Unintrusive performance profiling and testing in software development

University of Oulu
Department of Information Processing Science
Master’s Thesis
Janne Sakko
17.11.2014
Abstract

Performance is a complex topic in software development. Performance is a result of various interconnected properties of software and hardware. Risks and damages of badly performing software are well known and visible. Still, performance considerations are not thoroughly embedded to whole development life-cycle. Many projects start to consider performance only when issues emerge. Fixing performance problems at late phases of development can be very difficult and expensive.

When performance problems emerge, most important goal is to determine root causes of issues. Instrumenting software can be effective way to measure and analyse software, but if not implemented during development, it can be limited and laborious. Unintrusive software profilers don't require any modifications to the profiled software to be used. Profilers can provide various information about the software and the environment.

Performance testing aims to validate and verify that performance targets of a project are achieved. Regression testing is well known method for assuring that regressions are not introduced to the software during development. Performance regression testing has similar targets for performance.

This thesis explores usage of performance profilers and performance regression testing in UpWind project. UpWind is a sail boat chart navigation software project conducted in University of Oulu. Evaluation study in context of Design Science Research is used as a research method.

In this thesis navigation algorithm of UpWind project was profiled using OProfile and Valgrind profilers. Profiling provided new information about performance behaviour of UpWind project and also new insights about performance profiling. In order to prevent future performance regressions in UpWind project, performance tests and performance regression testing process were drafted. Performance tests were implemented using Qt framework's QTestLib.

Keywords
Software performance, performance profiling, performance regression testing, route planning algorithms
Foreword

I would like to thank my thesis supervisor, professor Samuli Saukkonen for his excellent guidance and advices during this process. I would also like to thank opponent Ari Vesanen for his expert critique, that helped to make this thesis better. Lastly, I would like to thank my family for their support during these years. I couldn’t have done this without you.

Janne Sakko

Oulu, November 17, 2014
1. Introduction

Performance problems seem to have persisted in software development for decades (Smith, 1990; McConnell, 2004; Molyneaux, 2009). Everyone in the software industry typically has some experiences about using or developing badly performing software. Whether it is a mobile platform that needs to be tuned and tuned again or travel billing system that keeps on loading for no apparent reason, everyone probably knows an application that’s a bit too ambitious for its resources. Excessive time, developers and money are spent on coping with and fixing badly performing software.

Performance problems affect the usability and image of software. Constant performance problems break the work-flow. Long response times cause frustration at minimum (Ceaparu, Lazar, Bessiere, Robinson & Shneiderman, 2004). Some performance problems have serious and expensive real-life effects (Molyneaux, 2009; Smith, 1990). Some performance problems may render software completely unusable for the purpose it was created. Software vendor’s reputation and reliability are at stake when poor performance of developed software is finally revealed in customers’ premises.

It would be important to understand what software performance actually is, how it’s measured and how to ensure acceptable performance level of software. Software performance might still not be fully or widely embedded to software development and it’s more of an afterthought. Software performance problems are usually detected late at the development life-cycle (Smith, 1990; Molyneaux, 2009) and fixing problems in late phases can be very expensive (McConnell, 2009). Software performance considerations should be taken in count already in planning of software among other quality attributes of software. But still many software projects rely solely on “fire-fighting”; fixing performance problems in delivered live environment (Molyneaux, 2009).

Software performance is a complex issue that has almost kind of special glamour around it. Only the most accomplished and technically savvy engineers and developers should dare to undertake performance improvement endeavours. Although that’s a coarse characterization, there’s undoubtedly some truth in those words. Indeed, often understanding of performance problems require well understanding of inner functions of hardware, specifics of developed software, third-party libraries, databases and specialities of underlying operating system (Liu, 2009). Even individual code changes - performance tunings - might require advanced knowledge of programming language specific techniques and how specific compiler processes source code to lower-level instructions (McConnell, 2004).

It can be very difficult and take a lot of time to fix complex performance issues after they are noticed. One reason for long fixing times are that root causes for performance issues are not always easy to determine and might be related to overall design and architecture of project. Not nearly all projects are designed with performance considerations and built-in performance measurement enablers (Smith, 1990; Molyneaux, 2009). Performance problems like other problems in software projects are usually unpredictable. Planning, estimation and assessing possible risks carefully can be very helpful in lessening the unpredictability (Smith, 1990). If project’s performance
problems are deeply rooted to the project’s architecture it can be very difficult and time consuming to fix them.

Open Source performance profilers are widely used, well available on Linux and highly useful in analysing performance problems in software development (Cohen, 2004; Novillo, 2005). Unintrusive profiling tools can help the developer to analyse program without having to do modifications to source code (Floyd, 2012). However, performance analysing tools are not perfect. Different tools use different methods for eliciting data and some tools are better in some cases than other tools. Usually there are trade-offs between tool functionality, accuracy and tool overhead. It’s important to understand limitations of different tools and suitability of those tools in different situations.

Performance degradations are a real threat and cause of worry in big software projects (Corbet, 2010; Bjedov, 2007; Baker, 2014). It’s well known that every new functional requirement, bug fix and maintenance work can introduce new, unexpected side effects and issues to software (McConnell, 2004). It’s customary to evaluate functionality and quality targets of project frequently in software evaluation phases in project’s life-cycle. Different project management methods from water-fall to various agile systems assess the project goal evaluation with different frequencies and with different corrective approaches. Regression unit testing for functional targets is very well known approach to assure that no functional regressions creep in during development or maintenance (McConnell, 2004). Unit testing is used to test individual elements, like functions of software. Same kind of regression testing for performance targets is not so well established or adopted method of quality assurance (Horký, Haas, Kotrč, Lacina, and Tůma, 2013).

This thesis tends to research usability of unintrusive performance profiling tools in analysing performance properties of software. Although, there’s vast amount of profiling tools available, the selection of tools is guided by practicability, licenses, popularity and eventually preferences of author. Performance profiling tools are used to analyse properties of route planning algorithm. Revealed properties of the algorithm will help to guide further performance work and also work as an example of successful usage of profiling tools.

Also, viable option for performance regression testing using Qt Test framework is introduced. There doesn't seem to be much previous literature about performance testing with Qt Test framework so this thesis will be an exciting introduction to a subject. This thesis shows that writing performance tests with Qt Test framework can be effective and beneficial for projects developed with Qt.

UpWind is a software project developed in Oulu University in the Department of Information Processing Science. Project’s main target has from the beginning been solving sail-boat navigation problems. Even though UpWind project has provided functionally satisfactory solutions for navigation, there are still some persisting quality issues in the program. The problems are most prominently related to time performance of calculating the short-term route (Kangastalo, 2013). It's expected that the system should be able to calculate the short-term route whilst the vessel is moving. In other words, the system should be usable for real time short term navigation.

UpWind project is used to explore introduced performance profiling and testing topics. UpWind is a perfect example of a project with a long development history and known performance problems. UpWind is fairly large and complex program with a database
and various dynamic properties. UpWind project is more thoroughly introduced in chapter 4.

This thesis follows the following structure. In second chapter research questions and research method are introduced. Third chapter gives a brief introduction to software performance, different profiling techniques and software performance testing. In fourth chapter performance profiling tools and testing are used to evaluate UpWind short-term route planning algorithm. In fifth chapter answers to research problem are explicitly stated and sixth chapter presents a conclusion of the work.
2. Research problem and method

In this chapter research problem and sub-problems are defined. Also, chosen research method is presented and justified.

2.1 Statement of research problem

Performance problems are common in software development and in order to effectively solve problems it’s important to analyse their root causes. Performance profilers provide interesting possibilities to analyse performance problems and behaviour of developed software without modifying software. Performance testing provides also a way to analyse software and verify that performance targets are met. The research problem of this thesis comes from these two realisations.

The research problem in this thesis can be defined as follows:

- How unintrusive profiling and performance testing can be used in evaluation of software artefact to produce new information about performance properties of the artefact and improve performance evolution of the artefact in development.

Research problem presented here is quite complicated with multiple possible approaches. It’s important to get a grasp on the basic tenets of software performance to acquire better understanding of complexities of the quite large subject. The focus of this thesis is to produce applicable information for software testing in an area of applications that fulfils a certain criteria. This thesis mainly studies practical applications of software performance analysis and tests on local, single-user applications.

Software performance evaluation through profiling and testing is quite a huge topic. In the next chapter evaluation study method is introduced to answer this research question in more manageable way.

2.2 Research method

This thesis aims to correspond to real problems encountered in software development of UpWind project. Performance problems are also not generally unknown to software development projects. Generalized idea of this thesis is to find techniques to ensure constant performance quality of software project in development and practical ways to elicit performance information about the developed software. Techniques for performance evaluation are used in evaluation of UpWind project.

Design Science Research (DSR) tries to solve human problems with creation of innovative artefacts (Hevner & Chatterjee, 2010). Designing, building and evaluation of meaningful artefacts are the key activities in many Design Science processes (Offermann, Levina, Schönherr & Udo Bub, 2009; Hevner & Chatterjee, 2010). The artefact evaluation study is a crucial of DSR (Hevner and Chatterjee, 2010). The solution that the artefact provides must be evaluated with relevant and measurable metrics (Hevner et al., 2004; Peffers et al., 2007). Since this thesis aims to evaluate a designed and implemented artefact, evaluation study in DSR context will be a proper approach.
Evaluation can cover technical aspects of an artefact or socio-technical aspects such as usefulness and organizational impact (Hevner and Chatterjee, 2010). Furthermore, according to Hevner and Chatterjee (2010) evaluation assists to understand the created designs more deeply and add to the body of knowledge for future designers to learn from. This thesis aims to answer to both technical and socio-technical aspects of software development. Technical side aims to bring more information about performance behaviour of UpWind system using profiling and testing as evaluation methods. In a socio-technical side results can be used to focus future work on UpWind project and also improve performance evaluation practices in the project. Also, applicability of different profiling tools and tests for UpWind project is evaluated. Performance is often denoted as a key criterion in the software design and design science researcher must have a basic understanding of subject (Hevner and Chatterjee, 2010).

Hevner and Chatterjee (2010) introduce basic structure for evaluation studies. First, designer or other stakeholders need to find out how the designed and implemented artefact performs. Typically, set of questions describing what is needed to know about the artefact and how the evaluation is conducted are drafted in negotiation with different stakeholders. Next, in the investigation phase, answers to the questions are searched through analysis of measurements and experiments. Finally, the report is created for interested parties that answers the questions through collected data. (Hevner and Chatterjee, 2010).

This thesis follows broadly the structure of an evaluation study presented by Hevner and Chatterjee (2010) applying presented research problem to UpWind project. In first phase, questions for evaluation are formulated:

- How to reliably measure performance of software?
- What attributes of program affect the performance?
- What kind of information can be elicited from software using profiling tools?
- What limitations performance profiling tools have?
- How to ensure that performance quality doesn’t drop during development?

These questions are answered in context of UpWind but are generalized when possible. In the second phase, required understanding of software performance subjects are acquired through existing literature. This aims to bring the evaluation study into a proper theoretical framework. In phase three, evaluation techniques from literature study are applied to UpWind project. Final phase is to report results in a form of this thesis.

### 2.3 The UpWind artefact

According to Hevner & Chatterjee (2010) IT artefacts can be broadly defined as constructs (vocabulary and symbols), models (abstractions and representations), methods (algorithms and practices) and instantiations (implemented and prototype systems). In this study the artefact under evaluation is instantiation of the UpWind project. Only source code of UpWind project is considered as an artefact in this thesis. In this thesis it’s defined so that the source code of UpWind contains all the modifications relevant to the performance and quality of the program. An artefact under development is an evolving creation. The UpWind artefact is a snapshot of the code
base in a certain point of history. Snapshot contains implemented performance tests and evaluation information about the project.

Since the code base and its history are stored using Git version control system, the appropriate term for snapshot is Git tag. Tag has a special meaning in the Git version control system\(^1\). Tags represents certain evolution points in a history of software that are of some significance like version release points\(^2\).

\(^1\)http://git-scm.com/docs/git-tag
3. Software performance

In performance work it’s important to understand what to measure as well as how to measure reliably. What are the factors that affect the performance of software? In this chapter different aspects of information system performance are identified based on the reviewed literature. Different methods to acquire performance metrics from information systems are introduced. Also, different approaches to achieve good performance in information systems are introduced.

3.1 Introduction to Software performance

Software performance seems to be one of those areas in software development that only gets attention when there are visible problems. Much like other software testing endeavours performance testing have been more of an afterthought in projects; only to be done late in project’s development cycle, when the program is functionally complete. Software performance is most visible to the end user of the product. Performance engineering is mostly thankless work; good performance is usually forgotten, but bad will be remembered. Bad performance can cause problems from minor irritation to totally unusable user experience. Bad performance at the smallest disrupt the work-flow of user. At worst it makes using the program for the work it was supposed to impossible. Almost everyone in the software industry seems to have an anecdote or a few about using or developing a product with suboptimal performance.

From where all these badly performing programs come from? Why performance problems are still so usual in the software business? In the early years of computing programs needed to be carefully crafted to fit in resource restricted systems (Smith, 1990). Modelling performance of software was labour intensive, difficult and very costly and thus it was left only to most critical systems while other programs adopted “fix-it-later” method (Smith, 1990). In the early years - 60’s to 70’s - results of “fix-it-later” were acceptable due to smaller scale of programs and software developers had attitude and skills building programs with adequate performance (Smith, 1990). After those years programs have scaled up and grown more complicated multi-layered designs with databases and networking. Attitude of “fix-it-later” is just not enough any more (Smith, 1990).

Smith (1990) presents misguided beliefs of early years that led to degradation of performance testing: performance problems are rare, problems can be compensated with hardware, building for performance is expensive, software can be tuned later and efficient designs tend to make “tricky” code. These beliefs seem to still plague projects of current days (Reddy and Kasi, 2014). Lack of performance considerations in design and late performance testing seems to contribute a great amount of performance problems in already deployed programs (Molyneaux, 2009). Often performance problems stem from designing software that doesn’t exploit the available hardware such as multi-core systems (Reddy and Kasi, 2014). Molyneaux (2009) further lists reasons for unexpected performance problems: system’s lack of capacity to handle all simultaneous clients, automated test tools are not used or are found to be incompatible for developed software, and performance testing is still informal discipline.
It has been shown many times in software development that fixing performance problems - or usually any software problem for that matter - gets ever more costly at the latter stages of development cycle (McConnell, 2004; Smith, 1990; Molyneaux, 2009). Performance problems that are detected late in the development can seriously affect already planned time estimates and delay delivery of project (Smith, 1990). Late tuning or optimization of project is time consuming, difficult and expensive, and can lead to complex, hard to maintain code (Smith, 1990; McConnell, 2004). Performance problems in deployed software cause increased maintenance needs with critical fixes in the live environment (Molyneaux, 2009). Functionally otherwise complete, but badly performing software can cause end-users to acquire negative attitude towards the software (Smith, 1990; Molyneaux, 2009). Even after the performance problems are fixed, end-users might hold on to their initial negative attitudes toward the software (Smith, 1990).

What is actually meant by performance and bad performance in particular? Performance is usually associated with speed of program’s execution; the system responds in timely manner to user’s requests. Bad performance usually manifests as slowness and unresponsiveness of program’s user interface. Well performing program lets user carry out user's tasks without undue perceived delay and irritation (Molyneaux, 2009). In software projects performance requirements are considered to be non-functional or quality requirements (Bourque & Fairley, 2014). Performance doesn't describe what tasks software should do but how those tasks should be executed. Performance requirements define constraints for acceptable quality level of program’s functional requirements with other non-functional requirements, such as maintainability, reliability, security, safety and interoperability (Bourque & Fairley, 2014).

Performance can also be used to refer memory footprint or amount of disk space that an installed software requires. Designs and architectures that increase performance of some attribute might have negative effect on another performance attribute. For example, by loading database assets to memory in advance so that they can be used without resorting to expensive IO calls to disk when the assets are needed, can increase the execution speed of software but also grow the memory footprint. These trade-offs with speed of execution and other software attributes should not come as a surprise in the latter phases of the development. Trade-offs between different design choices should be well known in advance when software is being designed. This assures that correct architectural choices are made respecting the individual software requirements.

3.2 Software performance testing

Software testing is essentially a process for finding situations in where the software behaves differently from specifications (Hailpern and Santhanam, 2002). When software project's performance doesn’t meet the specifications, it’s important to locate the sources of problems. This might seem as a daunting task as gathering and analysing performance data needs a lot of effort and knowledge. The sheer amount of moving interconnected parts of a software project can result a staggeringly complex creations. How to reliably spot bottlenecks in systems that have multiple components, layered in multiple tiers and connected with various ways? Which measures are relevant to the software performance? What data is important in solving and predicting performance problems? What are the suitable units of measure that represent the performance properties of a program? How those measures are extracted in reliable and rigorous manner? These are few of the questions faced in the performance work and discussed in this chapter.
3.2.1 Data of interest

Smith (1990) in book Performance Engineering of Software Systems introduces metrics that are pertinent in modelling, designing and evaluating software. Software Performance Engineering leans heavily towards modelling of software performance in early phases of software development cycle. Already in the early phases of the software project it’s important to plan for collecting performance and other required measurements from the software. Smith (1990) supports using instrumentation techniques to capture required measurements from the software, since existing tools might not be able to collect all measurements that are important for the developed software. The metrics to acquire according to Smith (1990) are workload data, data characteristics, execution characteristic and computer system usage.

Performance engineer should acquire information about the expected workload data for modelling. Also, data operation aspects of the software should be estimated. Workload data refers to workload caused by handling of data in different components; how frequently those components are used and how saturated they get. These components can be for example functions in program or calls to database management functions. Data characteristics refer to attributes of handled data items which should be measured. Attributes of data include for example total amount of data and size of each data item. Also, how frequent data is requested and locality of the requests can be useful information. (Smith, 1990).

According to Smith (1990) there exists three types of execution characteristics that can be measured: resource requirements, support software services and path characteristics. Execution characteristics help to model and estimate resource usage of the developed software as well as evaluating and validating implementation of software. Resource requirements refer to usage of computer system’s limited resources, such as CPU, I/O and memory. Resources used by support software should be measured if the program uses other supporting software extensively. For path characteristics one should count how many times major branching paths are executed in program during run. (Smith, 1990).

According to Smith (1990) computer system usage can be used mainly to evaluate performance and to identify bottlenecks in the processing. For computer system usage one should measure:

- The scenario response time - the elapsed time from scenario request until its completion (the actual service time and the time spent waiting in queues)
- Scenario throughput - the number of scenarios processed per time unit
- Key computer system resource usage by each workload scenario - the actual service time as well as the wait time
- Resource utilization - percentage of time busy
- Resource throughput - the rate at which the resource completes service requests
- Resource queue lengths - the average number of jobs waiting for service

(Smith, 1990)
3.2.2 Software performance metrics and measurements

Measurements in software development are used to verify and validate implementations and designs, and also locating fixing hotspots in the code (Reddy and Kasi, 2014). Quality requirements for software performance measurement doesn’t really differentiate from any scientific measurement (Smith, 1990). Performance measurements can be collected in highly controlled laboratory setting or in live environment (Reddy and Kasi, 2014).

Reddy and Kasi (2014) introduce best practices for measuring performance:

- Performance measurements should be planned so that results are both representative and repeatable. Representative means that tests must be designed so that only important metrics are collected to describe accurately the phenomena and to reduce the overhead for processing the data. Repeatability means that variables in tests must be controlled to produce similar measurements every time.

- Code should be instrumented for performance measurements. Smith (1990) also suggests using instrumentation.

- Critical components of developed system should be measured early and often to validate models and verify predictions. Models should be created meticulously not to omit any important factors affecting the performance. (Reddy and Kasi, 2014)

Software performance metrics relate to the performance targets of a software project. Different purposes and different goals of programs require different measurements to acquire meaningful performance data. If software intends to service customer as fast as possible service time and perhaps availability are the most important metrics to measure. If on the other hand software is supposed to apply operations to a large amount of data items as fast as possible, throughput might be the correct metric. This kind of metrics are called *key performance indicators* (KPI) (Molyneaux, 2009) since they are essential in describing if performance targets of a project are met. The metrics chosen must be descriptive towards the goals of the software and also metrics must be quantifiable (Smith, 1990). Quantifiability of performance measurement is of most importance in successful evaluation of performance (Liu, 2009; McConnell, 2004; Molyneaux, 2009).

Molyneaux (2009) divides indicators to two types: *service oriented* and *efficiency oriented*. Service oriented indicators measure how well the program services the user and efficiency oriented indicators measure how well the program uses it's running environment. Service oriented indicators are availability and response time, and efficiency oriented throughput and utilization. (Molyneaux, 2009) Liu (2009) also divides performance indicators to two categories: interactive user activities are measured as a response time and non-interactive batch jobs as a throughput (Liu, 2009).

*Response time.* Response time refers to elapsed time of user initiated, interactive tasks (Liu, 2009; Smith, 1990; Molyneaux, 2009). Response time can refer to total elapsed times of end-to-end user tasks, or to elapsed time of individual transactions (Smith, 1990). End-to-end user tasks might contain multiple transactions (Smith, 1990).
Focusing on end-to-end times helps to evaluate effectiveness of the system (Smith, 1990).

**Availability or uptime.** Availability is percentage of time system is operational (Smith, 1990; Molyneaux, 2009). Mostly concerns web servers and other services that are not hosted on local computers. Unavailability of the services must be minimized to planned maintenance downtimes only (Molyneaux, 2009). Availability must not suffer under increased stress (Molyneaux, 2009).

**Concurrency.** Usually concurrency is used to refer amount of simultaneous users the program must be able to handle without slowing down, but it’s not always that simple. When talking about actual concurrency, one must note that not all users in the system are active all the time (Molyneaux, 2009). There are different types of tasks that take different amount of time and use system resources differently (Liu, 2009). Hence, when deciding actual concurrency targets for a system, it’s good to know how the end-users will use the system.

**Scalability.** Scalability is a property of software, very closely related to performance. Scalability describes the trend of software performance with increasing load on the software (Liu, 2009). In related to response time: good scalability means that the response times of software grow only moderately and stays stable when load to the system increases (Molyneaux, 2009). Performance and scalability of software are tightly connected; scalable software continues to perform acceptably even in high loads but addressing scalability of badly performing software makes no sense (Liu, 2009).

If performance of the software starts to deteriorate quickly with increasing load, before the intended load capacity is achieved, then there is a scalability issue (Liu, 2009). Liu (2009) further divides scalability issues to two types: type 1 can be remedied by software and hardware tunings and type 2 cannot be. Obviously, type 2 scalability issues should be avoided at any cost during the design and development. Fixing them post development can lead to major and expensive architectural changes. (Liu, 2009)

**Throughput.** Usually throughput measures how many transactions the system is capable of handling in a time duration (Liu, 2009) or how fast the system handles a number of transactions (Molyneaux, 2009). Molyneaux (2009) uses capacity to measure how many transactions are handled in a given time duration. Throughput can be used in context of non-user-interactive batch-jobs (Liu, 2009) and also in context of user transactions (Molyneaux, 2009). For example, how many user transactions the system can handle in a time period.

**Resource utilization.** Different resources in the computer system are usually scarce and need to be consumed moderately. Utilization measures how much of the resources total capacity is used (Molyneaux, 2009). For example CPU utilization measures the usage of CPU resource in a given time and memory utilization how much of system’s memory is used by different programs. Monitoring resource utilization can really beneficial in finding bottlenecks in the system (Liu, 2009).

As noted at the start of this chapter, these measurements are important in setting performance targets of the developed software. Table 1 aggregates measurements introduced in this chapter and suggests template questions for helping to understand different design implications of different performance targets.
### Table 1: Key performance indicators

<table>
<thead>
<tr>
<th>KPI</th>
<th>Questions</th>
<th>Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability</td>
<td>How much time the system should be operational in a year? Or in a month? What is accepted downtime for the system? Does maintenance operations require downtime and how much?</td>
<td>Percentage of operational time.</td>
</tr>
<tr>
<td>Concurrency</td>
<td>How many concurrent users there are going to be? How many of them are going to be active on any given time? How long are concurrent users' average thinking or idle times?</td>
<td>Number of users.</td>
</tr>
<tr>
<td>Response time</td>
<td>How much time one user transaction takes? How long the user is willing to wait?</td>
<td>Time in appropriate unit.</td>
</tr>
<tr>
<td>Resource utilization</td>
<td>How much the program is going to use the system's resources? Is the program particularly CPU/memory/network/etc. intensive? Do some transactions or users consume considerably more resources than others? How to prioritize usage of scarce resources?</td>
<td>Percentage of used CPU/memory etc.</td>
</tr>
<tr>
<td>Scalability</td>
<td>How well the system performs when the load is increased? At what point response times grow too long when system load is increased? What is acceptable rate of slowing performance when system load is increased?</td>
<td>Response time in relation to increased system load.</td>
</tr>
<tr>
<td>Throughput</td>
<td>How many transactions the system must be able to handle in a given time? How fast system should handle a number of transactions?</td>
<td>Number of transactions in a time duration.</td>
</tr>
</tbody>
</table>

#### 3.2.3 Performance testing types

There are several types of performance tests for different purposes. Performance tests are useful for validating and verifying program’s performance, searching performance hotspots and assessing performance of third party solutions. Before starting tests the code-base should be stable enough to be tested for performance (Molyneaux, 2009). Buggy and unfinished code is somewhat useless to test for performance because bugs can perturb validity of tests and even prevent executing tests (Molyneaux, 2009). Code freeze during performance testing helps to avoid changes to the code-base that could affect performance results during test iterations (Molyneaux, 2009). Also, as performance work should consist the whole life-cycle of project, conscious decision should be made to improve tools and methods of testing to acquire more and more accurate and useful results (Molyneaux, 2009).

Similar to functional regression testing for program functionality tries performance regression testing catch possible performance problems caused by new modifications to the software. Performance optimization and tuning testing seeks to find places for performance improvements in software and offer recommendations how to fix performance problems. Performance benchmark testing is done on the system that mostly represents the situation of the customer. The purpose is to showcase the performance of program that fulfils customer’s needs. Scalability testing seeks to find limits of the program. What volume of transactions or users the program can handle without becoming unusably slow. When the product is deployed and taken in use there can always be unprecedented spike of users or customer eventually may eventually see growing amounts of users. Scalability testing helps to see how the program behaves in those situations. (Liu, 2009)

Baseline test is used to establish the default level of program’s performance. Baseline test can be performed in environment with optimal settings yielding best case performance. Degradation of performance in subsequent tests can then be compared to program’s optimal performance. Load test is used to validate that program’s performance targets of response time, throughput, scalability etc. are met with realistic amount of concurrent users. Stress test aims to find upper limits of load until the performance degradation grows to unbearable or the program, or some part of it, all together crashes. Soak or scalability test monitors the long time performance degradation rate of the program. For example, memory lead can gradually over time cause program’s performance to deteriorate. Smoke test seeks to test performance regressions caused by code changes. Only transactions that could be affected by code changes are tested. Isolation test is used to test only those transactions affected by performance problems in order to find the cause to performance problems. (Molyneaux, 2009)

Table 2 presents a mapping from performance categories by Liu (2009) to performance test types by (Molyneaux, 2009).

<table>
<thead>
<tr>
<th>Category (Liu, 2009)</th>
<th>Performance test (Molyneaux, 2009)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regression testing</td>
<td>Smoke test</td>
</tr>
<tr>
<td>Optimization and tuning</td>
<td>Isolation test</td>
</tr>
<tr>
<td>Benchmarking</td>
<td>Load test, Baseline test, Soak test</td>
</tr>
<tr>
<td>Scalability</td>
<td>Stress test</td>
</tr>
</tbody>
</table>
3.2.4 QTestLib

Particularly interesting possibility for benchmarking Qt programs are tools provided by Qt framework. The Qt framework provides comprehensive unit testing framework that also includes benchmarking options in its Qt Test module. As a part of the Qt framework, Qt Test is ideal for testing programs built on Qt components. It directly supports special Qt types, signal and slot system, GUI testing, is extensible by custom types, and provides atomic and thread safe error reporting (Qt Test Overview, 2013).

With Qt Test module it’s possible to create benchmarking scripts that are separate from the main program. In the script the benchmarked part of the code and possible input values are defined. The code to be benchmarked is inserted inside QBENCHMARK macro. This set some limitations to what parts and properties of the separate main program can be benchmarked. Benchmarking part of the code that is self-contained, like functions, is well supported by Qt Test module. Parts of the program that are not easily separable from the main program may need some other methods for benchmarking, like manual instrumentation. (Koehne, 2014) Of course, benefit of segregating benchmark scripts from the actual source code rather than implementing instrumentation directly into the source code is better maintainability. Also, then benchmark tests can be ran separately from the actual program. This functions as an enabler for a convenient automated performance regression testing.

Qt Test module’s benchmarking option provides multiple back-ends depending on the platform where the benchmarks are run presented in the Table 3.

Table 3: Available back-ends for QTestLib benchmark option. (Qt Test Overview, 2013)

<table>
<thead>
<tr>
<th>Back-end</th>
<th>Platform availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wall-time</td>
<td>All platforms</td>
</tr>
<tr>
<td>CPU tick counter</td>
<td>Windows, Mac OS X, Linux, UNIX</td>
</tr>
<tr>
<td>Event counter</td>
<td>All platforms</td>
</tr>
<tr>
<td>Valgrind Callgrind</td>
<td>Linux</td>
</tr>
<tr>
<td>Linux perf</td>
<td>Linux</td>
</tr>
</tbody>
</table>

To effectively use Qt’s benchmarking option, one must understand the specialities of the different back-ends. Different back-ends are different in how they operate and what results they provide. Some back-ends require multiple iterations over the benchmarked section in order to acquire reliable results. Wall-time is the default back-end and it’s based on Qt’s QTime component. Wall-time and Valgrind Callgrind back-ends provide execution time of benchmarked section as milliseconds. Wall-time possibly requires many iterations to be useful. On the other hand Valgrind gives exact results, but doesn’t contain I/O waits. Linux perf back-end counts CPU cycles. Event counter counts events originated from Qt program and the system. (Qt Test Overview, 2013). Results of the benchmark can also be exported to a spreadsheet in order to use results in other statistical tools (Koehne, 2014).

Valgrind is introduced in chapter 3.4.2
Unfortunately scientific literature and references considering Qt for testing or benchmarking seems to be greatly lacking. Informal search of papers that would research or use Qt Test module revealed no relevant results. Electronic databases of ACM, IEEE, EBSCO, Springer Link and ScienceDirect were searched for papers containing strings “Qt Test” or “QTestLib” or “QtTestLib”. Search strings “Qt Test” and “QTestLib” refer to the latest version of Qt 5 and “QtTestLib” was used in earlier versions of Qt. Since not all databases are dedicated to computers and engineering, filters were used to include matches only from only engineering, software engineering and computer science fields where appropriate. Different databases use different category conventions and those filter settings had to be modified for every database individually.

The search produced 8 matches in total. 7 scientific papers and one book. All papers were individually inspected, since there were so few results. Only in one of the inspected scientific papers Q framework was used but had no testing or benchmarking related material. The book contained section about unit testing with Qt but nothing about benchmarking. Based on the literature, no conclusion can be drawn about reliability of Qt Test module’s benchmarking option. Due to lack of scientific papers about the subject of Qt Test module’s benchmarking option it has been mandatory to rely on Digia’s documentation (Qt Test Overview, 2013) and blogs (Koehne, 2014) about Qt as well as our own tests and experiences.

Even though papers using Qt Test module’s benchmarking option couldn’t have been found, functions of Qt library can be used in an instrumentation of code. Paper by Pieterse, Kourie, Cleophas and Watson (2010) describe a method for benchmarking different bit-vector implementations with instrumentation using Qt library. The code signals starting point and ending time of the measured function. Qt library class that is used to measure used time is QTime. (Pieterse et. al, 2010). According to Pieterse et al. (2010) this approach to measurement results only insignificant performance overhead. In master thesis by Kangastalo (2013) certain navigation algorithms of UpWind project were instrumented using Qt library’s QElapsedTimer class. Also, Qt library’s official documentation suggests that QElapsedTimer should be used in benchmarking the code (Koehne, 2014) supporting the approach chosen by Kangastalo (2013).

### 3.3 Performance analysis

The Pareto Principle is very well known idea that can also be applied in software engineering. In general Pareto Principle states that 80 percent of results can be achieved with 20 percent of work (McConnell, 2004). According to McConnell (2004) and Weyuker and Vokolos (2000) this is also found to be true with software performance: small part of the program used largest amount of the program’s total execution time. This is an important guide to remember when targeting performance improvement efforts in projects with limited resources.

Performance analysis is observation of program’s behaviour in execution to locate possible performance hot spots (Bourque & Fairley, 2014). Much of the software performance work is already laid out in choosing appropriate designs, architectures, algorithms and data-structures for specific problems (Smith, 1990; Cormen, Leiserson, Rivest & Stein, 2009). It’s possible to detect inefficient algorithms by careful code inspection. Analysis of algorithms used in software can help in this (Cormen et al.,

4http://qt-project.org/doc/qt-5/qelapsedtimer.html#details
Improper design choices and architectures can be founded by comparing current designs to performance anti-patterns (Smith & Williams, 2000; Smith & Williams, 2002).

The next section introduces different data gathering techniques that are divided to four categories: profiling, code analysis, monitoring, instrumentation. The categories provided in this chapter are informal and can be quite broad and occasionally overlapping. Moreover, this is not a strict or bounding segregation of categories but rather an effort to describe different possible aspects to performance data gathering and analysis. Actual available tools for data gathering are presented in the following chapter. One must note that one data gathering tool may sit on multiple categories by providing multiple techniques for eliciting data.

As being discussed, there are multiple ways to collect and analyse information about program’s performance. For purposes of this thesis, it’s sufficient to concentrate only on methods that apply to functionally complete programs. For example, various available performance modelling techniques, that are suitable for programs in pre-implementation and major re-factoring phases are intentionally left out from this thesis.

As program’s performance depends of various factors from program’s software to environment where it’s ran, there are many sources of possible sources of performance issues. As a common source of performance inefficiency McConnell (2004) lists input/output operations, memory paging, system calls of operating system, interpreted languages and plain errors in the program. Also, poor design choices, or anti-patterns, can have negative effect on program’s performance (Smith & Williams, 2000; Smith & Williams, 2002).

Performance and scalability of the software doesn’t only depend on qualities of the software itself but on the qualities of the underlying software and the hardware platforms on where the software is executed (Liu, 2009). Hardware platform factors that can affect the performance of the software are CPU architecture, amount of CPUs, power of individual CPU, support for multi-threading and vendor independent technologies like Intel’s hyper-threading, amount and speed of RAM, read/write speed and storage capacity of HDD or SSD units, usage of hardware or software RAID controllers, properties of the motherboard etc. Software platform factors that might affect the performance are currently used operating system, hardware drivers, database programs, concurrently running or assisting programs etc. Interestingly enough, certain software licenses can place performance restrictions on the software, encouraging user to purchase more expensive licences to unleash the full performance potential of the program. (Liu, 2009).

The performance of program is thus highly dependant on hardware and software platform where it’s executed. It’s important to understand the functionality of tested program in order to choose the appropriate hardware components for program’s performance requirements. For example, if tested program doesn’t contain any threading support or actions it performs are mostly executed on a single CPU core, it doesn’t make sense to add multi core hardware with low single CPU core frequencies. Also, when measuring performance, this interconnectedness of tested program and hardware and software platforms should be considered in interpretation of the measurements. (Liu, 2009)

More complex systems might consist of separate components and thus introduce additional performance considerations in form of data transfer through different
networks. An example could be a multi-tier network systems that contains separated database, application logic and user interface layers. (Liu, 2009).

### 3.3.1 Profiling

Profiling tools are the most common performance analysis tools used (Bourque & Fairley, 2014). Execution profilers monitor the program’s code when it runs and collect information how much time is spent on different statements and execution paths (McConnell, 2004). Profilers usually work by inserting a piece of code at the start and end of every function, which enables profiler to record statistics of how many times the function was called and how much time it used. This can also be done manually by programmer. Inserting code to the program obviously lengthens the program’s overall execution time. (Atkins & Subramaniam, 1996). Although profilers may provide many other properties, in its barest profiling can be seen as automating the manual instrumentation done by the coder.

### 3.3.2 Code analysis

Static code analysers inspect the code when the program is not running and make estimations on where there might be possible performance problems and also give advices how to avoid them. Static code analysers don’t inspect the code of other libraries and/or calls to OS routines which may actually be the greatest cause of performance problems. (Atkins & Subramaniam, 1996).

### 3.3.3 Monitoring (sampling)

States of the executing program and system can be followed with monitors. Usually monitors are sampling monitors that record the state of the system at fixed intervals. (Smith, 1990). Monitors can be software monitors that execute independently from measured software or hardware monitors that are external devices attached to the system. Atkins & Subramaniam (1996) introduce a statistical sampling method that records states of the computer CPU at regular intervals. Statistical sampling of the CPU requires that hardware of the system facilitates the information gathering.

For performance analysis software system transitions between the states can offer a valuable information. These transitions are called events (Smith, 1990). Performance engineer can define interesting events and record every time they occur with related performance data. Events created by individual program that are pertinent for performance are for example I/O events and page faults. Events can also be collected from other programs executing in parallel to measured program. (Smith, 1990). Collecting information of events system-wide can reveal information how different parallel programs use and possible saturate system resources. Atkins & Subramaniam (1996) note that such dynamic way of collecting real-time data on different events yields less specific information than static code analysis.

### 3.3.4 Instrumentation

Instrumentation of the code refers to inserting code at the key probe points in the software (Smith, 1990). Instrumentations can be inserted manually (Smith, 1990) or
programmatically (Kumar, Childers & Soffa, 2005). Smith (1990) encourages implementing instrumentation for software during its development. It’s easier to define instrumentation probe points when logical functions are designed and taking instrumentation in count during design might incur overhead from instrumentation. With instrumentation Smith (1990) further suggests collecting number of times workload scenarios execute, how often software components execute and their resource requirements. Even though manual instrumentation can be more laborious than automatic and tool assisted performance data gathering, instrumentation provides various benefits to compensate the effort. Instrumentation gives full control of what data to gather, granularity over gathered data and reporting data (Smith, 1990).

3.3.5 Performance analysis considerations

One must be aware of possible impacts for the measurements that these data gathering techniques might incur. Software data gathering processes cause extra overhead in the system where the measured program is executed. Other processes in the system might delay the measurements or be erroneously accumulate the total time of measured process. (Smith, 1990). Liu (2009) suggests compensating overhead of performance work with hardware that’s more powerful than in the target environment.

It’s also important that the data gathered is reproducible. For this reason one might need to control different aspects of the environment where the measurements are made: the workload, the time of the day, and duration of the measurements. Rigorous method of gathering performance data doesn’t really differ from scientific data gathering. (Smith, 1990).

3.4 Performance analysis tools

Basic functionality of performance analysis tools is to show where in the code the program execution spends most of its time. Some of these tools may serve multiple purposes also other than performance related. They can be used to trace program behaviour, analyse memory consumption and spot memory leaks. Indeed, these tools can be very beneficial in whole process of software construction and not just in performance analysis. This helps in focusing the performance improvement work to correct places instead of relying intuition, and hence spare wasting of precious project resources.

There exists vast amount of tools to assist in performance benchmarking and performance data gathering. Hence, selecting the best tool for an individual job can be a very challenging task. Some tools are better at certain kind of tasks and might be lacking in others. To be able choose an appropriate tool for the task at hand at least basic knowledge of strengths and weaknesses of tools should be known. There’s no reason to limit performance analysis to only one tool, but a smart combination of tools can provide wide understanding of program. Of course, mere good tools are not enough. The interpretation of results of individual tools are not always straightforward and therefore require special knowledge of tools and targets of measurement.

Some tools are hardware or OS dependant. Hardware dependant tools might only work on certain processor architectures or might have restricted functionality on other processors. These tools use special properties of the processor to gather most accurate and detailed information about the execution of instructions in the processor. OS
dependant tools depend on special properties of the operating systems or use other tools available only to certain operating systems.

Selection of presented tools must be somehow limited to a manageable size, since available performance analysis tools are abundant. The selection of performance analysis tools in this thesis is guided by a practicality towards the properties of the tested program, used development environment and test setting. Therefore, the criteria for selecting the tools in this thesis stems from compatibility with the developed program, availability of tool and operating system of development environment. Respectively, tools must support C++ programmed locally deployed programs, be available for general usage through Open Source licensing or through some other licensing scheme that doesn’t contain fees or trial periods, and tool must be executable in Ubuntu Linux operating system. Due to these criteria, selection of performance analysis tools represented here contain only a small subset of all available commercial and non-commercial tools out in the wild.

3.4.1 GNU gprof

Gprof is a profiler for complex modular programs. Gprof was initially created by Graham, Kessler and McKusick in University of California, Berkeley on base of UNIX profiling tool prof (Graham, Kessler and McKusick, 2004). In modern GNU/Linux based operating systems like Ubuntu have GNU gprof version contained in GNU binutils binary tools collection. GNU gprof is written by Jay Fenlason (GNU gprof, 2013). Henceforth, when Gprof is mentioned, GNU gprof is actually meant if not explicitly stated otherwise.

Gprof provides flat profile and a call-graph information about the execution of a profiled program. Flat profile contains execution counts and durations of individual functions. This will help the user to spot which functions take most time and are called most during execution. Call-graph describes from where functions were called. With call-graphs erroneous behaviour or excessive usage of certain functions can be detected. (GNU gprof, 2013) There’s three steps that need to be taken in order to enable profiling: the program must be compiled and linked with profiling enabled, the program must be executed to generate profile data and Gprof must be ran to analyse the profile data (GNU gprof, 2013). Gprof provides results of profiling as a text only to a standard output (on Linux) and it itself doesn’t contain graphical visualization of profiling results.

When program is compiled with Gprof profiling option, every function call in the program is altered to catch information about from where it was called and how many times it was called. Additionally, Gprof uses statistical sampling to monitor and record program counter’s location. This is used to elicit run-time duration of program’s functions. Sampling process is subject of some statistical inaccuracy and functions that execute only briefest of time might not be catched by sampling. (GNU gprof, 2013) According to GNU gprof (2013) documentation the predicted sampling error for Gprof is square-root of collected samples. Also, profiling with Gprof causes program to run somewhat slower due to collecting and storing data (GNU gprof, 2013).

It’s also important to note that profiling of shared dynamic libraries is not supported. Libraries must be build as static libraries for Gprof profiling (GNU gprof, 2013).
Table 2 presents an example of flat profile provided by Gprof when profiling a small program that used insertion sort and merge sort to arrange 100,000 of randomly generated numbers. For testing purposes naïve insertion and merge sort algorithms were implemented with C++ according to definitions by Cormen et al. (2009). Specifications of computer used is presented in chapter 4.4 table 7.

<table>
<thead>
<tr>
<th>%</th>
<th>cumulative seconds</th>
<th>self seconds</th>
<th>calls</th>
<th>self ms/call</th>
<th>total ms/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>99.82</td>
<td>2.50</td>
<td>2.50</td>
<td></td>
<td></td>
<td></td>
<td>Sort::insertionSort (int*, int)</td>
</tr>
<tr>
<td>0.40</td>
<td>2.51</td>
<td>0.01</td>
<td>99998</td>
<td>100.23</td>
<td>100.23</td>
<td>Sort::mergeSort (int*, int, int, int)</td>
</tr>
<tr>
<td>0.00</td>
<td>2.51</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>GLOBAL_sub_I_Z DumpListPii</td>
</tr>
</tbody>
</table>

What’s interesting here is that call counts for all functions are not displayed, as if not all functions of a deterministic program were called. Even insertionSort function hasn’t apparently increased call count, even though it has taken 99.82% time of program’s whole execution. It must be noted that program was compiled with rather aggressive GNU C++ compiler optimization option of O2 and without debugging symbols for this profile. When the program was compiled with O0 optimization option and profile was taken again, all correct functions seemed to be called. It’s likely that aggressive optimization by compiler has affected how the functions are accessed and that has affected what kind of results Gprof produces. This is an important factor when interpreting results of Gprof.

3.4.2 Valgrind

Valgrind is an instrumentation framework that provides numerous ways for profiling and debugging programs with various plugins called tools (Valgrind User Manual, 2014). Valgrind in itself is only a framework for different tools and doesn’t in itself provide debugging or profiling functions. Valgrind is supplied with a standard set of tools: Memcheck, Cachecheck, Callgrind, Helgrind, DRD, Massif, DHAT, SGcheck and BBV. Valgrind is Open Source software licensed with The GNU General Public License, version 2. Valgrind runs on various flavours of Linux, Android and Darwin (OS X). Valgrind is widely used both in Open Source and proprietary software development5. Valgrind is a complex project with many tools and options. Explanation provided here is just a brief primer to basic functionality and standard tools.

Valgrind is called as a dynamic binary instrumentation (DBI) framework. It’s purpose is to make implementing dynamic binary analysis (DBA) tools easy. In DBI frameworks the analysis code is added to the profiled program’s code at run-time. Valgrind operates directly with program executable and therefore, the profiled program doesn’t require any compile time or linking preparations for profiling with Valgrind. (Nethercote and

5http://valgrind.org/gallery/users.html
Seward, 2007) Depending on the used tool profiling with Valgrind causes the profiled program to run 10 - 50 times slower than natively (Valgrind User Manual, 2014).

Valgrind standard tool set provides possibilities from detecting different memory errors to cache and branch-prediction profilers and to thread error detection. Table 5 lists all nine standard Valgrind tools.

*Table 5: Standard Valgrind tools and brief descriptions from Valgrind User Manual (2014)*

<table>
<thead>
<tr>
<th>Tool</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memcheck</td>
<td>Memory error detector. Default tool in Valgrind and probably Valgrind’s most used tool (Nethercote and Seward, 2007).</td>
</tr>
<tr>
<td>Cachegrind</td>
<td>Cache and branch-prediction profiler.</td>
</tr>
<tr>
<td>Callgrind</td>
<td>Call-graph generating cache profiler.</td>
</tr>
<tr>
<td>Helgrind</td>
<td>Thread error detector.</td>
</tr>
<tr>
<td>DRD</td>
<td>Thread error detector. Uses different analysis techniques than Helgrind.</td>
</tr>
<tr>
<td>Massif</td>
<td>Heap profiler.</td>
</tr>
<tr>
<td>DHAT</td>
<td>Heap profiler.</td>
</tr>
<tr>
<td>SGcheck</td>
<td>Experimental tool that can detect overruns of stack and global arrays.</td>
</tr>
<tr>
<td>BBV</td>
<td>Experimental SimPoint basic block vector generator.</td>
</tr>
</tbody>
</table>

All these Valgrind tools can be greatly helpful in writing good quality and more correct code. Different tools provide information about program’s behaviour and can reveal non-optimal solutions and various problems.

Memcheck is memory error detection tool. It addresses different illegal memory accesses, undefined values, erroneous heap memory frees, leaking memory and other issues common in C and C++ programs. (Valgrind User Manual, 2014)

Cachegrind tool provides information about program’s cache usage and conditional branch path detection. Cachegrind simulates machine’s first level instruction and data caches (I1 and D1) and unified last level cache (LL). Even on machines that have more than two cache levels only first and last level are simulated for they are most useful. It also provides possibility for source annotation. (Valgrind User Manual, 2014)

Callgrind is tool for producing a call-graph profile of an application. Graphical presentation of call-graph can be examined with additional KDE/Qt based program KCachegrind. Callgrind can also do cache simulation and conditional branch path detection similar to Cachegrind. (Valgrind User Manual, 2014)
3.4.3 OProfile

OProfile is a sampling profiler for Linux that can collect various hardware events system wide. OProfile is Open Source and released under GNU GPL license. It depends on performance counters provided by the CPU. System wide means that OProfile can collect information about the whole system; about every running process on a machine. OProfile can also be used to profile Linux kernel code which is not possible with Valgrind. OProfile has a very low overhead on the system, about 1 - 3%, depending on sample frequency. It can also be used to profile individual programs. Like Valgrind OProfile is unobtrusive; any executable can be profiled and no re-compiling or linking of profiled program is needed. Programs don’t even need to be compiled with debugging symbols. Without debug symbols the profile output is limited to shared libraries and other possible processes that the program uses. With debug symbols OProfile can output more specific annotated source of the program. (Levon, 2004)

OProfile count events such as CPU cycles, interrupts and cache misses. Available hardware events depend on used CPU architecture. With system wide profiling one can examine how much different processes and libraries use system CPU time. This helps to identify bottlenecks on system level. On an application level, annotated source profiling information shows number of samples of certain events collected on different lines of code. Annotation can also be extended to assembly language instructions produced from code. (Levon, 2004)

OProfile consists of different utility tools that provide profiling functionality and reporting. Here is briefly introduced the most relevant ones. operf is a tool to profile individual executables and whole system. ocount is used to simply count hardware events transpired from individual applications or from system-wide. ophelp lists the available events for used CPU architecture. oreport is a primary tool to represent profiled data. It supports various formatting options as well as flat and call-graph profiles. opannotate is used to get profiler report in form of annotated source code. (Levon, 2004)

Results from OProfile are not 100% accurate, as is usual with statistically sampling profilers. When interpreting annotated source code, one must note that compiler may change some lines of code, add glue code, change code-line order and annotate in-line functions in original source where they were defined instead in the calling source. Accurate debug information in annotated sources is always problem when compiler optimizations are used. (Levon, 2004) These are usually not a problem and results from OProfile can be very useful in tuning process nevertheless. (Levon, 2004; Cohen, 2004).

3.5 Performance tuning

Code tuning is process of modifying correct code to make it run more efficiently (McConnell, 2004). Performance tuning described here differ from actual performance defect fixing. Tunings are applied to the parts of the code that are otherwise correct and good, but just need to be made run faster. Performance defects are incorrect implementations in the code that can even be trivial to fix (Jin, Song, Shi, Scherpelz, & Lu, 2012). Code tunings are small scale changes to affect singular hot spots in the code-base (Bourque & Fairley, 2014; McConnell, 2004). Tuning can be highly effective in increasing program’s performance in certain situations. Especially algorithms dealing with repetitive data processing, searching and sorting can usually be improved. Also,
loop construct optimizations can also yield significant performance results. (Atkins & Subramaniam, 1996).

Code tuning can be a hazardous affair as tunings have a strong tendency to negatively effect on code maintainability and understandability (McConnell, 2004), sometimes tuning is last option to achieve acceptable level of performance. Tuning techniques vary from situation to situation, depending on coding languages, compilers and environments. Not all tuning techniques are applicable to all situations and between languages and even compilers. That’s why the most important thing is to always measure the effect on performance of every individual code tuning. As tuning techniques tend to make the code more complex, one should also always contemplate carefully the costs of code tunings to the clarity and maintainability of the code.

McConnell (2004) notes that usually program’s performance depends more on program architecture and design, and chosen data structures and algorithms than on the efficiency of the individual lines of code. It’s important not to start tuning too early or unnecessarily in the project because many techniques used to tune the code can make the code more complex and harder to maintain (McConnell, 2004). McConnell (2004) suggests that best way to prepare for performance work is to initially write clean code that is easy to understand and maintain. The idea of incorporating performance considerations to the whole project’s life-cycle is strongly supported by Software Performance Engineering methodology (Smith, 1990) and others (Molyneaux, 2009; Liu, 2009). Indeed, it seems that best way to advert performance problems in a project is to include performance awareness to the project’s life-cycle as early as possible. There are various ways of modelling and evaluating how different design choices affect to the performance before they are implemented (Smith, 1990).

Applicable code tuning techniques depend on the various aspect of the program and the environment. Because code tuning usually refers to local, code-level, changes in the code base rather than architectural or design choices, the tuning requires in depth knowledge about the coding language chosen for the project. Also, it’s important to understand how the language compilers try to optimize the code. Many times, it’s more beneficial to initially write code using the best practices of the language and instruct and let the compiler apply optimizations. Sometimes even, manual tuning efforts might even prevent the efficient optimization by the compiler and hence produce executable with actually poorer performance. That and other various, maybe unknown, properties of the program and environment factor in to the conclusion, that effects of every code tuning should be measured. (McConnell, 2004)
4. Performance analysis and testing of UpWind project

Performance degradation is a major threat and concern in many big projects (Corbet, 2004; Bjedov, 2007; Baker, 2014). Performance problems have also affected UpWind project. UpWind project has had major development team turnover. Continuity of the project has relied on sparse documentation and helpfulness of rare static personnel. It’s imperative that a decline of project quality must be stopped and quality consciousness must be embedded to team members. One way to combat unintentional quality and functional degradations is to implement a testing process that seeks to verify functional and quality targets of the project as the project evolves.

Even though functional targets of any program obviously benefit from testing process, in this thesis performance targets are focused exclusively. Firstly UpWind project is introduced. Then performance analysis of the project is conducted and suitable performance testing process is established to yield sustainable performance in the future.

4.1 UpWind project

UpWind is a university research project started in 2006 in Department of Information processing Sciences of Oulu university. UpWind is a free and open source software project although it’s not currently licensed under any Open Source or Free Software license. UpWind project’s code-base is publicly available for anyone to inspect in GitHub source repository and collaboration service. As a research project the main aim has been effective route planning algorithms for sailing boat navigation. Project implements standard waypoint navigation scheme with naïve Dijkstra’s algorithm to solve the shortest path along navigation lines (Gultekin, 2013). Waypoint navigation is called long-term route planning in context of UpWind. UpWind project also implements a novel algorithm to suggest optimal sailing tacking lines to reach the target destination as fast as possible in respect to wind and safe travelling (i Royo, 2010). Tacking line navigation is called short-term route planning in UpWind context. There are others commercial and non-commercial projects for sailing navigation but none of them, as far as I know, implement exactly similar tacking line optimization.

UpWind project is normally run with a dedicated hardware for sailing purposes. Maritime environment sets special challenges for computer hardware to be used in sailing boats. Marine computer hardware is prepared to endure vibration and shocks caused by waves and wind at the sea. Also, fragile hardware components must be

\[\text{https://github.com/UpWind/devel}\]

\[\text{http://uusiloisto.fi/}\]

\[\text{http://opencpn.org/ocpn/}\]
shielded from corrosion and water damage. It also sets restrictions what kind of modifications are available, since the hardware used in the UpWind system is specialized. Therefore, updates to UpWind project are restricted to software and performance increase through hardware modification is very limited at best and not considered in this thesis.

Project has been developed by different student groups as student projects and thesis writers. Project has been loosely controlled by few static key personnel that determine the project’s goals and broad technical aspects. Project has no formidable existing documentation or established development process. Usually new student teams work on a project for a very short time - usually around six months - and then move on. Even though static personnel have done their best, continuity and consistency between consecutive teams has been a persistent problem. The time new teams spend with UpWind project is hardly enough to form any deep understanding about ins and outs of such a complex system.

UpWind project has gone through many transformations, since it’s conception in 2006. At first UpWind was implemented with Java language but was later ported to C++ and Qt due to performance needs. Most recent major modification to the system is new plugin architecture. It provides easy flexibility and customizability to the project’s standard functionality by providing common interface for plugins to connect to core system.

UpWind project consists of system core, plugins and spatial database. Core initializes the program, load settings, creates main window and dynamically load plugins through it’s PluginManager component. Plugins handle everything else from drawing the chart, interpreting NMEA sentences to database access, and long-term and short-term route planning. Spatial database contains chart information in special format dedicated to geographic presentation of data. In UpWind project database usually resides in a separate virtual machine or a database server computer. Table 6 lists current plugins and their descriptions in UpWind system.

Table 6: Current plugins in UpWind project.

<table>
<thead>
<tr>
<th>Plugin</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NMEASerialPortReader</td>
<td>Important part of sailing boat navigation system is to read NMEA sentences from the boat’s interface. Sailing boat is equipped with different sensors sensing boat’s current GPS position, compass reading, wind speed and direction among others. This plugin connects to computer’s serial port and reads upcoming NMEA data sentences.</td>
</tr>
<tr>
<td>PostgreChartProvider</td>
<td>Loads chart data from database and process the raw data for later usage. Applies some filtering to the data and imports data to OGR Simple Features Library (<a href="http://www.gdal.org/">http://www.gdal.org/</a>) form.</td>
</tr>
<tr>
<td>QtRenderer</td>
<td>Is in charge of drawing chart, different chart objects, boat and long and short-term routes.</td>
</tr>
<tr>
<td>UpWindScene</td>
<td>Calculates long-term and short-term route plans.</td>
</tr>
</tbody>
</table>
Builds obstacle table from information from database to be used in short-term route planning. Also, handles boat’s polar diagram that describes boat’s target velocity with certain wind angle and wind speed.

| NMEA Instruments | Contains instruments that display information from different sensors on the boat. Current instruments include anemometer, clock, compass, gps and tool to handle boat during simulation. |

In addition to these plugins, UpWind project also contains simulator to produce mock NMEA sentences and send them over serial port connection.

### 4.2 UpWind project performance analysis

Like also so many other software projects, UpWind has woken up to its performance problems only very late in its development process. There hasn’t been any formal performance testing process and due to the lack of documentation there’s no indication how the performance has been considered during the design and development. But performance problems are visible even to the bare eye. The program boots up slowly, there’s a pause between zooming map levels and forming short-term route plans take a long time. While the other UI related performance problems can be annoying they are not most severe and not a target of this thesis. In this thesis performance of short-term algorithm is only focused, because that is the most essential component of UpWind software.

Kangastalo (2013) studied time performance of long-term and short-term route planning algorithms in his master thesis. Kangastalo (2013) found out that long-term route planning was mostly problem free but short-term algorithm suffered from bad performance problems. Sometimes short-term algorithm took over 20 seconds to complete with Lenovo T61p machine topping 10 seconds time limit that triggers the new route plan cycle (Kangastalo, 2013). Kangastalo (2013) further elaborated that majority (circa 99%) of time was constantly consumed by database queries.

Short-term route planning is an integral part of UpWind program. It’s a novel algorithm developed by I Royo (2010) for UpWind project. Route planning algorithm works by taking requested destination position and boat’s current position and calculates the turning points required to reach the destination while not colliding with obstacles. As this algorithm is designed for maritime usage of sailing boats, wind plays a major part in what points are selected as turning points. In sailing, turning is an operation that slows the speed of the boat, so number of turnings should be minimal to optimize the

---

9 Navigating a sailboat depends on from which side of the boat wind is blowing. Sailing boat cannot move directly against the wind (or upwind). Usually sailboat can move against wind in 45 degree angles. Corrective manoeuvres must be taken to reach the position in upwind. These manoeuvres are called tacks. Moving in upwind creates a zigzag movement pattern that eventually zeroes into the destination position. Sailing upwind is called beating in sailing jargon. When the wind comes from behind the boat, the situation is pretty much the same. Zigzag movements must be used to yield optimal speeds and to make sailing safer. This is called running. In running turnings are called gybes. When wind comes perpendicular to the sailboat, the boat is said to be reaching. No turns are then needed to reach the target position. More elaborate descriptions can be found from works by Gultekin (2013) and I Royo (2010).
speed on the route. Also, for security reasons boat must always have a direct line of sight to long-term navigation route; algorithm should not suggest a route where some object at some point obstructs the view to long-term route.

Database that contains geographical information about current area of sailing is accessed when UpWind is started. Geographical information is extracted from database to show land and sea areas. Also, different maritime signs and navigation lines that are contained in the database are also loaded. Obstacle table is created from chart data, containing all relevant hazards sailing boat needs to avoid when navigating. This includes for example rocks, known shipwrecks, depth contours and masses of land.

The short-term algorithm starts by determining the furthest point on the long-term route that can be seen from the boat without obstructions. This is done by finding obstacles inside triangles formed by boat position, point on the long-term route which has the shortest distance to the boat and succeeding long-term route waypoints. If obstacles are detected, new destination point is assigned to position on the long-term route prior to area with obstacles. Then wind angle is used to determine required turning points to reach the new destination point. More in-depth description of short-term algorithm can be found in study by Royo (2010).

Currently algorithm is triggered and re-ran every ten seconds to see further destination points on the long-term route and tune turning points reflecting the changes on the wind angle and new observed obstacles. Ten seconds can be thought as a limit of acceptable time of one short-term route planning cycle. Kangastalo (2013) analysed the short-term route planning algorithm and determined that time complexity of whole algorithm is \( O(n^2) \), where the \( n \) is number of long-term route waypoints. Running times of different sub-algorithms varied wildly depending on obstacles on the inspected area (Kangastalo, 2013).

A defect was spotted in a creation of obstacle table, when the database queries were investigated in short-term route planning. Obstacle table that was introduced earlier, is not a memory entity but an SQL table that is written back to chart database. The defect caused the obstacle table to grow in size every time UpWind program was started. The table was never cleared nor the new table replaced the old one, but new table was always added to old table. Obstacle table is individual and immutable for certain database chart, but duplicate tables were not checked so eventually obstacle table in database would grow to be quite huge.

This defect had a large impact on performance of short-term algorithm. Short-term algorithm in a typical case creates many geographical intersection-queries to SQL database looking for obstacles in the boat’s path. When the obstacle table grew, also the time that took to complete database queries using obstacle table grew. Performance profiling and testing presented in this thesis is conducted with short-term algorithm where the described deficit is fixed.

4.3 Performance engineering considerations in UpWind

Every software project is essentially different. From different software properties to widely differing environments. Performance testing for every project requires understanding and considering properties of every individual project. Performance analysis must be tailored for each project individually and UpWind is no exception.
These project properties eventually define which performance analysis tools and which performance tests are suitable for project.

As UpWind is currently available exclusively to Linux, only Linux compatible analysis tools can be used. UpWind supports only one user at time so there’s no need to take in care for extra concurrent users. UpWind is executed on a local machine. UpWind is not a web service nor is it ran on some external server. Also, UpWind relies thoroughly on dynamically loaded and linked plugins everything from drawing the chart to calculating route plans. Furthermore, only a part of the program - short-term route planning - will be under inspection.

From early introduced performance analysis tools GProf cannot be used due to lack of support for dynamic libraries. Valgrind and OProfile are left as viable options for performance analysis of UpWind project. Not being able to use GProf in this case is regrettable but all similar functionality can be found from Valgrind and OProfile.

The testing and analysis environment also requires some consideration. While it’s obvious that hardware plays major part in performance, the essential properties of the source code are immutable between different environments. Valgrind and OProfile both cause some overhead to the system that slows the execution of analysed application. This is not a problem because aim of analysis tools is not to benchmark the application but to analyse the behaviour of the code. Nevertheless, hardware properties are still not completely irrelevant. Testing environment should have enough resources to handle overhead caused by analysis tools (Liu, 2009). Hardware properties also specify which hardware events are available for profiling and some hardware just isn’t compatible with analysis tools. Lastly compilers might differ to some degree between machine architectures as they interpret higher level code to hardware specific machine code instructions.

4.4 Test case design

Hailpern and Santhanam (2002) quote Dijkstra’s criticism of testing “Program testing can be used to show presence of bugs, but never to show their absence”. Performance tests include tests against the software specification requirements and against known defects. The coverage of performance tests or any test can’t probably ever be total 100% (Bjedov, 2007). Designing, implementing and running test cases always use resources. Hence, tests are always trade-off between coverage and available resources.

Software development project should have specifications with verifiable performance requirements, but this is frequently not the case (Weyuker and Vokolos, 2000). Also in UpWind there’s no official performance requirements other than that the program could be used in actual navigation of sailing boat. Kangastalo (2013) used ten second as upper-limit for one short-term route calculation cycle. This time limit might now be deprecated, since after the fix of defect described earlier. Also, it must be noted, that it can be difficult to assign strict performance targets for performance unit tests, because there are too many factors that can affect the results (Horký et al., 2013).

Since there is so little previous performance work done on UpWind project, the objective for performance tests in this thesis is to determine current performance of UpWind project’s short-term algorithm as realistically as possible. These tests can also be used as a baseline and reference for further performance work in UpWind project and can also be used profiling of short-term route planning algorithm. Kangastalo (2013)
measured performance of short-term algorithm, but those results were affected by the previously mentioned defect. In these tests the defect is fixed.

According to Hailpern & Santhanam (2002) there’s four major parts in any testing effort: test case design, test case creation, test case execution and debugging. Test case design is discussed in this chapter. Firstly, test cases are created using instrumentation to isolate and control testing process. Instrumentation is used to get preliminary performance data from different test cases. This will work as a starting point for performance profiling efforts. In chapter 4.7.3 test cases are implemented with QTestLib for performance regression testing.

When designing and creating dedicated performance test cases there are few issues that need to be solved. What kind of workload should be used in tests? Should the average workload be used or a very heavy load (Weyuker and Vokolos, 2000). In these performance test cases we are going to use average, or “typical” loads encountered during normal usage. Different test data are used to simulate different situations and bring variety to typical situations. Next choice is, should the tests and application run in the system in isolation or with normal background noise from the system (Weyuker and Vokolos, 2000). Because the UpWind program is going to typically run in non-isolated environment with background noise, these performance tests are also run in typical environment.

Weyuker and Vokolos (2000) suggest determining realistic values for input test data. The idea is to test calculation of short-term route plan with different boat positions in relation to long-term route on a chart with realistic obstacle data. During that route there are many situations when short-term route is generated. How the short-term route is calculated in each situations is dependant on obstacles, boat’s position in relation to long-term route and wind. These three items affect how navigation turning points are chosen in every calculated short-term route planning iteration.

The approach chosen in this thesis is to generate a reference long-term route which is then traversed with a sailing boat in realistic manner. Position of the boat is stored at certain points when the route is travelled. Those stored points along the long-term route are used in test cases. In a test case short-term route is calculated and elapsed calculation time is recorded at one stored point. Similar approach was used in study by Kangastalo (2013) in which elapsed times of short-term algorithm calculation were recorded during live run of the program. This kind of approach will yield realistic approximation of performance in typical real-world usage scenario of UpWind application. Figure 1 presents the long-term route and boat positions for test cases.
Individual test case boat positions shown in figure 1 represent different situations when sailing near the coast. Dark blue line denotes the long-term route. Different obstacles like land, depth contours, rocks and navigation signs set limits to generated short-term routes. Yellow arrows point the coarse direction of boat at every test case and light green arrow shows the wind direction. Wind speed and angle remain constant during the test. Wind in current scenario is blowing constantly from east at constant velocity. Wind direction is in relation to the map where north is denoted as 0 degrees and east is 90 degrees and so forth. Wind angle is therefore constant 90 degrees.

Short-term route planning is calculated in plugin library UpWindScene. Library contains class CalculateLaylines that has a function startCalc(). CalculateLaylines::startCalc() is the function that is responsible for calculating short-term route planning. startCalc() calls other sub-functions that are executed linearly to calculate route in parts. At the end of startCalc() function new short-term route navigation points are emitted through Qt's signal-slot system for further processing. startCalc() function is called in ten second intervals. In test cases function CalculateLaylines::startCalc() is isolated and executed separately with boat positions and long-term route as an input data. These five test cases are used in profiling and testing.

The relevant computer specifications of the system used for performance analyses and testing is presented in table 7.
Table 7: Testing and analysis computer specifications.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel Core i7 3517U @ 2.00GHz × 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chipset</td>
<td>Intel HM76 Express Chipset</td>
</tr>
<tr>
<td>Memory</td>
<td>DDR3 1600 MHz SDRAM</td>
</tr>
<tr>
<td>Storage</td>
<td>SATA III SSD 128GB RPM</td>
</tr>
<tr>
<td>OS</td>
<td>Ubuntu Linux 14.04 LTS 64-bit</td>
</tr>
</tbody>
</table>

It must be noted that the computer in table 7 is much more powerful than the system where the UpWind is normally run. This guarantees that there’s enough capacity to handle overhead of profiling.

4.5 Preliminary analysis of short term algorithm with Valgrind and OProfile

In this chapter OProfile and Valgrind are used to profile short-term route planning algorithm of UpWind project. This provides preliminary information for analysis based on results from performance test cases.

4.5.1 OProfile

Events that OProfile can detect depend on the CPU architecture used. ophelp tool detects the CPU type correctly as Intel Ivy Bridge and determines that the default counted event is CPU_CLK_UNHALTED. CPU_CLK_UNHALTED counts core cycles when not in halt state (Intel Architecture Software Developer’s Manual, 2014). There are many other events as well that could be counted, but with CPU_CLK_UNHALTED it can be determined how much of CPU resources different parts of the program are using and that is sufficient for this analysis.

OProfile’s manual instructs to profile realistic situations, profile different scenarios and profile for as long as possible (Levon, 2004). Also, it’s required to focus profiling exclusively to short-term algorithm, since that’s what we are interested in. Profiling data is collected from executing the algorithm 100 times on each test case boat position. Wouldn’t fewer iterations of tests be enough? OProfile has some statistical inaccuracy due to its nature. More iterations simulate effects of a longer trip and also increases possibility that all even small functions are more accurately sampled. This, of course doesn’t represent the reality 100%, but should provide a reasonable accurate result and it fulfils suggestions for profiling described in OProfile’s manual. For OProfile analysis UpWind was built with O2 level of compiler optimization and without debugging symbols.

Profiling results are generated with opreport tool containing symbol summary of binary used in test. Only symbols from library libUpWindScene that contains the short-term route planning were included in the report. Table 8 lists five functions that counted most of the samples during short-term route planning. The event counted was CPU_CLK_UNHALTED.
Table 8: OProfile results from short-term route planning.

<table>
<thead>
<tr>
<th>Samples #</th>
<th>%</th>
<th>Symbol name</th>
</tr>
</thead>
<tbody>
<tr>
<td>566</td>
<td>31.5320</td>
<td>CalculateLaylines::buildWKTPolygon(QPolygonF const&amp;)</td>
</tr>
<tr>
<td>271</td>
<td>15.0975</td>
<td>CalculateLaylines::checkIntersection(QString const&amp;, QString const&amp;, QString)</td>
</tr>
<tr>
<td>183</td>
<td>10.1959</td>
<td>CalculateLaylines::getNextPoint(QVector&lt;QPointF&gt; const&amp;, QPointF const&amp;, float const&amp;)</td>
</tr>
<tr>
<td>179</td>
<td>9.9721</td>
<td>CalculateLaylines::checkIntersection(QString const&amp;, QPolygonF const&amp;, QPolygonF const&amp;)</td>
</tr>
<tr>
<td>84</td>
<td>4.6797</td>
<td>QVector&lt;QPointF&gt;::operator=(QVector&lt;QPointF&gt; const&amp;)</td>
</tr>
</tbody>
</table>

It can be seen from table 8 that short-term algorithm spent most of its time in `buildWKTPolygon(QPolygonF const&)` function taking roughly 32 percent of whole execution time. Second most time-intensive function is `checkIntersection(QString const&, QString const&, QString)` that uses SQL database functions to calculate intersections between geography objects. This result is interesting because it was expected that functions using the database would use most of the execution time based on study by Kangastalo (2013).

`getNextPoint(QVector<QPointF> const&, QPointF const&, float const&)` is a lengthy function that finds next destination point on the long-term route. It uses variations of `checkIntersection` function many times, thus contributing to high ranking of those functions. Fourth function is `checkIntersection(QString const&, QPolygonF const&, QPolygonF const&)`. This is an intermediate function that processes inputs to a form that can be used in SQL queries. For this purpose function uses `buildWKTPolygon(QPolygonF const&)` and `checkIntersection(QString const&, QString const&, QString)` contributing to their high rankings.

`QVector<QPointF>::operator=(QVector<QPointF> const&)` is a vector copy assignment operator from Qt library. Various instances of navigation point information are copied from vector to vector many times during short-term algorithm. Maybe this is not so effective and other solutions could be tried. Also, it’s probably good idea to check if usage of `buildWKTPolygon(QPolygonF const&)` could be made more effective.

4.5.2 Valgrind

**Cachegrind**

Cachegrind tool can help to find non-optimal cases of CPU cache usage and conditional branch detection. Cache misses and wrongly evaluated branch conditions are known to cause extra CPU cycles in computation.

Valgrind User Manual (2014) suggests compiling to be profiled program with debugging symbols and also with compiler optimizations turned on. In this case O2
option was used. Also, -fno-inline option was added to lessen the confusion from interpreting the results. Results were inspected and interpreted with KCachegrind tool that provides graphical interface to Cachegrind’s reports. Profiling information was gathered from one iteration of all test cases. While Cachegrind’s results might not be “super-accurate” (Valgrind User Manual, 2014) it doesn’t use statistical sampling like OProfile and hence one iteration should be enough. Furthermore, including all test cases assures that all branches traversed in tests are also profiled.

Table 9 represents Cachegrind results from function level. According to Valgrind User Manual (2014) most useful metrics are instruction reads (Ir), last-level cache misses (DLmr and DLmw) and conditional branches mispredicted (Bcm) and indirect branches mispredicted (Bim). Table 9 is sorted according to the Ir count and only top five instruction heavy functions are shown.

<table>
<thead>
<tr>
<th>Function of CalculateLayline class</th>
<th>Ir</th>
<th>DLmr</th>
<th>DLmw</th>
<th>Bcm</th>
<th>Bim</th>
</tr>
</thead>
<tbody>
<tr>
<td>buildWKTPolygon(QPolygonF const&amp;)</td>
<td>77 456</td>
<td>0</td>
<td>0</td>
<td>582</td>
<td>0</td>
</tr>
<tr>
<td>checkIntersection(QString const&amp;, QString const&amp;, QString)</td>
<td>15 061</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>checkIntersection(QString const&amp;, QPolygonF const&amp;, QPolygonF const&amp;)</td>
<td>8 487</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>getNextPoint(QVector&lt;QPointF&gt; const&amp;, QPointF const&amp;, float const&amp;)</td>
<td>6 548</td>
<td>0</td>
<td>0</td>
<td>57</td>
<td>0</td>
</tr>
<tr>
<td>getNearestPoint(QVector&lt;QPointF&gt; const&amp;, PointF const&amp;)</td>
<td>5 403</td>
<td>0</td>
<td>0</td>
<td>181</td>
<td>0</td>
</tr>
</tbody>
</table>

The function which accounted great majority of executed instructions (Ir) is buildWKTPolygon(QPolygonF const&). Also, no other function had so many conditional branches mispredicted (Bcm). Last level cache read and write (DLmr and DLmw) misses were virtually non-existent in short-term algorithm. Also, no mispredicted indirect branches (Bim) were identified.

Quite expectedly high number of executed instructions seems to correlate with OProfile’s results of high counted samples. Four of the five functions are same in both OProfile’s results in Table 8 and in cachegrind’s results in Table 9. Due to high number of executed instructions and mispredicted conditional branches, buildWKTPolygon requires a closer look. Table 10 shows line-by-line listing of function buildWKTPolygon(QPolygonF const&).

<table>
<thead>
<tr>
<th>Bcm</th>
<th>Ir</th>
<th>Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 392</td>
<td>Qstring CalculateLaylines::buildWKTPolygon(const QPolygonF &amp;homboid) {</td>
<td></td>
</tr>
<tr>
<td>Line</td>
<td>Code</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>848</td>
<td><code>QString WKTPolygon = &quot;POLYGON(&quot;;</code></td>
<td></td>
</tr>
<tr>
<td>582</td>
<td><code>for (int i = 0; i &lt; rhomboid.size(); i++) {</code></td>
<td></td>
</tr>
<tr>
<td>18792</td>
<td><code>WKTPolygon.append(QString::number(rhomboid[i].x(), 'f', WKT_P));</code></td>
<td></td>
</tr>
<tr>
<td>3936</td>
<td><code>WKTPolygon.append(&quot; &quot;);</code></td>
<td></td>
</tr>
<tr>
<td>18792</td>
<td><code>WKTPolygon.append(QString::number(rhomboid[i].y(), 'f', WKT_P));</code></td>
<td></td>
</tr>
<tr>
<td>3936</td>
<td><code>WKTPolygon.append( &quot;);&quot;;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>}</code></td>
<td></td>
</tr>
<tr>
<td>6360</td>
<td><code>WKTPolygon.append(QString::number(rhomboid.at(0).x(), 'f', WKT_P));</code></td>
<td></td>
</tr>
<tr>
<td>1272</td>
<td><code>WKTPolygon.append(&quot; &quot;);</code></td>
<td></td>
</tr>
<tr>
<td>6360</td>
<td><code>WKTPolygon.append(QString::number(rhomboid.at(0).y(), 'f', WKT_P));</code></td>
<td></td>
</tr>
<tr>
<td>1272</td>
<td><code>WKTPolygon.append( &quot;));&quot;;</code></td>
<td></td>
</tr>
<tr>
<td>3392</td>
<td><code>return WKTPolygon;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>}</code></td>
<td></td>
</tr>
</tbody>
</table>

From code listing of `buildWKTPolygon` it can be seen that most of the instructions are executed when `WKTPolygon` string is appended. Also, all the conditional branch mispredictions are happening in for loop’s condition. Maybe `size()` function of `QPolygonF` prevents compiler from optimizing the loop and hence CPU cannot reliably predict does the loop execution continue or end.

### 4.6 Performance analysis of instrumentation measurements in short-term algorithm

Kangastalo (2013) used instrumentation of short-term route algorithm to collect performance measurements. In the same vein, this thesis uses instrumentation to collect performance measurements from short-term algorithm. For instrumentation measurements five test cases were again used. `CalculateLaylines::start()` function that does the route planning was measured by starting `QElapsedTimer` just before the `startCalc()` function and elapsed time was retrieved right after the function returned from execution. Results were collected to CSV file that was further processed with statistical
tool R. These results can be compared to performance results from Qt Test framework’s benchmarks in chapter 4.7.3.

Fluctuation in performance measurements is an important issue to consider. Liu (2009) considers that performance measurement results are stochastic by nature. Even if environment and software stays the same, performance measurements won’t always yield the exactly same results (Liu, 2009). It’s important to try to mitigate the fluctuation of performance measurement results. To minimize effects of fluctuation of individual measurements each test case was executed 100 consecutive times. Also, no other work was executed on machine while the tests were running. These are quite minimal preventive measures but on the other hand, UpWind is meant to be run locally on a general computer.

Table 11 presents performance measurement results from all test cases with 100 consecutive runs. Figure 2 presents box plot of measured times in different boat positions.

**Table 11: Mean short-term route calculation times from instrumentation in msecs.**

<table>
<thead>
<tr>
<th>Boat position</th>
<th>Mean</th>
<th>Min</th>
<th>1st Qu.</th>
<th>Median</th>
<th>3rd Qu.</th>
<th>Max</th>
<th>Sd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>88.11</td>
<td>75.00</td>
<td>85.00</td>
<td>87.00</td>
<td>91.00</td>
<td>110.00</td>
<td>5.75</td>
</tr>
<tr>
<td>2</td>
<td>149.9</td>
<td>140.0</td>
<td>146.0</td>
<td>148.0</td>
<td>151.0</td>
<td>189.0</td>
<td>7.30</td>
</tr>
<tr>
<td>3</td>
<td>118.7</td>
<td>107.0</td>
<td>115.0</td>
<td>117.0</td>
<td>120.0</td>
<td>140.0</td>
<td>5.93</td>
</tr>
<tr>
<td>4</td>
<td>164.9</td>
<td>152.0</td>
<td>161.0</td>
<td>163.5</td>
<td>167.2</td>
<td>184.0</td>
<td>6.25</td>
</tr>
<tr>
<td>5</td>
<td>125.7</td>
<td>116.0</td>
<td>123.0</td>
<td>125.0</td>
<td>128.0</td>
<td>151.0</td>
<td>4.16</td>
</tr>
</tbody>
</table>
As can be seen from figure 2 and table 11 there’s quite much interference in measurements. Box plot figure 2 exhibits various outliers with a wide spread especially in 2. test case. However, standard deviations (Sd) of measurements seem to be quite reasonable. This seems to indicate that even though there are many iterations with exceptional measurement results, results are overall quite stable and reliable enough for comparison.

Table 11 shows distinct differences in achieved mean times between tested boat positions. Measured time in first position is quite small in relation to other boat positions and fourth value is greater than in any other test case. Boat positions 3 and 5 are pretty much in same ballpark. These distinctions in measured values reserve some closer inspection. Table 12 shows number of generated navigation points (turning points and route start and end points) in every boat position.

---

Figure 2: Box plot of test case measurements.
Table 12: Navigation points generated in different test cases.

<table>
<thead>
<tr>
<th>Test case / Boat position</th>
<th>Left path navigation points</th>
<th>Right path navigation points</th>
<th>Total navigation points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Comparing tables 11 and 12 it seems that test case 4 stands out both in execution time and in amount of generated navigation points. Exceptionally large measured time and high number of generated navigation points seem to be connected in this case, although first test case has second highest count of generated navigation points but lowest measured time. On test cases 2, 3 and 5 have only four generated navigation points. Test cases 3 and 5 have rather similar measurement times and test case 2 has around 20 millisecond longer time. Let’s first investigate the first and fourth boat positions since they have the largest number of generated navigation points and also the largest difference in measured calculation times.

Let’s use Valgrind’s Callgrind tool to get more information what happens inside those points. Table 13 presents results for first boat position and table 14 presents results for fourth boat position.

Table 13: Valgrind Callgrind results for test case 1.

<table>
<thead>
<tr>
<th>Ir</th>
<th>Called</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>12 452</td>
<td>52</td>
<td>CalculateLaylines::buildWKTPolygon(QPolygonF const&amp;)</td>
</tr>
<tr>
<td>2371</td>
<td>28</td>
<td>CalculateLaylines::checkIntersection(QString const&amp;, QString const&amp;, QString)</td>
</tr>
<tr>
<td>1263</td>
<td>1</td>
<td>CalculateLaylines::getNearestPoint(QVector&lt;QPointF&gt; const&amp;, QPointF const&amp;)</td>
</tr>
<tr>
<td>1200</td>
<td>25</td>
<td>CalculateLaylines::checkIntersection(QString const&amp;, QPolygonF const&amp;, QPolygonF const&amp;)</td>
</tr>
<tr>
<td>907</td>
<td>2</td>
<td>CalculateLaylines::getPath(bool const&amp;, float const&amp;, int const&amp;, double const&amp;, double const&amp;, QPointF const&amp;, QPointF const&amp;, QPolygonF const&amp;, QVector&lt;QPointF&gt;&amp;)</td>
</tr>
</tbody>
</table>
Table 14: Valgrind Callgrind results for test case 4.

<table>
<thead>
<tr>
<th>Ir</th>
<th>Called</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 116</td>
<td>138</td>
<td>CalculateLaylines::buildWKTPolygon(QPolygonF const&amp;)</td>
</tr>
<tr>
<td>5 983</td>
<td>71</td>
<td>CalculateLaylines::checkIntersection(QString const&amp;, QString const&amp;, QString)</td>
</tr>
<tr>
<td>3 264</td>
<td>68</td>
<td>CalculateLaylines::checkIntersection(QString const&amp;, QPolygonF const&amp;, QPolygonF const&amp;)</td>
</tr>
<tr>
<td>2 064</td>
<td>1</td>
<td>CalculateLaylines::getNextPoint(QVector&lt;QPointF&gt; const&amp;, QPointF const&amp;, float const&amp;)</td>
</tr>
<tr>
<td>1 327</td>
<td>1</td>
<td>CalculateLaylines::getNearestPoint(QVector&lt;QPointF&gt; const&amp;, QPointF const&amp;)</td>
</tr>
</tbody>
</table>

Tables 13 and 14 are sorted by instruction counts (Ir) per function. Only the five most instruction heavy functions from both positions are shown. It's obvious that buildWKTPolygon(QPolygonF const&) dominates the calculation process in both occasions (test case 1: 12 452 Ir and test case 4: 32 116 Ir). Also, OProfile profile in table 7 confirms that most of the calculation time is spent in buildWKTPolygon(QPolygonF const&) when cumulation of all test cases are considered. Higher instruction counts in fourth boat position seems to be caused by the fact that buildWKTPolygon(QPolygonF const&) is called 138 times compared to modest 52 of first boat position. This obviously requires more investigation. What affect those call counts?

Figures 3 and 4 show Callgrind call-graphs from both test cases, showing from which functions the function buildWKTPolygon(QPolygonF const&) was called. Pictures are generated with KCachegrind.
It seems that getNextPoint(...) and getPath(...) functions through checkIntersection(QString const&, QString const&, QString) function are the culprits behind high call counts. Now the functions that use most of the time in calculation are known and order in which they are called. After this point, there’s clear guidance where to focus the investigation next. Code level inspection is now required.

Function buildWKTPolygon(QPolygonF const&) turns 2D polygon objects into a string format that can be fed to SQL database. This is used in checkIntersection(QString const&, QPolygonF const&) function that queries SQL database if geographical objects intersect. getNextPoint(...) function searches next point in long-term route that can be used as a next destination point. getPath(...) function calculates tacking points to get to the next destination point when sailing upwind. Both getNextPoint(...) and getPath(...) use checkIntersection(QString const&, QPolygonF const&, QPolygonF const&) extensively.

ggetNextPoint(...) function determines the next destination point on the long-term route in two phases. First, the algorithm tries to find utmost long-term navigation point that can be seen from boat without obstacles. This is done forming triangles using the boat position, the nearest position on the long-term route and consecutive long-term route waypoints until obstacles are detected inside a triangle. Finding an obstacle inside triangle means that latest triangle’s long-term navigation point cannot be seen directly from the boat. Obstacles are searched from every triangle using checkIntersection(...) function. Next, the triangle must then be “fine-tuned” so that the closest position on the long-term route to the utmost long-term navigation point can be directly seen from the boat. Fine-tuning halves the last triangle until the triangle contains no more obstacles. checkIntersection(...) function is also used to find obstacles in fine-tuning process.

From this getNextPoint(...) description it can be deduced that the amount of unobstructed long-term navigation points and how many triangle halvings must be done in fine-tuning affect how many times checkIntersection(QString const&, QPolygonF const&, QPolygonF const&) is called. Indeed, in cases 1 and 4 it can be verified from Callgrind...
source annotation that in test case 1 obstacles are searched for 3 times and in test case 4 for 20 times in the first phase. In fine-tuning obstacles are searched 2 times in test case 1 and 3 times in test case 4.

getPath(...) function determines turning points to reach the next destination point. Boat’s position in relation to next destination point, boat’s direction, wind and obstacles in the area affect how many tacks are required to reach the next destination. Every new tacking point calculation that is calculated requires additional checkIntersection(...) function calls. Boat’s position that generates short-term route with more tacking points, makes more SQL queries and spends more time.

Both test cases 1 and 4 generate quite lot of navigation points; 12 and 14 respectively. This makes sense when the wind direction and direction of navigation are considered. More turnings (tacks) are required to reach the destination, when sailing towards wind. In test case 4 boat position is especially difficult. Path for navigation is very narrow with obstacles on both sides towards the next destination point. In other test cases 2, 3 and 5 boat is either reaching which requires less tacking points. Obviously, amount of tacking points affect the amount of SQL queries and thus the measured time, but between test cases 1 and 4 it’s obvious that most of the difference comes from searching the utmost point on the long-term route.

In test cases 2, 3 and 5 there are only 4 short-term navigation points created in each case, but test case 2 has 20 milliseconds higher average measurements than in test cases 3 and 5. These are all reaching situations when the wind blows perpendicular to the boat and no tacking points are created needed in upwind sailing. Let’s investigate test cases 2 and 3 more closely to find out where the difference in time comes from. Table 15 presents five of most instructions heavy functions from test case 2 and 3 from Callgrind.

<table>
<thead>
<tr>
<th>Function</th>
<th>Test case 2 (Ir)</th>
<th>Test case 3 (Ir)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CalculateLaylines::buildWKTPolygon(QPolygonF const&amp;)</td>
<td>12 548</td>
<td>24 196</td>
</tr>
<tr>
<td>CalculateLaylines::checkIntersection(QString const&amp;, QString const&amp;, QString)</td>
<td>2 539</td>
<td>4 723</td>
</tr>
<tr>
<td>CalculateLaylines::getNextPoint(QVector&lt;QPointF&gt; const&amp;, QPointF const&amp;, float const&amp;)</td>
<td>2 382</td>
<td>3 928</td>
</tr>
<tr>
<td>CalculateLaylines::checkIntersection(QString const&amp;, QPolygonF const&amp;, QPolygonF const&amp;)</td>
<td>1 296</td>
<td>2 544</td>
</tr>
<tr>
<td>CalculateLaylines::getNearestPoint(QVector&lt;QPointF&gt; const&amp;, QPolygonF const&amp;)</td>
<td>1 269</td>
<td>1 283</td>
</tr>
</tbody>
</table>

From table 15 can be seen that even though, test case 2 took on average more time than test case 3, test case 2 has much lower count of instruction fetches. getNextPoint(...) seems to be the biggest source of instruction counts in both test cases, so let’s examine what happens in that function. From Callgrind’s source annotation it can be seen that in
finding the utmost unobstructed point on the long-term route in test case 2 obstacles are searched for 9 times and in test case 3 for 23 times. In fine-tuning obstacles are searched for 7 and 5 times respectively. Call counts and instruction counts from Callgrind doesn’t seem to explain time usage in this case. When the wall-time to do obstacle searches in getNextPoint(...) function in both test cases were measured, it was noticed that checkIntersection(...) function calls in test case 2 in fact took constantly more time than in test case 3. This would indicate that SQL queries generated in test case 2 took more time to complete than queries from test case 3.

Results from profiling tools, such as Valgrind must be interpreted cautiously. If checkIntersection(QString const&, QString const&, QString) is waiting for answer from database to an SQL query, does it execute instructions? The function uses PQExec function from libpq to execute the query on the database. PQExec is a synchronous function and while it's executing the client application is suspended\(^\text{10}\). Valgrind doesn’t account for kernel activity or activity of other processes (Valgrind User Manual, 2014). This might explain why Callgrind’s results were confusing at first in this case.

4.7 UpWind performance regression testing

UpWind project has suffered performance problems for a long time. It’s obvious that software project evolves though code changes. But code changes also always introduce a risk of functional or quality degradations. Regression tests addresses directly to degradations due to code changes in development. In this chapter, performance regression testing and testing process for UpWind are introduced.

4.7.1 Performance regression tests

Performance regressions during development are known but complex issue. Source code changes in a project can cause severe performance degradations (Liu, 2009). Performance problems can accumulate slowly over time from various sources without any specific instance when performance drops dramatically (Corbet, 2010). If not rigorously addressed the performance of software may degrade significantly without being noticed (Liu, 2009).

Functional regression tests are important in maintaining functionality of software and that specification requirements are met when the development is ongoing (Leung and White, 1989; Wong, Horgan, London and Agrawal, 1997; Hailpern and Santhanam, 2002). Often fixing one problem or implementing new functionality can introduce new unexpected issues that are not immediately visible. Regression testing may be used frequently in development phase when new features are added or in maintenance phase (Leung and White, 1989). Regression tests must usually be modified if software specifications change (Leung and White, 1989). Tests that check important functionality and quality targets of developed software increase trust that new modifications to the code-base won’t break any important part of the system (Leung and White, 1989). However, because more regression tests take more resources and time to run, selected test cases can be a trade-off between test coverage and resources (Wong et al., 1997).

Functional performance testing is well known by the developers, and same principles apply to performance regression testing also (Liu, 2009). Performance regression testing

\(^{10}\)http://www.postgresql.org/docs/current/static/libpq-async.html
can even be integrated to traditional regression testing (Foo, Jiang, Adams, Hassan, Zou & Flora, 2010). Performance regression testing directly addresses the performance degradations during development. The performance regression tests aim to detect performance degradations caused by new software modifications (Liu, 2009). Performance regression testing is important for minimizing risks of the deployed software as well as keeping track of performance evolution of software (Liu, 2009).

Functional unit testing is used to test individual units or components of code, but can also be used in context of performance testing (Horký et al., 2013). Also, Molyneaux (2009) discusses about performance testing of only the components that have changed. He calls this kind of testing smoke testing. Performance unit testing might not be that simple. That’s partly due to notion that it’s difficult to determine what’s adequate performance for individual code elements for functions instead to whole end-to-end performance (Horký et al., 2013) and performance regression testing still might need manual judgement of possible performance problems (Foo, Jiang, Adams, Hassan, Zou & Flora, 2010). Implication is that it’s difficult to determine good performance goals for individual software components.

4.7.2 Performance regression testing process

Performance regression tests need to assure that new source code modifications won't cause performance degradations. It's important that regression tests cover important user scenarios (Liu, 2009). It's also important to keep performance regression tests up-to-date when software project evolves. Also, performance tests must be evaluated against comparable previous values. For this reason, it's important to always use the same testing procedure (Liu, 2009).

Foo et al. (2010) outline typical process of performance regression testing:

1. Performance tests are started and relevant metrics are collected
2. Averages of result metrics are compared against pre-defined threshold values, after the tests are completed
3. Past and new performance metrics are compared visually for performance regression between test runs
4. All performance data is archived in central repository for bookkeeping purposes

For UpWind performance regression testing a process is suggested. It’s a modification of testing process by Foo et al. (2010). The biggest change to testing process by Foo et al. (2010) is that there’s currently no threshold values for performance tests. Liu (2009) suggests running performance regression tests before and after the code changes to see the effects of changes in current environment. Furthermore, performance regression tests can be used as a baseline for further tests. Suggested performance regression testing process for UpWind project:

1. Performance tests are started and relevant metrics are collected as baseline results.
2. Modifications to the project are made.
3. Performance tests are run again and the same metrics collected as in the first step.

4. After tests performance results from steps 1 and 3 are compared to catch performance regressions.

5. Performance results should be archived to monitor the performance evolution of software.

Performance tests are stochastic in nature, there's going to be some amount of fluctuation in measurements in any case (Liu, 2009). Liu (2009) suggests that measurements of less than 10% performance degradation or improvement should be interpreted cautiously.

This process is not automated and requires attention of the developer in running the tests and in interpreting the results. Machines used by different developers are bound to be different and with different calculation powers. That's why it's important that each developer runs the tests on their own environment before and after the code modifications. It's important that also the environment where the tests are run stays comparable between test runs. Developer must be sure that noise of the system stays relatively same between the test runs. Separate, dedicated machine for tests would be the best solution.

No method for archiving the performance results is suggested in this thesis, although it is highly recommended in order to follow the performance evolution of project.

4.7.3 Performance tests using QTestLib

Since UpWind is developed using Qt libraries, it’s most natural to create performance regression tests using Qt’s QTestLib testing framework. QTestLib allows testing parts of classes in a function level granularity. QTestLib supports different performance test back-ends and can provide results in various ways. QTestLib was more thoroughly described in chapter 3.2.4.

Performance regression test with QTestLib consists of initializing the test session and environment, executing the test and then interpreting the results. Database containing the chart data must be started with realistic chart data before the tests are run. UpWind’s plugin system should be initialized to be able to test short-term route planning. When the system is initialized, the function that calculates short-term route is tested with Qt’s benchmarking tools. Tests provide wall-times of short-term route calculation algorithm. QTestLib should provide pretty reliable results since framework automatically repeat tests to minimize randomness in results and iteration count can also be manually set.

It must be noted that while in complete implementation of UpWind, short-term route calculation is done in separate thread from the main UI this is not to be case with the tests. In these tests we are only interested in throughput of short-term route planning algorithm and not the function or quality of the main UX experience, although improving short-term algorithm enables faster update frequency of short-term navigation aides on UI and prevent possible queuing of short-term route planning events.
It's quite convenient to write simple performance tests with QTestLib where Performance testing is done in bm_CalculateLaylines::testShortTermPerfRegression().

```cpp
void bm_CalculateLaylines::testShortTermPerfRegression() {
    QFETCH(QPointF, boatPosition);

    CalculateLaylines cl;
    cl.pathPoints = bm_CalculateLaylines::longTermRoute;
    cl.startPoint = boatPosition;
    cl.calculationOnGoing = false;

    QBENCHMARK {
        cl.startCalc();
    }
}
```

First CalculateLaylines class is initialized with long-term route and current boat position. It's required to make bm_CalculateLaylines a friend class to CalculateLaylines to expose private members pathPoints, startPoint and calculationOnGoing in tests. QFETCH macro is used to acquire test data for different test cases 1 – 5. Between test cases only input data is different and calling of startCalc() stays the same. Only the code inside QBENCHMARK macro is measured in tests. Appendix B contains full code listing bm_CalculateLaylines class.

For building UpWind release configuration was used, meaning that no debug symbols were inserted and O2 level compiler optimizations were used. Qt 5.2.1 version for Linux 64-bit was used in building UpWind and test cases. Table 16 presents the measured times from performance regression testing.

**Table 16: Example performance regression test results.**

<table>
<thead>
<tr>
<th>Test case</th>
<th>Mean wall-time (msecs) by QTestLib benchmarks from 100 iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>88.42</td>
</tr>
<tr>
<td>2</td>
<td>148.93</td>
</tr>
<tr>
<td>3</td>
<td>118.87</td>
</tr>
<tr>
<td>4</td>
<td>165.09</td>
</tr>
<tr>
<td>5</td>
<td>123.56</td>
</tr>
</tbody>
</table>

Each test case was executed consecutively 100 times. Table 16 presents mean times of iterations automatically calculated by QTestLib. When compared to results from code instrumentation in table 11, it can be seen that even though individual test case times are slightly different, the overall test results follow the same form.
5. Discussion

The research problem for this thesis was stated as: *How unintrusive profiling and performance testing can be used in evaluation of software artefact to produce new information about performance properties of the artefact and in improving performance evolution of the artefact in development.*

Evaluation study in Design Science Research was chosen to be the research method that guided the thesis. Evaluation study is based on questions that the study seeks to answer (Hevner & Chatterjee, 2010). The research problem of this thesis can be answered by answering these questions:

- How to reliably measure performance of software?
- What attributes of program affect the performance?
- What kind of information can be elicited from software using profiling tools?
- What limitations performance profiling tools have?
- How to ensure that performance quality doesn't drop during development?

Literature research and UpWind software project were used to explore the questions in-depth. Literature research provided a theoretical background for performance analysis and testing. This information was used in UpWind project to acquire new information about the performance properties and also to create performance regression tests to improve stability of performance evolution of the project. From this work the following answers to the questions could’ve been found.

*How to reliably measure performance of software?* It's very important to measure performance reliably to catch the real problems and to prevent false alarms. Measuring software performance accurately and reliably shouldn't differ from any scientific measuring in rigour (Smith, 1990). Performance measurements are stochastic in nature (Liu, 2009). Some fluctuation between the test runs is almost impossible prevent. The fluctuation should be minimized by running the tests in static environment, keeping the testing process same and running tests for enough time to minimize the effect of randomness (Liu, 2009). All tests and profiling in this thesis was done on same computer, under similar workloads. The tests were always run consecutive hundred times to minimize the random effect from mean times. This method provided comparable performance results on one machine. It must be noted that measurements obtained in this thesis are not directly comparable in another computers, but measurements should still follow the same pattern.

Smith (1990) and Reddy & Kasi (2014) encourage using instrumentation in software performance measurements. Undoubtedly, instrumentation still has its place in performance engineering. Instrumentation can be laborious to implement in late phases of the development cycle. If it is designed carefully into the system, instrumentation can provide very specific information in a controlled manner (Smith, 1990).
Also, suitable performance metrics should be used. Performance metrics for different situations identified in this thesis are response time, throughput, availability, concurrency, scalability, resource utilization.

What attributes of program affect the performance? Performance of a particular program is a result from various factors. Poor design and architecture choices in a software are probably the biggest contributors to bad performance (Smith, 1990; McConnell, 2004; Smith & Williams, 2000; Smith & Williams, 2002). Usually a small part of the program causes most of the performance problems (McConnell, 2004; Weyuker & Vokolos, 2000). Choosing an improper data-structure or an algorithm for a task in software can cause performance effects on whole software if tasks start to queue. Even though, considering performance in every phase of the development is so important (Smith, 1990) it still can be an afterthought in many projects (Molyneaux, 2009).

What kind of information can be elicited from software using profiling tools? According to literature performance profiler tools are very beneficial in investigating software performance problems (Cohen, 2004; Floyd, 2012). Indeed, in this thesis UpWind project’s short-term route planning algorithm was analysed using Valgrind and OProfile. OProfile was very useful in pointing out the parts of the code that took most of the CPU time. This information can be used to direct the more fine grained performance analysis to appropriate places. Valgrind proved out to be a diverse but complicated tool in aiding to understand what happens in the code; how the functions are related, from where and how many times they were called.

What limitations performance profiling tools have? It's clear that profiling tools are not perfect. Valgrind has some well known limitations in its profiling coverage (Valgrind User Manual, 2014) and its handling of floating points (Floyd, 2012). Although Valgrind was generally useful in explaining the inner workings of software, during profiling it was noticed that Valgrind’s Callgrind tool doesn't always directly explain measured performance times. Acquired results always require interpretation and good knowledge of software and environment. Valgrind provides quite specialized information about events such as different levels of CPU cache usage, instruction reads and conditional branch executions. Interpreting these accurately might require information about the specific hardware and compiler. Valgrind doesn’t tell the developer how to fix problems it merely shows where they potentially are, provided the developer can interpret the results correctly.

OProfile relies on system hardware events in profiling, meaning that different systems might have different hardware events available. Descriptions and implications of those events must be checked from hardware manufacturer’s manual. Different profilers use different methods to collect the measurements. OProfile is a statistical profiler and Valgrind uses binary instrumentation. Statistical sampling methods are vulnerable to statistical errors, but accuracy should be enough for most uses (Levon, 2004). On the other hand, OProfile is quite unobtrusive. It produces only minimal overhead and can hardly be noticed when profiled program is ran. Valgrind instruments the profiled program’s binary in runtime and will slow the execution down significantly 10 - 50 times compared to running the program natively. Obviously, Valgrind tools are not ideal for measuring direct execution times, but they can give various information about including memory usage, cache usage, conditional branch behaviour, function call graphs and threading. GProf is also a viable tool to profile smaller programs, but couldn’t be used with UpWind because GProf lacks support for dynamic libraries.
How to ensure that performance quality doesn’t drop during development? Regression testing is very well known principle in software development. It ensures to prevent functional and quality regressions caused by functional implementations and maintenance. Performance regression testing is probably not so well known, even though performance regressions do happen (Corbet, 2010). Performance regression testing implemented in this thesis aims to maintain performance of particular component of UpWind project, the short-term route planning algorithm. The performance test runs short-term algorithm with some typical input data, in hopes to catch possible performance regressions in those cases. Performance tests were implemented using Qt Test framework, which proved to be quite efficient way to do performance tests in Qt project.

Not much previous literature was found about using Qt Test framework. Results acquired from tests created with Qt Test framework were comparable to results from manual instrumentation of code in terms of accuracy. Compared to instrumentation Qt Test framework offers few benefits. Firstly, source code doesn’t need to be modified to benchmark program. This helps to keep code base clean and separate module responsibilities in project. Secondly, performance tests can be executed using various different back-ends. There’s no need to settle to only simple wall-time measurement, when developers can also measure system events or use accurate measurements with Valgrind back-end without changing the code. Thirdly, Qt Test framework runs the test case multiple times if it’s required to get accurate measurements. Users can also set the iteration count without changing the code. Fourthly, Qt Test framework supports different ways to print benchmark information from plain text to XML.

Also, Qt Test framework has some limitations. Benchmarking can only be done on function level granularity. If performance of only part of the function should be benchmarked, instrumentation might still be the only way. It seems that development of Qt Test performance benchmarking option is still ongoing. At Qt version 5.2.1 not all command line options were functional, or they didn’t work properly. At the time, Qt Test framework calculates only simple mean times from benchmarks with multiple iterations. No more fine grained information can be got for statistical analysis. Also, Qt test framework didn't detect the installed Valgrind.

Wall-time measurements gathered performance regression tests should still be enough to spot drastic changes in performance. Because performance measurements are so prone to random fluctuation, therein lies the question: how large anomaly in performance measurements require further action? There probably are no one right answer to this. Liu (2009) suggests that one should be cautious in interpreting any performance measurement change less than 10% as an improvement or degradation of system performance.
6. Conclusion

Software performance is a complex matter. Factors that affect the software performance are abundant (Liu, 2009). There's no one magic bullet to cure all performance defects or to guarantee good performance in software project. Projects of today are still affected by performance problems (Molyneaux, 2009; Weyuker and Vokolos, 2000; Corbet, 2010). Performance defects caused by poor design and architecture choices can be very hard and expensive to fix. It seems that best way to mitigate performance problems is careful planning and considering performance in every phase of the project’s life-cycle (Smith, 1990). Good designs, architecture and sound principles of coding are a good way to ensure the best possible quality outcome of software development project (McConnell, 2004). However, software tuning can’t always be avoided, but when it's done, it usually comes with a price. Software tunings tend to work against the good code writing policies. Tunings can make code more complex, confusing and hard to maintain (McConnell, 2004).

The purpose of this thesis was to research how unintrusive performance profiling and testing can be used in acquiring new information about an implemented artefact and to improve performance evolution of the artefact in development. Evaluation study approach in context of Design Science Research was chosen as research method. UpWind project was used to explore profilers and performance testing. Also, new information about performance behaviour was acquired from the UpWind system. Hevner and Chatterjee (2010) suggest that DSR researchers should know basics of software performance. To cover the basics of software performance and topics of profiling and testing a literature review was conducted. Also, this helped to put the study on a theoretical background. As a part DSR evaluation study questions for evaluation were formed. These questions guided both the literature review and the evaluation of performance. The evaluation study focused on the most essential part of UpWind project, the short-term route planning algorithm.

Firstly performance test cases were created. Representative long-term route was created and five points chosen on it formed the test cases used through the profiling and testing of program. Initial performance information in those five test cases were acquired by Qt timers. It's well known that performance measurements are stochastic in nature (Liu, 2009). Indeed, performance measurements contained some interference but they were accurate enough for comparing differences between test cases. These differences were used as a basis for performance profiling. Valgrind and OProfile were chosen as suitable profiling tools for UpWind project. Using OProfile the most time consuming functions were able to be identified. Valgrind's Callgrind tool provided more in depth information about function call traces and behaviour of the algorithm.

Instrumentation is seen as highly valuable way to acquire performance measurements from software (Smith, 1990; Reddy & Kasi, 2014). Implementing instrumentation in late development phases can be difficult and laborious. It's also an intrusive action, it's required to compile the program to see the effects. Unintrusive profilers like OProfile, Valgrind and GProf remove this step. These profilers provide fast access to performance
information of an application or even whole system. Especially, using some graphical user interface, like KCachegrind can really help in analysing bigger programs.

To ensure that in future development software performance wouldn’t degrade uncontrollably performance regression tests were implemented using Qt’s QTestLib testing framework. Also, a simple process was suggested to how to use regression testing without automation.

This thesis contributes to the existing knowledge by providing a thorough analysis of performance behaviour of UpWind’s short-term route planning algorithm. Also, a situation where Valgrind’s Callgrind tool provided unexpected information was found during the profiling. Also, no previous studies using QTestLib for performance testing were found and as far as I know this is the first one. QTestLib provides easy and fast way to create performance benchmark tests in applications using Qt.

This thesis didn’t produce a method for automating performance regression tests. It’s suggested that tests are executed every time before and after new implementation or modification to the code base. At least every code change to UpWindScene plugin should be tested for performance regressions. Executing performance regression tests for changes to other parts of the code is left at the judgement of individual developers.

There are some limitations to this thesis. Choosing test cases is a tricky question. Only five boat positions were tested for short-term algorithm performance in this thesis. It’s probable that more test cases would bring more varied examples about performance in different situations. With a sample of five randomly chosen positions, there were huge variations in performance results between test cases. Also, testing the same boat positions with different wind angles might provide further information about how the short-term algorithm performs in different situations.

Performance tests and profiling weren’t done on the maritime computer that would usually be used in running the UpWind. This, of course, has effect on wall-time measurements gathered from instrumentation and performance regression tests, but it doesn’t affect how the code fundamentally works. Logic of the code is not affected by hardware platform change and measured wall-times should essentially follow the same figure, although the scale might change. To find the performance problems affecting in the real environment the tests should be ran also on maritime computer. Further study on finding difference between these results and results on maritime computer might prove to be interesting.

Actual memory usage or resource utilization weren't assessed in this thesis. How UpWind uses system resources should probably be done on whole system level in maritime computer. This would give insights where to focus future performance work. Also, it would be beneficial to monitor resource utilization and performance behaviour over longer periods of time.
Reference


Appendix A. Code listing of short-term algorithm

Contains selected functions from calculatelaylines.cpp. Functions are presented here as they are in the code base. I have not implemented these code lines. They are printed here to give the full disclosure about the analysis discussed in this thesis. Small layout modifications are made to make them more readable on these pages. Selected functions are presented in this order: startCalc, updateCheckPoint, getNextPoint, getNearestPoint, updateLaylines, getPath, checkIntersection and buildWKTPolygon. Full source code can be found from: https://github.com/UpWind/devel/blob/master/upwind/src/UWPlugins/UpWindScene/S

cene/calculatelaylines.cpp.

```cpp
void CalculateLaylines::startCalc(){
    if (!this->calculationOnGoing){
        this->calculationOnGoing = true;
        this->obstacleFound = false;
        QVector<QPointF> layLines;
        QVector<QPointF> rightpath;
        QVector<QPointF> leftpath;

        if (this->pathPoints.size() > 0){
            this->openPostgreConnection();
            this->ACCU_OFFSET = 1;
            this->MAX_TURNING_POINTS = 5;
            this->geoBoatPos = this->startPoint;
            this->pathPoints = this->pathPoints;
            /*
             * Start process of calculating laylines.
             * First get the destination point on long term route
             * (no obstacles between boat and long term route point).
             */
            this->updateCheckPoint();
            /*
             * Then calculate the laylines that takes
             * the boat to that destination point
             */
            this->updateLaylines();
            rightpath = *pRightPath;
            leftpath = *pLeftPath;

            for (int i = 0; i < rightpath.size(); i++) {
                layLines.append(rightpath.at(i));
            }

            for (int i = 0; i < leftpath.size(); i++) {
                layLines.append(leftpath.at(i));
            }

            this->layLines = layLines;
        }
        this->calculationOnGoing = false;
        this->closePostgreConnection();
        emit emitLaylines(this->layLines);
    } else {
```
this->calculationOnGoing = false;
}
}

void CalculateLaylines::updateCheckPoint()
{
    // Let's find out which is the next point in our route.
    // Find the destiny check point in geographical format:
    this->geoDestinyPos = this->getNextPoint(this->pathPoints, geoBoatPos, ACCU_OFFSET);
    this->destinyPos = UwMath::toConformalInverted((const QPointF)geoDestinyPos);
}

QPointF CalculateLaylines::getNextPoint( const QVector<QPointF> &route, const QPointF &boatPos, const float &offset)
{
    int nearest_point = getNearestPoint(route, boatPos);
    if ( checkIntersection( "obstacles_r", boatPos ) ) {
        // if we are inside an obstacle, don't even try
        return boatPos;
    }
    // we got the nearest point in the route
    // but could be far, let's find out the real
    // nearest point of the route by making a
    // projection towards it...
    // START SEARCH OF PROJECTION FOR FINE TUNE
    // 2. We'll get projections to each point on the route:
    QPointF projection_point;
    QLineF route_line, projection_line;
    QPointF a, b, c;

    if ( route.size() >= 2 ) {
        // If nearest is the last one:
        if ( route.size() - nearest_point == 1 ) {
            a = UwMath::toConformal( route.at( nearest_point - 1 ) );
            b = UwMath::toConformal( route.at( nearest_point ) );
            c = UwMath::toConformal( (const QPointF)boatPos );
            projection_point = UwMath::getProjectionPoint( a, b, c);
            UwMath::fromConformal( projection_point);
            route_line.setP1( route.at( nearest_point - 1 ) );
            route_line.setP2( route.at( nearest_point ) );
            projection_line.setP1( boatPos );
            projection_line.setP2( projection_point );
            // If nearest is the first one:
        } else if ( nearest_point == 0 ) {
            a = UwMath::toConformal( route.at( nearest_point ) );
            b = UwMath::toConformal( route.at( nearest_point + 1 ) );
            c = UwMath::toConformal( (const QPointF)boatPos );
            projection_point = UwMath::getProjectionPoint( a, b, c);
            UwMath::fromConformal( projection_point);
            route_line.setP1( route.at( nearest_point ) );
            route_line.setP2( route.at( nearest_point + 1 ) );
            projection_line.setP1( boatPos );
            projection_line.setP2( projection_point );
            // If nearest is not first or last one:
        } else {
            a = UwMath::toConformal( route.at( nearest_point ) );
            b = UwMath::toConformal( route.at( nearest_point + 1 ) );
        }
    }
c = UwMath::toConformal((const QPointF)boatPos);
projection_point = UwMath::getProjectionPoint(a, b, c);
UwMath::fromConformal(projection_point);
route_line.setP1(route.at(nearest_point));
route_line.setP2(route.at(nearest_point + 1));
projection_line.setP1(boatPos);
projection_line.setP2(projection_point);

// We must check if our position is between nearest and nearest + 1 or between nearest - 1 and nearest:
if (!checkIntersection(route_line, projection_line)) {
    a = UwMath::toConformal(route.at(nearest_point - 1));
b = UwMath::toConformal(route.at(nearest_point));
c = UwMath::toConformal((const QPointF)boatPos);
projection_point = UwMath::getProjectionPoint(a, b, c);
UwMath::fromConformal(projection_point);
route_line.setP1(route.at(nearest_point - 1));
route_line.setP2(route.at(nearest_point));
projection_line.setP1(boatPos);
projection_line.setP2(projection_point);
}

// Next we'll find the checkpoint:
//3. We'll check if there are obstacles in the projected triangle:
if (route.size() < 1) {
} else if (route.size() == 1) {
    // Route to process only has one point:
    QLineF heading(boatPos, route.at(nearest_point));
    bool hobs_r = checkIntersection("obstacles_r", heading);
    bool hobs_l = checkIntersection("obstacles_l", heading);
    // Finetune checkpoint in the heading:
    if (hobs_r || hobs_l) {
        this->obstacleFound = true;
        bool ready = false;
        QLineF last_heading;
        while (!ready) {
            if ((hobs_r &&
                checkIntersection("obstacles_r", heading)) ||
            (hobs_l &&
                checkIntersection("obstacles_l", heading))) {
                last_heading = heading;
                heading = UwMath::lineToHalf(heading);
            } else {
                if (checkOffset(heading, last_heading, offset)) {
                    ready = true;
                } else {
                    heading = UwMath::avgLine(heading, last_heading);
                }
            }
        }
    }
}
// We won't return the point between boat and checkpoint as the next checkpoint on long-term route:
return heading.p2();
} else {
    // We won't return the point between boat and checkpoint as the next checkpoint on long-term route:
    return heading.p2();
}
} else if ( route.size() > 1 ) {
    // Route to process has more than 1 point:
    int i = nearest_point;
    QPolygonF triangle;
    // 1st point:
    triangle << boatPos;

    // 2nd point:
    if ( checkIntersection( route_line, projection_line) ) {
        triangle << projection_point;
    } else {
        triangle << route.at( nearest_point);
    }

    // 3rd point:
    if ( route.at( nearest_point) == route_line.p2() ) {
        triangle << route.at( nearest_point);
        i = nearest_point - 1;
    } else if ( route.at( nearest_point) == route_line.p1() ) {
        triangle << route.at( nearest_point + 1);
    }

    // SPECIAL CHECK to find obstacles in the heading of the first
    // triangle
    if ( triangle.at( 0 ) == boatPos ) {
        QLineF heading( triangle.at( 0), triangle.at( 1));
        QPolygonF last_triangle;
        bool hobs_r = checkIntersection( "obstacles_r", heading,
                                   triangle );
        bool hobs_l = checkIntersection( "obstacles_l", heading,
                                   triangle );
        // FINETUNE CHECKPOINT IN THE HEADING
        if ( hobs_r || hobs_l ) {
            bool ready = false;
            QLineF last_heading;
            while ( !ready ) {
                if ( ( hobs_r && checkIntersection( "obstacles_r",
                                  heading, triangle) ) ||
                     ( hobs_l && checkIntersection( "obstacles_l",
                                  heading, triangle)) )
                {
                    last_heading = heading;
                    heading = UwMath::lineToHalf( heading);
                    last_triangle = triangle;
                    triangle = UwMath::triangleToHalf(triangle);
                } else {
                    if (checkOffset_OnePointVersion(
                        last_triangle, triangle, offset ))
                        { ready = true;
                    } else {
                        triangle =
                        UwMath::avgTriangle_OnePointVersion
                        ( triangle, last_triangle);
                    }
                }
            }
            return heading.p2();
        }
    }
bool obs_r = checkIntersection( "obstacles_r", triangle, triangle );
bool obs_l = checkIntersection( "obstacles_l", triangle, triangle );

while ( !obs_r && !obs_l && i < (route.size() - 2) ) {
    triangle.clear();
    triangle << boatPos;
    triangle << route.at( i );
    triangle << route.at( i + 1 );
    obs_r = checkIntersection( "obstacles_r", triangle, triangle );
    obs_l = checkIntersection( "obstacles_l", triangle, triangle );
    // Finetune checkpoint:
    if ( obs_r || obs_l ) {
        bool ready = false;
        QPolygonF last_triangle;
        while ( !ready ) {
            if ( ( obs_r && checkIntersection( "obstacles_r", triangle, triangle ) ) ||
                ( obs_l && checkIntersection( "obstacles_l", triangle, triangle ) ) )
            {
                last_triangle = triangle;
                triangle =
                    UwMath::triangleToHalf_OnePointVersion( triangle );
            } else {
                if (checkOffset_OnePointVersion( last_triangle, triangle, offset)){
                    ready = true;
                } else {
                    triangle =
                        UwMath::avgTriangle_OnePointVersion( triangle, last_triangle );
                }
            }
            //If the points in the line segment on the side of
            //the route there is no space for the boat in
            //the map anymore and we can quit the search:
            if(triangle.at(1) == triangle.at(2))
                ready = true;
        }
        i++;
    }
}
return triangle.at( 2 );

// End the process of searching for the checkpoints.
return QPointF();

} // END SPECIAL CHECK
int CalculateLaylines::getNearestPoint( const QVector<QPointF> &route,
        const QPointF &boatPos) {
    // Input should be in GEOGRAPHICAL format.
    // We receive a route here as a list of points.
    // Let's see at what point of the route we are:
    double distance = std::numeric_limits<double>::max();
    int nearest_point = 0;
    // 1. We'll get the shortest distance to boat:
    // Here we get a point where we "join" the route:
    for ( int i = 0; i < route.size(); i++) {
        double temp = UwMath::getDistance( boatPos, route.at(i));
        if ( temp < distance ) {
            distance = temp;
            nearest_point = i;
        }
    }
    return nearest_point;
}

void CalculateLaylines::updateLaylines() {
    this->pPolarDiagram->populate();
    // LayLines are not calculated with the actual TWA,
    // but the TWA that we will have when heading towards our destiny.
    //************HARDCODED VALUE FOR futureTrueWindAngle*************
    this->trueWindDirection = 90;
    float futureTrueWindAngle = UwMath::getTWA(geoBoatPos,
        geoDestinyPos, trueWindDirection );
    //************HARDCODED VALUE FOR windSpeed*************
    windSpeed = 10;
    layLinesAngle = pPolarDiagram->getAngle(windSpeed,
        futureTrueWindAngle);
    // new paths...
    pLeftPath = new QVector<QPointF> ;
    pRightPath = new QVector<QPointF> ;
    if ( layLinesAngle != 0 ) {
        // Here we are not reaching:
        QPointF TPleft, TPright;
        // Get the turning point without taking care of obstacles:
        UwMath::getTurningPoints( TPleft, TPright,
            trueWindDirection, layLinesAngle,
            geoBoatPos, geoDestinyPos);
        // ORDER IS VERY IMPORTANT!!
        // This is our area of interest for obstacles checking
        QPolygonF rhomboid;
        rhomboid << geoBoatPos;
        rhomboid << TPleft;
        rhomboid << geoDestinyPos;
        rhomboid << TPright;
        // Checking of the heading is made when getting
        // the checkpoint here.
        
        // Next we'll go for obstacles checking:
        if ( checkIntersection( "obstacles_r", rhomboid, rhomboid ) ||
             checkIntersection( "obstacles_l", rhomboid, rhomboid ) ) {
            // If we have polygon obstacles in the area
            // OR we have line obstacles in the area:
// Populate the right path:
getPath( true, ACCU_OFFSET, MAX_TURNING_POINTS,
 trueWindDirection, layLinesAngle,
 geoBoatPos, geoDestinyPos, rhomboid,
 *pRightPath);

// Populate the left path:
getPath( false, ACCU_OFFSET, MAX_TURNING_POINTS,
 trueWindDirection, layLinesAngle,
 geoBoatPos, geoDestinyPos, rhomboid, *pLeftPath);

} else {
 // If we don't have any obstacle in our area
 // use master turning points for the path:
 pRightPath->append( geoBoatPos);
pRightPath->append( TPright );
pRightPath->append( geoDestinyPos);
pLeftPath->append( geoBoatPos);
pLeftPath->append( TLeft );
pLeftPath->append( geoDestinyPos);
}

// End of checking
} else {
 // LayLines angle = 0
 // Here we are reaching:
pRightPath->append( geoBoatPos);
pRightPath->append( geoDestinyPos);
pLeftPath->append( geoBoatPos);
pLeftPath->append( geoDestinyPos);
}

void CalculateLaylines::getPath( const bool &side, const float &offset, const int &max_turns,
 const double &windAngle, const double &layLinesAngle, const QPointF &boatPos,
 const QPointF &destinyPos, const QPolygonF &obstacles_shape,
 QVector<QPointF> &Path )
{
 //Input has to be Geographical data because
 // we check against PostGIS
 QPointF TLeft, TRight;
 bool sideRight = side;
 bool ready, obs_r, obs_l = false;
 int count = 0;
 // Path is returned by reference, clear:
 Path.clear();
 // First point in the path is the origin:
 Path.append( boatPos);
 QPolygonF present_triangle;
 QPolygonF last_triangle;
 // While the last point in the path is not the destiny
 // AND we don't exceed the maximum turning points setting
 // we'll execute following:

 while ( Path.last() != destinyPos && count < max_turns ) {
 // Clear triangle:
 present_triangle.clear();
 // Get the turning points between the last point and dest.
 // For the first run last point in the path is origin:
 UwMath::getTurningPoints( TLeft, TRight,
 windAngle, layLinesAngle,
Path.last(), destinyPos);  

// Build a new triangle  
present_triangle << Path.last();  
if (sideRight) present_triangle << TPright;  
else present_triangle << TPleft;  
present_triangle << destinyPos;  

// Check for obstacles in the triangle  
obstacleFound = false;  
obs_r = checkIntersection( "obstacles_r", present_triangle, obstacles_shape );  
obstacleFound = true;  
obs_l = checkIntersection( "obstacles_l", present_triangle, obstacles_shape );  
if ( obs_r || obs_l ) {  
  obstacleFound = true;  
  // Obstacles found: let's reduce the triangle  
  // until they disappear:  
  ready = false;  
  last_triangle = present_triangle;  
  // Reduce triangle to half:  
  present_triangle = UwMath::triangleToHalf(  
    present_triangle  
  );  
  while ( !ready ) {  
    // Check again, put booleans first and  
    // checkIntersection won't even be called:  
    if ( ( obs_r && checkIntersection( "obstacles_r",  
      present_triangle, obstacles_shape ) ) ||  
      ( obs_l && checkIntersection( "obstacles_l",  
      present_triangle, obstacles_shape ) ) )  
    {  
      // Still obstacles: reduce again!  
      last_triangle = present_triangle;  
      present_triangle = UwMath::triangleToHalf(  
        present_triangle  
      );  
    } else {  
      // No more obstacles:  
      if ( checkOffset( last_triangle,  
        present_triangle, destinyPos, offset ) )  
      {  
        // The distance is acceptable, finetune done.  
        ready = true;  
      } else {  
        // We are too far, let's increase the triangle  
        // a bit:  
        present_triangle = UwMath::avgTriangle(  
          present_triangle, last_triangle);  
      }  
    }  
  }  
}  

// This triangle has no obstacles inside:  
Path.append( present_triangle.at( 1 ) );  
Path.append( present_triangle.at( 2 ) );  
// Add a turn:  
count++;  
// We continue in the other side:  
sideRight = !sideRight;
bool CalculateLaylines::checkIntersection( const QString &layerName,
    const QString &object, QString shape = QString() ) {
    QString sql("SELECT * FROM ( SELECT DISTINCT Intersects(
        wkb_geometry, ");
    sql.append( object);
    sql.append( " ) AS result FROM ");

    // if shape is empty string, then there is no area of restriction
    if ( shape.isEmpty() ) {
        sql.append( layerName);
        sql.append( " ) AS intersection ");
    } else {
        sql.append( layerName);
        sql.append( " WHERE wkb_geometry && GeometryFromText( ");
        sql.append( shape);
        sql.append( " ) AS intersection ");
    }

    sql.append( " WHERE result = TRUE" );
    res = PQexec(conn, sql.toLatin1() );
    bool intersection = ( PQntuples(res) > 0 );
    PQclear(res);
    return intersection;
}

QString CalculateLaylines::buildWKTPolygon(const QPolygonF &rhomboid)
{
    QString WKTPolygon = "POLYGON("
    for ( int i = 0; i < rhomboid.size(); i++ ) {
        WKTPolygon.append(QString::number(rhomboid[i].x(), 'f',
            WKT_P));
        WKTPolygon.append(" ");
        WKTPolygon.append(QString::number(rhomboid[i].y(), 'f',
            WKT_P));
        WKTPolygon.append("," );
    }
    WKTPolygon.append(QString::number( rhomboid.at(0).x(), 'f',
            WKT_P));
    WKTPolygon.append(" ");
    WKTPolygon.append(QString::number( rhomboid.at(0).y(), 'f',
            WKT_P));
    WKTPolygon.append(" )" );
    return WKTPolygon;
Appendix B. Code listing of performance regression test with QTestLib

bm_calculatelaylines.h:

```cpp
class bm_CalculateLaylines : public QObject {
    Q_OBJECT
    static QVector<QPointF> longTermRoute;

private slots:
    void testShortTermPerfRegression_data();
    void testShortTermPerfRegression();
};
```

bm_calculatelaylines.cpp:

```cpp
QVector<QPointF> bm_CalculateLaylines::longTermRoute;

void bm_CalculateLaylines::testShortTermPerfRegression_data() {
    bm_CalculateLaylines::longTermRoute << QPointF(25.1216, 65.0567)
        << QPointF(25.1219, 65.0569)
        ... << QPointF(25.0129, 65.2415);

    QTest::addColumn<QPointF>("boatPosition");
    QTest::newRow("1. boat position") << QPointF(25.087, 65.0584);
    QTest::newRow("2. boat position") << QPointF(25.216, 65.0702);
    QTest::newRow("3. boat position") << QPointF(25.104, 65.1128);
    QTest::newRow("4. boat position") << QPointF(25.1209, 65.1201);
    QTest::newRow("5. boat position") << QPointF(25.1893, 65.1278);
}

void bm_CalculateLaylines::testShortTermPerfRegression() {
    QFETCH(QPointF, boatPosition);
    CalculateLaylines cl;
    cl.pathPoints = bm_CalculateLaylines::longTermRoute;
    cl.startPoint = boatPosition;
    cl.calculationOnGoing = false;

    QBENCHMARK {
        cl.startCalc();
    }
}
```

QTEST_APPLESS_MAIN(bm_CalculateLaylines)