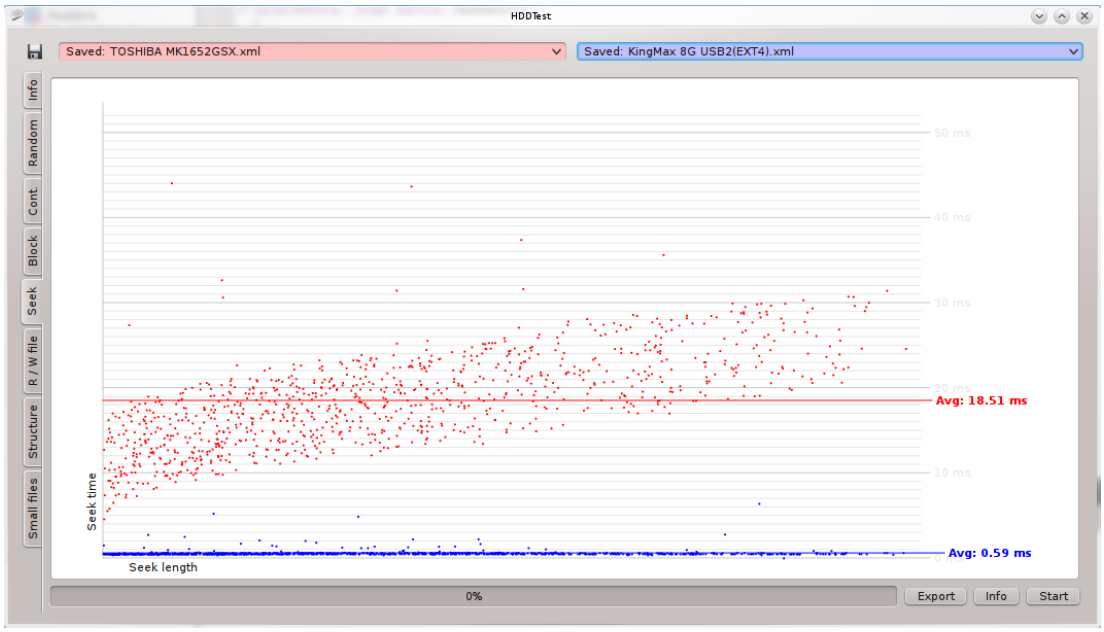


Figure 2.3: Seeker benchmark controls

All benchmarks look similar. They consist of a graph area where the results are displayed, progress bar at the bottom and three buttons placed bottom right.

The Info button displays benchmark description. The Export button exports graph area of the benchmark as an image and opens file save dialog to save it. The last button is used to start the benchmark if it is not running or stop it otherwise.

2.1.3 Basic usage

This is a step by step guide how to compare performance of the two devices. Before benchmarking the devices please consult Starting application chapter, make sure devices being tested are not used and mount filesystems on devices wanted to pass the filesystem benchmarks.

After starting the application please select first device from left(red) combobox. If a filesystem is going to be benchmarked the selected partition has to be mounted. If everything works and the application was started with sufficient privileges no warning should have occurred. The information tab should display information about device selected. Some devices cannot be identified but at last size should be displayed and match device or partition size.

When device is selected the benchmarks can be run on it. When a tab is opened with the wanted benchmark, it can be run by pressing the start button. The button disables itself and changes its text to "Starting". It can stay like this for a few seconds before device is flushed and caches dropped. Then text changes to "Stop" and button is enabled again. The benchmark is running and the results should be displayed as process progresses. Once progress reaches 100% the button changes to "Start" again and the benchmark can be run again.

Once all desired benchmarks have been done the results can be saved using button with diskette icon placed top left.

When results are saved the second device can be benchmarked. The process is all the same as with the first device. Once first device is finished just select another from left(red) combobox. All unsaved information about first device are forgotten when another device is chosen. Before or during benchmarking of the second device results saved first time can be selected to be displayed simultaneously. This can be done by selecting saved results from right (blue) combobox. Newly saved results are not displayed and have to be open by selecting "launch file open dialog" option in blue combobox.

Raw device tests just read from device thus do not change data on it. Despite of this filesystem benchmarks read and write to files. HDDTest creates directory "tmp/hddtest.temp.dir" on filesystem being benchmarked. Once benchmarking is finished "hddtest.temp.dir" can be removed.

Listing 2.3: Removing HDDTest temp

```
# cd /path/to/device/root
# rmdir tmp/hddtest.temp.dir
```

2.1.4 Limitations

While in most cases benchmarks work as desired it may happen they will not. Under some situation it may happen the benchmarks do not work correctly or not at all. In most cases this is caused by running HDDTest without access rights to

devices which causes device access to fail and end with an error. The access can also fail because of device being too small for benchmark. It can also happen that rights are sufficient to access device but not to drop caches. In such situations a warning is displayed. Running benchmarks when such warning is displayed is not recommended as results will most probably be inaccurate.

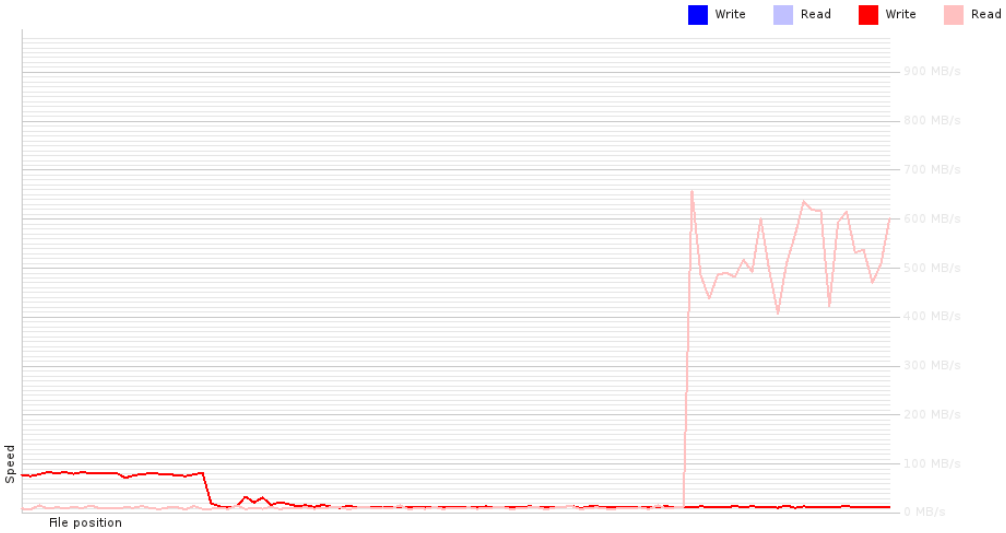


Figure 2.4: Failed benchmark when caches were not dropped. The NTFS driver cached last bytes written and satisfied read requests with them much faster than the device would be able.

It can also happen that a filesystem caches data and does not respects cache empty commands. In such situations no warnings or errors are displayed as this cannot be detected. Such problems are visible from results. For example on figure 2.4.

2.1.5 Advanced usage

Even when HDDTest tries to be simple to use for user and does not contain a lot of options, it can be used to do something more than just what it was designed for. Instructions below are just tips and should not be followed without understanding.

Benchmarking file

HDDTest can be made to benchmark a file with its raw device tests. It just needs the system to pretend that the file is block device. Assuming the losetup is available on the system, the first loop device is not used and the file that should be used is located at "/testfile". All commands should be used with appropriate user privileges.

Listing 2.4: Setting up loop device

```
# losetup /dev/loop0 /testfile
```

Now file can be benchmarked by typing "/dev/loop0" in device selection combobox in HDDTest. Once finished, loop device should be detached.

Listing 2.5: Unsetting up loop device

```
# losetup -d /dev/loop0
```

Testing system latency

Sometimes it is not sure what the bottleneck actually is, whether it is device itself or the rest of the system. Some interesting values that help to understand this situation can be gained by running benchmarks on a ram-disk. Many Linux distributions come with 16MB ram-disk located at /dev/ram0. Benchmarks can be run on such device by typing "/dev/ram0" in device selection combobox. Please note that many benchmarks will refuse to work on such a small device. Interesting results can be gained from "Cont" and "Seek" benchmarks.

Benchmarking remote share

The same idea as used when Benchmarking file can be used to benchmark file located on remote share. Assuming share is located at "/mnt/share", first loop device is unused, losetup and dd utilities are present.

Listing 2.6: Benchmark remote file

```
# cd /mnt/share
# dd if=/dev/zero of=./file.2G bs=1M count=2k
# losetup /dev/loop0 /mnt/share/file.2G
```

Now file on share can be benchmarked using raw device benchmarks by typing "/dev/loop0" in device selection combobox. When benchmarking is finished loop can be unset and file deleted.

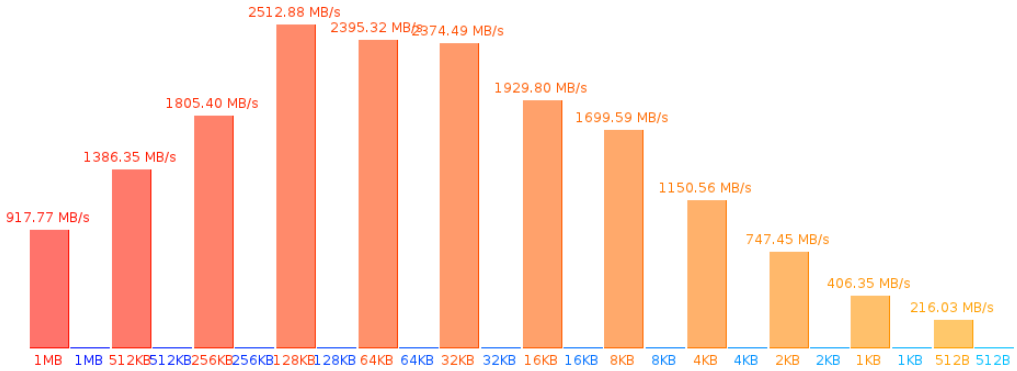


Figure 2.5: RAM device random blocks benchmark. Limitations of system can be seen. Now it is clear that it makes no sense to benchmark any faster device as its performance would be limited.

Listing 2.7: Cleanup after benchmarking remote file

```
# losetup -d /dev/loop0
# rm /mnt/share/file.2G
```

2.2 Programmer documentation

This section explains how to build HDDTest application, how it works and how to extend it.

2.2.1 Building application

To successfully build HDDTest application several tools are required. To compile code g++ is needed. Build was tested with GNU Compiler Collection (GCC) version 4.5.2. HDDTest uses Qt libraries, so development version of Qt libraries is required. It was tested with version 4.7.3. To carry off build process qmake tool is needed, tested with version 2.01a. Exact versions of tools specified above are not required. During development the tools were upgraded several times thus sources are most probably compatible with previous versions. Versions specified should be used only when installed versions do not work.

A clean Ubuntu 10.04 installation required these packages to be installed: qt4-qmake, libqt4dev, g++.

Application was developed with use of Qt Creator IDE and project description file is included with sources. Building application with Qt Creator IDE is quite simple. Opening the project and hitting Ctrl+r is enough. When the Qt Creator IDE is not available the application can be built using command line tools. To build application this way run the shell interpreter in source root directory and type the following commands.

Listing 2.8: Building HDDTest from commandline

```
# make clean
# qmake -Wall CONFIG+=debug hddtest.pro
# make
```

Typical build should not throw any warnings and should result in hddtest executable. The whole build process was tested on up-to-date 64bit Gentoo linux system and 32bit Ubuntu 10.04 Lucid Lynx.

2.2.2 Basic ideas and solutions

Classes

Classes in the project copy separated functional parts of the application. There are simple classes for the main application window called "HDDTestWidget" and random number generator called "RandomGenerator". Basic primitives used by benchmarks were also moved to separated classes for file and device access called "File" and "Device".

Every benchmark has its own class which covers both benchmarks GUI and benchmark process itself. Benchmark classes are based on a class common to all benchmarks. The class is called "TestWidget". It handles GUI elements that are common to all benchmarks, graph drawing and benchmark state changes.

There are more helper classes described in generated Doxygen documentation that are not important to understand basic principles of the application.

Qt

Qt framework is used to handle application GUI and everything that Qt libraries can handle in order to make application depend more on Qt than Linux specific things. Only raw device access used by benchmarks was done as low-level as possible because it is not sure how Qt handles it internally.

The most interesting part of GUI is graph drawing. All the graphs and measures are drawn by HDDTest itself. Graph primitives are implemented in TestWidget class and provided to benchmark classes. Graph drawing is done via QGraphicsView object. QGraphicsView provides drawing geometrical primitives and fonts [4]. Graph primitives are based on those QGraphicsView provides. QGraphicsView also supports scaling and anti aliasing which is used to smooth graphs.

File and Device classes

These two classes provide basic functions needed by benchmarks. Provided functions are mostly wrappers to file read and write functions. Wrappers do not care about data being transferred. Instead of this only sizes and times are important. Methods of this classes are used exclusively by benchmarks and device enumeration in main application window. Even when moving code to benchmark functions would be possible, the code was left separated.

Device and file access and enumeration code was moved to this separated classes in order to make application ready for supporting multiple platforms in future. Separating this code could also help when low-level access code needs to be changed because of bugs. Also the benchmark functions are much more simple when not including time measurement calls.

TestWidget class

TestWidget class implements common code from all benchmarks and graph drawing code. This class has three basic functions. The first one is to provide callback to benchmark GUI elements. The second one is to start and stop the test in separate thread. The third is to provide graph drawing.

The actual benchmarks are classes derived from TestWidget class which implement a few methods specific to them. When benchmark is started a benchmark specific method containing benchmarking code is run in separate thread and another benchmark specific method is called periodically to redraw the results in graph. The graph drawing method uses basic graph parts provided by base TestWiget class such as Bar graph, Line Graph and more support elements.

Benchmark specific classes

Every benchmark has its own class that contains its GUI handling and benchmark function. Benchmark class also defines class that holds results in benchmark specific way. This class is based on TestWidget class. Benchmark class implements several virtual methods defined by TestWidget. These methods define benchmark specific behaviour. Every benchmark defines TestLoop method which contains

benchmark code. InitScene, UpdateScene and GetProgress methods that handles GUI specific things. And WriteResults, RestoreResults and EraseResults to handle results in benchmark specific way.

2.2.3 Device access, cache disabling

For a benchmark, getting as close to hardware as possible is very important. Accessing devices as files was chosen as it proved to work in every situation tested. There are more ways how to access device than this but others would limit benchmarks or make them more complicated.

Listing 2.9: Sample opening and reading device

```
fd = open("/dev/sda", O_RDONLY | O_LARGEFILE);
posix_fadvise(fd, 0, 0, POSIX_FADV_DONTNEED);
read(fd, buffer, buffer_size);
close(fd);
```

A more accurate benchmarks would use O_SYNC operation while opening the device but it would cause device access to stop working under some circumstances. The HDDTest application is intended to be run by users not understanding things behind on many different systems so reliability is very important. Moreover results measured with and without this option do not differ a lot. All the raw device access primitives used by benchmarks are implemented in Device class and C functions like open, read, write and lseek64 are used to build them.

Higher level access to filesystem uses Qt classes to access filesystem. In this case Qt was used because it provides simple way to do the job and seems to be as fast as "native" approach. Even when simple constructions were used filesystem access code was encapsulated in File and Device classes in order to be altered easily if needed.

There are several problems with operating system caching and buffering data being read and written. There are problems with raw device access and problems with filesystem.

The problems with raw device access are quite easy to solve. This is caused by the limitation of raw device benchmarks. Raw benchmarks are read only so only problem with caching is that Linux kernel caches data in memory and satisfies read requests with cached data. This behaviour is exposed when same benchmark is run twice. The random sequence used by benchmarks is every time the same in order to make results reproducible. The kernel caches all data being read during the first pass and then the second pass is incredibly fast. This effect is effectively removed by dropping system caches before every benchmarks or its separated part. The only remaining problem is when the same data are hit twice during one pass. This do not happen often so this problem is usually not noticeable from results. To limit this effect an advice to kernel is given not to cache data. It is only an advice but it seems working on tested systems. And even if it would not work the results were not affected a lot.

The problems with access to files and filesystem is much more complicated as it is read-write. The system is not only using cached data when reading, it also accelerates writes to files by keeping changes in memory. The good thing is that filesystem benchmarks are working in phases. And the results are shown only one

times per phase. To limit the cache effect a sync command is called and caches are dropped before every phase. Doing so removes the problem while reading, but writes seemed to be too fast. In order to avoid cache accelerated writes a final sync was added behind every phase. The sync time is added to phase time in the results. The sync before the phase ensures that non-synced data not belonging to the benchmark are not synced at benchmark end.

The only benchmark not working in phases is File RW. In order to disable write caching for this benchmark the benchmarked file is open with O_SYNC flag. Using this flag while working with raw device caused problems, but while used on real files no problems were detected.

Listing 2.10: Example benchmark phase with cache prevention

```
Drop caches
Sync filesystem
Take start timestamp
Do some timed job like creating 1000 directories
Sync filesystem
Take end timestamp
```

Even when cache prevention mentioned above seems to work on many filesystems, it looks at least Filesystem in userspace (FUSE) based filesystem do some extra caching that cannot be disabled this way. This was noticed when using NTFS and sometimes with Extended File Allocation Table (exFAT). Both of them are using FUSE.

Error and warning handling

The application uses callbacks for reporting errors to user. Only errors and warnings from device access are reported to user as they can influence results. Other problems are silently ignored. When an critical error occurs during the benchmark it is stopped and the user is notified. When a non-critical error occurs a warning is displayed and benchmarks process continues.

2.2.4 Known problems in implementation

Device listing and identification

Devices are listed by scanning content of "/dev/disk/by-path" directory. This method relies on presence of link to the device in that directory. Some devices that are suitable for benchmarking seem to be not present in this directory. Moreover, sometimes devices that cannot be benchmarked are present (usually optical drives).

To solve this problem another source of available devices has to be used. Recent Linux desktop systems use udisks for such purpose. Using udisks would also improve ability to identify the device. Current identification based on ATA command does not work as well as udisks would.

Random generator

To achieve reproducibility, the benchmarks that depend on random operations have to perform pseudo-random operations with fixed seed. Current implementation uses standard random and srandom C function calls to obtain random values. Even when seed is set every time the same random number generator can differ system to system. This can cause that benchmarks can be a bit different when C libraries change.

To obtain more stable benchmarks implementing simple fixed seed random number generator in HDDTest code is needed.

Device access

Even when current device access code with cache prevention measures works in most cases some improvement is possible. In some cases cache preventing fails. See Device access, cache disabling. Using synchronous I/O is one of possible solutions. This solution was not used as it caused other problems but maybe this can be worked out.

3. Results

In this chapter some results measured with HDDTest application are presented. To clarify measured results a description of both hardware and software caching layers of tested devices is present.

3.1 Media caching layers

This section explains caching layers on media used to obtain results presented in this chapter. The most results were captured with two devices. The first one is notebook hard-drive with rotating platters TOSHIBA MK1652GSX. The second one is USB flash drive Kingmax USB2.0 FlashDisk.

Kingmax USB2.0 FlashDisk

This device is a typical example of a flash based USB mass storage drive. Manufacturer do not provide much more information than storage capacity. From the tests follows that the device does not have any detectable cache or write buffer. Probing the device with hdparm reports write caching is not supported.

Listing 3.1: Testing write cache presence with hdparm.

```
# hdparm -W /dev/sdb

/dev/sdb:
SG_IO: bad/missing sense data, sb[]:  f0 00 05 00 00 00 00 0a
 write-caching = not supported
```

From all the signs above it looks like this device does not have a cache of size that can influence results.

TOSHIBA MK1652GSX

This is a standard notebook rotating platters drive. The hdparm command reports 8MB write cache is present. Even when cache is present its effect seems minimum.

Listing 3.2: Getting cache size with hdparm.

```
# hdparm -I /dev/sda|grep cache
        cache/buffer size  = 8192 KBytes
           *    Write cache
```

If the 8MB storage were be used to cache Seek benchmark traffic then second pass of the benchmark would be much faster. This happens when OS caching is not disabled. Assuming the cache would store the whole block which is 512 bytes in size. The whole Seek benchmark pass counting 1000 seeks would make the cache to store 1000 blocks. This proves that the 8MB storage is not used this way.

The benchmark with write cache enabled and disabled using hdparm shows huge performance difference. This difference is present only when writing data to the device. The improvement when cache is used is permanent even when long burst is written. All the read only benchmarks do not show any significant difference.

Looking at the beginning of the graphs there is a peak that indicates some cache is being used. But multiple tests shows that those peaks are sometimes positive and sometimes negative. So it is not clear what causes them.

Figure 3.1: TOSHIBA MK1652GSX Write cache impact. Red - cache enabled. Blue - cache disabled

SSD drives

No real SSD drive was benchmarked. A benchmark of a CF card connected over ATA interface had similar results as benchmark of USB flash drive.

SSD devices usually do wear levelling. This includes complicated systems of mapping real blocks to virtual ones. It makes results of benchmarks unpredictable. Moreover, some SSD drives include cache of significant size. [1] It is unclear how this cache can affect results.

3.2 Linux I/O layers

There are many caching layer present in the Linux operating system. The layers are presented one by one from user-space to device. Their caching effect on benchmarks is discussed.

3.2.1 Virtual filesystem

Userspace applications talks to this layer. According to Linux Kernel documentation [2] this layer provides cache for directories and inodes. This cache provides fast directory lookup. This cache is not used by raw device benchmarks as they

do not use filesystem. This cache is used by filesystem benchmarks. There is no need to disable this cache as most probably this cache can be provided to all open directories and to every application used. Benchmarks simulating daily filesystem usage should use this cache.

3.2.2 Real filesystem

There are many filesystems supported by the Linux kernel. There are even many used by systems, this benchmarking software is expected to be running on. Among others ext2, ext3, ext4, btrfs, XFS, ReiserFS. Every filesystem uses a bit different strategy. Some of them are use cache to speed things up. There are two different kinds of data to cache.

The first one is directory structure and data related to it. This cache is usually small and does fit in Random access memory (RAM). It means that this cache will most probably accelerate every operation done by user. The HDDTest application is going to simulate daily system usage so it should use this cache too.

The second one is cache of data stored in files. This is much different than the first case. In most cases system RAM is not big enough to cache all data user is working with. When filesystem usage is just a bit heavier the operations tend to be accelerated by cache in the beginning. But very lazy to complete as the cache is full somewhere in the middle of the operation. When benchmarking a filesystem, a performance that can be kept all the time during the operation is important. This means that HDDTest should try to disable such cache.

3.2.3 Page cache

A block device associated with real device uses buffer to cache requests to blocks done by upper layer [3]. This caching impacts both filesystem and raw benchmarks. The effect of this cache has been minimized by various measures in HDDTest code. There is no detectable effect on raw device benchmarks and very limited effect on most filesystems.

3.2.4 I/O Scheduler and drivers

Schedulers mostly come into play when more processes are sharing access to one device. There are more schedulers supported by Linux kernel. When device is benchmarked it should not be used by any other application. If it is used exclusively the scheduler should not impact results a lot. The benchmarks provided by HDDTest are not threaded. This means they do not test device performance when used by more threads. So the only thing that can happen when scheduler comes into play is when some other application is using device while benchmark is in progress. In this case result of benchmark is unclear. Benchmarks should not be run on the device used by other processes in the system.

3.3 Description of results measured

This section describes some results measured with the HDDTest application. The results are presenting both the features of the application and interesting behaviour of devices being tested. The set of the devices being tested is limited because the topic of this thesis is benchmarking software and not comparing devices.

Benchmarks were run on the Dell Vostro 1510 notebook with Intel(R) Core(TM)2 Duo T5670 @ 1.80GHz CPU and 4GB of RAM. Most benchmarks were run with up to date 64bit Gentoo system. The results of internal hard drive on which the Gentoo system resides were measured with System Rescue CD in order to make sure nothing is using it.

3.3.1 Device comparison

There are results of a few benchmarks run with different devices in this subsection. The expected values were measured. The results illustrate how HDDTest can be used to compare device performance.

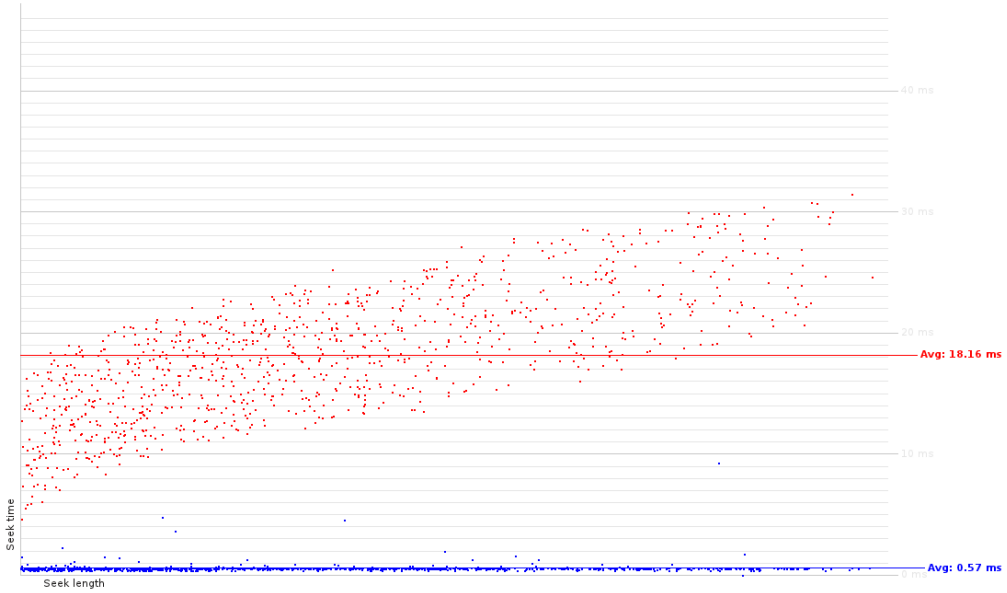


Figure 3.2: Comparison of TOSHIBA MK1652GSX (red) with KingMax 8G USB flash (blue) in Seek. Flash disc is clear winner is Seek benchmark.

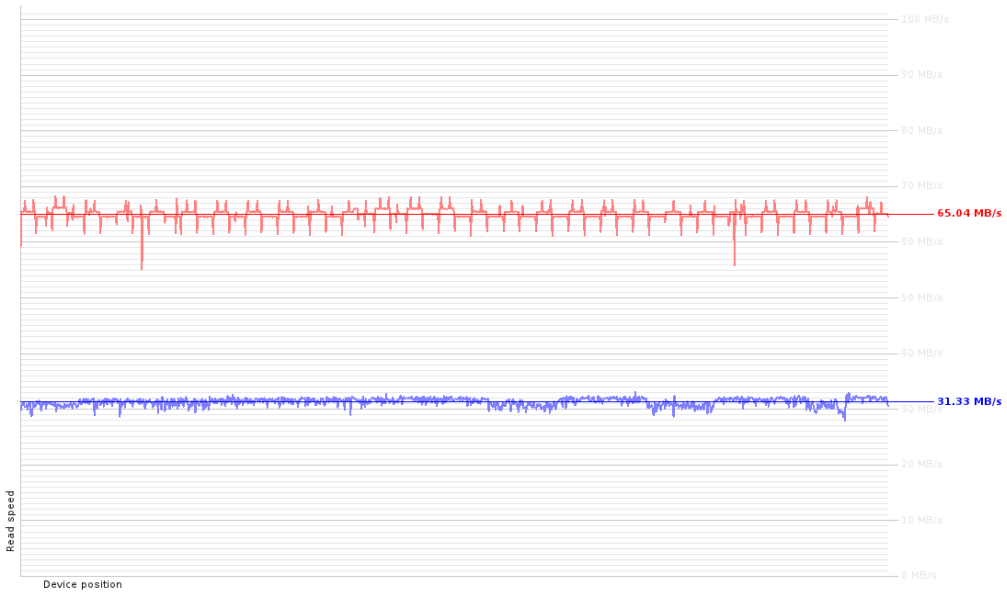


Figure 3.3: Comparison of TOSHIBA MK1652GSX (red) with KingMax 8G USB flash (blue) in Continuous read. The continuous read speed performance of flash drive is limited by USB 2.0 bus speed.

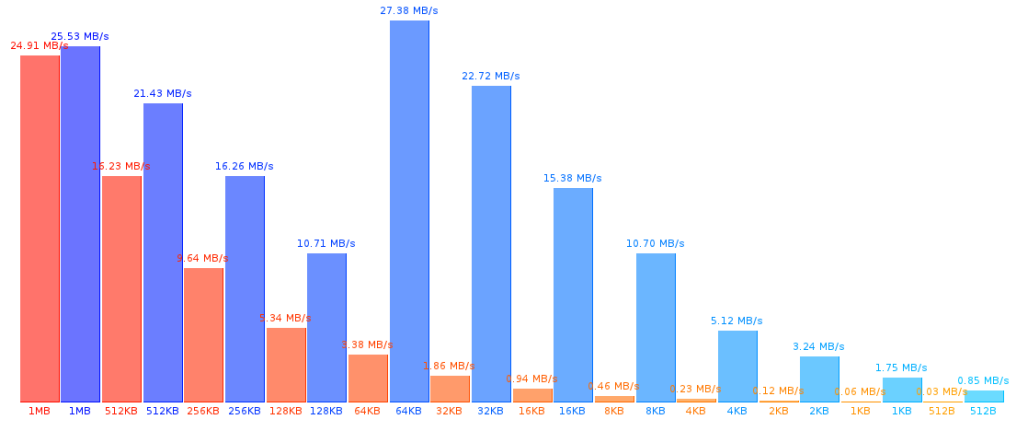


Figure 3.4: Comparison of TOSHIBA MK1652GSX (red) with KingMax 8G USB flash (blue) in Random placed block read. Flash performance with different block sizes is interesting.

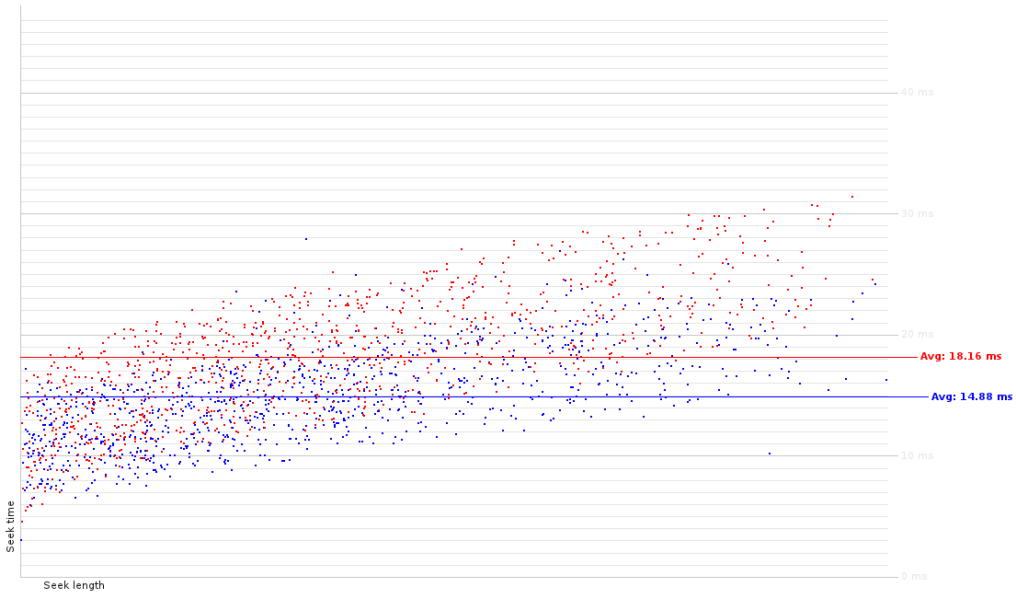


Figure 3.5: Comparison of TOSHIBA MK1652GSX (red) with Maxtor 6 Y080P0 (blue) in Seek. It shows how rotation speed of platters impacts seek performance. The Toshiba is 5400 RPM and the Maxtor is 7200 RPM.

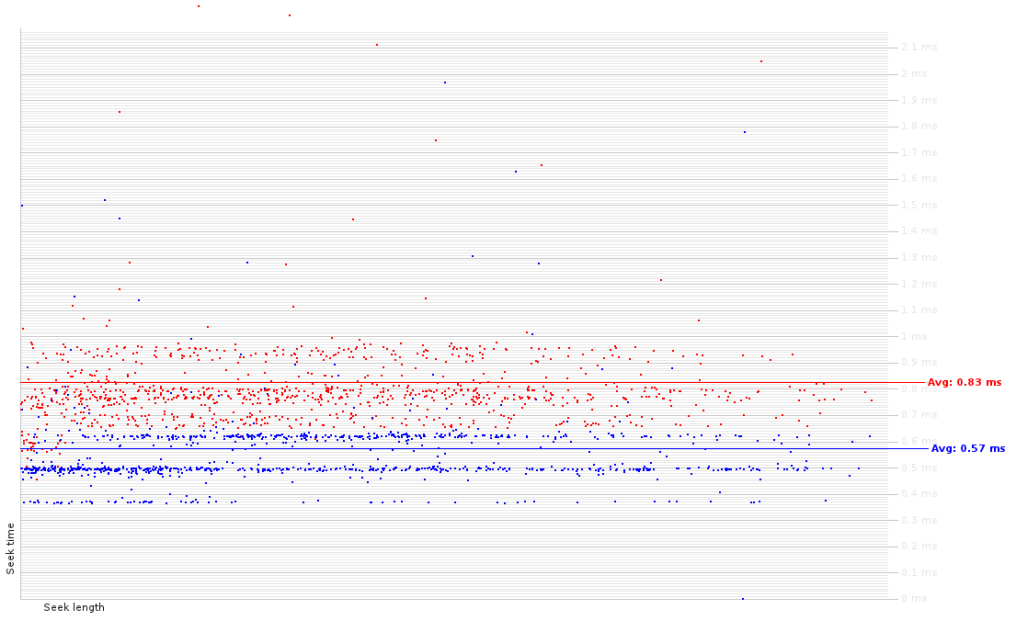


Figure 3.6: Comparison of KingMax 8G USB flash (blue) and 8GB TakeMS microSD (red) in Seek

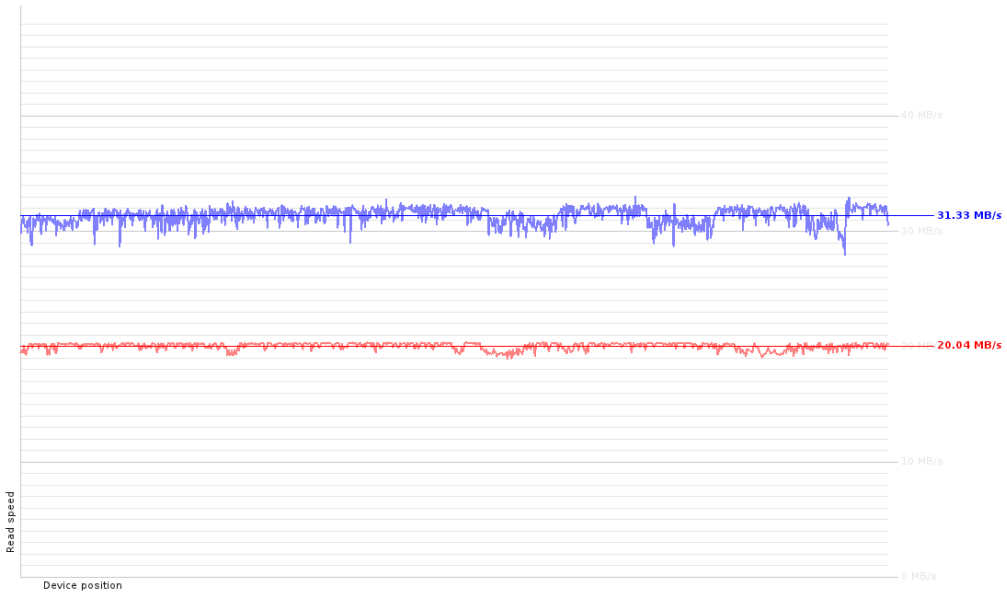


Figure 3.7: Comparison of KingMax 8G USB flash (blue) and 8GB TakeMS microSD (red) in continuous read

3.3.2 Reproducibility of results

These figures show how similar results measured by two passes of the same benchmark are.

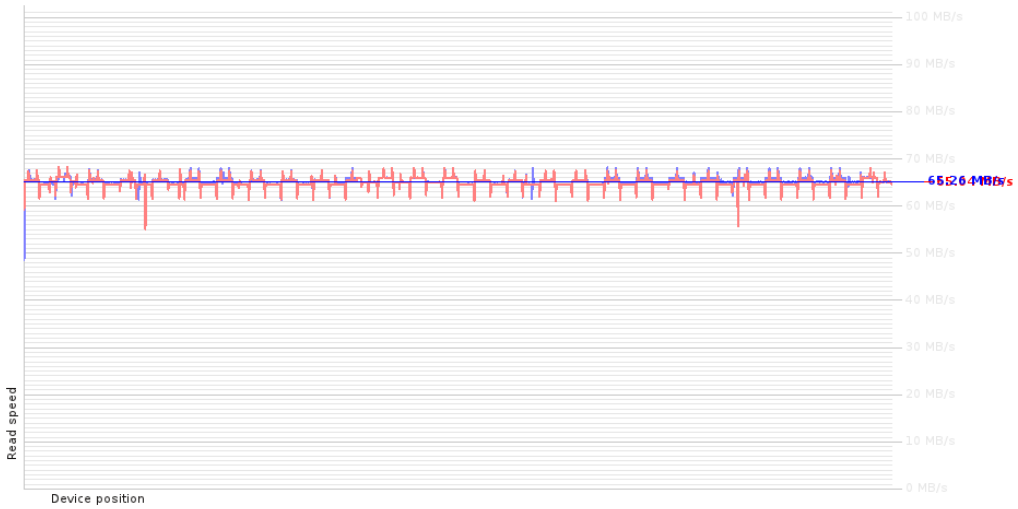


Figure 3.8: Two passes of continuous read with the Toshiba MK1652GSX. Most of the peaks on graph are the same.

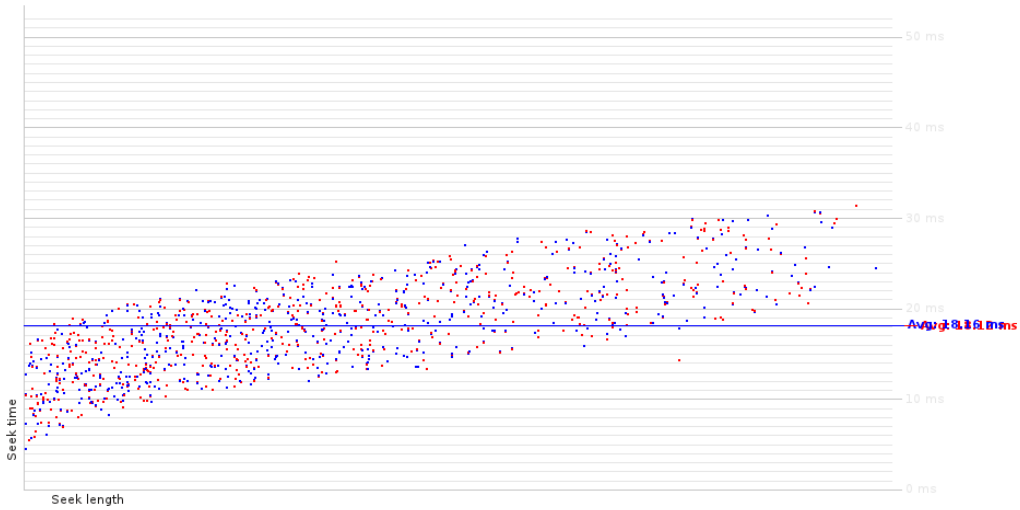


Figure 3.9: Two passes of seek with the Toshiba MK1652GSX. The dots are almost the same for both passes.

3.3.3 Interesting results

Two interesting things were found in the results. Even when discovering uncommon hardware behaviour is not the goal of this thesis, results showing this behaviour were included. The description of the results is based on the benchmark results and basic knowledge of hardware. Most probably the things are much more complicated than described here. The deep insight is beyond the scope of this thesis.

The first interesting result was measured with TakeMS 8GB microSD card. This media was used during development of the application for testing, so it was benchmarked many times. Some places on the card are every time slower than others. There is no pattern in distribution of such places. There are more possible reasons for this behaviour. Maybe some blocks takes longer time to be read, or more probably some blocks are bad and requests to them are mapped to spare blocks. The mapping then causes degraded performance.

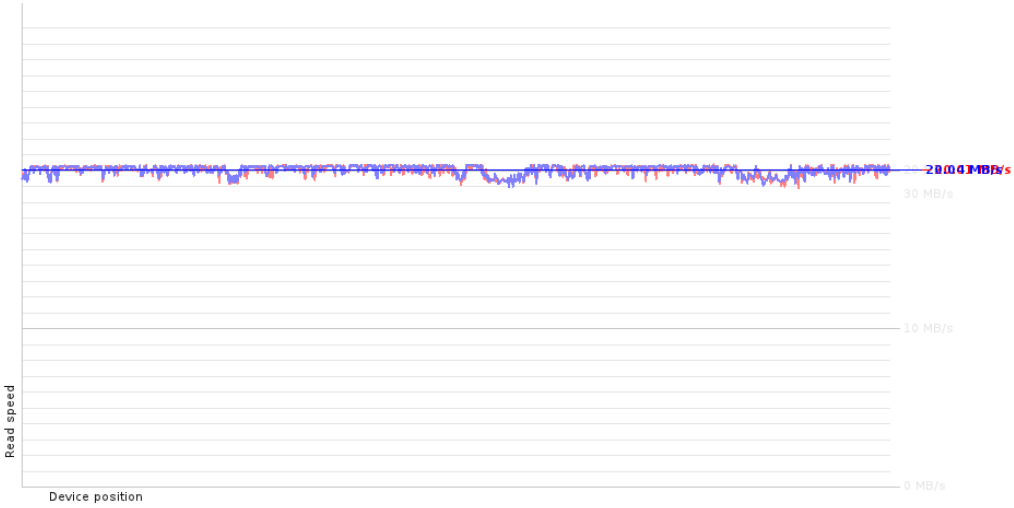


Figure 3.10: TakeMS microSD two passes - Continuous read. There are some parts of the graph where transfer speed was a bit lower. Interesting thing is that the parts are the same on both passes of benchmark. This means it is not a random phenomenon. Most probably some of internal flash blocks are bit slower than others.

The second interesting feature is demonstrated on KingMax 8GB USB flash drive, even when it was present on other two flash based devices. The "Random" benchmark reads blocks of different sizes from random positions on device. Classic rotating platters hard disc performs the better the bigger blocks are. It has to perform time consuming seek to next block position every time new block is being read. This causes almost linear dependence of transfer speed on block size, when suboptimal block size is used. When this benchmark is run on flash based device the situation is much different as seek times are much more smaller. Even when performance of flash device in this benchmark is much better it also shows degraded performance when blocks are small. The interesting thing is that reads are getting slower with blocks from 1MB to 128KB, then restores to maximum with 64KB blocks, and then getting slower again. This means that reading one

block of size 128KB is actually much slower than reading two 64KB blocks from different positions.

This seems to be related to internal flash block sizes. The blocks are read from random positions during the test. They are not aligned at all. When the benchmark aligns block position at their size the results is different. The block size at which the performance reverts back to maximum is not 64KB but 128KB and the whole effect is not as huge as it used to be.

This supports the idea that differences in performance are caused by possible relative positions between internal flash blocks and the benchmark blocks. If the internal block size is 128KB then this happens: Blocks larger than 128KB crosses internal blocks boundaries once or more times. 128KB blocks crosses it once with exception of those aligned at 128KB(this happens rarely). 64KB blocks will hit boundary in 50% of reads. Smaller blocks have even lower chance of hitting the boundary. When a boundary is hit extra internal block needs to be read.

When benchmark reads a block, all internal blocks needed have to be read. This means there is a smaller average overhead for really big blocks. Almost 100% chance of 100% overhead for 128KB blocks. And the bigger overhead the smaller than 128KB the blocks are.

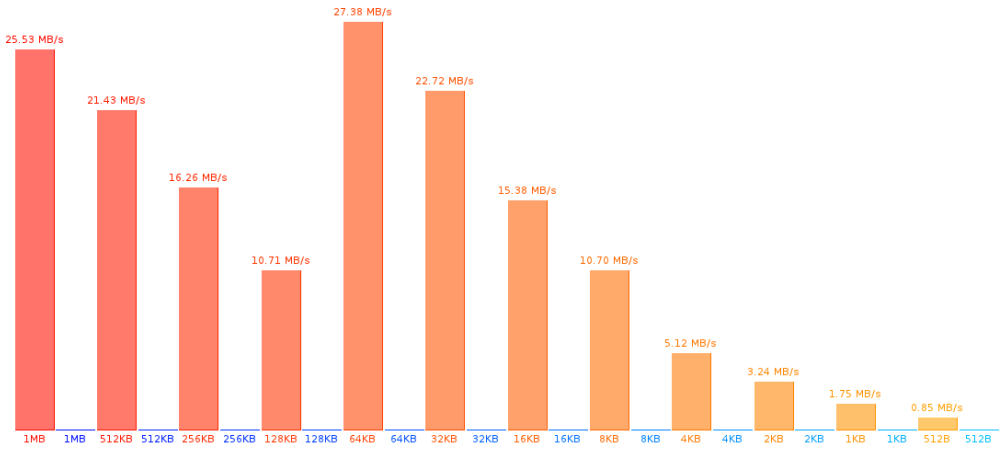


Figure 3.11: KingMax 8GB USB flash random block read shows interesting results. The performence do not follows the expectation that smaller blocks causes overhead thus degrade speed. The block benchmark shows equal performance on all block sizes when blocks are read continuously. This benchmark which reads blocks from random locations seems to be slower on smaller block sizes as expected. But it seems to revert full speed on 64KB block size. This beahvoiur is the same as with another USB flash and TakeMS micrSD card.

Conclusion

This thesis focused on three main targets. The first was implementing user friendly hard drive benchmarking software called HDDTest. The second was providing documentation in order to make the software easier to use by inexperienced users. The third one was measuring some results with the software and describing what is behind them in order to prove the software is usable. Even if documentation is important and writing good documentation is a challenge the software itself and measured results seem to be more interesting.

For more information on this thesis you can contact the author of this thesis vlada.matena@gmail.com.

Software

After a long development final versions of the HDDTest seem to be stable. Even when not intensively tested on different Linux distributions and hardware the benchmarks look to be safe to run. During early development just one incident happened when a partition was overwritten due to bug in HDDTest code. The problem was found and fixed, and measures were taken to make sure this will not happen again.

The most interesting features of the application are its simplicity and interactivity. These features are making HDDTest to be a good tool for an average user to compare drive performance or check whether the drive is working optimally. The lack of such a tool in the Linux environment was actually motivation for this thesis. The live view of running benchmark should make benchmarking process more interesting to user and simple graphs should explain drive performance in a simple way.

Results

The results, even when measured with limited choice of hardware, demonstrate intended usage of the software. Some interesting characteristics of tested devices were found. In order to clarify the results on overview of hardware and software layers dealing with I/O was included.

Possible extensions

There are several places where the HDDTest application can be extended or improved. Among others these improvements seem to be important.

Using udisks to handle device enumeration and identification would give the user better list of available devices. Moreover, identification of non ATA compliant devices would be supported.

Also more work could be done on device access code in general. Some more testing needs to be done in order to use best device access code available.

The matter that was not solved in this thesis is the way how to provide this application to public. The current deployment method expects HDDTest to be

built on the target system. This method is not acceptable for intended users of this software. In order to provide application to endusers licensing problems need to be resolved and packages for commonly used Linux distributions need to be made.

Bibliography

[1] OCZ Vertex Plus Series SATA II 2.5" SSD
http://www.ocztechnology.com/ocz-vertex-plus-series-sata-ii-2-5-ssd.html

[2] The Linux kernel documentation - Virtual file system
http://www.kernel.org/doc/Documentation/filesystems/vfs.txt

[3] The Linux kernel documentation - Block device
http://www.kernel.org/doc/Documentation/block

[4] Qt documentation - QtGraphicsView
http://doc.qt.nokia.com/latest/qgraphicsview.html

[5] HDTach
http://www.simplisoftware.com/Public/index.php?request=HdTach

[6] HDTune
http://www.hdtune.com/

List of Figures

Glossary

combobox Is a GUI element. It allows user to enter data to the application. The data can be chosen from dropdown selection or entered directly into editable field.. 8, 9, 11–14, 16

dd File copy utility. `http://www.gnu.org/software/coreutils/`. 16

GTK Cross platform widget toolkit for C programming language. 9

hdparm Commandline interface to kernel IDE and SATA APIs. `http://sourceforge.net/projects/hdparm`. 23, 24

losetup A tool for controlling loop devices in Linux system. It is part of util-linux package. `ftp://ftp.kernel.org/pub/linux/utils/util-linux/`. 16

QGraphicsView Is a GUI component provided by Qt which allows arbitrary graphics objects to be rendered. [4]. 3, 19

Qt Cross platform application and user interface framework targeting on C++ programming language. 2–4, 9, 11, 18, 19, 37

ReiserFS General purpose journaled filesystem. 25

System Rescue CD Live Linux distribution `http://www.sysresccd.org/`. 26

udisks Storage daemon with support for PolycyKit and D-Bus. 21, 33

XFS High performance journaling filesystem. 25

Acronyms

ATA Advanced Technology Attachment. 21, 24

btrfs B-tree file system. 25

CF Compact flash. 24

CPU Central processing unit. 10, 26

exFAT Extended File Allocation Table. 21

ext2 Second extended filesystem. 25

ext3 Third extended filesystem. 25

ext4 Fourth extended filesystem. 25

FUSE Filesystem in userspace. 21

GCC GNU Compiler Collection. 18

GUI Graphical user interface. 2, 4, 5, 9, 10, 18–20, 37

HDD Hard disk drive. 4, 7

IDE Integrated development environment. 18

KDE K Desktop Environment. 9

NTFS New Technology File System. 9, 15, 21

RAM Random access memory. 25, 26

RPM Revolutions per minute. 28

S.M.A.R.T Self-Monitoring, Analysis and Reporting Technology. 10

SD Secure Digital. 3

SSD Solid state drive. 7, 24

USB Universal Serial Bus. 3, 23, 24, 27, 32

XML Extensible Markup Language. 3, 13

Appendices

The CD attached to this thesis contains source code of software called HDDTest developed as part of this thesis, documentation generated from source code and electronic version of this thesis.

The content of the CD is following:

hddtest - HDDTest source code with GIT repository included

documentation - Doxygen generated documentation

thesis.pdf - PDF version of this thesis