

# 1.

## Optimizing software in C++

### An optimization guide for Windows, Linux and Mac platforms

By Agner Fog. Technical University of Denmark.  
Copyright © 2004 - 2015. Last updated 2015-12-23.

#### Contents

1	Introduction .....	3
1.1	The costs of optimizing .....	4
2	Choosing the optimal platform .....	5
2.1	Choice of hardware platform .....	5
2.2	Choice of microprocessor .....	6
2.3	Choice of operating system.....	6
2.4	Choice of programming language .....	8
2.5	Choice of compiler .....	10
2.6	Choice of function libraries.....	12
2.7	Choice of user interface framework.....	14
2.8	Overcoming the drawbacks of the C++ language.....	14
3	Finding the biggest time consumers .....	16
3.1	How much is a clock cycle? .....	16
3.2	Use a profiler to find hot spots .....	16
3.3	Program installation .....	18
3.4	Automatic updates .....	19
3.5	Program loading .....	19
3.6	Dynamic linking and position-independent code .....	20
3.7	File access.....	20
3.8	System database .....	20
3.9	Other databases .....	21
3.10	Graphics .....	21
3.11	Other system resources .....	21
3.12	Network access .....	21
3.13	Memory access.....	22
3.14	Context switches.....	22
3.15	Dependency chains .....	22
3.16	Execution unit throughput .....	22
4	Performance and usability .....	23
5	Choosing the optimal algorithm .....	24
6	Development process.....	25
7	The efficiency of different C++ constructs.....	26
7.1	Different kinds of variable storage.....	26
7.2	Integers variables and operators.....	29
7.3	Floating point variables and operators .....	32
7.4	Enums .....	33
7.5	Booleans.....	34
7.6	Pointers and references .....	36
7.7	Function pointers .....	37
7.8	Member pointers.....	38
7.9	Smart pointers .....	38
7.10	Arrays .....	39
7.11	Type conversions.....	40
7.12	Branches and switch statements.....	44
7.13	Loops.....	45

7.14	Functions	48
7.15	Function parameters	50
7.16	Function return types	50
7.17	Structures and classes	51
7.18	Class data members (properties)	52
7.19	Class member functions (methods)	53
7.20	Virtual member functions	54
7.21	Runtime type identification (RTTI)	54
7.22	Inheritance	54
7.23	Constructors and destructors	55
7.24	Unions	55
7.25	Bitfields	56
7.26	Overloaded functions	56
7.27	Overloaded operators	56
7.28	Templates	57
7.29	Threads	60
7.30	Exceptions and error handling	61
7.31	Other cases of stack unwinding	65
7.32	Preprocessing directives	65
7.33	Namespaces	65
8	Optimizations in the compiler	66
8.1	How compilers optimize	66
8.2	Comparison of different compilers	74
8.3	Obstacles to optimization by compiler	77
8.4	Obstacles to optimization by CPU	81
8.5	Compiler optimization options	81
8.6	Optimization directives	82
8.7	Checking what the compiler does	84
9	Optimizing memory access	87
9.1	Caching of code and data	87
9.2	Cache organization	87
9.3	Functions that are used together should be stored together	88
9.4	Variables that are used together should be stored together	88
9.5	Alignment of data	90
9.6	Dynamic memory allocation	90
9.7	Container classes	93
9.8	Strings	96
9.9	Access data sequentially	96
9.10	Cache contentions in large data structures	96
9.11	Explicit cache control	99
10	Multithreading	101
10.1	Hyperthreading	103
11	Out of order execution	103
12	Using vector operations	105
12.1	AVX instruction set and YMM registers	107
12.2	AVX-512 instruction set and ZMM registers	107
12.3	Automatic vectorization	107
12.4	Using intrinsic functions	109
12.5	Using vector classes	113
12.6	Transforming serial code for vectorization	117
12.7	Mathematical functions for vectors	119
12.8	Aligning dynamically allocated memory	120
12.9	Aligning RGB video or 3-dimensional vectors	120
12.10	Conclusion	120
13	Making critical code in multiple versions for different instruction sets	122
13.1	CPU dispatch strategies	122
13.2	Model-specific dispatching	124
13.3	Difficult cases	125

13.4 Test and maintenance .....	126
13.5 Implementation .....	127
13.6 CPU dispatching in Gnu compiler .....	129
13.7 CPU dispatching in Intel compiler .....	130
14 Specific optimization topics .....	132
14.1 Use lookup tables .....	132
14.2 Bounds checking .....	134
14.3 Use bitwise operators for checking multiple values at once.....	135
14.4 Integer multiplication .....	136
14.5 Integer division.....	138
14.6 Floating point division .....	139
14.7 Don't mix float and double.....	140
14.8 Conversions between floating point numbers and integers .....	141
14.9 Using integer operations for manipulating floating point variables .....	142
14.10 Mathematical functions .....	146
14.11 Static versus dynamic libraries.....	146
14.12 Position-independent code.....	148
14.13 System programming .....	150
15 Metaprogramming .....	151
16 Testing speed.....	154
16.1 Using performance monitor counters .....	156
16.2 The pitfalls of unit-testing .....	156
16.3 Worst-case testing .....	157
17 Optimization in embedded systems.....	159
18 Overview of compiler options.....	161
19 Literature .....	164
20 Copyright notice .....	165

## 1 Introduction

This manual is for advanced programmers and software developers who want to make their software faster. It is assumed that the reader has a good knowledge of the C++ programming language and a basic understanding of how compilers work. The C++ language is chosen as the basis for this manual for reasons explained on page 8 below.

This manual is based mainly on my study of how compilers and microprocessors work. The recommendations are based on the x86 family of microprocessors from Intel, AMD and VIA including the 64-bit versions. The x86 processors are used in the most common platforms with Windows, Linux, BSD and Mac OS X operating systems, though these operating systems can also be used with other microprocessors. Many of the advices may apply to other platforms and other compiled programming languages as well.

This is the first in a series of five manuals:

1. Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms.
2. Optimizing subroutines in assembly language: An optimization guide for x86 platforms.
3. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers.
4. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs.
5. Calling conventions for different C++ compilers and operating systems.

The latest versions of these manuals are always available from [www.agner.org/optimize](http://www.agner.org/optimize). Copyright conditions are listed on page 165 below.

Those who are satisfied with making software in a high-level language need only read this first manual. The subsequent manuals are for those who want to go deeper into the technical details of instruction timing, assembly language programming, compiler technology, and microprocessor microarchitecture. A higher level of optimization can sometimes be obtained by the use of assembly language for CPU-intensive code, as described in the subsequent manuals.

Please note that my optimization manuals are used by thousands of people. I simply don't have the time to answer questions from everybody. So please don't send your programming questions to me. You will not get any answer. Beginners are advised to seek information elsewhere and get a good deal of programming experience before trying the techniques in the present manual. There are various discussion forums on the Internet where you can get answers to your programming questions if you cannot find the answers in the relevant books and manuals.

I want to thank the many people who have sent me corrections and suggestions for my optimization manuals. I am always happy to receive new relevant information.

## **1.1 The costs of optimizing**

University courses in programming nowadays stress the importance of structured and object-oriented programming, modularity, reusability and systematization of the software development process. These requirements are often conflicting with the requirements of optimizing the software for speed or size.

Today, it is not uncommon for software teachers to recommend that no function or method should be longer than a few lines. A few decades ago, the recommendation was the opposite: Don't put something in a separate subroutine if it is only called once. The reasons for this shift in software writing style are that software projects have become bigger and more complex, that there is more focus on the costs of software development, and that computers have become more powerful.

The high priority of structured software development and the low priority of program efficiency is reflected, first and foremost, in the choice of programming language and interface frameworks. This is often a disadvantage for the end user who has to invest in ever more powerful computers to keep up with the ever bigger software packages and who is still frustrated by unacceptably long response times, even for simple tasks.

Sometimes it is necessary to compromise on the advanced principles of software development in order to make software packages faster and smaller. This manual discusses how to make a sensible balance between these considerations. It is discussed how to identify and isolate the most critical part of a program and concentrate the optimization effort on that particular part. It is discussed how to overcome the dangers of a relatively primitive programming style that doesn't automatically check for array bounds violations, invalid pointers, etc. And it is discussed which of the advanced programming constructs are costly and which are cheap, in relation to execution time.

## 2 Choosing the optimal platform

### 2.1 Choice of hardware platform

The choice of hardware platform has become less important than it used to be. The distinctions between RISC and CISC processors, between PC's and mainframes, and between simple processors and vector processors are becoming increasingly blurred as the standard PC processors with CISC instruction sets have got RISC cores, vector processing instructions, multiple cores, and a processing speed exceeding that of yesterday's big mainframe computers.

Today, the choice of hardware platform for a given task is often determined by considerations such as price, compatibility, second source, and the availability of good development tools, rather than by the processing power. Connecting several standard PC's in a network may be both cheaper and more efficient than investing in a big mainframe computer. Big supercomputers with massively parallel vector processing capabilities still have a niche in scientific computing, but for most purposes the standard PC processors are preferred because of their superior performance/price ratio.

The CISC instruction set (called x86) of the standard PC processors is not optimal from a technological point of view. This instruction set is maintained for the sake of backwards compatibility with a lineage of software that dates back to around 1980 where RAM memory and disk space were scarce resources. However, the CISC instruction set is better than its reputation. The compactness of the code makes caching more efficient today where cache size is a limited resource. The CISC instruction set may actually be better than RISC in situations where code caching is critical. The worst problem of the x86 instruction set is the scarcity of registers. This problem has been alleviated in the 64-bit extension to the x86 instruction set where the number of registers has been doubled.

Thin clients that depend on network resources are not recommended for critical applications because the response times for network resources cannot be controlled.

Small hand-held devices are becoming more popular and used for an increasing number of purposes such as email and web browsing that previously required a PC. Similarly, we are seeing an increasing number of devices and machines with embedded microcontrollers. I am not making any specific recommendation about which platforms and operating systems are most efficient for such applications, but it is important to realize that such devices typically have much less memory and computing power than PCs. Therefore, it is even more important to economize the resource use on such systems than it is on a PC platform. However, with a well optimized software design, it is possible to get a good performance for many applications even on such small devices, as discussed on page 159.

This manual is based on the standard PC platform with an Intel, AMD or VIA processor and a Windows, Linux, BSD or Mac operating system running in 32-bit or 64-bit mode. Much of the advice given here may apply to other platforms as well, but the examples have been tested only on PC platforms.

#### Graphics accelerators

The choice of platform is obviously influenced by the requirements of the task in question. For example, a heavy graphics application is preferably implemented on a platform with a graphics coprocessor or graphics accelerator card. Some systems also have a dedicated physics processor for calculating the physical movements of objects in a computer game or animation.

It is possible in some cases to use the high processing power of the processors on a graphics accelerator card for other purposes than rendering graphics on the screen. However, such applications are highly system dependent and therefore not recommended if portability is important. This manual does not cover graphics processors.

[Click here to download full PDF material](#)