1.
Optimizing software in C++
An optimization guide for Windows, Linux and Mac platforms

By Agner Fog. Technical University of Denmark.

Contents

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. 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.

Programmable logic devices

A programmable logic device is a chip that can be programmed in a hardware definition language, such as VHDL or Verilog. Common devices are CPLDs and FPGAs. The difference between a software programming language, e.g. C++, and a hardware definition language is that the software programming language defines an algorithm of sequential instructions, where a hardware definition language defines hardware circuits consisting of digital building blocks such as gates, flip-flops, multiplexers, arithmetic units, etc. and the wires that connect them. The hardware definition language is inherently parallel because it defines electrical connections rather than sequences of operations.

A complex digital operation can often be executed faster in a programmable logic device than in a microprocessor because the hardware can be wired for a specific purpose.

It is possible to implement a microprocessor in an FPGA as a so-called soft processor. Such a soft processor is much slower than a dedicated microprocessor and therefore not advantageous by itself. But a solution where a soft processor activates critical application-specific instructions that are coded in a hardware definition language in the same chip can be a very efficient solution in some cases. An even more powerful solution is the combination of a dedicated microprocessor core and an FPGA in the same chip. Such hybrid solutions are now used in some embedded systems.

A look in my crystal ball reveals that similar solutions may some day be implemented in PC processors. The application program will be able to define application-specific instructions that can be coded in a hardware definition language. Such a processor will have an extra cache for the hardware definition code in addition to the code cache and the data cache.

2.2 Choice of microprocessor

The benchmark performance of competing brands of microprocessors are very similar thanks to heavy competition. Processors with multiple cores are advantageous for applications that can be divided into multiple threads that run in parallel. Small lightweight processors with low power consumption are actually quite powerful and may be sufficient for less intensive applications.

Some systems have a graphics processing unit, either on a graphics card or integrated in the CPU chip. Such units can be used as coprocessors to take care of some of the heavy graphics calculations. In some cases it is possible to utilize the computational power of the graphics processing unit for other purposes than it is intended for. Some systems also have a physics processing unit intended for calculating the movements of objects in computer games. Such a coprocessor might also be used for other purposes. The use of coprocessors is beyond the scope of this manual.

2.3 Choice of operating system

All newer microprocessors in the x86 family can run in both 16-bit, 32-bit and 64-bit mode.

16-bit mode is used in the old operating systems DOS and Windows 3.x. These systems use segmentation of the memory if the size of program or data exceeds 64 kbytes. This is quite inefficient. The modern microprocessors are not optimized for 16-bit mode and some operating systems are not backwards compatible with 16-bit programs. It is not recommended to make 16-bit programs, except for small embedded systems.

Today (2013) both 32-bit and 64-bit operating systems are common, and there is no big difference in performance between the systems. There is no heavy marketing of 64-bit software, but it is quite certain that the 64-bit systems will dominate in the future.

The 64-bit systems can improve the performance by 5-10% for some CPU-intensive applications with many function calls. If the bottleneck is elsewhere then there is no difference in performance between 32-bit and 64-bit systems. Applications that use large amounts of memory will benefit from the larger address space of the 64-bit systems.

A software developer may choose to make memory-hungry software in two versions. A 32-bit version for the sake of compatibility with existing systems and a 64-bit version for best performance.

The Windows and Linux operating systems give almost identical performance for 32-bit software because the two operating systems are using the same function calling conventions. FreeBSD and Open BSD are identical to Linux in almost all respects relevant to software optimization. Everything that is said here about Linux also applies to BSD systems.

The Intel-based Mac OS X operating system is based on BSD, but the compiler uses position-independent code and lazy binding by default, which makes it less efficient. The performance can be improved by using static linking and by not using position-independent code (option `-fno-pic`).

64 bit systems have several advantages over 32 bit systems:

- The number of registers is doubled. This makes it possible to store intermediate data and local variables in registers rather than in memory.
- Function parameters are transferred in registers rather than on the stack. This makes function calls more efficient.
- The size of the integer registers is extended to 64 bits. This is only an advantage in applications that can take advantage of 64-bit integers.
- The allocation and deallocation of big memory blocks is more efficient.
- The SSE2 instruction set is supported on all 64-bit CPUs and operating systems.
- The 64 bit instruction set supports self-relative addressing of data. This makes position-independent code more efficient.

64 bit systems have the following disadvantages compared to 32 bit systems:

- Pointers, references, and stack entries use 64 bits rather than 32 bits. This makes data caching less efficient.
- Access to static or global arrays require a few extra instructions for address calculation in 64 bit mode if the image base is not guaranteed to be less than 2^{31}. This extra cost is seen in 64 bit Windows and Mac programs but rarely in Linux.
- Address calculation is more complicated in a large memory model where the combined size of code and data can exceed 2 Gbytes. This large memory model is hardly ever used, though.
- Some instructions are one byte longer in 64 bit mode than in 32 bit mode.
- Some 64-bit compilers are inferior to their 32-bit counterparts.

In general, you can expect 64-bit programs to run a little faster than 32-bit programs if there are many function calls, if there are many allocations of large memory blocks, or if the

program can take advantage of 64-bit integer calculations. It is necessary to use 64-bit systems if the program uses more than 2 gigabytes of data.

The similarity between the operating systems disappears when running in 64-bit mode because the function calling conventions are different. 64-bit Windows allows only four function parameters to be transferred in registers, whereas 64-bit Linux, BSD and Mac allow up to fourteen parameters to be transferred in registers (6 integer and 8 floating point). There are also other details that make function calling more efficient in 64-bit Linux than in 64-bit Windows (See page 50 and manual 5: "Calling conventions for different C++ compilers and operating systems"). An application with many function calls may run slightly faster in 64-bit Linux than in 64-bit Windows. The disadvantage of 64-bit Windows may be mitigated by making critical functions inline or static or by using a compiler that can do whole program optimization.

2.4 Choice of programming language

Before starting a new software project, it is important to decide which programming language is best suited for the project at hand. Low-level languages are good for optimizing execution speed or program size, while high-level languages are good for making clear and well-structured code and for fast and easy development of user interfaces and interfaces to network resources, databases, etc.

The efficiency of the final application depends on the way the programming language is implemented. The highest efficiency is obtained when the code is compiled and distributed as binary executable code. Most implementations of C++, Pascal and Fortran are based on compilers.

Several other programming languages are implemented with interpretation. The program code is distributed as it is and interpreted line by line when it is run. Examples include JavaScript, PHP, ASP and UNIX shell script. Interpreted code is very inefficient because the body of a loop is interpreted again and again for every iteration of the loop.

Some implementations use just-in-time compilation. The program code is distributed and stored as it is, and is compiled when it is executed. An example is Perl.

Several modern programming languages use an intermediate code (byte code). The source code is compiled into an intermediate code, which is the code that is distributed. The intermediate code cannot be executed as it is, but must go through a second step of interpretation or compilation before it can run. Some implementations of Java are based on an interpreter which interprets the intermediate code by emulating the so-called Java virtual machine. The best Java machines use just-in-time compilation of the most used parts of the code. C#, managed C++, and other languages in Microsoft's .NET framework are based on just-in-time compilation of an intermediate code.

The reason for using an intermediate code is that it is intended to be platform-independent and compact. The biggest disadvantage of using an intermediate code is that the user must install a large runtime framework for interpreting or compiling the intermediate code. This framework typically uses much more resources than the code itself.

Another disadvantage of intermediate code is that it adds an extra level of abstraction which makes detailed optimization more difficult. On the other hand, a just-in-time compiler can optimize specifically for the CPU it is running on, while it is more complicated to make CPU-specific optimizations in precompiled code.

The history of programming languages and their implementations reveal a zigzag course that reflects the conflicting considerations of efficiency, platform independence, and easy development. For example, the first PC's had an interpreter for Basic. A compiler for Basic soon became available because the interpreted version of Basic was too slow. Today, the

most popular version of Basic is Visual Basic .NET, which is implemented with an intermediate code and just-in-time compilation. Some early implementations of Pascal used an intermediate code like the one that is used for Java today. But this language gained remarkably in popularity when a genuine compiler became available.

It should be clear from this discussion that the choice of programming language is a compromise between efficiency, portability and development time. Interpreted languages are out of the question when efficiency is important. A language based on intermediate code and just-in-time compilation may be a viable compromise when portability and ease of development are more important than speed. This includes languages such as C#, Visual Basic .NET and the best Java implementations. However, these languages have the disadvantage of a very large runtime framework that must be loaded every time the program is run. The time it takes to load the framework and compile the program are often much more than the time it takes to execute the program, and the runtime framework may use more resources than the program itself when running. Programs using such a framework sometimes have unacceptably long response times for simple tasks like pressing a button or moving the mouse. The .NET framework should definitely be avoided when speed is critical.

The fastest execution is no doubt obtained with a fully compiled code. Compiled languages include C, C++, D, Pascal, Fortran and several other less well-known languages. My preference is for C++ for several reasons. C++ is supported by some very good compilers and optimized function libraries. C++ is an advanced high-level language with a wealth of advanced features rarely found in other languages. But the C++ language also includes the low-level C language as a subset, giving access to low-level optimizations. Most C++ compilers are able to generate an assembly language output, which is useful for checking how well the compiler optimizes a piece of code. Furthermore, most C++ compilers allow assembly-like intrinsic functions, inline assembly or easy linking to assembly language modules when the highest level of optimization is needed. The C++ language is portable in the sense that C++ compilers exist for all major platforms. Pascal has many of the advantages of C++ but is not quite as versatile. Fortran is also quite efficient, but the syntax is very old-fashioned.

Development in C++ is quite efficient thanks to the availability of powerful development tools. One popular development tool is Microsoft Visual Studio. This tool can make two different implementations of C++, directly compiled code and intermediate code for the common language runtime of the .NET framework. Obviously, the directly compiled version is preferred when speed is important.

An important disadvantage of C++ relates to security. There are no checks for array bounds violation, integer overflow, and invalid pointers. The absence of such checks makes the code execute faster than other languages that do have such checks. But it is the responsibility of the programmer to make explicit checks for such errors in cases where they cannot be ruled out by the program logic. Some guidelines are provided below, on page 15.

C++ is definitely the preferred programming language when the optimization of performance has high priority. The gain in performance over other programming languages can be quite substantial. This gain in performance can easily justify a possible minor increase in development time when performance is important to the end user.

There may be situations where a high level framework based on intermediate code is needed for other reasons, but part of the code still needs careful optimization. A mixed implementation can be a viable solution in such cases. The most critical part of the code can be implemented in compiled C++ or assembly language and the rest of the code, including user interface etc., can be implemented in the high level framework. The optimized part of the code can possibly be compiled as a dynamic link library (DLL) which is called by the rest of the code. This is not an optimal solution because the high level framework still consumes a lot of resources, and the transitions between the two kinds of code gives an

extra overhead which consumes CPU time. But this solution can still give a considerable improvement in performance if the time-critical part of the code can be completely contained in a DLL.

Another alternative worth considering is the D language. D has many of the features of Java and C# and avoids many of the drawbacks of C++. Yet, D is compiled to binary code and can be linked together with C or C++ code. Compilers and IDE's for D are not yet as well developed as C++ compilers.

2.5 Choice of compiler

There are several different C++ compilers to choose between. It is difficult to predict which compiler will do the best job optimizing a particular piece of code. Each compiler does some things very smart and other things very stupid. Some common compilers are mentioned below.

Microsoft Visual Studio

This is a very user friendly compiler with many features, but also very expensive. A limited "express" edition is available for free. Visual Studio can build code for the .NET framework as well as directly compiled code. (Compile *without* the Common Language Runtime, CLR, to produce binary code). Supports 32-bit and 64-bit Windows. The integrated development environment (IDE) supports multiple programming languages, profiling and debugging. A command-line version of the C++ compiler is available for free in the Microsoft platform software development kit (SDK or PSDK). Supports the OpenMP directives for multi-core processing. Visual Studio optimizes reasonably well, but it is not the best optimizer.

Borland/CodeGear/Embarcadero C++ builder

Has an IDE with many of the same features as the Microsoft compiler. Supports only 32-bit Windows. Does not support the SSE and later instruction sets. Does not optimize as good as the Microsoft, Intel, Gnu and PathScale compilers.

Intel C++ compiler (parallel composer)

This compiler does not have its own IDE. It is intended as a plug-in to Microsoft Visual Studio when compiling for Windows and to Eclipse when compiling for Linux. It can also be used as a stand alone compiler when called from a command line or a make utility. It supports 32-bit and 64-bit Windows and 32-bit and 64-bit Linux as well as Intel-based Mac OS and Itanium systems.

The Intel compiler supports vector intrinsics, automatic vectorization (see page 107), OpenMP and automatic parallelization of code into multiple threads. The compiler supports CPU dispatching to make multiple code versions for different CPUs. (See page 130 for how to make this work on non-Intel processors). It has excellent support for inline assembly on all platforms and the possibility of using the same inline assembly syntax in both Windows and Linux. The compiler comes with some of the best optimized math function libraries available.

The most important disadvantage of the Intel compiler is that the compiled code may run with reduced speed or not at all on AMD and VIA processors. It is possible to avoid this problem by bypassing the so-called CPU-dispatcher that checks whether the code is running on an Intel CPU. See page 130 for details).

The Intel compiler is a good choice for code that can benefit from its many optimization features and for code that is ported to multiple operating systems.

Gnu

This is one of the best optimizing compilers available, though less user friendly. It is free and open source. It comes with most distributions of Linux, BSD and Mac OS X, 32-bit and 64-bit. Supports OpenMP and automatic parallelization. Supports vector intrinsics and automatic vectorization (see page 107). The Gnu function libraries are not fully optimized yet. Supports both AMD and Intel vector math libraries. The Gnu C++ compiler is available for many platforms, including 32-bit and 64-bit Linux, BSD, Windows and Mac. The Gnu compiler is a very good choice for all Unix-like platforms.

Clang

The Clang compiler combined with the LLVM is a new compiler which is similar to the Gnu compiler in many respects and highly compatible with Gnu. It is expected to replace the Gnu compiler on the Mac platform, but also supports Linux and Windows platforms. The Clang compiler is a good choice for all platforms.

PathScale

C++ compiler for 32- and 64-bit Linux. Has many good optimization options. Supports parallel processing, OpenMP and automatic vectorization. It is possible to insert optimization hints as pragmas in the code to tell the compiler e.g. how often a part of the code is executed. Optimizes very well. This compiler is a good choice for Linux platforms if the bias of the Intel compiler in favor of Intel CPUs cannot be tolerated.

PGI

C++ compiler for 32- and 64-bit Windows, Linux and Mac. Supports parallel processing, OpenMP and automatic vectorization. Optimizes reasonably well. Very poor performance for vector intrinsics.

Digital Mars

This is a cheap compiler for 32-bit Windows, including an IDE. Does not optimize well.

Open Watcom

Another open source compiler for 32-bit Windows. Does not, by default, conform to the standard calling conventions. Optimizes reasonably well.

Codeplay VectorC

A commercial compiler for 32-bit Windows. Integrates into the Microsoft Visual Studio IDE. Has not been updated since 2004. Can do automatic vectorization. Optimizes moderately well. Supports three different object file formats.

Comments

All of these compilers can be used as command-line versions without an IDE. Free trial versions are available for the commercial compilers.

Mixing object files from different compilers is generally possible on Linux platforms, and in some cases on Windows platforms. The Microsoft and Intel compilers for Windows are fully compatible on the object file level, and the Digital Mars compiler is mostly compatible with these. The CodeGear, Codeplay and Watcom compilers are not compatible with other compilers at the object file level.

My recommendation for good code performance is to use the Gnu, Clang, Intel or PathScale compiler for Unix applications and the Gnu, Clang, Intel or Microsoft compiler for Windows applications.

The choice of compiler may in some cases be determined by the requirements of compatibility with legacy code, specific preferences for the IDE, for debugging facilities, easy GUI development, database integration, web application integration, mixed language

programming, etc. In cases where the chosen compiler doesn't provide the best optimization it may be useful to make the most critical modules with a different compiler. Object files generated by the Intel and PathScale compilers can in most cases be linked into projects made with Microsoft or Gnu compilers without problems if the necessary library files are also included. Combining the Borland compiler with other compilers or function libraries is more difficult. The functions must have `extern "C"` declaration and the object files need to be converted to OMF format. Alternatively, make a DLL with the best compiler and call it from a project built with another compiler.

2.6 Choice of function libraries

Some applications spend most of their execution time on executing library functions. Time-consuming library functions often belong to one of these categories:

- File input/output
- Graphics and sound processing
- Memory and string manipulation
- Mathematical functions
- Encryption, decryption, data compression

Most compilers include standard libraries for many of these purposes. Unfortunately, the standard libraries are not always fully optimized.

Library functions are typically small pieces of code that are used by many users in many different applications. Therefore, it is worthwhile to invest more efforts in optimizing library functions than in optimizing application-specific code. The best function libraries are highly optimized, using assembly language and automatic CPU-dispatching (see page 122) for the latest instruction set extensions.

If a profiling (see page 16) shows that a particular application uses a lot of CPU-time in library functions, or if this is obvious, then it may be possible to improve the performance significantly simply by using a different function library. If the application uses most of its time in library functions then it may not be necessary to optimize anything else than finding the most efficient library and economize the library function calls. It is recommended to try different libraries and see which one works best.

Some common function libraries are discussed below. Many libraries for special purposes are also available.

Microsoft

Comes with Microsoft compiler. Some functions are optimized well, others are not. Supports 32-bit and 64-bit Windows.

Borland / CodeGear / Embarcadero

Comes with the Borland C++ builder. Not optimized for SSE2 and later instruction sets. Supports only 32-bit Windows.

Gnu

Comes with the Gnu compiler. Not optimized as good as the compiler itself is. The 64-bit version is better than the 32-bit version. The Gnu compiler often inserts built-in code instead of the most common memory and string instructions. The built-in code is not optimal. Use

option `-fno-builtin` to get library versions instead. The Gnu libraries support 32-bit and 64-bit Linux and BSD. The Windows version is currently not up to date.

Mac

The libraries included with the Gnu compiler for Mac OS X (Darwin) are part of the Xnu project. Some of the most important functions are included in the operating system kernel in the so-called commpage. These functions are highly optimized for the Intel Core and later Intel processors. AMD processors and earlier Intel processors are not supported at all. Can only run on Mac platform.

Intel

The Intel compiler includes standard function libraries. Several special purpose libraries are also available, such as the "Intel Math Kernel Library" and "Integrated Performance Primitives". These function libraries are highly optimized for large data sets. However, the Intel libraries do not always work well on AMD and VIA processors. See page 130 for an explanation and possible workaround. Supports all x86 and x86-64 platforms.

AMD

AMD Math core library contains optimized mathematical functions. It also works on Intel processors. The performance is inferior to the Intel libraries. Supports 32- and 64-bit Windows and Linux.

Asmlib

My own function library made for demonstration purposes. Available from www.agner.org/optimize/asmlib.zip. Currently includes optimized versions of memory and string functions and some other functions that are difficult to find elsewhere. Faster than most other libraries when running on the newest processors. Supports all x86 and x86-64 platforms.

Comparison of function libraries

Test	Processor	Microsoft	CodeGear	Intel	Mac	Gnu 32-bit	Gnu 32-bit -fno-builtin	Gnu 64 bit -fno-builtin	Asmlib
`memcpy` 16kB aligned operands	Intel Core 2	0.12	0.18	0.12	0.11	0.18	0.18	0.18	0.11
`memcpy` 16kB unaligned op.	Intel Core 2	0.63	0.75	0.18	0.11	1.21	0.57	0.44	0.12
`memcpy` 16kB aligned operands	AMD Opteron K8	0.24	0.25	0.24	n.a.	1.00	0.25	0.28	0.22
`memcpy` 16kB unaligned op.	AMD Opteron K8	0.38	0.44	0.40	n.a.	1.00	0.35	0.29	0.28
`strlen` 128 bytes	Intel Core 2	0.77	0.89	0.40	0.30	4.5	0.82	0.59	0.27
`strlen` 128 bytes	AMD Opteron K8	1.09	1.25	1.61	n.a.	2.23	0.95	0.6	1.19

Table 2.1. Comparing performance of different function libraries.
Numbers in the table are core clock cycles per byte of data (low numbers mean good performance). Aligned operands means that source and destination both have addresses divisible by 16.
Library versions tested (not up to date):
Microsoft Visual studio 2008, v. 9.0
CodeGear Borland bcc, v. 5.5
Mac: Darwin8 g++ v 4.0.1.
Gnu: Glibc v. 2.7, 2.8.
Asmlib: v. 2.00.
Intel C++ compiler, v. 10.1.020. Functions `_intel_fast_memcpy` and `_intel_new_strlen` in library `libircmt.lib`. Function names are undocumented.

2.7 Choice of user interface framework

Most of the code in a typical software project goes to the user interface. Applications that are not computationally intensive may very well spend more CPU time on the user interface than on the essential task of the program.

Application programmers rarely program their own graphical user interfaces from scratch. This would not only be a waste of the programmers' time, but also inconvenient to the end user. Menus, buttons, dialog boxes, etc. should be as standardized as possible for usability reasons. The programmer can use standard user interface elements that come with the operating system or libraries that come with compilers and development tools.

A popular user interface library for Windows and C++ is Microsoft Foundation Classes (MFC). A competing product is Borland's now discontinued Object Windows Library (OWL). Several graphical interface frameworks are available for Linux systems. The user interface library can be linked either as a runtime DLL or a static library. A runtime DLL takes more memory resources than a static library, except when several applications use the same DLL at the same time.

A user interface library may be bigger than the application itself and take more time to load. A light-weight alternative is the Windows Template Library (WTL). A WTL application is generally faster and more compact than an MFC application. The development time for WTL applications can be expected to be higher due to poor documentation and lack of advanced development tools.

The simplest possible user interface is obtained by dropping the graphical user interface and use a console mode program. The inputs for a console mode program are typically specified on a command line or an input file. The output goes to the console or to an output file. A console mode program is fast, compact, and simple to develop. It is easy to port to different platforms because it doesn't depend on system-specific graphical interface calls. The usability may be poor because it lacks the self-explaining menus of a graphical user interface. A console mode program is useful for calling from other applications such as a make utility.

The conclusion is that the choice of user interface framework must be a compromise between development time, usability, program compactness, and execution time. No universal solution is best for all applications.

2.8 Overcoming the drawbacks of the C++ language

While C++ has many advantages when it comes to optimization, it does have some disadvantages that make developers choose other programming languages. This section discusses how to overcome these disadvantages when C++ is chosen for the sake of optimization.

Portability

C++ is fully portable in the sense that the syntax is fully standardized and supported on all major platforms. However, C++ is also a language that allows direct access to hardware interfaces and system calls. These are of course system-specific. In order to facilitate porting between platforms, it is recommended to place the user interface and other system-specific parts of the code in a separate module, and to put the task-specific part of the code, which supposedly is system-independent, in another module.

The size of integers and other hardware-related details depend on the hardware platform and operating system. See page 29 for details.

Development time

Some developers feel that a particular programming language and development tool is faster to use than others. While some of the difference is simply a matter of habit, it is true that some development tools have powerful facilities that do much of the trivial programming work automatically. The development time and maintainability of C++ projects can be improved by consistent modularity and reusable classes.

Security

The most serious problem with the C++ language relates to security. Standard C++ implementations have no checking for array bounds violations and invalid pointers. This is a frequent source of errors in C++ programs and also a possible point of attack for hackers. It is necessary to adhere to certain programming principles in order to prevent such errors in programs where security matters.

Problems with invalid pointers can be avoided by using references instead of pointers, by initializing pointers to zero, by setting pointers to zero whenever the objects they point to become invalid, and by avoiding pointer arithmetics and pointer type casting. Linked lists and other data structures that typically use pointers may be replaced by more efficient container class templates, as explained on page 93. Avoid the function `scanf`.

Violation of array bounds is probably the most common cause of errors in C++ programs. Writing past the end of an array can cause other variables to be overwritten, and even worse, it can overwrite the return address of the function in which the array is defined. This can cause all kinds of strange and unexpected behaviors. Arrays are often used as buffers for storing text or input data. A missing check for buffer overflow on input data is a common error that hackers often have exploited.

A good way to prevent such errors is to replace arrays by well-tested container classes. The standard template library (STL) is a useful source of such container classes. Unfortunately, many standard container classes use dynamic memory allocation in an inefficient way. See page and 90 for examples of how to avoid dynamic memory allocation. See page 93 for discussion of efficient container classes. An appendix to this manual at www.agner.org/optimize/cppexamples.zip contains examples of arrays with bounds checking and various efficient container classes.

Text strings are particularly problematic because there may be no certain limit to the length of a string. The old C-style method of storing strings in character arrays is fast and efficient, but not safe unless the length of each string is checked before storing. The standard solution to this problem is to use string classes, such as `string` or `CString`. This is safe and flexible, but quite inefficient in large applications. The string classes allocate a new memory block every time a string is created or modified. This can cause the memory to be fragmented and involve a high overhead cost of heap management and garbage collection. A more efficient solution that doesn't compromise safety is to store all strings in one memory pool. See the examples in the appendix at www.agner.org/optimize/cppexamples.zip for how to store strings in a memory pool.

Integer overflow is another security problem. The official C standard says that the behavior of signed integers in case of overflow is "undefined". This allows the compiler to ignore overflow or assume that it doesn't occur. In the case of the Gnu compiler, the assumption that signed integer overflow doesn't occur has the unfortunate consequence that it allows the compiler to optimize away an overflow check. There are a number of possible remedies against this problem: (1) check for overflow before it occurs, (2) use unsigned integers - they are guaranteed to wrap around, (3) trap integer overflow with the option `-ftrapv`, but this is extremely inefficient, (4) get a compiler warning for such optimizations with option `-Wstrict-overflow=2`, or (5) make the overflow behavior well-defined with option `-fwrapv` or `-fno-strict-overflow`.

You may deviate from the above security advices in critical parts of the code where speed is important. This can be permissible if the unsafe code is limited to well-tested functions, classes, templates or modules with a well-defined interface to the rest of the program.

3 Finding the biggest time consumers

3.1 How much is a clock cycle?

In this manual, I am using CPU clock cycles rather than seconds or microseconds as a time measure. This is because computers have very different speeds. If I write that something takes 10 µs today, then it may take only 5 µs on the next generation of computers and my manual will soon be obsolete. But if I write that something takes 10 clock cycles then it will still take 10 clock cycles even if the CPU clock frequency is doubled.

The length of a clock cycle is the reciprocal of the clock frequency. For example, if the clock frequency is 2 GHz then the length of a clock cycle is

$$\frac{1}{2\mathrm{GHz}} = 0.5\mathrm{ns}.$$

A clock cycle on one computer is not always comparable to a clock cycle on another computer. The Pentium 4 (NetBurst) CPU is designed for a higher clock frequency than other CPUs, but it uses more clock cycles than other CPUs for executing the same piece of code in general.

Assume that a loop in a program repeats 1000 times and that there are 100 floating point operations (addition, multiplication, etc.) inside the loop. If each floating point operation takes 5 clock cycles, then we can roughly estimate that the loop will take 1000 * 100 * 5 * 0.5 ns = 250 µs on a 2 GHz CPU. Should we try to optimize this loop? Certainly not! 250 µs is less than 1/50 of the time it takes to refresh the screen. There is no way the user can see the delay. But if the loop is inside another loop that also repeats 1000 times then we have an estimated calculation time of 250 ms. This delay is just long enough to be noticeable but not long enough to be annoying. We may decide to do some measurements to see if our estimate is correct or if the calculation time is actually more than 250 ms. If the response time is so long that the user actually has to wait for a result then we will consider if there is something that can be improved.

3.2 Use a profiler to find hot spots

Before you start to optimize anything, you have to identify the critical parts of the program. In some programs, more than 99% of the time is spent in the innermost loop doing mathematical calculations. In other programs, 99% of the time is spent on reading and writing data files while less than 1% goes to actually doing something on these data. It is very important to optimize the parts of the code that matters rather than the parts of the code that use only a small fraction of the total time. Optimizing less critical parts of the code

will not only be a waste of time, it also makes the code less clear and more difficult to debug and maintain.

Most compiler packages include a profiler that can tell how many times each function is called and how much time it uses. There are also third-party profilers such as AQtime, Intel VTune and AMD CodeAnalyst.

There are several different profiling methods:

- Instrumentation: The compiler inserts extra code at each function call to count how many times the function is called and how much time it takes.

- Debugging. The profiler inserts temporary debug breakpoints at every function or every code line.

- Time-based sampling: The profiler tells the operating system to generate an interrupt, e.g. every millisecond. The profiler counts how many times an interrupt occurs in each part of the program. This requires no modification of the program under test, but is less reliable.

- Event-based sampling: The profiler tells the CPU to generate interrupts at certain events, for example every time a thousand cache misses have occurred. This makes it possible to see which part of the program has most cache misses, branch mispredictions, floating point exceptions, etc. Event-based sampling requires a CPU-specific profiler. For Intel CPUs use Intel VTune, for AMD CPUs use AMD CodeAnalyst.

Unfortunately, profilers are often unreliable. They sometimes give misleading results or fail completely because of technical problems.

Some common problems with profilers are:

- Coarse time measurement. If time is measured with millisecond resolution and the critical functions take microseconds to execute then measurements can become imprecise or simply zero.

- Execution time too small or too long. If the program under test finishes in a short time then the sampling generates too little data for analysis. If the program takes too long time to execute then the profiler may sample more data than it can handle.

- Waiting for user input. Many programs spend most of their time waiting for user input or network resources. This time is included in the profile. It may be necessary to modify the program to use a set of test data instead of user input in order to make profiling feasible.

- Interference from other processes. The profiler measures not only the time spent in the program under test but also the time used by all other processes running on the same computer, including the profiler itself.

- Function addresses are obscured in optimized programs. The profiler identifies any hot spots in the program by their address and attempts to translate these addresses to function names. But a highly optimized program is often reorganized in such a way that there is no clear correspondence between function names and code addresses. The names of inlined functions may not be visible at all to the profiler. The result will be misleading reports of which functions take most time.

- Uses debug version of the code. Some profilers require that the code you are testing contains debug information in order to identify individual functions or code lines. The