Team D Performance Report

20/11/2019

Contents

[Code Optimisers 3](#_Toc25145781)

[Types and Levels of Optimization 3](#_Toc25145782)

[What to Optimize 3](#_Toc25145783)

[Good and Bad Outcomes of Optimization 3](#_Toc25145784)

[Performance Tools 3](#_Toc25145785)

[WebLOAD 3](#_Toc25145786)

[SmartMeter.io 3](#_Toc25145787)

[Apache JMeter 3](#_Toc25145788)

# Code Optimisers

Code optimization is any method of code modification to improve code quality and efficiency. A program may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer input/output operations.

The basic requirements optimization methods should comply with, is that an optimized program must have the same output and side effects as its non-optimized version. This requirement, however, may be ignored in the case that the benefit from optimization, is estimated to be more important than probable consequences of a change in the program behaviour.

## Types and Levels of Optimization

Optimization can be performed by automatic optimizers, or programmers. An optimizer is either a specialized software tool or a built-in unit of a compiler (the so-called optimizing compiler). Modern processors can also optimize the execution order of code instructions.

Optimizations are classified into high-level and low-level optimizations. High-level optimizations are usually performed by the programmer, who handles abstract entities (functions, procedures, classes, etc.) and keeps in mind the general framework of the task to optimize the design of a system. Optimizations performed at the level of elementary structural blocks of source code - loops, branches, etc. - are usually referred to as high-level optimizations too, while some authors classify them into a separate ("middle") level (N. Wirth?). Low-level optimizations are performed at the stage when source code is compiled into a set of machine instructions, and it is at this stage that automated optimization is usually employed. Assembler programmers believe however, that no machine, however perfect, can do this better than a skilled programmer (yet everybody agrees that a poor programmer will do much worse than a computer).

## What to Optimize

With manual code optimization, one faces another problem: one doesn't just need to know how exactly optimization should be done, but also what particular part of the program should be optimized. Due to various reasons (slow input operations, the difference in the working speed of a human operator and a computer, and so on), 90% of the execution time of a program is spent executing only 10% of the code (this statement is rather speculative, with the Pareto principle as a quite doubtful ground, but A. Tanenbaum makes it sound convincing). Since optimization takes additional time aside from the time you've spent on developing the program, you'd better focus on optimizing this time-critical 10% of code rather than try to optimize the whole program. These code fragments are known as bottlenecks, and can be detected by special utilities - profilers - which can measure the time taken by various parts of the program to execute.

In practice, however, optimization is usually done after the stage of "chaotic" programming (including such methods as "Copy-Paste", "we'll see later", "it's OK this way"), and therefore is a mixture of optimization as such, refactoring and bugfixes: simplification of "queer" constructs like strlen(path.c\_str()), logical conditions like (a.x != 0 && a.x != 0), and so on. Profilers are of little help with this kind of optimization. Nevertheless, you can detect these issues with static analysis tools, i.e. tools designed to search for semantic errors, relying on deep analysis of source code. As you can see from the above mentioned example with the strange condition, inefficient code may appear as a result of errors (like a misprint in our example, where a.x != 0 && a.y != 0 should be instead). A powerful static analyser will detect such code fragments and draw your attention to them by producing warning messages.

## Good and Bad Outcomes of Optimization

In programming, almost everything should be treated from the viewpoint of rationality - optimization is no exception. There is a belief that code written by an inexperienced Assembler programmer is 3-5 times slower than code generated by the compiler (Zubkov). Widely known is a phrase by Knuth regarding early low-level optimizations (such as attempts to save on operators or variables): "Premature optimization is the root of all evil".

Most programmers don't complain about optimizations performed by the optimizer, some of which are conventional and obligatory. Such as, for instance, tail call optimization in functional languages (tail call is a special case of recursion, which can be represented as a loop).

However, one should understand that multiple complex optimizations at the level of machine code may cause a great slow-down of compilation. The benefit they allow you to gain may be much too insignificant, when compared to general system design optimizations (Wirth). One should also keep in mind that modern languages, with all their syntactic and semantic "frills", have many nuances and subtleties, so that a programmer who isn't familiar with them may be surprised by an outcome of optimization.

# Performance Tools

## WebLOAD

Enterprise-grade load and performance testing tool for web applications. WebLOAD is the tool of choice for enterprises with heavy user load and complex testing requirements. It allows you to perform load and stress testing on any internet application by generating load from the cloud and on-premises machines.

WebLOAD’s strengths are its flexibility and ease of use – enabling you to quickly define the tests you need with features like DOM-based recording/playback, automatic correlation, and JavaScript scripting language.

The tool provides a clear analysis of your web application performance, pinpointing issues and bottlenecks that may stand in the way of achieving your load and response requirements.

WebLOAD supports hundreds of technologies – from web protocols to enterprise applications and has built-in integration with Jenkins, Selenium and many other tools to enable continuous load testing for DevOps.

## SmartMeter.io

This load and performance testing tool provides advanced testing functions. With JMeter at its core, it will be instantly familiar to any of its users.

Creating a test in SmartMeter.io is very simple. You can make test scenarios without scripting just by clicking in an embedded browser. There’s also no proxy setup or browser plugin necessary.

It features automatically generated reports with all details about the test and its results. The results contain auto-evaluated acceptance criteria, statistics, graph comparison tool, and trend analysis of multiple test runs.

The tool is also strong in distributed testing, CI integration, and offers unparalleled performance testing support for Vaadin apps.

## Apache JMeter

**Open source load testing tool:** It is a Java platform application. It is mainly considered as a performance testing tool and it can also be integrated with the test plan. In addition to the load Test plan, you can also create a functional test plan. This tool has the capacity to be loaded into a server or network so as to check on its performance and analyse its working under different conditions.

Initially, it was introduced for testing the web applications, but later its scope had widened. It is of great use in testing the functional performance of the resources such as Servlets, Perl Scripts and JAVA objects. Need JVM 1.4 or higher to run.