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 168 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 162.

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 49 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. The full version is very expensive, but a limited non-commercial version 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. 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 latest instruction sets. Does not optimize as good as the Microsoft, Intel, and Gnu 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 110), OpenMP and automatic parallelization of code into multiple threads. The compiler supports CPU dispatching to make multiple code versions for different CPUs. (See page 133 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 133 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 110). 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 platforms. The stand-alone compiler is run from the command line, but several IDE's (integrated development environments) are available, including Eclipse, NetBeans, CodeBlocks, and BloodShed.

Clang

The Clang compiler is based on the LLVM (Low Level Virtual Machine). It is similar to the Gnu compiler in many respects and highly compatible with Gnu. It is the most common compiler on the Mac platform, but also supports Linux and Windows platforms. The Clang compiler is a good choice for all platforms. It can be used with the Eclipse IDE.

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. Apparently no longer updated. 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 Embarcadero, 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, or Intel compiler for Unix applications and the Gnu, Clang, Intel, or Microsoft compiler for Windows applications. Do not use the Intel compiler if you want your code to run efficiently on AMD microprocessors.

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 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. 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 125) 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 133 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 95. 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 92 for examples of how to avoid dynamic memory allocation. See page 95 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 debug version of the code is not optimized.

- Jumps between CPU cores. A process or thread does not necessarily stay in the same processor core on multi-core CPUs, but event-counters do. This results in meaningless event counts for threads that jump between multiple CPU cores. You may need to lock a thread to a specific CPU core by setting a thread affinity mask.

- Poor reproducibility. Delays in program execution may be caused by random events that are not reproducible. Such events as task switches and garbage collection can occur at random times and make parts of the program appear to take longer time than normally.

There are various alternatives to using a profiler. A simple alternative is to run the program in a debugger and press break while the program is running. If there is a hot spot that uses 90% of the CPU time then there is a 90% chance that the break will occur in this hot spot. Repeating the break a few times may be enough to identify a hot spot. Use the call stack in the debugger to identify the circumstances around the hot spot.

Sometimes, the best way to identify performance bottlenecks is to put measurement instruments directly into the code rather than using a ready-made profiler. This does not solve all the problems associated with profiling, but it often gives more reliable results. If you are not satisfied with the way a profiler works then you may put the desired measurement instruments into the program itself. You may add counter variables that count how many times each part of the program is executed. Furthermore, you may read the time before and after each of the most important or critical parts of the program to measure how much time each part takes. See page 157 for further discussion of this method.

Your measurement code should have `#if` directives around it so that it can be disabled in the final version of the code. Inserting your own profiling instruments in the code itself is a very useful way to keep track of the performance during the development of a program.

The time measurements may require a very high resolution if time intervals are short. In Windows, you can use the `GetTickCount` or `QueryPerformanceCounter` functions for millisecond resolution. A much higher resolution can be obtained with the time stamp counter in the CPU, which counts at the CPU clock frequency (in Windows: `__rdtsc()`).

The time stamp counter becomes invalid if a thread jumps between different CPU cores. You may have to fix the thread to a specific CPU core during time measurements to avoid this. (In Windows, `SetThreadAffinityMask`, in Linux, `sched_setaffinity`).

The program should be tested with a realistic set of test data. The test data should contain a typical degree of randomness in order to get a realistic number of cache misses and branch mispredictions.

When the most time-consuming parts of the program have been found, then it is important to focus the optimization efforts on the time consuming parts only. Critical pieces of code can be further tested and investigated by the methods described on page 157.

A profiler is most useful for finding problems that relate to CPU-intensive code. But many programs use more time loading files or accessing databases, network and other resources than doing arithmetic operations. The most common time-consumers are discussed in the following sections.

3.3 Program installation

The time it takes to install a program package is not traditionally considered a software optimization issue. But it is certainly something that can steal the user's time. The time it takes to install a software package and make it work cannot be ignored if the goal of software optimization is to save time for the user. With the high complexity of modern software, it is not unusual for the installation process to take more than an hour. Neither is it unusual that a user has to reinstall a software package several times in order to find and resolve compatibility problems.

Software developers should take installation time and compatibility problems into account when deciding whether to base a software package on a complex framework requiring many files to be installed.

The installation process should always use standardized installation tools. It should be possible to select all installation options at the start so that the rest of the installation process can proceed unattended. Uninstallation should also proceed in a standardized manner.

3.4 Automatic updates

Many software programs automatically download updates through the Internet at regular time intervals. Some programs search for updates every time the computer starts up, even if the program is never used. A computer with many such programs installed can take several minutes to start up, which is a total waste of the user's time. Other programs use time searching for updates each time the program starts. The user may not need the updates if the current version satisfies the user's needs. The search for updates should be optional and off by default unless there is a compelling security reason for updating. The update process should run in a low priority thread, and only if the program is actually used. No program should leave a background process running when it is not in use. The installation of downloaded program updates should be postponed until the program is shut down and restarted anyway.

Updates to the operating system can be particularly time consuming. Sometimes it takes hours to install automatic updates to the operating system. This is very problematic because these time consuming updates may come unpredictably at inconvenient times. This can be a very big problem if the user has to turn off or log off the computer for security reasons before leaving their workplace and the system forbids the user to turn off the computer during the update process.

3.5 Program loading

Often, it takes more time to load a program than to execute it. The load time can be annoyingly high for programs that are based on big runtime frameworks, intermediate code, interpreters, just-in-time compilers, etc., as is commonly the case with programs written in Java, C#, Visual Basic, etc.

But program loading can be a time-consumer even for programs implemented in compiled C++. This typically happens if the program uses a lot of runtime DLL's (dynamically linked libraries or shared objects), resource files, configuration files, help files and databases. The operating system may not load all the modules of a big program when the program starts up. Some modules may be loaded only when they are needed, or they may be swapped to the hard disk if the RAM size is insufficient.

The user expects immediate responses to simple actions like a key press or mouse move. It is unacceptable to the user if such a response is delayed for several seconds because it requires the loading of modules or resource files from disk. Memory-hungry applications force the operating system to swap memory to disk. Memory swapping is a frequent cause of unacceptably long response times to simple things like a mouse move or key press.

Avoid an excessive number of DLLs, configuration files, resource files, help files etc. scattered around on the hard disk. A few files, preferably in the same directory as the `.exe` file, is acceptable.

3.6 Dynamic linking and position-independent code

Function libraries can be implemented either as static link libraries (`*.lib`, `*.a`) or dynamic link libraries, also called shared objects (`*.dll`, `*.so`). There are several factors that can make dynamic link libraries slower than static link libraries. These factors are explained in detail on page 149 below.

Position-independent code is used in shared objects in Unix-like systems. Mac systems often use position-independent code everywhere by default. Position-independent code is inefficient, especially in 32-bit mode, for reasons explained on page 149 below.

3.7 File access

Reading or writing a file on a hard disk often takes much more time than processing the data in the file, especially if the user has a virus scanner that scans all files on access.

Sequential forward access to a file is faster than random access. Reading or writing big blocks is faster than reading or writing a small bit at a time. Do not read or write less than a few kilobytes at a time.

You may mirror the entire file in a memory buffer and read or write it in one operation rather than reading or writing small bits in a non-sequential manner.

It is usually much faster to access a file that has been accessed recently than to access it the first time. This is because the file has been copied to the disk cache.

Files on remote or removable media such as floppy disks and USB sticks may not be cached. This can have quite dramatic consequences. I once made a Windows program that created a file by calling `WritePrivateProfileString`, which opens and closes the file for each line written. This worked sufficiently fast on a hard disk because of disk caching, but it took several minutes to write the file to a floppy disk.

A big file containing numerical data is more compact and efficient if the data are stored in binary form than if the data are stored in ASCII form. A disadvantage of binary data storage is that it is not human readable and not easily ported to systems with big-endian storage.

Optimizing file access is more important than optimizing CPU use in programs that have many file input/output operations. It can be advantageous to put file access in a separate thread if there is other work that the processor can do while waiting for disk operations to finish.

3.8 System database

It can take several seconds to access the system database in Windows. It is more efficient to store application-specific information in a separate file than in the big registration database in the Windows system. Note that the system may store the information in the database anyway if you are using functions such as `GetPrivateProfileString` and `WritePrivateProfileString` to read and write configuration files (`*.ini` files).

3.9 Other databases

Many software applications use a database for storing user data. A database can consume a lot of CPU time, RAM and disk space. It may be possible to replace a database by a plain old data file in simple cases. Database queries can often be optimized by using indexes, working with sets rather than loops, etc. Optimizing database queries is beyond the scope of this manual, but you should be aware that there is often a lot to gain by optimizing database access.

3.10 Graphics

A graphical user interface can use a lot of computing resources. Typically, a specific graphics framework is used. The operating system may supply such a framework in its API. In some cases, there is an extra layer of a third-party graphics framework between the

operating system API and the application software. Such an extra framework can consume a lot of extra resources.

Each graphics operation in the application software is implemented as a function call to a graphics library or API function which then calls a device driver. A call to a graphics function is time consuming because it may go through multiple layers and it needs to switch to protected mode and back again. Obviously, it is more efficient to make a single call to a graphics function that draws a whole polygon or bitmap than to draw each pixel or line separately through multiple function calls.

The calculation of graphics objects in computer games and animations is of course also time consuming, especially if there is no graphics processing unit.

Various graphics function libraries and drivers differ a lot in performance. I have no specific recommendation of which one is best.

3.11 Other system resources

Writes to a printer or other device should preferably be done in big blocks rather than a small piece at a time because each call to a driver involves the overhead of switching to protected mode and back again.

Accessing system devices and using advanced facilities of the operating system can be time consuming because it may involve the loading of several drivers, configuration files and system modules.

3.12 Network access

Some application programs use internet or intranet for automatic updates, remote help files, data base access, etc. The problem here is that access times cannot be controlled. The network access may be fast in a simple test setup but slow or completely absent in a use situation where the network is overloaded or the user is far from the server.

These problems should be taken into account when deciding whether to store help files and other resources locally or remotely. If frequent updates are necessary then it may be optimal to mirror the remote data locally.

Access to remote databases usually requires log on with a password. The log on process is known to be an annoying time consumer to many hard working software users. In some cases, the log on process may take more than a minute if the network or database is heavily loaded.

3.13 Memory access

Accessing data from RAM memory can take quite a long time compared to the time it takes to do calculations on the data. This is the reason why all modern computers have memory caches. Typically, there is a level-1 data cache of 8 - 64 Kbytes and a level-2 cache of 256 Kbytes to 2 Mbytes. There may also be a level-3 cache.

If the combined size of all data in a program is bigger than the level-2 cache and the data are scattered around in memory or accessed in a non-sequential manner then it is likely that memory access is the biggest time-consumer in the program. Reading or writing to a variable in memory takes only 2-3 clock cycles if it is cached, but several hundred clock cycles if it is not cached. See page 26 about data storage and page 89 about memory caching.

3.14 Context switches

A context switch is a switch between different tasks in a multitasking environment, between different threads in a multithreaded program, or between different parts of a big program. Frequent context switches can reduce the performance because the contents of data cache, code cache, branch target buffer, branch pattern history, etc. may have to be renewed.

Context switches are more frequent if the time slices allocated to each task or thread are smaller. The lengths of the time slices is determined by the operating system, not by the application program.

The number of context switches is smaller in a computer with multiple CPUs or a CPU with multiple cores.

3.15 Dependency chains

Modern microprocessors can do out-of-order execution. This means that if a piece of software specifies the calculation of A and then B, and the calculation of A is slow, then the microprocessor can begin the calculation of B before the calculation of A is finished. Obviously, this is only possible if the value of A is not needed for the calculation of B.

In order to take advantage of out-of-order execution, you have to avoid long dependency chains. A dependency chain is a series of calculations, where each calculation depends on the result of the preceding one. This prevents the CPU from doing multiple calculations simultaneously or out of order. See page 105 for examples of how to break a dependency chain.

3.16 Execution unit throughput

There is an important distinction between the latency and the throughput of an execution unit. For example, it may take 3 - 5 clock cycles to do a floating point addition on a modern CPU. But it is possible to start a new floating point addition every clock cycle. This means that if each addition depends on the result of the preceding addition then you will have only one addition every three clock cycles. But if all the additions are independent then you can have one addition every clock cycle.

The highest performance that can possibly be obtained in a computationally intensive program is achieved when none of the time-consumers mentioned in the above sections are dominating and there are no long dependency chains. In this case, the performance is limited by the throughput of the execution units rather than by the latency or by memory access.

The execution core of modern microprocessors is split between several execution units. Typically, there are two or more integer units, one or two floating point addition units, and one or two floating point multiplication units. This means that it is possible to do an integer addition, a floating point addition, and a floating point multiplication at the same time.

A code that does floating point calculations should therefore preferably have a balanced mix of additions and multiplications. Subtractions use the same unit as additions. Divisions take longer time. It is possible to do integer operations in-between the floating point operations without reducing the performance because the integer operations use different execution units. For example, a loop that does floating point calculations will typically use integer operations for incrementing a loop counter, comparing the loop counter with its limit, etc. In most cases, you can assume that these integer operations do not add to the total computation time.

4 Performance and usability

A better performing software product is one that saves time for the user. Time is a precious resource for many computer users and much time is wasted on software that is slow, difficult to use, incompatible or error prone. All these problems are usability issues, and I believe that software performance should be seen in the broader perspective of usability.

This is not a manual on usability, but I think that it is necessary here to draw the attention of software programmers to some of the most common obstacles to efficient use of software. For more on this topic, see my free E-book Usability for Nerds at Wikibooks.

The following list points out some typical sources of frustration and waste of time for software users as well as important usability problems that software developers should be aware of.

- Big runtime frameworks. The .NET framework and the Java virtual machine are frameworks that typically take much more resources than the programs they are running. Such frameworks are frequent sources of resource problems and compatibility problems and they waste a lot of time both during installation of the framework itself, during installation of the program that runs under the framework, during start of the program, and while the program is running. The main reason why such runtime frameworks are used at all is for the sake of cross-platform portability. Unfortunately, the cross-platform compatibility is not always as good as expected. I believe that the portability could be achieved more efficiently by better standardization of programming languages, operating systems, and API's.

- Memory swapping. Software developers typically have more powerful computers with more RAM than end users have. The developers may therefore fail to see the excessive memory swapping and other resource problems that cause the resource-hungry applications to perform poorly for the end user.

- Installation problems. The procedures for installation and uninstallation of programs should be standardized and done by the operating system rather than by individual installation tools.

- Automatic updates. Automatic updating of software can cause problems if the network is unstable or if the new version causes problem that were not present in the old version. Updating mechanisms often disturb the users with nagging pop-up messages saying please install this important new update or even telling the user to restart the computer while he or she is busy concentrating on important work. The updating mechanism should never interrupt the user but only show a discrete icon signaling the availability of an update, or update automatically when the computer is restarted anyway. Software distributors are often abusing the update mechanism to advertise new versions of their software. This is annoying to the user.

- Compatibility problems. All software should be tested on different platforms, different screen resolutions, different system color settings and different user access rights. Software should use standard API calls rather than self-styled hacks and direct hardware access. Available protocols and standardized file formats should be used. Web systems should be tested in different browsers, different platforms, different screen resolutions, etc. Accessibility guidelines should be obeyed.

- Copy protection. Some copy protection schemes are based on hacks that violate or circumvent operating system standards. Such schemes are frequent sources of compatibility problems and system breakdown. Many copy protection schemes are based on hardware identification. Such schemes cause problems when the hardware is updated. Most copy protection schemes are annoying to the user and prevent legitimate backup copying without effectively preventing illegitimate copying. The benefits of a

copy protection scheme should be weighed against the costs in terms of usability problems and necessary support.

- Hardware updating. The change of a hard disk or other hardware often requires that all software be reinstalled and user settings are lost. It is not unusual for the reinstallation work to take a whole workday or more. Many software applications need better backup features, and current operating systems need better support for hard disk copying.

- Security. The vulnerability of software with network access to virus attacks and other abuse is extremely costly to many users. Firewalls, virus scanners and other protection means are among the most frequent causes of compatibility problems and system crash. Furthermore, it is not uncommon for virus scanners to consume more time than anything else on a computer. Security software that is part of the operating system is often more reliable than third party security software.

- Background services. Many services that run in the background are unnecessary for the user and a waste of resources. Consider running the services only when activated by the user.

- Feature bloat. It is common for software to add new features to each new version for marketing reasons. This may cause the software to be slower or require more resources, even if the user never uses the new features.

- Take user feedback seriously. User complaints should be regarded as a valuable source of information about bugs, compatibility problems, usability problems and desired new features. User feedback should be handled in a systematic manner to make sure the information is utilized appropriately. Users should get a reply about investigation of the problems and planned solutions. Patches should be easily available from a website.

5 Choosing the optimal algorithm

The first thing to do when you want to optimize a piece of CPU-intensive software is to find the best algorithm. The choice of algorithm is very important for tasks such as sorting, searching, and mathematical calculations. In such cases, you can obtain much more by choosing the best algorithm than by optimizing the first algorithm that comes to mind. In some cases you may have to test several different algorithms in order to find the one that works best on a typical set of test data.

That being said, I must warn against overkill. Don't use an advanced and complicated algorithm if a simple algorithm can do the job fast enough. For example, some programmers use a hash table for even the smallest list of data. A hash table can improve search times dramatically for very large data bases, but there is no reason to use it for lists that are so small that a binary search, or even a linear search, is fast enough. A hash table increases the size of the program as well as the size of data files. This can actually reduce speed if the bottleneck is file access or cache access rather than CPU time. Another disadvantage of complicated algorithms is that it makes program development more expensive and more error prone.

A discussion of different algorithms for different purposes is beyond the scope of this manual. You have to consult the general literature on algorithms and data structures for standard tasks such as sorting and searching, or the specific literature for more complicated mathematical tasks.

Before you start to code, you may consider whether others have done the job before you. Optimized function libraries for many standard tasks are available from a number of sources. For example, the Boost collection contains well-tested libraries for many common

purposes (www.boost.org). The "Intel Math Kernel Library" contains many functions for common mathematical calculations including linear algebra and statistics, and the "Intel Performance Primitives" library contains many functions for audio and video processing, signal processing, data compression and cryptography (www.intel.com). If you are using an Intel function library then make sure it works well on non-Intel processors, as explained on page 133.

It is often easier said than done to choose the optimal algorithm before you start to program. Many programmers have discovered that there are smarter ways of doing things only after they have put the whole software project together and tested it. The insight you gain by testing and analyzing program performance and studying the bottlenecks can lead to a better understanding of the whole structure of the problem. This new insight can lead to a complete redesign of the program, for example when you discover that there are smarter ways of organizing the data.

A complete redesign of a program that already works is of course a considerable job, but it may be quite a good investment. A redesign can not only improve the performance, it is also likely to lead to a more well-structured program that is easier to maintain. The time you spend on redesigning a program may in fact be less than the time you would have spent fighting with the problems of the original, poorly designed program.

6 Development process

There is a considerable debate about which software development process and software engineering principles to use. I am not going to recommend any specific model. Instead, I will make a few comments about how the development process can influence the performance of the final product.

It is good to do a thorough analysis of the data structure, data flow and algorithms in the planning phase in order to predict which resources are most critical. However, there may be so many unknown factors in the early planning stage that a detailed overview of the problem cannot easily be obtained. In the latter case, you may view the software development work as a learning process where the main feedback comes from testing. Here, you should be prepared for several iterations of redesign.

Some software development models have a strict formalism that requires several layers of abstraction in the logical architecture of the software. You should be aware that there are inherent performance costs to such a formalism. The splitting of software into an excessive number of separate layers of abstraction is a common cause of reduced performance.

Since most development methods are incremental or iterative in nature, it is important to have a strategy for saving a backup copy of every intermediate version. For one-man projects, it is sufficient to make a zip file of every version. For team projects, it is recommended to use a version control tool.

7 The efficiency of different C++ constructs

Most programmers have little or no idea how a piece of program code is translated into machine code and how the microprocessor handles this code. For example, many programmers do not know that double precision calculations are just as fast as single precision. And who would know that a template class is more efficient than a polymorphous class?

This chapter is aiming at explaining the relative efficiency of different C++ language elements in order to help the programmer choosing the most efficient alternative. The theoretical background is further explained in the other volumes in this series of manuals.

7.1 Different kinds of variable storage

Variables and objects are stored in different parts of the memory, depending on how they are declared in a C++ program. This has influence on the efficiency of the data cache (see page 89). Data caching is poor if data are scattered randomly around in the memory. It is therefore important to understand how variables are stored. The storage principles are the same for simple variables, arrays and objects.

Storage on the stack

Variables and objects declared inside a function are stored on the stack, except for the cases described in the sections below.

The stack is a part of memory that is organized in a first-in-last-out fashion. It is used for storing function return addresses (i.e. where the function was called from), function parameters, local variables, and for saving registers that have to be restored before the function returns. Every time a function is called, it allocates the required amount of space on the stack for all these purposes. This memory space is freed when the function returns. The next time a function is called, it can use the same space for the parameters of the new function.

The stack is the most efficient memory space to store data because the same range of memory addresses is reused again and again. If there are no big arrays, then it is almost certain that this part of the memory is mirrored in the level-1 data cache, where it is accessed quite fast.

The lesson we can learn from this is that all variables and objects should preferably be declared inside the function in which they are used.

It is possible to make the scope of a variable even smaller by declaring it inside `{}` brackets. However, most compilers do not free the memory used by a variable until the function returns even though it could free the memory when exiting the `{}` brackets in which the variable is declared. If the variable is stored in a register (see below) then it may be freed before the function returns.

Global or static storage

Variables that are declared outside of any function are called global variables. They can be accessed from any function. Global variables are stored in a static part of the memory. The static memory is also used for variables declared with the `static` keyword, for floating point constants, string constants, array initializer lists, `switch` statement jump tables, and virtual function tables.

The static data area is usually divided into three parts: one for constants that are never modified by the program, one for initialized variables that may be modified by the program, and one for uninitialized variables that may be modified by the program.

The advantage of static data is that it can be initialized to desired values before the program starts. The disadvantage is that the memory space is occupied throughout the whole program execution, even if the variable is only used in a small part of the program. This makes data caching less efficient.

Do not make variables global if you can avoid it. Global variables may be needed for communication between different threads, but that's about the only situation where they are unavoidable. It may be useful to make a variable global if it is accessed by several different

functions and you want to avoid the overhead of transferring the variable as function parameter. But it may be a better solution to make the functions that access the saved variable members of the same class and store the shared variable inside the class. Which solution you prefer is a matter of programming style.

It is often preferable to make a lookup-table static. Example:

```
// Example 7.1
float SomeFunction (int x) {
   static float list[] = {1.1, 0.3, -2.0, 4.4, 2.5};
   return list[x];
}
```

The advantage of using `static` here is that the list does not need to be initialized when the function is called. The values are simply put there when the program is loaded into memory. If the word `static` is removed from the above example, then all five values have to be put into the list every time the function is called. This is done by copying the entire list from static memory to stack memory. Copying constant data from static memory to the stack is a waste of time in most cases, but it may be optimal in special cases where the data are used many times in a loop where almost the entire level-1 cache is used in a number of arrays that you want to keep together on the stack.

String constants and floating point constants are stored in static memory in optimized code. Example:

```
// Example 7.2
a = b * 3.5;
c = d + 3.5;
```

Here, the constant `3.5` will be stored in static memory. Most compilers will recognize that the two constants are identical so that only one constant needs to be stored. All identical constants in the entire program will be joined together in order to minimize the amount of cache space used for constants.

Integer constants are usually included as part of the instruction code. You can assume that there are no caching problems for integer constants.

Register storage

A limited number of variables can be stored in registers instead of main memory. A register is a small piece of memory inside the CPU used for temporary storage. Variables that are stored in registers are accessed very fast. All optimizing compilers will automatically choose the most often used variables in a function for register storage. The same register can be used for multiple variables as long as their uses (live ranges) do not overlap.

The number of registers is very limited. There are approximately six integer registers available for general purposes in 32-bit operating systems and fourteen integer registers in 64-bit systems.

Floating point variables use a different kind of registers. There are eight floating point registers available in 32-bit operating systems and sixteen in 64-bit operating systems. Some compilers have difficulties making floating point register variables in 32-bit mode unless the SSE2 instruction set (or higher) is enabled.

Volatile

The `volatile` keyword specifies that a variable can be changed by another thread. This prevents the compiler from making optimizations that rely on the assumption that the variable always has the value it was assigned previously in the code. Example:

```
// Example 7.3. Explain volatile
volatile int seconds;  // incremented every second by another thread

void DelayFiveSeconds() {
   seconds = 0;
   while (seconds < 5) {
      // do nothing while seconds count to 5
   }
}
```

In this example, the `DelayFiveSeconds` function will wait until `seconds` has been incremented to 5 by another thread. If `seconds` was not declared `volatile` then an optimizing compiler would assume that `seconds` remains zero in the while loop because nothing inside the loop can change the value. The loop would be `while (0 < 5) {}` which would be an infinite loop.

The effect of the keyword `volatile` is that it makes sure the variable is stored in memory rather than in a register and prevents all optimizations on the variable. This can be useful in test situations to avoid that some expression is optimized away.

Note that `volatile` doesn't mean atomic. It doesn't prevent two threads from attempting to write the variable at the same time. The code in the above example may fail in the event that it attempts to set `seconds` to zero at the same time as the other thread increments `seconds`. A safer implementation would only read the value of `seconds` and wait until the value has changed five times.

Thread-local storage

Most compilers can make thread-local storage of static and global variables by using the keyword `__thread` or `__declspec(thread)`. Such variables have one instance for each thread. Thread-local storage is inefficient because it is accessed through a pointer stored in a thread environment block. Thread-local storage should be avoided, if possible, and replaced by storage on the stack (see above, p. 26). Variables stored on the stack always belong to the thread in which they are created.

Far

Systems with segmented memory, such as DOS and 16-bit Windows, allow variables to be stored in a far data segment by using the keyword `far` (arrays can also be `huge`). Far storage, far pointers, and far procedures are inefficient. If a program has too much data for one segment then it is recommended to use a different operating systems that allows bigger segments (32-bit or 64-bit systems).

Dynamic memory allocation

Dynamic memory allocation is done with the operators `new` and `delete` or with the functions `malloc` and `free`. These operators and functions consume a significant amount of time. A part of memory called the heap is reserved for dynamic allocation. The heap can easily become fragmented when objects of different sizes are allocated and deallocated in random order. The heap manager can spend a lot of time cleaning up spaces that are no longer used and searching for vacant spaces. This is called garbage collection. Objects that are allocated in sequence are not necessarily stored sequentially in memory. They may be scattered around at different places when the heap has become fragmented. This makes data caching inefficient.

Dynamic memory allocation also tends to make the code more complicated and error-prone. The program has to keep pointers to all allocated objects and keep track of when they are no longer used. It is important that all allocated objects are also deallocated in all possible cases of program flow. Failure to do so is a common source of error known as memory leak.

An even worse kind of error is to access an object after it has been deallocated. The program logic may need extra overhead to prevent such errors.

See page 92 for a further discussion of the advantages and drawbacks of using dynamic memory allocation.

Some programming languages, such as Java, use dynamic memory allocation for all objects. This is of course inefficient.

Variables declared inside a class

Variables declared inside a class are stored in the order in which they appear in the class declaration. The type of storage is determined where the object of the class is declared. An object of a class, structure or union can use any of the storage methods mentioned above. An object cannot be stored in a register except in the simplest cases, but its data members can be copied into registers.

A class member variable with the `static` modifier will be stored in static memory and will have one and only one instance. Non-static members of the same class will be stored with each instance of the class.

Storing variables in a class or structure is a good way of making sure that variables that are used in the same part of the program are also stored near each other. See page 52 for the pros and cons of using classes.

7.2 Integers variables and operators

Integer sizes

Integers can be different sizes, and they can be signed or unsigned. The following table summarizes the different integer types available.

declaration	size, bits	minimum value	maximum value	in stdint.h
`char`	8	-128	127	`int8_t`
`short int` in 16-bit systems: `int`	16	-32768	32767	`int16_t`
`int` in 16-bit systems: `long int`	32	-2^{31}	$2^{31}-1$	`int32_t`
`long long or int64_t` MS compiler: `__int64` 64-bit Linux: `long int`	64	-2^{63}	$2^{63}-1$	`int64_t`
`unsigned char`	8	0	255	`uint8_t`
`unsigned short int` in 16-bit systems: `unsigned int`	16	0	65535	`uint16_t`
`unsigned int` in 16-bit systems: `unsigned long`	32	0	$2^{32}-1$	`uint32_t`
`unsigned long long or uint64_t` MS compiler: `unsigned __int64` 64-bit Linux: `unsigned long int`	64	0	$2^{64}-1$	`uint64_t`

Table 7.1. Sizes of different integer types

Unfortunately, the way of declaring an integer of a specific size is different for different platforms as shown in the above table. If the standard header file `stdint.h` or

`inttypes.h` is available then it is recommended to use that for a portable way of defining integer types of a specific size.

Integer operations are fast in most cases, regardless of the size. However, it is inefficient to use an integer size that is larger than the largest available register size. In other words, it is inefficient to use 32-bit integers in 16-bit systems or 64-bit integers in 32-bit systems, especially if the code involves multiplication or division.

The compiler will always select the most efficient integer size if you declare an `int`, without specifying the size. Integers of smaller sizes (`char`, `short int`) are only slightly less efficient. In many cases, the compiler will convert these types to integers of the default size when doing calculations, and then use only the lower 8 or 16 bits of the result. You can assume that the type conversion takes zero or one clock cycle. In 64-bit systems, there is only a minimal difference between the efficiency of 32-bit integers and 64-bit integers, as long as you are not doing divisions.

It is recommended to use the default integer size in cases where the size doesn't matter and there is no risk of overflow, such as simple variables, loop counters, etc. In large arrays, it may be preferred to use the smallest integer size that is big enough for the specific purpose in order to make better use of the data cache. Bit-fields of sizes other than 8, 16, 32 and 64 bits are less efficient. In 64-bit systems, you may use 64-bit integers if the application can make use of the extra bits.

The unsigned integer type `size_t` is 32 bits in 32-bit systems and 64 bits in 64-bit systems. It is intended for array sizes and array indices when you want to make sure that overflow never occurs, even for arrays bigger than 2 GB.

When considering whether a particular integer size is big enough for a specific purpose, you must consider if intermediate calculations can cause overflow. For example, in the expression `a = (b*c)/d`, it can happen that `(b*c)` overflows, even if `a`, `b`, `c` and `d` would all be below the maximum value. There is no automatic check for integer overflow.

Signed versus unsigned integers

In most cases, there is no difference in speed between using signed and unsigned integers. But there are a few cases where it matters:

- Division by a constant: Unsigned is faster than signed when you divide an integer with a constant (see page 141). This also applies to the modulo operator `%`.

- Conversion to floating point is faster with signed than with unsigned integers for most instruction sets (see page 145).

- Overflow behaves differently on signed and unsigned variables. An overflow of an unsigned variable produces a low positive result. An overflow of a signed variable is officially undefined. The normal behavior is wrap-around of positive overflow to a negative value, but the compiler may optimize away branches that depend on overflow, based on the assumption that overflow does not occur.

The conversion between signed and unsigned integers is costless. It is simply a matter of interpreting the same bits differently. A negative integer will be interpreted as a very large positive number when converted to unsigned.

```
// Example 7.4. Signed and unsigned integers
int a, b;
double c;
b = (unsigned int)a / 10;     // Convert to unsigned for fast division
```

```
c = a * 2.5;                    // Use signed when converting to double
```

In example 7.4 we are converting `a` to unsigned in order to make the division faster. Of course, this works only if it is certain that `a` will never be negative. The last line is implicitly converting `a` to `double` before multiplying with the constant `2.5`, which is `double`. Here we prefer `a` to be signed.

Be sure not to mix signed and unsigned integers in comparisons, such as `<`. The result of comparing signed with unsigned integers is ambiguous and may produce undesired results.

Integer operators

Integer operations are generally very fast. Simple integer operations such as addition, subtraction, comparison, bit operations and shift operations take only one clock cycle on most microprocessors.

Multiplication and division take longer time. Integer multiplication takes 11 clock cycles on Pentium 4 processors, and 3 - 4 clock cycles on most other microprocessors. Integer division takes 40 - 80 clock cycles, depending on the microprocessor. Integer division is faster the smaller the integer size on AMD processors, but not on Intel processors. Details about instruction latencies are listed in manual 4: "Instruction tables". Tips about how to speed up multiplications and divisions are given on page 139 and 141, respectively.

Increment and decrement operators

The pre-increment operator `++i` and the post-increment operator `i++` are as fast as additions. When used simply to increment an integer variable, it makes no difference whether you use pre-increment or post-increment. The effect is simply identical. For example,
`for (i=0; i<n; i++)` is the same as `for (i=0; i<n; ++i)`. But when the result of the expression is used, then there may be a difference in efficiency. For example,
`x = array[i++]` is more efficient than `x = array[++i]` because in the latter case, the calculation of the address of the array element has to wait for the new value of `i` which will delay the availability of `x` for approximately two clock cycles. Obviously, the initial value of `i` must be adjusted if you change pre-increment to post-increment.

There are also situations where pre-increment is more efficient than post-increment. For example, in the case `a = ++b;` the compiler will recognize that the values of `a` and `b` are the same after this statement so that it can use the same register for both, while the expression `a = b++;` will make the values of `a` and `b` different so that they cannot use the same register.

Everything that is said here about increment operators also applies to decrement operators on integer variables.

7.3 Floating point variables and operators

Modern microprocessors in the x86 family have two different types of floating point registers and correspondingly two different types of floating point instructions. Each type has advantages and disadvantages.

The original method of doing floating point operations involves eight floating point registers organized as a register stack. These registers have long double precision (80 bits). The advantages of using the register stack are:

- All calculations are done with long double precision.

- Conversions between different precisions take no extra time.

- There are intrinsic instructions for mathematical functions such as logarithms and trigonometric functions.

- The code is compact and takes little space in the code cache.

The register stack also has disadvantages:

- It is difficult for the compiler to make register variables because of the way the register stack is organized.

- Floating point comparisons are slow unless the Pentium-II or later instruction set is enabled.

- Conversions between integers and floating point numbers is inefficient.

- Division, square root and mathematical functions take more time to calculate when long double precision is used.

A newer method of doing floating point operations involves eight or sixteen vector registers (XMM or YMM) which can be used for multiple purposes. Floating point operations are done with single or double precision, and intermediate results are always calculated with the same precision as the operands. The advantages of using the vector registers are:

- It is easy to make floating point register variables.

- Vector operations are available for doing parallel calculations on vectors of two double precision or four single precision variables in the XMM registers (see page 107). If the AVX instruction set is available then each vector can hold four double precision or eight single precision variables in the YMM registers.

Disadvantages are:

- Long double precision is not supported.

- The calculation of expressions where operands have mixed precision require precision conversion instructions which can be quite time-consuming (see page 143).

- Mathematical functions must use a function library, but this is often faster than the intrinsic hardware functions.

The floating point stack registers are available in all systems that have floating point capabilities (except in device drivers for 64-bit Windows). The XMM vector registers are available in 64-bit systems and in 32-bit systems when the SSE2 or later instruction set is enabled (single precision requires only SSE). The YMM registers are available if the AVX instruction set is supported by the processor and the operating system. See page 125 for how to test for the availability of these instruction sets.

Most compilers will use the XMM registers for floating point calculations whenever they are available, i.e. in 64-bit mode or when the SSE2 instruction set is enabled. Few compilers are able to mix the two types of floating point operations and choose the type that is optimal for each calculation.

In most cases, double precision calculations take no more time than single precision. When the floating point registers are used, there is simply no difference in speed between single and double precision. Long double precision takes only slightly more time. Single precision

division, square root and mathematical functions are calculated faster than double precision when the XMM registers are used, while the speed of addition, subtraction, multiplication, etc. is still the same regardless of precision on most processors (when vector operations are not used).

You may use double precision without worrying too much about the costs if it is good for the application. You may use single precision if you have big arrays and want to get as much data as possible into the data cache. Single precision is good if you can take advantage of vector operations, as explained on page 107.

Floating point addition takes 3 - 6 clock cycles, depending on the microprocessor. Multiplication takes 4 - 8 clock cycles. Division takes 14 - 45 clock cycles. Floating point comparisons are inefficient when the floating point stack registers are used. Conversions of float or double to integer takes a long time when the floating point stack registers are used.

Do not mix single and double precision when the XMM registers are used. See page 143.

Avoid conversions between integers and floating point variables, if possible. See page 144.

Applications that generate floating point underflow in XMM registers can benefit from setting the flush-to-zero mode rather than generating subnormal numbers in case of underflow:

```
// Example 7.5. Set flush-to-zero mode (SSE):
#include <xmmintrin.h>
_MM_SET_FLUSH_ZERO_MODE(_MM_FLUSH_ZERO_ON);
```

It is strongly recommended to set the flush-to-zero mode unless you have special reasons to use subnormal numbers. You may, in addition, set the denormals-are-zero mode if SSE2 is available:

```
// Example 7.6. Set flush-to-zero and denormals-are-zero mode (SSE2):
#include <xmmintrin.h>
_mm_setcsr(_mm_getcsr() | 0x8040);
```

See page 149 and 122 for more information about mathematical functions.

7.4 Enums

An `enum` is simply an integer in disguise. Enums are exactly as efficient as integers.

Note that the enumerators (value names) will clash with any variable or function having the same name. Enums in header files should therefore have long and unique enumerator names or be put into a namespace.

7.5 Booleans

The order of Boolean operands

The operands of the Boolean operators `&&` and `||` are evaluated in the following way. If the first operand of `&&` is false, then the second operand is not evaluated at all because the result is known to be false regardless of the value of the second operand. Likewise, if the first operand of `||` is true, then the second operand is not evaluated, because the result is known to be true anyway.

It may be advantageous to put the operand that is most often true last in an `&&` expression, or first in an `||` expression. Assume, for example, that `a` is true 50% of the time and `b` is true 10% of the time. The expression `a && b` needs to evaluate `b` when `a` is true, which is

50% of the cases. The equivalent expression `b && a` needs to evaluate `a` only when `b` is true, which is only 10% of the time. This is faster if `a` and `b` take the same time to evaluate and are equally likely to be predicted by the branch prediction mechanism. See page 43 for an explanation of branch prediction.

If one operand is more predictable than the other, then put the most predictable operand first.

If one operand is faster to calculate than the other then put the operand that is calculated the fastest first.

However, you must be careful when swapping the order of Boolean operands. You cannot swap the operands if the evaluation of the operands has side effects or if the first operand determines whether the second operand is valid. For example:

```
// Example 7.7
unsigned int i;  const int ARRAYSIZE = 100;  float list[ARRAYSIZE];
if (i < ARRAYSIZE && list[i] > 1.0) { ...
```

Here, you cannot swap the order of the operands because the expression `list[i]` is invalid when `i` is not less than `ARRAYSIZE`. Another example:

```
// Example 7.8
if (handle != INVALID_HANDLE_VALUE && WriteFile(handle, ...)) { ...
```

Here you cannot swap the order of the Boolean operands because you should not call `WriteFile` if the handle is invalid.

Boolean variables are overdetermined

Boolean variables are stored as 8-bit integers with the value 0 for false and 1 for true.

Boolean variables are overdetermined in the sense that all operators that have Boolean variables as input check if the inputs have any other value than 0 or 1, but operators that have Booleans as output can produce no other value than 0 or 1. This makes operations with Boolean variables as input less efficient than necessary. Take the example:

```
// Example 7.9a
bool a, b, c, d;
c = a && b;
d = a || b;
```

This is typically implemented by the compiler in the following way:

```
bool a, b, c, d;
if (a != 0) {
   if (b != 0) {
      c = 1;
   }
   else {
      goto CFALSE;
   }
}
else {
   CFALSE:
   c = 0;
}
if (a == 0) {
   if (b == 0) {
      d = 0;
   }
```

```
      else {
         goto DTRUE;
      }
   }
   else {
      DTRUE:
      d = 1;
   }
```

This is of course far from optimal. The branches may take a long time in case of mispredictions (see page 43). The Boolean operations can be made much more efficient if it is known with certainty that the operands have no other values than 0 and 1. The reason why the compiler doesn't make such an assumption is that the variables might have other values if they are uninitialized or come from unknown sources. The above code can be optimized if `a` and `b` have been initialized to valid values or if they come from operators that produce Boolean output. The optimized code looks like this:

```
// Example 7.9b
char a = 0, b = 0, c, d;
c = a & b;
d = a | b;
```

Here, I have used `char` (or `int`) instead of `bool` in order to make it possible to use the bitwise operators (`&` and `|`) instead of the Boolean operators (`&&` and `||`). The bitwise operators are single instructions that take only one clock cycle. The OR operator (`|`) works even if `a` and `b` have other values than 0 or 1. The AND operator (`&`) and the EXCLUSIVE OR operator (`^`) may give inconsistent results if the operands have other values than 0 and 1.

Note that there are a few pitfalls here. You cannot use `~` for NOT. Instead, you can make a Boolean NOT on a variable which is known to be 0 or 1 by XOR'ing it with 1:

```
// Example 7.10a
bool a, b;
b = !a;
```

can be optimized to:

```
// Example 7.10b
char a = 0, b;
b = a ^ 1;
```

You cannot replace `a && b` with `a & b` if `b` is an expression that should not be evaluated if `a` is false. Likewise, you cannot replace `a || b` with `a | b` if `b` is an expression that should not be evaluated if `a` is true.

The trick of using bitwise operators is more advantageous if the operands are variables than if the operands are comparisons, etc. For example:

```
// Example 7.11
bool a; float x, y, z;
a = x > y && z != 0;
```

This is optimal in most cases. Don't change `&&` to `&` unless you expect the `&&` expression to generate many branch mispredictions.

Boolean vector operations

An integer may be used as a Boolean vector. For example, if `a` and `b` are 32-bit integers, then the expression `y = a & b;` will make 32 AND-operations in just one clock cycle. The operators `&`, `|`, `^`, `~` are useful for Boolean vector operations.

7.6 Pointers and references

Pointers versus references

Pointers and references are equally efficient because they are in fact doing the same thing. Example:

```
// Example 7.12
void FuncA (int * p) {
   *p = *p + 2;
}

void FuncB (int & r) {
   r = r + 2;
}
```

These two functions are doing the same thing and if you look at the code generated by the compiler you will notice that the code is exactly identical for the two functions. The difference is simply a matter of programming style. The advantages of using pointers rather than references are:

- When you look at the function bodies above, it is clear that `p` is a pointer, but it is not clear whether `r` is a reference or a simple variable. Using pointers makes it more clear to the reader what is happening.

- It is possible to do things with pointers that are impossible with references. You can change what a pointer points to and you can do arithmetic operations with pointers.

The advantages of using references rather than pointers are:

- The syntax is simpler when using references.

- References are safer to use than pointers because in most cases they are sure to point to a valid address. Pointers can be invalid and cause fatal errors if they are uninitialized, if pointer arithmetic calculations go outside the bounds of valid addresses, or if pointers are type-casted to a wrong type.

- References are useful for copy constructors and overloaded operators.

- Function parameters that are declared as constant references accept expressions as arguments while pointers and non-constant references require a variable.

Efficiency

Accessing a variable or object through a pointer or reference may be just as fast as accessing it directly. The reason for this efficiency lies in the way microprocessors are constructed. All non-static variables and objects declared inside a function are stored on the stack and are in fact addressed relative to the stack pointer. Likewise, all non-static variables and objects declared in a class are accessed through the implicit pointer known in C++ as '`this`'. We can therefore conclude that most variables in a well-structured C++ program are in fact accessed through pointers in one way or another. Therefore, micro-processors have to be designed so as to make pointers efficient, and that's what they are.

However, there are disadvantages of using pointers and references. Most importantly, it requires an extra register to hold the value of the pointer or reference. Registers are a scarce resource, especially in 32-bit mode. If there are not enough registers then the pointer has to be loaded from memory each time it is used and this will make the program slower. Another disadvantage is that the value of the pointer is needed a few clock cycles before the time the variable pointed to can be accessed.

Pointer arithmetic

A pointer is in fact an integer that holds a memory address. Pointer arithmetic operations are therefore as fast as integer arithmetic operations. When an integer is added to a pointer then its value is multiplied by the size of the object pointed to. For example:

```
// Example 7.13
struct abc {int a; int b; int c;};
abc * p; int i;
p = p + i;
```

Here, the value that is added to `p` is not `i` but `i*12`, because the size of `abc` is 12 bytes. The time it takes to add `i` to `p` is therefore equal to the time it takes to make a multiplication and an addition. If the size of `abc` is a power of 2 then the multiplication can be replaced by a shift operation which is much faster. In the above example, the size of `abc` can be increased to 16 bytes by adding one more integer to the structure.

Incrementing or decrementing a pointer does not require a multiplication but only an addition. Comparing two pointers requires only an integer comparison, which is fast. Calculating the difference between two pointers requires a division, which is slow unless the size of the type of object pointed to is a power of 2 (See page 141 about division).

The object pointed to can be accessed approximately two clock cycles after the value of the pointer has been calculated. Therefore, it is recommended to calculate the value of a pointer well before the pointer is used. For example, `x = *(p++)` is more efficient than `x = *(++p)` because in the latter case the reading of `x` must wait until a few clock cycles after the pointer `p` has been incremented, while in the former case `x` can be read before `p` is incremented. See page 31 for more discussion of the increment and decrement operators.

7.7 Function pointers

Calling a function through a function pointer typically takes a few clock cycles more than calling the function directly if the target address can be predicted. The target address is predicted if the value of the function pointer is the same as last time the statement was executed. If the value of the function pointer has changed then the target address is likely to be mispredicted, which causes a long delay. See page 43 about branch prediction. A Pentium M processor may be able to predict the target if the changes of the function pointer follows a simple regular pattern, while Pentium 4 and AMD processors are sure to make a misprediction every time the function pointer has changed.

7.8 Member pointers

In simple cases, a data member pointer simply stores the offset of a data member relative to the beginning of the object, and a member function pointer is simply the address of the member function. But there are special cases such as multiple inheritance where a much more complicated implementation is needed. These complicated cases should definitely be avoided.

A compiler has to use the most complicated implementation of member pointers if it has incomplete information about the class that the member pointer refers to. For example:

```
// Example 7.14
class c1;
int c1::*MemberPointer;
```

Here, the compiler has no information about the class `c1` other than its name at the time `MemberPointer` is declared. Therefore, it has to assume the worst possible case and make a complicated implementation of the member pointer. This can be avoided by making the full declaration of `c1` before `MemberPointer` is declared. Avoid multiple inheritance, virtual functions, and other complications that make member pointers less efficient.

Most C++ compilers have various options to control the way member pointers are implemented. Use the option that gives the simplest possible implementation if possible, and make sure you are using the same compiler option for all modules that use the same member pointer.

7.9 Smart pointers

A smart pointer is an object that behaves like a pointer. It has the special feature that the object it points to is deleted when the pointer is deleted. Smart pointers are used only for objects stored in dynamically allocated memory, using `new`. The purpose of using smart pointers is to make sure the object is deleted properly and the memory released when the object is no longer used. A smart pointer may be considered a container that contains only a single element.

The most common implementations of smart pointers are `auto_ptr` and `shared_ptr`. `auto_ptr` has the feature that there is always one, and only one, `auto_ptr` that owns the allocated object, and ownership is transferred from one `auto_ptr` to another by assignment. `shared_ptr` allows multiple pointers to the same object.

There is no extra cost to accessing an object through a smart pointer. Accessing an object by `*p` or `p->member` is equally fast whether `p` is a simple pointer or a smart pointer. But there is an extra cost whenever a smart pointer is created, deleted, copied or transferred from one function to another. These costs are higher for `shared_ptr` than for `auto_ptr`.

Smart pointers can be useful in the situation where the logic structure of a program dictates that an object must be dynamically created by one function and later deleted by another function and these two functions are unrelated to each other (not member of the same class). If the same function or class is responsible for creating and deleting the object then you don't need a smart pointer.

If a program uses many small dynamically allocated objects with each their smart pointer then you may consider if the cost of this solution is too high. It may be more efficient to pool all the objects together into a single container, preferably with contiguous memory. See the discussion of container classes on page 95.

7.10 Arrays

An array is implemented simply by storing the elements consecutively in memory. No information about the dimensions of the array is stored. This makes the use of arrays in C and C++ faster than in other programming languages, but also less safe. This safety problem can be overcome by defining a container class that behaves like an array with bounds checking, as illustrated in this example:

```
// Example 7.15a. Array with bounds checking
template <typename T, unsigned int N> class SafeArray {
protected:
   T a[N];                      // Array with N elements of type T
```

```
public:
   SafeArray() {                  // Constructor
      memset(a, 0, sizeof(a));    // Initialize to zero
   }
   int Size() {                   // Return the size of the array
      return N;
   }
   T & operator[] (unsigned int i) { // Safe [] array index operator
      if (i >= N) {
         // Index out of range. The next line provokes an error.
         // You may insert any other error reporting here:
         return *(T*)0;    // Return a null reference to provoke error
      }
      // No error
      return a[i];      // Return reference to a[i]
   }
};
```

More examples of container classes are given in www.agner.org/optimize/cppexamples.zip.

An array using the above template class is declared by specifying the type and size as template parameters, as example 7.15b below shows. It is accessed with a square brackets index, just as a normal array. The constructor sets all elements to zero. You may remove the `memset` line if you don't want this initialization, or if the type `T` is a class with a default constructor that does the necessary initialization. The compiler may report that `memset` is deprecated. This is because it can cause errors if the size parameter is wrong, but it is still the fastest way to set an array to zero. The `[]` operator will detect an error if the index is out of range (see page 137 on bounds checking). An error message is provoked here in a rather unconventional manner by returning a null reference. This will provoke an error message in a protected operating system if the array element is accessed, and this error is easy to trace with a debugger. You may replace this line by any other form of error reporting. For example, in Windows, you may write `FatalAppExitA(0,"Array index out of range");` or better, make your own error message function.

The following example illustrates how to use `SafeArray`:

```
// Example 7.15b
SafeArray <float, 100> list;            // Make array of 100 floats
for (int i = 0; i < list.Size(); i++) { // Loop through array
   cout << list[i] << endl;             // Output array element
}
```

An array initialized by a list should preferably be static, as explained on page 27. An array can be initialized to zero by using `memset`:

```
// Example 7.16
float list[100];
memset(list, 0, sizeof(list));
```

A multidimensional array should be organized so that the last index changes fastest:

```
// Example 7.17
const int rows = 20, columns = 50;
float matrix[rows][columns];
int i, j;  float x;
   for (i = 0; i < rows; i++)
      for (j = 0; j < columns; j++)
         matrix[i][j] += x;
```

This makes sure that the elements are accessed sequentially. The opposite order of the two loops would make the access non-sequential which makes the data caching less efficient.

The size of all but the first dimension may preferably be a power of 2 if the rows are indexed in a non-sequential order in order to make the address calculation more efficient:

```
// Example 7.18
int FuncRow(int);  int FuncCol(int);
const int rows = 20, columns = 32;
float matrix[rows][columns];
int i;  float x;
for (i = 0; i < 100; i++)
   matrix[FuncRow(i)][FuncCol(i)] += x;
```

Here, the code must compute `(FuncRow(i)*columns + FuncCol(i)) * sizeof(float)` in order to find the address of the matrix element. The multiplication by `columns` in this case is faster when columns is a power of two. In the preceding example, this is not an issue because an optimizing compiler can see that the rows are accessed consecutively and can calculate the address of each row by adding the length of a row to the address of the preceding row.

The same advice applies to arrays of structure or class objects. The size (in bytes) of the objects should preferably be a power of 2 if the elements are accessed in a non-sequential order.

The advice of making the number of columns a power of 2 does not always apply to arrays that are bigger than the level-1 data cache and accessed non-sequentially because it may cause cache contentions. See page 89 for a discussion of this problem.

7.11 Type conversions

The C++ syntax has several different ways of doing type conversions:

```
// Example 7.19
int i;  float f;
f = i;                       // Implicit type conversion
f = (float)i;                // C-style type casting
f = float(i);                // Constructor-style type casting
f = static_cast<float>(i);   // C++ casting operator
```

These different methods have exactly the same effect. Which method you use is a matter of programming style. The time consumption of different type conversions is discussed below.

Signed / unsigned conversion

```
// Example 7.20
int i;
if ((unsigned int)i < 10) { ...
```

Conversions between signed and unsigned integers simply makes the compiler interpret the bits of the integer in a different way. There is no checking for overflow, and the code takes no extra time. These conversions can be used freely without any cost in performance.

Integer size conversion

```
// Example 7.21
int i;  short int s;
i = s;
```

An integer is converted to a longer size by extending the sign-bit if the integer is signed, or by extending with zero-bits if unsigned. This typically takes one clock cycle if the source is

an arithmetic expression. The size conversion often takes no extra time if it is done in connection with reading the value from a variable in memory, as in example 7.22.

```
// Example 7.22
short int a[100];  int i, sum = 0;
for (i=0; i<100; i++) sum += a[i];
```

Converting an integer to a smaller size is done simply by ignoring the higher bits. There is no check for overflow. Example:

```
// Example 7.23
int i;  short int s;
s = (short int)i;
```

This conversion takes no extra time. It simply stores the lower 16 bits of the 32-bit integer.

Floating point precision conversion

Conversions between `float`, `double` and `long double` take no extra time when the floating point register stack is used. It takes between 2 and 15 clock cycles (depending on the processor) when the XMM registers are used. See page 31 for an explanation of register stack versus XMM registers. Example:

```
// Example 7.24
float a; double b;
a += b;
```

In this example, the conversion is costly if XMM registers are used. `a` and `b` should be of the same type to avoid this. See page 143 for further discussion.

Integer to float conversion

Conversion of a signed integer to a `float` or `double` takes 4 - 16 clock cycles, depending on the processor and the type of registers used. Conversion of an unsigned integer takes longer time. It is faster to first convert the unsigned integer to a signed integer if there is no risk of overflow:

```
// Example 7.25
unsigned int u;  double d;
d = (double)(signed int)u;  // Faster, but risk of overflow
```

Integer to float conversions can sometimes be avoided by replacing an integer variable by a float variable. Example:

```
// Example 7.26a
float a[100];  int i;
for (i = 0;  i < 100;  i++) a[i] = 2 * i;
```

The conversion of `i` to float in this example can be avoided by making an additional floating point variable:

```
// Example 7.26b
float a[100];  int i;  float i2;
for (i = 0, i2 = 0;  i < 100;  i++, i2 += 2.0f) a[i] = i2;
```

Float to integer conversion

Conversion of a floating point number to an integer takes a very long time unless the SSE2 or later instruction set is enabled. Typically, the conversion takes 50 - 100 clock cycles. The reason is that the C/C++ standard specifies truncation so the floating point rounding mode has to be changed to truncation and back again.

If there are floating point-to-integer conversions in the critical part of a code then it is important to do something about it. Possible solutions are:

Avoid the conversions by using different types of variables.

Move the conversions out of the innermost loop by storing intermediate results as floating point.

Use 64-bit mode or enable the SSE2 instruction set (requires a microprocessor that supports this).

Use rounding instead of truncation and make a round function using assembly language. See page 144 for details about rounding.

Pointer type conversion

A pointer can be converted to a pointer of a different type. Likewise, a pointer can be converted to an integer, or an integer can be converted to a pointer. It is important that the integer has enough bits for holding the pointer.

These conversions do not produce any extra code. It is simply a matter of interpreting the same bits in a different way or bypassing syntax checks.

These conversions are not safe, of course. It is the responsibility of the programmer to make sure the result is valid.

Re-interpreting the type of an object

It is possible to make the compiler treat a variable or object as if it had a different type by type-casting its address:

```
// Example 7.27
float x;
*(int*)&x |= 0x80000000;   // Set sign bit of x
```

The syntax may seem a little odd here. The address of `x` is type-casted to a pointer to an integer, and this pointer is then de-referenced in order to access `x` as an integer. The compiler does not produce any extra code for actually making a pointer. The pointer is simply optimized away and the result is that `x` is treated as an integer. But the `&` operator forces the compiler to store `x` in memory rather than in a register. The above example sets the sign bit of `x` by using the `|` operator which otherwise can only be applied to integers. It is faster than `x = -abs(x);`.

There are a number of dangers to be aware of when type-casting pointers:

- The trick violates the strict aliasing rule of standard C, specifying that two pointers of different types cannot point to the same object (except for char pointers). An optimizing compiler might store the floating point and integer representations in two different registers. You need to check if the compiler does what you want it to. It is safer to use a union, as in example 14.23 page 146.
- The trick will fail if the object is treated as bigger than it actually is. This above code will fail if an `int` uses more bits than a `float`. (Both use 32 bits in x86 systems).
- If you access part of a variable, for example 32 bits of a 64-bit double, then the code will not be portable to platforms that use big endian storage.
- If you access a variable in parts, for example if you write a 64-bit double 32 bits at a time, then the code is likely to execute slower than intended because of a store forwarding delay in the CPU (See manual 3: "The microarchitecture of Intel, AMD and VIA CPUs").

Const cast

The `const_cast` operator is used for relieving the `const` restriction from a pointer. It has some syntax checking and is therefore more safe than the C-style type-casting without adding any extra code. Example:

```
// Example 7.28
class c1 {
   const int x;        // constant data
   public:
   c1() : x(0) {};     // constructor initializes x to 0
   void xplus2() {     // this function can modify x
      *const_cast<int*>(&x) += 2;}  // add 2 to x
};
```

The effect of the `const_cast` operator here is to remove the `const` restriction on `x`. It is a way of relieving a syntax restriction, but it doesn't generate any extra code and doesn't take any extra time. This is a useful way of making sure that one function can modify `x`, while other functions can not.

Static cast

The `static_cast` operator does the same as the C-style type-casting. It is used, for example, to convert `float` to `int`.

Reinterpret cast

The `reinterpret_cast` operator is used for pointer conversions. It does the same as C-style type-casting with a little more syntax check. It does not produce any extra code.

Dynamic cast

The `dynamic_cast` operator is used for converting a pointer to one class to a pointer to another class. It makes a runtime check that the conversion is valid. For example, when a pointer to a base class is converted to a pointer to a derived class, it checks whether the original pointer actually points to an object of the derived class. This check makes `dynamic_cast` more time-consuming than a simple type casting, but also safer. It may catch programming errors that would otherwise go undetected.

Converting class objects

Conversions involving class objects (rather than pointers to objects) are possible only if the programmer has defined a constructor, an overloaded assignment operator, or an overloaded type casting operator that specifies how to do the conversion. The constructor or overloaded operator is as efficient as a member function.

7.12 Branches and switch statements

The high speed of modern microprocessors is obtained by using a pipeline where instructions are fetched and decoded in several stages before they are executed. However, the pipeline structure has one big problem. Whenever the code has a branch (e.g. an if-else structure), the microprocessor doesn't know in advance which of the two branches to feed into the pipeline. If the wrong branch is fed into the pipeline then the error is not detected until 10 - 20 clock cycles later and the work it has done by fetching, decoding and perhaps speculatively executing instructions during this time has been wasted. The consequence is that the microprocessor wastes several clock cycles whenever it feeds a branch into the pipeline and later discovers that it has chosen the wrong branch.

Microprocessor designers have gone to great lengths to reduce this problem. The most important method that is used is branch prediction. Modern microprocessors are using advanced algorithms to predict which way a branch will go based on the past history of that

branch and other nearby branches. The algorithms used for branch prediction are different for each type of microprocessor. These algorithms are described in detail in manual 3: "The microarchitecture of Intel, AMD and VIA CPUs".

A branch instruction takes typically 0 - 2 clock cycles in the case that the microprocessor has made the right prediction. The time it takes to recover from a branch misprediction is approximately 12 - 25 clock cycles, depending on the processor. This is called the branch misprediction penalty.

Branches are relatively cheap if they are predicted most of the time, but expensive if they are often mispredicted. A branch that always goes the same way is predicted well, of course. A branch that goes one way most of the time and rarely the other way is mispredicted only when it goes the other way. A branch that goes many times one way, then many times the other way is mispredicted only when it changes. A branch that follows a simple periodic pattern can also be predicted quite well if it is inside a loop with few or no other branches. A simple periodic pattern can be, for example, to go one way two times and the other way three times. Then again two times the first way and three times the other way, etc. The worst case is a branch that goes randomly one way or the other with a 50-50 chance of going either way. Such a branch will be mispredicted 50% of the time.

A for-loop or while-loop is also a kind of branch. After each iteration it decides whether to repeat or to exit the loop. The loop-branch is usually predicted well if the repeat count is small and always the same. The maximum loop count that can be predicted perfectly varies between 9 and 64, depending on the processor. Nested loops are predicted well only on some processors. On many processors, a loop that contains several branches is not predicted well.

A switch statements is a kind of branch that can go more than two ways. Switch statements are most efficient if the case labels follow a sequence where each label is equal to the preceding label plus one, because it can be implemented as a table of jump targets. A switch statement with many labels that have values far from each other is inefficient because the compiler must convert it to a branch tree.

On older processors, a switch statement with sequential labels is simply predicted to go the same way as last time it was executed. It is therefore certain to be mispredicted whenever it goes another way than last time. Newer processors are sometimes able to predict a switch statement if it follows a simple periodic pattern or if it is correlated with preceding branches and the number of different targets is small.

The number of branches and switch statements should preferably be kept small in the critical part of a program, especially if the branches are poorly predictable. It may be useful to roll out a loop if this can eliminate branches, as explained in the next paragraph.

The target of branches and function calls are saved in a special cache called the branch target buffer. Contentions in the branch target buffer can occur if a program has many branches or function calls. The consequence of such contentions is that branches can be mispredicted even if they otherwise would be predicted well. Even function calls can be mispredicted for this reason. A program with many branches and function calls in the critical part of the code can therefore suffer from mispredictions.

In some cases it is possible to replace a poorly predictable branch by a table lookup. For example:

```
// Example 7.29a
float a;  bool b;
a = b ? 1.5f : 2.6f;
```

The `?:` operator here is a branch. If it is poorly predictable then replace it by a table lookup:

```
// Example 7.29b
float a;  bool b = 0;
const float lookup[2] = {2.6f, 1.5f};
a = lookup[b];
```

If a `bool` is used as an array index then it is important to make sure it is initialized or comes from a reliable source so that it can have no other values than 0 or 1. See page 34.

In some cases the compiler can automatically replace a branch by a conditional move, depending on the specified instruction set.

The examples on page 137 and 139 show various ways of reducing the number of branches.

Manual 3: "The microarchitecture of Intel, AMD and VIA CPUs" gives more details on branch predictions in the different microprocessors.

7.13 Loops

The efficiency of a loop depends on how well the microprocessor can predict the loop control branch. See the preceding paragraph and manual 3: "The microarchitecture of Intel, AMD and VIA CPUs" for an explanation of branch prediction. A loop with a small and fixed repeat count and no branches inside can be predicted perfectly. As explained above, the maximum loop count that can be predicted depends on the processor. Nested loops are predicted well only on some processors that have a special loop predictor. On other processors, only the innermost loop is predicted well. A loop with a high repeat count is mispredicted only when it exits. For example, if a loop repeats a thousand times then the loop control branch is mispredicted only one time in thousand so the misprediction penalty is only a negligible contribution to the total execution time.

Loop unrolling

In some cases it can be an advantage to unroll a loop. Example:

```
// Example 7.30a
int i;
for (i = 0; i < 20; i++) {
   if (i % 2 == 0) {
      FuncA(i);
   }
   else {
      FuncB(i);
   }
   FuncC(i);
}
```

This loop repeats 20 times and calls alternately `FuncA` and `FuncB`, then `FuncC`. Unrolling the loop by two gives:

```
// Example 7.30b
int i;
for (i = 0; i < 20; i += 2) {
   FuncA(i);
   FuncC(i);
   FuncB(i+1);
   FuncC(i+1);
}
```

This has three advantages:

- The `i<20` loop control branch is executed 10 times rather than 20.
- The fact that the repeat count has been reduced from 20 to 10 means that it can be predicted perfectly on a Pentium 4.
- The `if` branch is eliminated.

Loop unrolling also has disadvantages:

- The unrolled loop takes up more space in the code cache or micro-op cache.
- The Core2 processor performs better on very small loops (less than 65 bytes of code).
- If the repeat count is odd and you unroll by two then there is an extra iteration that has to be done outside the loop. In general, you have this problem when the repeat count is not certain to be divisible by the unroll factor.

Loop unrolling should only be used if there are specific advantages that can be obtained. If a loop contains floating point calculations and the loop counter is an integer, then you can generally assume that the overall computation time is determined by the floating point code rather than by the loop control branch. There is nothing to gain by unrolling the loop in this case.

Loop unrolling should preferably be avoided on processors with a micro-op cache (e.g. Sandy Bridge) because it is important to economize the use of the micro-op cache.

Compilers will usually unroll a loop automatically if this appears to be profitable (see page 72). The programmer does not have to unroll a loop manually unless there is a specific advantage to obtain, such as eliminating the `if`-branch in example 7.30b.

The loop control condition

The most efficient loop control condition is a simple integer counter. A microprocessor with out-of-order capabilities (see page 105) will be able to evaluate the loop control statement several iterations ahead.

It is less efficient if the loop control branch depends on the calculations inside the loop. The following example converts a zero-terminated ASCII string to lower case:

```
// Example 7.31a
char string[100], *p = string;
while (*p != 0) *(p++) |= 0x20;
```

If the length of the string is already known then it is more efficient to use a loop counter:

```
// Example 7.31b
char string[100], *p = string;  int i, StringLength;
for (i = StringLength; i > 0; i--) *(p++) |= 0x20;
```

A common situation where the loop control branch depends on calculations inside the loop is in mathematical iterations such as Taylor expansions and Newton-Raphson iterations. Here the iteration is repeated until the residual error is lower than a certain tolerance. The time it takes to calculate the absolute value of the residual error and compare it to the tolerance may be so high that it is more efficient to determine the worst-case maximum repeat count and always use this number of iterations. The advantage of this method is that the microprocessor can execute the loop control branch ahead of time and resolve any branch misprediction long before the floating point calculations inside the loop are finished. This method is advantageous if the typical repeat count is near the maximum repeat count

and the calculation of the residual error for each iteration is a significant contribution to the total calculation time.

A loop counter should preferably be an integer. If a loop needs a floating point counter then make an additional integer counter. Example:

```
// Example 7.32a
double x, n, factorial = 1.0;
for (x = 2.0; x <= n; x++) factorial *= x;
```

This can be improved by adding an integer counter and using the integer in the loop control condition:

```
// Example 7.32b
double x, n, factorial = 1.0;  int i;
for (i = (int)n - 2, x = 2.0; i >= 0; i--, x++) factorial *= x;
```

Note the difference between commas and semicolons in a loop with multiple counters, as in example 7.32b. A `for`-loop has three clauses: initialization, condition, and increment. The three clauses are separated by semicolons, while multiple statements within each clause are separated by commas. There should be only one statement in the condition clause.

Comparing an integer to zero is sometimes more efficient than comparing it to any other number. Therefore, it is slightly more efficient to make a loop count down to zero than making it count up to some positive value, `n`. But not if the loop counter is used as an array index. The data cache is optimized for accessing arrays forwards, not backwards.

Copying or clearing arrays

It may not be optimal to use a loop for trivial tasks such as copying an array or setting an array to all zeroes. Example:

```
// Example 7.33a
const int size = 1000;  int i;
float a[size], b[size];
// set a to zero
for (i = 0; i < size; i++) a[i] = 0.0;
// copy a to b
for (i = 0; i < size; i++) b[i] = a[i];
```

It is often faster to use the functions `memset` and `memcpy`:

```
// Example 7.33b
const int size = 1000;
float a[size], b[size];
// set a to zero
memset(a, 0, sizeof(a));
// copy a to b
memcpy(b, a, sizeof(b));
```

Most compilers will automatically replace such loops by calls to `memset` and `memcpy`, at least in simple cases. The explicit use of `memset` and `memcpy` is unsafe because serious errors can happen if the size parameter is bigger than the destination array. But the same errors can happen with the loops if the loop count is too big.

7.14 Functions

Function calls may slow down a program for the following reasons:

- The function call makes the microprocessor jump to a different code address and back again. This may take up to 4 clock cycles. In most cases the microprocessor is able to overlap the call and return operations with other calculations to save time.

- The code cache works less efficiently if the code is fragmented and scattered around in memory.

- Function parameters are stored on the stack in 32-bit mode. Storing the parameters on the stack and reading them again takes extra time. The delay is significant if a parameter is part of a critical dependency chain.

- Extra time is needed for setting up a stack frame, saving and restoring registers, and possibly save exception handling information.

- Each function call statement occupies a space in the branch target buffer (BTB). Contentions in the BTB can cause branch mispredictions if the critical part of a program has many calls and branches.

The following methods may be used for reducing the time spent on function calls in the critical part of a program.

Avoid unnecessary functions

Some programming textbooks recommend that every function that is longer than a few lines should be split up into multiple functions. I disagree with this rule. Splitting up a function into multiple smaller functions only makes the program less efficient. Splitting up a function just because it is long does not make the program more clear unless the function is doing multiple logically distinct tasks. A critical innermost loop should preferably be kept entirely inside one function, if possible.

Use inline functions

An inline function is expanded like a macro so that each statement that calls the function is replaced by the function body. A function is usually inlined if the `inline` keyword is used or if its body is defined inside a class definition. Inlining a function is advantageous if the function is small or if it is called only from one place in the program. Small functions are often inlined automatically by the compiler. On the other hand, the compiler may in some cases ignore a request for inlining a function if the inlining causes technical problems or performance problems.

Avoid nested function calls in the innermost loop

A function that calls other functions is called a *frame function*, while a function that does not call any other function is called a *leaf function*. Leaf functions are more efficient than frame functions for reasons explained on page 63. If the critical innermost loop of a program contains calls to frame functions then the code can probably be improved by inlining the frame function or by turning the frame function into a leaf function by inlining all the functions that it calls.

Use macros instead of functions

A macro declared with `#define` is certain to be inlined. But beware that macro parameters are evaluated every time they are used. Example:

```
// Example 7.34a. Use macro as inline function
#define MAX(a,b) (a > b ? a : b)
y = MAX(f(x), g(x));
```

In this example, `f(x)` or `g(x)` is calculated twice because the macro is referencing it twice.

You can avoid this by using an inline function instead of a macro. If you want the function to work with any type of parameters then make it a template:

```
// Example 7.34b. Replace macro by template
template <typename T>
static inline T max(T const & a, T const & b) {
   return a > b ? a : b;
}
```

Another problem with macros is that the name cannot be overloaded or limited in scope. A macro will interfere with any function or variable having the same name, regardless of scope or namespaces. Therefore, it is important to use long and unique names for macros, especially in header files.

Use fastcall functions

The keyword `__fastcall` changes the function calling method in 32-bit mode so that the first two (three on CodeGear compiler) integer parameters are transferred in registers rather than on the stack. This can improve the speed of functions with integer parameters.

Floating point parameters are not affected by `__fastcall`. The implicit '`this`' pointer in member functions is also treated like a parameter, so there may be only one free register left for transferring additional parameters. Therefore, make sure that the most critical integer parameter comes first when you are using `__fastcall`. Function parameters are transferred in registers by default in 64-bit mode. Therefore, the `__fastcall` keyword is not recognized in 64-bit mode.

Make functions local

A function that is used only within the same module (i.e. the current `.cpp` file) should be made local. This makes it easier for the compiler to inline the function and to optimize across function calls. There are three ways to make a function local:

1. Add the keyword `static` to the function declaration. This is the simplest method, but it doesn't work with class member functions, where `static` has a different meaning.

2. Put the function or class into an anonymous namespace.

3. The Gnu compiler allows `"__attribute__((visibility("hidden")))"`.

Use whole program optimization

Some compilers have an option for whole program optimization or for combining multiple `.cpp` files into a single object file. This enables the compiler to optimize register allocation and parameter transfer across all `.cpp` modules that make up a program. Whole program optimization cannot be used for function libraries distributed as object or library files.

Use 64-bit mode

Parameter transfer is more efficient in 64-bit mode than in 32-bit mode, and more efficient in 64-bit Linux than in 64-bit Windows. In 64-bit Linux, the first six integer parameters and the first eight floating point parameters are transferred in registers, totaling up to fourteen register parameters. In 64-bit Windows, the first four parameters are transferred in registers, regardless of whether they are integers or floating point numbers. Therefore, 64-bit Linux is more efficient than 64-bit Windows if functions have more than four parameters. There is no difference between 32-bit Linux and 32-bit Windows in this respect.

7.15 Function parameters

Function parameters are transferred by value in most cases. This means that the value of the parameter is copied to a local variable. This is efficient for simple types such as `int`, `float`, `double`, `bool`, `enum` as well as pointers and references.

Arrays are always transferred as pointers unless they are wrapped into a class or structure.

The situation is more complex if the parameter has a composite type such as a structure or class. The transfer of a parameter of composite type is most efficient if all of the following conditions are met:

- the object is so small that it fits into a single register
- the object has no copy constructor and no destructor
- the object has no virtual member
- the object does not use runtime type identification (RTTI)

If any of these conditions is not met then it is usually faster to transfer a pointer or reference to the object. If the object is large then it obviously takes time to copy the entire object. Any copy constructor must be called when the object is copied to the parameter, and the destructor, if any, must be called before the function returns.

The preferred method for transferring composite objects to a function is by a const reference. A const reference makes sure that the original object is not modified. Unlike a pointer or a non-const reference, a const reference allows the function argument to be an expression or an anonymous object. The compiler can easily optimize away a const reference if the function is inlined.

An alternative solution is to make the function a member of the object's class or structure. This is equally efficient.

Simple function parameters are transferred on the stack in 32-bit systems, but in registers in 64-bit systems. The latter is more efficient. 64-bit Windows allows a maximum of four parameters to be transferred in registers. 64-bit Unix systems allow up to fourteen parameters to be transferred in registers (8 float or double plus 6 integer, pointer or reference parameters). The `this` pointer in member functions counts a one parameter. Further details are given in manual 5: "Calling conventions for different C++ compilers and operating systems".

7.16 Function return types

The return type of a function should preferably be a simple type, a pointer, a reference, or void. Returning objects of a composite type is more complex and often inefficient.

Objects of a composite type can be returned in registers only in the simplest cases. See manual 5: "Calling conventions for different C++ compilers and operating systems" for details on when objects can be returned in registers.

Except for the simplest cases, composite objects are returned by copying them into a place indicated by the caller through a hidden pointer. The copy constructor, if any, is usually called in the copying process, and the destructor is called when the original is destroyed. In simple cases, the compiler may be able to avoid the calls to the copy constructor and the destructor by constructing the object on its final destination, but don't count on it.

Instead of returning a composite object, you may consider the following alternatives:

- Make the function a constructor for the object.

- Make the function modify an existing object rather than making a new one. The existing object can be made available to the function through a pointer or reference, or the function could be a member of the object's class.

- Make the function return a pointer or reference to a static object defined inside the function. This is efficient, but risky. The returned pointer or reference is valid only until the next time the function is called and the local object is overwritten, possibly in a different thread. If you forget to make the local object static then it becomes invalid as soon as the function returns.

- Make the function construct an object with `new` and return a pointer to it. This is inefficient because of the costs of dynamic memory allocation. This method also involves the risk of memory leaks if you forget to delete the object.

7.17 Function tail calls

A tail call is a way of optimizing function calls. If the last statement of a function is a call to another function, then the compiler can replace the call by a jump to the second function. An optimizing compiler will do this automatically. The second function will not return to the first function, but directly to the place where the first function was called from. This is more efficient because it eliminates a return. Example:

```
// Example 7.35. Tail call
void function2(int x);

void function1(int y) {
  ...
   function2(y+1);
}
```

Here, the return from `function1` is eliminated by jumping directly to `function2`. This works even if there is a return value:

```
// Example 7.36. Tail call with return value
int function2(int x);

int function1(int y) {
  ...
   return function2(y+1);
}
```

The tail call optimization works only if the two functions have the same return type. If the functions have parameters on the stack (which is mostly the case in 32-bit mode) then the two functions must use the same amount of stack space for parameters.

7.18 Recursive functions

A recursive function is a function that calls itself. Recursive function calls can be useful for handling recursive data structures. The cost of recursive functions is that all parameters and local variables get a new instance for every recursion, and this takes up stack space. Deep recursions also makes the prediction of return addresses less efficient. This problem typically appears with recursion levels deeper than 16 (see the explanation of *return stack buffer* in manual 3: "The microarchitecture of Intel, AMD and VIA CPUs").

Recursive function calls can still be the most efficient solution for handling a branching data tree structure. Recursion is more efficient if the tree structure is broad than if it is deep. A non-branching recursion can always be replaced by a loop, which is more efficient. A common textbook example of a recursive function is the factorial function:

```
// Example 7.37. Factorial as recursive function
unsigned long int factorial(unsigned int n) {
   if (n < 2) return 1;
   return n * factorial(n-1);
}
```

This implementation is very inefficient because all the instances of `n` and all the return addresses take up storage space on the stack. It is more efficient to use a loop:

```
// Example 7.38. Factorial function as loop
unsigned long int factorial(unsigned int n) {
   unsigned long int product = 1;
   while (n > 1) {
      product *= n;
      n--;
   }
   return product;
}
```

Recursive tail calls are more efficient than other recursive calls, but still less efficient than a loop.

Novice programmers sometimes make a call to `main` in order to restart their program. This is a bad idea because the stack becomes filled up with new instances of all local variables for every recursive call to `main`. The proper way to restart a program is to make a loop in `main`.

7.19 Structures and classes

Nowadays, programming textbooks recommend object oriented programming as a means of making software more clear and modular. The so-called objects are instances of structures and classes. The object oriented programming style has both positive and negative impacts on program performance. The positive effects are:

- Variables that are used together are also stored together if they are members of the same structure or class. This makes data caching more efficient.

- Variables that are members of a class need not be passed as parameters to a class member function. The overhead of parameter transfer is avoided for these variables.

The negative effects of object oriented programming are:

- Non-static member functions have a '`this`' pointer which is transferred as an implicit parameter to the function. The overhead of parameter transfer for '`this`' is incurred on all non-static member functions.

- The '`this`' pointer takes up one register. Registers are a scarce resource in 32-bit systems.

- Virtual member functions are less efficient (see page 55).

No general statement can be made about whether the positive or the negative effects of object oriented programming are dominating. At least, it can be said that the use of classes and member functions is not expensive. You may use an object oriented programming style if it is good for the logical structure and clarity of the program as long as you avoid an excessive number of function calls in the most critical part of the program. The use of structures (without member functions) has no negative effect on performance.

7.20 Class data members (instance variables)

The data members of a class or structure are stored consecutively in the order in which they are declared whenever an instance of the class or structure is created. There is no performance penalty for organizing data into classes or structures. Accessing a data member of a class or structure object takes no more time than accessing a simple variable.

Most compilers will align data members to round addresses in order to optimize access, as given in the following table.

Type	size, bytes	alignment, bytes
bool	1	1
char, signed or unsigned	1	1
short int, signed or unsigned	2	2
int, signed or unsigned	4	4
64-bit integer, signed or unsigned	8	8
pointer or reference, 32-bit mode	4	4
pointer or reference, 64-bit mode	8	8
float	4	4
double	8	8
long double	8, 10, 12 or 16	8 or 16

Table 7.2. Alignment of data members.

This alignment can cause holes of unused bytes in a structure or class with members of mixed sizes. For example:

```
// Example 7.39a
struct S1 {
   short int a;  // 2 bytes. first byte at  0, last byte at  1
                 // 6 unused bytes
   double b;     // 8 bytes. first byte at  8, last byte at 15
   int d;        // 4 bytes. first byte at 16, last byte at 19
                 // 4 unused bytes
};
S1 ArrayOfStructures[100];
```

Here, there are 6 unused bytes between `a` and `b` because `b` has to start at an address divisible by 8. There are also 4 unused bytes in the end. The reason for this is that the next instance of `S1` in the array must begin at an address divisible by 8 in order to align its `b` member by 8. The number of unused bytes can be reduced to 2 by putting the smallest members last:

```
// Example 7.39b
struct S1 {
   double b;     // 8 bytes. first byte at  0, last byte at  7
   int d;        // 4 bytes. first byte at  8, last byte at 11
   short int a;  // 2 bytes. first byte at 12, last byte at 13
                 // 2 unused bytes
};
S1 ArrayOfStructures[100];
```

This reordering has made the structure 8 bytes smaller and the array 800 bytes smaller.

Structure and class objects can often be made smaller by reordering the data members. If the class has at least one virtual member functions then there is a pointer to a virtual table before the first data member or after the last member. This pointer is 4 bytes in 32-bit systems and 8 bytes in 64-bit systems. If you are in doubt how big a structure or each of its

members are then you may make some tests with the `sizeof` operator. The value returned by the `sizeof` operator includes any unused bytes in the end of the object.

The code for accessing a data member is more compact if the offset of the member relative to the beginning of the structure or class is less than 128 because the offset can be expressed as an 8-bit signed number. If the offset relative to the beginning of the structure or class is 128 bytes or more then the offset has to be expressed as a 32-bit number (the instruction set has nothing between 8 bit and 32 bit offsets). Example:

```
// Example 7.40
class S2 {
   public:
   int a[100];  // 400 bytes. first byte at   0, last byte at 399
   int b;       // 4 bytes.   first byte at 400, last byte at 403
   int ReadB() {return b;}
};
```

The offset of `b` is 400 here. Any code that accesses `b` through a pointer or a member function such as `ReadB` needs to code the offset as a 32-bit number. If `a` and `b` are swapped then both can be accessed with an offset that is coded as an 8-bit signed number, or no offset at all. This makes the code more compact so that the code cache is used more efficiently. It is therefore recommended that big arrays and other big objects come last in a structure or class declaration and the most often used data members come first. If it is not possible to contain all data members within the first 128 bytes then put the most often used members in the first 128 bytes.

7.21 Class member functions (methods)

Each time a new object of a class is declared or created it will generate a new instance of the data members. But each member function has only one instance. The function code is not copied because the same code can be applied to all instances of the class.

Calling a member function is as fast as calling a simple function with a pointer or reference to a structure. For example:

```
// Example 7.41
class S3 {
   public:
   int a;
   int b;
   int Sum1() {return a + b;}
};
int Sum2(S3 * p) {return p->a + p->b;}
int Sum3(S3 & r) {return  r.a +  r.b;}
```

The three functions `Sum1`, `Sum2` and `Sum3` are doing exactly the same thing and they are equally efficient. If you look at the code generated by the compiler, you will notice that some compilers will make exactly identical code for the three functions. `Sum1` has an implicit '`this`' pointer which does the same thing as `p` and `r` in `Sum2` and `Sum3`. Whether you want to make the function a member of the class or give it a pointer or reference to the class or structure is simply a matter of programming style. Some compilers make `Sum1` slightly more efficient than `Sum2` and `Sum3` in 32-bit Windows by transferring '`this`' in a register rather than on the stack.

A `static` member function cannot access any non-static data members or non-static member functions. A static member function is faster than a non-static member function because it doesn't need the '`this`' pointer. You may make member functions faster by making them static if they don't need any non-static access.

7.22 Virtual member functions

Virtual functions are used for implementing polymorphic classes. Each instance of a polymorphic class has a pointer to a table of pointers to the different versions of the virtual functions. This so-called virtual table is used for finding the right version of the virtual function at runtime. Polymorphism is one of the main reasons why object oriented programs can be less efficient than non-object oriented programs. If you can avoid virtual functions then you can obtain most of the advantages of object oriented programming without paying the performance costs.

The time it takes to call a virtual member function is a few clock cycles more than it takes to call a non-virtual member function, provided that the function call statement always calls the same version of the virtual function. If the version changes then you may get a misprediction penalty of 10 - 20 clock cycles. The rules for prediction and misprediction of virtual function calls is the same as for switch statements, as explained on page 44.

The dispatching mechanism can be bypassed when the virtual function is called on an object of known type, but you cannot always rely on the compiler bypassing the dispatch mechanism even when it would be obvious to do so. See page 75.

Runtime polymorphism is needed only if it cannot be known at compile time which version of a polymorphic member function is called. If virtual functions are used in a critical part of a program then you may consider whether it is possible to obtain the desired functionality without polymorphism or with compile-time polymorphism.

It is sometimes possible to obtain the desired polymorphism effect with templates instead of virtual functions. The template parameter should be a class containing the functions that have multiple versions. This method is faster because the template parameter is always resolved at compile time rather than at runtime. Example 7.47 on page 59 shows an example of how to do this. Unfortunately, the syntax is so kludgy that it may not be worth the effort.

7.23 Runtime type identification (RTTI)

Runtime type identification adds extra information to all class objects and is not efficient. If the compiler has an option for RTTI then turn it off and use alternative implementations.

7.24 Inheritance

An object of a derived class is implemented in the same way as an object of a simple class containing the members of both parent and child class. Members of parent and child class are accessed equally fast. In general, you can assume that there is hardly any performance penalty to using inheritance.

There may be a slight degradation in code caching for the following reasons:

- The size of the parent class data members is added to the offset of the child class members. The code that accesses data members with a total offset bigger than 127 bytes is slightly less compact. See page 54.

- The member functions of parent and child are typically stored in different modules. This may cause a lot of jumping around and less efficient code caching. This problem can be solved by making sure that functions which are called near each other are also stored near each other. See page 90 for details.

Inheritance from multiple parent classes in the same generation can cause complications with member pointers and virtual functions or when accessing an object of a derived class through a pointer to one of the base classes. You may avoid multiple inheritance by making objects inside the derived class:

```
// Example 7.42a. Multiple inheritance
class B1; class B2;
class D : public B1, public B2 {
public:
   int c;
};
```

Replace with:

```
// Example 7.42b. Alternative to multiple inheritance
class B1; class B2;
class D : public B1 {
public:
   B2 b2;
   int c;
};
```

7.25 Constructors and destructors

A constructor is implemented internally as a member function which returns a reference to the object. The allocation of memory for a new object is not necessarily done by the constructor itself. Constructors are therefore as efficient as any other member functions. This applies to default constructors, copy constructors, and any other constructors.

A class doesn't need a constructor. A default constructor is not needed if the object doesn't need initialization. A copy constructor is not needed if the object can be copied simply by copying all data members. A simple constructor may be inlined for improved performance.

A copy constructor may be called whenever an object is copied by assignment, as a function parameter, or as a function return value. The copy constructor can be a time consumer if it involves allocation of memory or other resources. There are various ways to avoid this wasteful copying of memory blocks, for example:

- Use a reference or pointer to the object instead of copying it
- Use a "move constructor" to transfer ownership of the memory block. This requires a compiler with C++0x support.
- Make a member function or friend function or operator that transfers ownership of the memory block from one object to another. The object that looses ownership of the memory block should have its pointer set to NULL. There should of course be a destructor that destroys any memory block that the object owns.

A destructor is as efficient as a member function. Do not make a destructor if it is not necessary. A virtual destructor is as efficient as a virtual member function. See page 55.

7.26 Unions

A union is a structure where data members share the same memory space. A union can be used for saving memory space by allowing two data members that are never used at the same time to share the same piece of memory. See page 91 for an example.

A union can also be used for accessing the same data in different ways. Example:

```
// Example 7.43
union {
   float f;
```

```
   int i;
} x;
x.f = 2.0f;
x.i |= 0x80000000;  // set sign bit of f
cout << x.f;        // will give -2.0
```

In this example, the sign bit of `f` is set by using the bitwise OR operator, which can only be applied to integers.

7.27 Bitfields

Bitfields may be useful for making data more compact. Accessing a member of a bitfield is less efficient than accessing a member of a structure. The extra time may be justified in case of large arrays if it can save cache space or make files smaller.

It is faster to compose a bitfield by the use of `<<` and `|` operations than to write the members individually. Example:

```
// Example 7.44a
struct Bitfield {
   int a:4;
   int b:2;
   int c:2;
};
Bitfield x;
int A, B, C;
x.a = A;
x.b = B;
x.c = C;
```

Assuming that the values of `A`, `B` and `C` are too small to cause overflow, this code can be improved in the following way:

```
// Example 7.44b
union Bitfield {
   struct {
      int a:4;
      int b:2;
      int c:2;
   };
   char abc;
};
Bitfield x;
int A, B, C;
x.abc = A | (B << 4) | (C << 6);
```

Or, if protection against overflow is needed:

```
// Example 7.44c
x.abc = (A & 0x0F) | ((B & 3) << 4) | ((C & 3) <<6 );
```

7.28 Overloaded functions

The different versions of an overloaded function are simply treated as different functions. There is no performance penalty for using overloaded functions.

7.29 Overloaded operators

An overloaded operator is equivalent to a function. Using an overloaded operator is exactly as efficient as using a function that does the same thing.

An expression with multiple overloaded operators will cause the creation of temporary objects for intermediate results, which may be undesired. Example:

```
// Example 7.45a
class vector {                                    // 2-dimensional vector
public:
   float x, y;                                    // x,y coordinates
   vector() {}                                    // default constructor
   vector(float a, float b) {x = a; y = b;}       // constructor
   vector operator + (vector const & a) {         // sum operator
      return vector(x + a.x, y + a.y);}           // add elements
};

vector a, b, c, d;
a = b + c + d;          // makes intermediate object for (b + c)
```

The creation of a temporary object for the intermediate result `(b+c)` can be avoided by joining the operations:

```
// Example 7.45b
a.x = b.x + c.x + d.x;
a.y = b.y + c.y + d.y;
```

Fortunately, most compilers will do this optimization automatically in simple cases.

7.30 Templates

A template is similar to a macro in the sense that the template parameters are replaced by their values before compilation. The following example illustrates the difference between a function parameter and a template parameter:

```
// Example 7.46
int Multiply (int x, int m) {
   return x * m;}

template <int m>
int MultiplyBy (int x) {
   return x * m;}

int a, b;
a = Multiply(10,8);
b = MultiplyBy<8>(10);
```

`a` and `b` will both get the value 10 * 8 = 80. The difference lies in the way `m` is transferred to the function. In the simple function, `m` is transferred at runtime from the caller to the called function. But in the template function, `m` is replaced by its value at compile time so that the compiler sees the constant 8 rather than the variable `m`. The advantage of using a template parameter rather than a function parameter is that the overhead of parameter transfer is avoided. The disadvantage is that the compiler needs to make a new instance of the template function for each different value of the template parameter. If `MultiplyBy` in this example is called with many different factors as template parameters then the code can become very big.

In the above example, the template function is faster than the simple function because the compiler knows that it can multiply by a power of 2 by using a shift operation. `x*8` is replaced by `x<<3`, which is faster. In the case of the simple function, the compiler doesn't know the value of `m` and therefore cannot do the optimization unless the function can be

inlined. (In the above example, the compiler is able to inline and optimize both functions and simply put 80 into `a` and `b`. But in more complex cases it might not be able to do so).

A template parameter can also be a type. The example on page 38 shows how you can make arrays of different types with the same template.

Templates are efficient because the template parameters are always resolved at compile time. Templates make the source code more complex, but not the compiled code. In general, there is no cost in terms of execution speed to using templates.

Two or more template instances will be joined into one if the template parameters are exactly the same. If the template parameters differ then you will get one instance for each set of template parameters. A template with many instances makes the compiled code big and uses more cache space.

Excessive use of templates makes the code difficult to read. If a template has only one instance then you may as well use a `#define`, `const` or `typedef` instead of a template parameter.

Templates may be used for metaprogramming, as explained at page 154.

Using templates for polymorphism

A template class can be used for implementing a compile-time polymorphism, which is more efficient than the runtime polymorphism that is obtained with virtual member functions. The following example shows first the runtime polymorphism:

```
// Example 7.47a. Runtime polymorphism with virtual functions
class CHello {
public:
   void NotPolymorphic();     // Non-polymorphic functions go here
   virtual void Disp();       // Virtual function
   void Hello() {
      cout << "Hello ";
      Disp();                 // Call to virtual function
   }
};

class C1 : public CHello {
   public:
   virtual void Disp() {
      cout << 1;
   }
};

class C2 : public CHello {
   public:
   virtual void Disp() {
      cout << 2;
   }
};

void test () {
   C1 Object1;  C2 Object2;
   CHello * p;
   p = &Object1;
   p->NotPolymorphic();       // Called directly
   p->Hello();                // Writes "Hello 1"
   p = &Object2;
   p->Hello();                // Writes "Hello 2"
}
```

The dispatching to `C1::Disp()` or `C2::Disp()` is done at runtime here if the compiler doesn't know what class of object `p` points to (see page 75). Current compilers are not very good at optimizing away `p` and inlining the call to `Object1.Hello()`, though future compilers may be able to do so.

If it is known at compile-time whether the object belongs to class `C1` or `C2`, then we can avoid the inefficient virtual function dispatch process. This can be done with a special trick which is used in the Active Template Library (ATL) and Windows Template Library (WTL):

```
// Example 7.47b. Compile-time polymorphism with templates

// Place non-polymorphic functions in the grandparent class:
class CGrandParent {
public:
   void NotPolymorphic();
};

// Any function that needs to call a polymorphic function goes in the
// parent class. The child class is given as a template parameter:
template <typename MyChild>
class CParent : public CGrandParent {
public:
   void Hello() {
      cout << "Hello ";
      // call polymorphic child function:
      (static_cast<MyChild*>(this))->Disp();
   }
};

// The child classes implement the functions that have multiple
// versions:
class CChild1 : public CParent<CChild1> {
   public:
   void Disp() {
      cout << 1;
   }
};

class CChild2 : public CParent<CChild2> {
   public:
   void Disp() {
      cout << 2;
   }
};

void test () {
   CChild1 Object1;  CChild2 Object2;
   CChild1 * p1;
   p1 = &Object1;
   p1->Hello();              // Writes "Hello 1"
   CChild2 * p2;
   p2 = &Object2;
   p2->Hello();              // Writes "Hello 2"
}
```

Here `CParent` is a template class which gets information about its child class through a template parameter. It can call the polymorphic member of its child class by type-casting its '`this`' pointer to a pointer to its child class. This is only safe if it has the correct child class name as template parameter. In other words, you must make sure that the declaration

```
class CChild1 : public CParent<CChild1> {
```

has the same name for the child class name and the template parameter.

The order of inheritance is now as follows. The first generation class (`CGrandParent`) contains any non-polymorphic member functions. The second generation class (`CParent<>`) contains any member functions that need to call a polymorphic function. The third generations classes contain the different versions of the polymorphic functions. The second generation class gets information about the third generation class through a template parameter.

No time is wasted on runtime dispatch to virtual member functions if the class of the object is known. This information is contained in `p1` and `p2` having different types. A disadvantage is that `CParent::Hello()` has multiple instances that take up cache space.

The syntax in example 7.47b is admittedly very kludgy. The few clock cycles that we may save by avoiding the virtual function dispatch mechanism is rarely enough to justify such a complicated code that is difficult to understand and therefore difficult to maintain. If the compiler is able to do the devirtualization (see page 75) automatically then it is certainly more convenient to rely on compiler optimization than to use this complicated template method.

7.31 Threads

Threads are used for doing two or more jobs simultaneously or seemingly simultaneously. If the computer has only one CPU core then it is not possible to do two jobs simultaneously. Each thread will get time slices of typically 30 ms for foreground jobs and 10 ms for background jobs. The context switches after each time slice are quite costly because all caches have to adapt to the new context. It is possible to reduce the number of context switches by making longer time slices. This will make applications run faster at the cost of longer response times for user input. (In Windows you can increase the time slices to 120 ms by selecting optimize performance for background services under advanced system performance options. I don't know if this is possible in Linux).

Threads are useful for assigning different priorities to different tasks. For example, in a word processor the user expects an immediate response to pressing a key or moving the mouse. This task must have a high priority. Other tasks such as spell-checking and repagination are running in other threads with lower priority. If the different tasks were not divided into threads with different priorities then the user might experience unacceptably long response times to keyboard and mouse inputs when the program is busy doing the spell checking.

Any task that takes a long time, such as heavy mathematical calculations, should be scheduled in a separate thread if the application has a graphical user interface. Otherwise the program will be unable to respond quickly to keyboard or mouse input.

It is possible to make a thread-like scheduling in an application program without invoking the overhead of the operating system thread scheduler. This can be accomplished by doing the heavy background calculations piece by piece in a function that is called from the message loop of a graphical user interface (`OnIdle` in Windows MFC). This method may be faster than making a separate thread in systems with only one CPU core, but it requires that the background job can be divided into small pieces of a suitable duration.

The best way to fully utilize systems with multiple CPU cores is to divide the job into multiple threads. Each thread can then run on its own CPU core.

There are four kinds of costs to multithreading that we have to take into account when optimizing multithreaded applications:

- The cost of starting and stopping threads. Don't put a task into a separate thread if it is short in duration compared with the time it takes to start and stop the thread.

- The cost of task switching. This cost is minimized if the number of threads with the same priority is no more than the number of CPU cores.

- The cost of synchronizing and communicating between threads. The overhead of semaphores, mutexes, etc. is considerable. If two threads are often waiting for each other in order to get access to the same resource then it may be better to join them into one thread. A variable that is shared between multiple threads must be declared `volatile`. This prevents the compiler from doing optimizations on that variable.

- The different threads need separate storage. No function or class that is used by multiple threads should rely on static or global variables. (See thread-local storage p. 28) The threads have each their stack. This can cause cache contentions if the threads share the same cache.

Multithreaded programs must use thread-safe functions. A thread-safe function should never use static variables.

See chapter 10 page 103 for further discussion of the techniques of multithreading.

7.32 Exceptions and error handling

Runtime errors cause exceptions which can be detected in the form of traps or software interrupts. These exceptions can be caught with the use of try-catch blocks. The program will crash with an error message if exception handling is enabled and there is no try-catch block.

Exception handling is intended for detecting errors that seldom occur and recovering from error conditions in a graceful way. You may think that exception handling takes no extra time as long as the error doesn't occur, but unfortunately this is not always true. The program may have to do a lot of bookkeeping in order to know how to recover in the event of an exception. The costs of this bookkeeping depends very much on the compiler. Some compilers have efficient table-based methods with little or no overhead while other compilers have inefficient code-based methods or require runtime type identification (RTTI), which affects other parts of the code. See ISO/IEC TR18015 Technical Report on C++ Performance for further explanation.

The following example explains why bookkeeping is needed:

```
// Example 7.48
class C1 {
   public:
   ...
   ~C1();
};

void F1() {
   C1 x;
   ...
}

void F0() {
   try {
      F1();
   }
   catch (...) {
   ...
```

```
    }
}
```

The function `F1` is supposed to call the destructor for the object `x` when it returns. But what if an exception occurs somewhere in `F1`? Then we are breaking out of `F1` without returning. `F1` is prevented from cleaning up because it has been interrupted. Now it is the responsibility of the exception handler to call the destructor of `x`. This is only possible if `F1` has saved all information about the destructor to call or any other cleanup that may be necessary. If `F1` calls another function which in turn calls another function, etc., and if an exception occurs in the innermost function, then the exception handler needs all information about the chain of function calls and it needs to follow the track backwards though the function calls to check for all the necessary cleanup jobs to do. This is called stack unwinding.

All functions have to save some information for the exception handler, even if no exception ever happens. This is the reason why exception handling can be expensive in some compilers. If exception handling is not necessary for your application then you should disable it in order to make the code smaller and more efficient. You can disable exception handling for the whole program by turning off the exception handling option in the compiler. You can disable exception handling for a single function by adding `throw()` to the function prototype:

```
void F1() throw();
```

This allows the compiler to assume that `F1` will never throw any exception so that it doesn't have to save recovery information for function `F1`. However, if `F1` calls another function `F2` that can possibly throw an exception then `F1` has to check for exceptions thrown by `F2` and call the `std::unexpected()` function in case `F2` actually throws an exception. Therefore, you should apply the empty `throw()` specification to `F1` only if all functions called by `F1` also have an empty `throw()` specification. The empty `throw()` specification is useful for library functions.

The compiler makes a distinction between *leaf functions* and *frame functions*. A frame function is a function that calls at least one other function. A leaf function is a function that doesn't call any other function. A leaf function is simpler than a frame function because the stack unwinding information can be left out if exceptions can be ruled out or if there is nothing to clean up in case of an exception. A frame function can be turned into a leaf function by inlining all the functions that it calls. The best performance is obtained if the critical innermost loop of a program contains no calls to frame functions.

While an empty `throw()` statement can improve optimizations in some cases, there is no reason to add statements like `throw(A,B,C)` to tell explicitly what kind of exceptions a function can throw. In fact, the compiler may actually add extra code to check that thrown exceptions are indeed of the specified types (See Sutter: A Pragmatic Look at Exception Specifications, Dr Dobbs Journal, 2002).

In some cases, it is optimal to use exception handling even in the most critical part of a program. This is the case if alternative implementations are less efficient and you want to be able to recover from errors. The following example illustrates such a case:

```
// Example 7.49
// Portability note: This example is specific to Microsoft compilers.
// It will look different in other compilers.
#include <excpt.h>
#include <float.h>
#include <math.h>
#define EXCEPTION_FLT_OVERFLOW  0xC0000091L
```

```
void MathLoop() {
   const int arraysize = 1000;  unsigned int dummy;
   double  a[arraysize], b[arraysize], c[arraysize];

   // Enable exception for floating point overflow:
   _controlfp_s(&dummy, 0, _EM_OVERFLOW);
   // _controlfp(0, _EM_OVERFLOW); // if above line doesn't work

   int i = 0;   // Initialize loop counter outside both loops
   // The purpose of the while loop is to resume after exceptions:
   while (i < arraysize) {
      // Catch exceptions in this block:
      __try {
         // Main loop for calculations:
         for ( ; i < arraysize; i++) {

            // Overflow may occur in multiplication here:
            a[i] = log (b[i] * c[i]);
         }
      }
      // Catch floating point overflow but no other exceptions:
      __except (GetExceptionCode() == EXCEPTION_FLT_OVERFLOW
      ? EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) {
         // Floating point overflow has occurred.
         // Reset floating point status:
         _fpreset();
         _controlfp_s(&dummy, 0, _EM_OVERFLOW);
         // _controlfp(0, _EM_OVERFLOW); // if above doesn't work

         // Re-do the calculation in a way that avoids overflow:
         a[i] = log(b[i]) + log(c[i]);

         // Increment loop counter and go back into the for-loop:
         i++;
      }
   }
}
```

Assume that the numbers in `b[i]` and `c[i]` are so big that overflow can occur in the multiplication `b[i]*c[i]`, though this only happens rarely. The above code will catch an exception in case of overflow and redo the calculation in a way that takes more time but avoids the overflow. Taking the logarithm of each factor rather than the product makes sure that no overflow can occur, but the calculation time is doubled.

The time it takes to make support for the exception handling is negligible because there is no `try` block or function call (other than `log`) inside the critical innermost loop. `log` is a library function which we assume is optimized. We cannot change its possible exception handling support anyway. The exception is costly when it occurs, but this is not a problem since we are assuming that the occurrence is rare.

Testing for the overflow condition inside the loop does not cost anything here because we are relying on the microprocessor hardware for raising an exception in case of overflow. The exception is caught by the operating system which redirects it to the exception handler in the program if there is a `try` block.

There is a portability issue to catching hardware exceptions. The mechanism relies on non-standardized details in both compiler, operating system and CPU hardware. Porting such an application to a different platform is likely to require modifications in the code.

Let's look at the possible alternatives to exception handling in this example. We might check for overflow by checking if `b[i]` and `c[i]` are too big before multiplying them. This would

require two floating point comparisons, which are relatively costly because they must be inside the innermost loop. Another possibility is to always use the safe formula `a[i] = log(b[i]) + log(c[i]);`. This would double the number of calls to `log`, and logarithms take a long time to calculate. If there is a way to check for overflow outside the loop without checking all the array elements then this might be a better solution. It might be possible to do such a check before the loop if all the factors are generated from the same few parameters. Or it might be possible to do the check after the loop if the results are combined by some formula into a single result.

Exceptions and vector code

Vector instructions are useful for doing multiple calculations in parallel. This is described in chapter 12 below. Exception handling does not work well with vector code because a single element in a vector may cause an exception while perhaps the other vector elements do not. You may even generate an exception in a not-taken branch because of the way branches are implemented in vector code. If the code can benefit from vector instructions then it is better to disable the trapping of exceptions and rely on the propagation of NAN and INF instead. See chapter 7.34 below. This is further discussed in the document www.agner.org/optimize/nan_propagation.pdf.

Avoiding the cost of exception handling

Exception handling is not necessary when no attempt is made to recover from errors. If you just want the program to issue an error message and stop the program in case of an error then there is no reason to use `try`, `catch`, and `throw`. It is more efficient to define your own error-handling function that simply prints an appropriate error message and then calls `exit`.

Calling `exit` may not be safe if there are allocated resources that need to be cleaned up, as explained below. There are other possible ways of handling errors without using exceptions. The function that detects an error can return with an error code which the calling function can use for recovering or for issuing an error message.

It is recommended to use a systematic and well thought-through approach to error handling. You have to distinguish between recoverable and non-recoverable errors; make sure allocated resources are cleaned up in case of an error; and make appropriate error messages to the user.

Making exception-safe code

Assume that a function opens a file in exclusive mode, and an error condition terminates the program before the file is closed. The file will remain locked after the program is terminated and the user will be unable to access the file until the computer is rebooted. To prevent this kind of problems you must make your program exception safe. In other words, the program must clean up everything in case of an exception or other error condition. Things that may need to be cleaned up include:

- Memory allocated with `new` or `malloc`.
- Handles to windows, graphic brushes, etc.
- Locked mutexes.
- Open database connections.
- Open files and network connections.
- Temporary files that need to be deleted.

- User work that needs to be saved.
- Any other allocated resource.

The C++ way of handling cleanup jobs is to make a destructor. A function that reads or writes a file can be wrapped into a class with a destructor that makes sure the file is closed. The same method can be used for any other resource, such as dynamically allocated memory, windows, mutexes, database connections, etc.

The C++ exception handling system makes sure that all destructors for local objects are called. The program is exception safe if there are wrapper classes with destructors to take care of all cleanup of allocated resources. The system is likely to fail if the destructor causes another exception.

If you make your own error handling system instead of using exception handling then you cannot be sure that all destructors are called and resources cleaned up. If an error handler calls `exit()`, `abort()`, `_endthread()`, etc. then there is no guarantee that all destructors are called. The safe way to handle an unrecoverable error without using exceptions is to return from the function. The function may return an error code if possible, or the error code may be stored in a global object. The calling function must then check for the error code. If the latter function also has something to clean up then it must return to its own caller, and so on.

7.33 Other cases of stack unwinding

The preceding paragraph described a mechanism called stack unwinding that is used by exception handlers for cleaning up and calling any necessary destructors after jumping out of a function in case of an exception without using the normal return route. This mechanism is also used in two other situations:

The stack unwinding mechanism may be used when a thread is terminated. The purpose is to detect if any objects declared in the thread have a destructor that needs to be called. It is recommended to return from functions that require cleanup before terminating a thread. You cannot be certain that a call to `_endthread()` cleans up the stack. This behaviour is implementation dependent.

The stack unwinding mechanism is also used when the function `longjmp` is used for jumping out of a function. Avoid the use of `longjmp` if possible. Don't rely on `longjmp` in time-critical code.

7.34 Propagation of NAN and INF

Floating point errors will propagate to the end result of a series of calculations in most cases. This is a very efficient alternative to exceptions and fault trapping.

Floating point overflow and division by zero gives infinity. If you add or multiply infinity with something you get infinity as a result. The INF code may propagate to the end result in this way. However, not all operations with INF input will give INF as a result. If you divide a normal number by INF you get zero. The special cases INF-INF and INF/INF give NAN (not-a-number). The special code NAN also occurs when you divide zero by zero and when the input of a function is out of range, such as sqrt(-1) and log(-1).

Most operations with a NAN input will give a NAN output, so that the NAN will propagate to the end result. This is a simple and efficient way of detecting floating point errors. Almost all floating point errors will propagate to the end result where they appear as INF or NAN. If you print out the results, you will see INF or NAN instead of a number. No extra code is

needed to keep track of the errors, and there is no extra cost to the propagation of INF and NAN.

A NAN can contain a payload with extra information. A function library can put an error code into this payload in case of an error, and this payload will propagate to the end result.

The function `finite()` will return false when the parameter is INF or NAN, and true if it is a normal floating point number. This can be used for detecting errors before a floating point number is converted to an integer and in other cases where we want to check for errors.

The details of INF and NAN propagation are further explained in the document "NAN propagation versus fault trapping in floating point code" at www.agner.org/optimize/nan_propagation.pdf. This document also discusses situations where the propagation of INF and NAN fails, as well as compiler optimization options that influence the propagation of these codes.

7.35 Preprocessing directives

Preprocessing directives (everything that begins with `#`) are costless in terms of program performance because they are resolved before the program is compiled.

`#if` directives are useful for supporting multiple platforms or multiple configurations with the same source code. `#if` is more efficient than `if` because `#if` is resolved at compile time while `if` is resolved at runtime.

`#define` directives are equivalent to `const` definitions when used for defining constants. For example, `#define ABC 123` and `const int ABC = 123;` are equally efficient because, in most cases, an optimizing compiler can replace an integer constant with its value. However, the `const int` declaration may in some cases take memory space where a `#define` directive never takes memory space. A floating point constant always takes memory space, even when it has not been given a name.

`#define` directives when used as macros are sometimes more efficient than functions. See page 48 for a discussion.

7.36 Namespaces

There is no cost in terms of execution speed to using namespaces.

8 Optimizations in the compiler

8.1 How compilers optimize

Modern compilers can do a lot of modifications to the code in order to improve performance. It is useful for the programmer to know what the compiler can do and what it can not do. The following sections describe some of the compiler optimizations that it is relevant for the programmer to know about.

Function inlining

The compiler can replace a function call by the body of the called function. Example:

```
// Example 8.1a
float square (float a) {
   return a * a;}
```

```
float parabola (float x) {
   return square(x) + 1.0f;}
```

The compiler may replace the call to square by the code inside square:

```
// Example 8.1b
float parabola (float x) {
   return x * x + 1.0f;}
```

The advantages of function inlining are:

The overhead of call and return and parameter transfer are eliminated.
Code caching will be better because the code becomes contiguous.
The code becomes smaller if there is only one call to the inlined function.
Function inlining can open the possibility for other optimizations, as explained below.

The disadvantage of function inlining is that the code becomes bigger if there is more than one call to the inlined function and the function is big. The compiler is more likely to inline a function if it is small or if it is called from only one or a few places.

Constant folding and constant propagation

An expression or subexpression containing only constants will be replaced by the calculated result. Example:

```
// Example 8.2a
double a, b;
a = b + 2.0 / 3.0;
```

The compiler will replace this by

```
// Example 8.2b
a = b + 0.66666666666666666667;
```

This is actually quite convenient. It is easier to write `2.0/3.0` than to calculate the value and write it with many decimals. It is recommended to put a parenthesis around such a subexpression to make sure the compiler recognizes it as a subexpression. For example, `b*2.0/3.0` will be calculated as `(b*2.0)/3.0` rather than as `b*(2.0/3.0)` unless you put a parenthesis around the constant subexpression.

A constant can be propagated through a series of calculations:

```
// Example 8.3a
float parabola (float x) {
   return x * x + 1.0f;}

float a, b;
a = parabola (2.0f);
b = a + 1.0f;
```

The compiler may replace this by

```
// Example 8.3b
a = 5.0f;
b = 6.0f;
```

Constant folding and constant propagation is not possible if the expression contains a function which cannot be inlined or cannot be calculated at compile time. For example:

```
// Example 8.4
double a = sin(0.8);
```

The `sin` function is defined in a separate function library and you cannot expect the compiler to be able to inline this function and calculate it at compile time. Some compilers are able to calculate the most common math functions such as `sqrt` and `pow` at compile-time, but not the more complicated functions like `sin`.

Pointer elimination

A pointer or reference can be eliminated if the target pointed to is known. Example:

```
// Example 8.5a
void Plus2 (int * p) {
   *p = *p + 2;}

int a;
Plus2 (&a);
```

The compiler may replace this by

```
// Example 8.5b
a += 2;
```

Common subexpression elimination

If the same subexpression occurs more than once then the compiler may calculate it only once. Example:

```
// Example 8.6a
int a, b, c;
b = (a+1) * (a+1);
c = (a+1) / 4;
```

The compiler may replace this by

```
// Example 8.6b
int a, b, c, temp;
temp = a+1;
b = temp * temp;
c = temp / 4;
```

Register variables

The most commonly used variables are stored in registers (see page 27).

The maximum number of integer register variables is approximately six in 32-bit systems and fourteen in 64-bit systems.

The maximum number of floating point register variables is eight in 32-bit systems and sixteen in 64-bit systems. Some compilers have difficulties making floating point register variables in 32-bit systems unless the SSE2 (or later) instruction set is enabled.

The compiler will choose the variables that are used most for register variables. This includes pointers and references, which can be stored in integer registers. Typical candidates for register variables are temporary intermediates, loop counters, function parameters, pointers, references, '`this`' pointer, common subexpressions, and induction variables (see below).

A variable cannot be stored in a register if its address is taken, i.e. if there is a pointer or reference to it. Therefore, you should avoid making any pointer or reference to a variable that could benefit from register storage.

Live range analysis

The live range of a variable is the range of code in which the variable is used. An optimizing compiler can use the same register for more than one variable if their live-ranges do not overlap or if they are sure to have the same value. This is useful when the number of available registers is limited. Example:

```
// Example 8.7
int SomeFunction (int a, int x[]) {
   int b, c;
   x[0] = a;
   b = a + 1;
   x[1] = b;
   c = b + 1;
   return c;
}
```

In this example, `a`, `b` and `c` can share the same register because their live ranges do not overlap. If `c = b + 1` is changed to `c = a + 2` then `a` and `b` cannot use the same register because their live ranges now overlap.

Compilers do not normally use this principle for objects stored in memory. It will not use the same memory area for different objects even when their live ranges do not overlap. See page 91 for an example of how to make different objects share the same memory area.

Join identical branches

The code can be made more compact by joining identical pieces of code. Example:

```
// Example 8.8a
double x, y, z;  bool b;
if (b) {
   y = sin(x);
   z = y + 1.;
}
else {
   y = cos(x);
   z = y + 1.;
}
```

The compiler may replace this by

```
// Example 8.8b
double x, y;  bool b;
if (b) {
   y = sin(x);
}
else {
   y = cos(x);
}
z = y + 1.;
```

Eliminate jumps

Jumps can be avoided by copying the code that it jumps to. Example:

```
// Example 8.9a
int SomeFunction (int a, bool b) {
   if (b) {
      a = a * 2;
   }
   else {
      a = a * 3;
```

```
   }
   return a + 1;
}
```

This code has a jump from `a=a*2;` to `return a+1;`. The compiler can eliminate this jump by copying the return statement:

```
// Example 8.9b
int SomeFunction (int a, bool b) {
   if (b) {
      a = a * 2;
      return a + 1;
   }
   else {
      a = a * 3;
      return a + 1;
   }
}
```

A branch can be eliminated if the condition can be reduced to always true or always false:

```
// Example 8.10a
if (true) {
   a = b;
}
else {
   a = c;
}
```

Can be reduced to:

```
// Example 8.10b
a = b;
```

A branch can also be eliminated if the condition is known from a previous branch. Example:

```
// Example 8.11a
int SomeFunction (int a, bool b) {
   if (b) {
      a = a * 2;
   }
   else {
      a = a * 3;
   }
   if (b) {
      return a + 1;
   }
   else {
      return a - 1;
   }
}
```

The compiler may reduce this to:

```
// Example 8.11b
int SomeFunction (int a, bool b) {
   if (b) {
      a = a * 2;
      return a + 1;
   }
   else {
      a = a * 3;
      return a - 1;
```

```
   }
}
```

Loop unrolling

Some compilers will unroll loops if a high degree of optimization is requested. See page 45. This may be advantageous if the loop body is very small or if it opens the possibility for further optimizations. Loops with a very low repeat count may be completely unrolled to avoid the loop overhead. Example:

```
// Example 8.12a
int i, a[2];
for (i = 0; i < 2; i++) a[i] = i+1;
```

The compiler may reduce this to:

```
// Example 8.12b
int a[2];
a[0] = 1; a[1] = 2;
```

Unfortunately, some compilers unroll too much. Excessive loop unrolling is not optimal because it takes too much space in the code cache and it fills up the loop buffer that some microprocessors have. In some cases it can be useful to turn off the loop unroll option in the compiler.

Loop invariant code motion

A calculation may be moved out of a loop if it is independent of the loop counter. Example:

```
// Example 8.13a
int i, a[100], b;
for (i = 0; i < 100; i++) {
   a[i] = b * b + 1;
}
```

The compiler may change this to:

```
// Example 8.13b
int i, a[100], b, temp;
temp = b * b + 1;
for (i = 0; i < 100; i++) {
   a[i] = temp;
}
```

Induction variables

An expression that is a linear function of a loop counter can be calculated by adding a constant to the previous value. Example:

```
// Example 8.14a
int i, a[100];
for (i = 0; i < 100; i++) {
   a[i] = i * 9 + 3;
}
```

The compiler may avoid the multiplication by changing this to:

```
// Example 8.14b
int i, a[100], temp;
temp = 3;
for (i = 0; i < 100; i++) {
   a[i] = temp;
   temp += 9;
```

```
}
```

Induction variables are often used for calculating the addresses of array elements. Example:

```
// Example 8.15a
struct S1 {double a; double b;};
S1 list[100];  int i;
for (i = 0; i < 100; i++) {
   list[i].a = 1.0;
   list[i].b = 2.0;
}
```

In order to access an element in `list`, the compiler must calculate its address. The address of `list[i]` is equal to the address of the beginning of `list` plus `i*sizeof(S1)`. This is a linear function of `i` which can be calculated by an induction variable. The compiler can use the same induction variable for accessing `list[i].a` and `list[i].b`. It can also eliminate `i` and use the induction variable as loop counter when the final value of the induction variable can be calculated in advance. This reduces the code to:

```
// Example 8.15b
struct S1 {double a; double b;};
S1 list[100], *temp;
for (temp = &list[0]; temp < &list[100]; temp++) {
   temp->a = 1.0;
   temp->b = 2.0;
}
```

The factor `sizeof(S1)` = 16 is actually hidden behind the C++ syntax in example 8.15b. The integer representation of `&list[100]` is `(int)(&list[100]) = (int)(&list[0]) + 100*16`, and `temp++` actually adds 16 to the integer value of `temp`.

The compiler doesn't need induction variables to calculate the addresses of array elements of simple types because the CPU has hardware support for calculating the address of an array element if the address can be expressed as a base address plus a constant plus an index multiplied by a factor of 1, 2, 4 or 8, but not any other factor. If `a` and `b` in example 8.15a were `float` instead of `double`, then `sizeof(S1)` would be 8 and no induction variable would be needed because the CPU has hardware support for multiplying the index by 8.

The compilers I have studied do not make induction variables for floating point expressions or more complex integer expressions. See page 81 for an example of how to use induction variables for calculating a polynomial.

Scheduling

A compiler may reorder instructions for the sake of parallel execution. Example:

```
// Example 8.16
float a, b, c, d, e, f, x, y;
x = a + b + c;
y = d + e + f;
```

The compiler may interleave the two formulas in this example so that `a+b` is calculated first, then `d+e`, then `c` is added to the first sum, then `f` is added to the second sum, then the first result is stored in `x`, and last the second result is stored in `y`. The purpose of this is to help the CPU doing multiple calculations in parallel. Modern CPUs are actually able to reorder instructions without help of the compiler (see page 105), but the compiler can make this reordering easier for the CPU.

Algebraic reductions

Most compilers can reduce simple algebraic expressions using the fundamental laws of algebra. For example, a compiler may change the expression `-(-a)` to `a`.

I don't think that programmers write expressions like `-(-a)` very often, but such expressions may occur as a result of other optimizations such as function inlining. Reducible expressions also occur quite often as a result of macro expansions.

Programmers do, however, often write expressions that can be reduced. This may be because the non-reduced expression better explains the logic behind the program or because the programmer hasn't thought about the possibility of algebraic reduction. For example, a programmer may prefer to write `if(!a && !b)` rather than the equivalent `if(!(a || b))` even though the latter has one operator less. Fortunately, all compilers are able to do the reduction in this case.

You cannot expect a compiler to reduce complicated algebraic expressions. For example, only one of the compilers I have tested were able to reduce `(a*b*c)+(c*b*a)` to `a*b*c*2`. It is quite difficult to implement the many rules of algebra in a compiler. Some compilers can reduce some types of expressions and other compilers can reduce other types of expressions, but no compiler I have ever seen can reduce them all. In the case of Boolean algebra, it is possible to implement a universal algorithm (e.g. Quine–McCluskey or Espresso) that can reduce any expression, but none of the compilers I have tested seem to do so.

The compilers are better at reducing integer expressions than floating point expressions, even though the rules of algebra are the same in both cases. This is because algebraic manipulations of floating point expressions may have undesired effects. This effect can be illustrated by the following example:

```
// Example 8.17
char a = -100, b = 100, c = 100, y;
y = a + b + c;
```

Here, `y` will get the value -100+100+100 = 100. Now, according to the rules of algebra, we may write:

```
y = c + b + a;
```

This may be useful if the subexpression `c+b` can be reused elsewhere. In this example, we are using 8-bit integers which range from -128 to +127. An integer overflow will make the value wrap around. Adding 1 to 127 will generate -128, and subtracting 1 from -128 generates 127. The calculation of `c+b` will generate an overflow and give the result -56 rather than 200. Next, we are adding -100 to -56 which will generate an underflow and give the result 100 rather than -156. Surprisingly, we end up with the correct result because the overflow and underflow neutralize each other. This is the reason why it is safe to use algebraic manipulations on integer expressions (except for the <, <=, > and >= operators).

The same argument does not apply to floating point expressions. Floating point variables do not wrap around on overflow and underflow. The range of floating point variables is so large that we do not have to worry much about overflow and underflow except in special mathematical applications. But we do have to worry about loss of precision. Let's repeat the above example with floating point numbers:

```
// Example 8.18
float a = -1.0E8, b = 1.0E8, c = 1.23456, y;
y = a + b + c;
```

The calculation here gives `a+b=0`, and then `0+1.23456 = 1.23456`. But we will not get the same result if we change the order of the operands and add `b` and `c` first. `b+c = 100000001.23456`. The `float` type holds a precision of approximately seven significant digits, so the value of `b+c` will be rounded to `100000000`. When we add `a` to this number we get `0` rather than `1.23456`.

The conclusion to this argument is that the order of floating point operands cannot be changed without the risk of losing precision. The compilers will not do so unless you specify an option that allows less precise floating point calculations. Even with all relevant optimization options turned on, the compilers will not do such obvious reductions as `0/a = 0` because this would be invalid if `a` was zero or infinity or NAN (not a number). Different compilers behave differently because there are different opinions on which imprecisions should be allowed and which not.

You cannot rely on the compiler to do any algebraic reductions on floating point code and you can rely on only the most simple reductions on integer code. It is more safe to do the reductions manually. I have tested the capability to reduce various algebraic expressions on seven different compilers. The results are listed in table 8.1 below.

Devirtualization

An optimizing compiler can bypass the virtual table lookup for a virtual function call if it is known which version of the virtual function is needed. Example:

```
// Example 8.19. Devirtualization
class C0 {
   public:
   virtual void f();
};

class C1 : public C0 {
   public:
   virtual void f();
};

void g() {
   C1 obj1;
   C0 * p = & obj1;
   p->f();                // Virtual call to C1::f
}
```

Without optimization, the compiler needs to look up in a virtual table to see whether the call `p->f()` goes to `C0::f` or `C1::f`. But an optimizing compiler will see that `p` always points to an object of class `C1`, so it can call `C1::f` directly without using the virtual table. Unfortunately, few compilers are able to do this optimization.

8.2 Comparison of different compilers

I have made a series of experiments on seven different brands of C++ compilers to see whether they were able to do different kinds of optimizations. The results are summarized in table 8.1. The table shows whether the different compilers succeeded in applying the various optimization methods and algebraic reductions in my test examples.

The table can give some indication of which optimizations you can expect a particular compiler to do and which optimizations you have to do manually.

It must be emphasized that the compilers may behave differently on different test examples. You cannot expect a compiler to always behave according to the table.

Optimization method	Microsoft	Borland	Intel	Gnu	PathScale	PGI	Digital Mars	Watcom	Codeplay
Function inlining	x	-	x	x	x	x	-	-	x
Constant folding	x	x	x	x	x	x	x	x	x
Constant propagation	x	-	x	x	x	x	-	-	x
Pointer elimination	x	x	x	x	x	x	x	x	x
Common subexpression elimin., integer	x	(x)	x	x	x	x	x	x	x
Common subexpression elimin., float	x	-	x	x	x	x	-	x	x
Register variables, integer	x	x	x	x	x	x	x	x	x
Register variables, float	x	-	x	x	x	x	-	x	x
Live range analysis	x	x	x	x	x	x	x	x	x
Join identical branches	x	-	-	x	-	-	-	x	-
Eliminate jumps	x	x	x	x	x	x	-	x	x
Eliminate branches	x	-	x	x	x	x	-	-	-
Remove branch that is always true/false	x	-	x	x	x	x	x	x	x
Loop unrolling	x	-	x	x	x	x	-	-	x
Loop invariant code motion	x	-	x	x	x	x	x	x	x
Induction variables for array elements	x	x	x	x	x	x	x	x	x
Induction variables for other integer expressions	x	-	x	x	x	-	x	x	x
Induction variables for float expressions	-	-	-	-	-	-	-	-	-
Automatic vectorization	-	-	x	x	x	x	-	-	x
Devirtualization	-	-	-	x	-	-	-	-	-
Profile-guided optimization	x	-	x	x	x	x	-	-	-
Whole program optimization	x	-	x	x	x	-	-	-	-
Integer algebra reductions:									
a+b = b+a	x	(x)	x	x	x	x	-	x	x
a*b = b*a	x	(x)	x	x	x	x	-	x	x
(a+b)+c = a+(b+c)	x	-	x	x	-	-	x	x	-
a+b+c = c+b+a	x	-	-	x	-	-	-	-	-
a+b+c+d = (a+b)+(c+d)	-	-	x	x	-	-	-	-	-
a*b+a*c = a*(b+c)	x	-	x	x	x	-	-	-	x
a*x*x*x + b*x*x + c*x + d = ((a*x+b)*x+c)*x+d	x	-	x	x	x	-	-	-	x
x*x*x*x*x*x*x*x = $((x^2)^2)^2$	-	-	x	-	-	-	-	-	-
a+a+a+a = a*4	x	-	x	x	-	-	-	-	x
-(-a) = a	x	-	x	x	x	x	x	x	-
a-(-b) = a+b	x	-	x	x	x	x	-	x	-
a-a = 0	x	-	x	x	x	x	x	x	x
a+0 = a	x	x	x	x	x	x	x	x	x
a*0 = 0	x	x	x	x	x	x	x	-	x
a*1 = a	x	x	x	x	x	x	x	x	x
(-a)*(-b) = a*b	x	-	x	x	x	-	-	-	-
a/a = 1	-	-	-	-	x	-	-	-	x
a/1 = a	x	x	x	x	x	x	x	x	x
0/a = 0	-	-	-	x	-	-	-	x	x
(-a == -b) = (a == b)	-	-	-	x	x	-	-	-	-
(a+c == b+c) = (a == b)	-	-	-	-	x	-	-	-	-
!(a < b) = (a >= b)	x	x	x	x	x	x	x	x	x
(a<b && b<c && a<c) = (a<b && b<c)	-	-	-	-	-	-	-	-	-
Multiply by constant = shift and add	x	x	x	x	-	x	x	x	-

Divide by constant = multiply and shift	x	-	x	x	x	(-)	x	-	-
Floating point algebra reductions:									
a+b = b+a	x	-	x	x	x	x	-	-	x
a*b = b*a	x	-	x	x	x	x	-	-	x
a+b+c = a+(b+c)	x	-	x	x	-	-	-	-	-
(a+b)+c = a+(b+c)	-	-	x	x	-	-	-	-	-
a*b*c = a*(b*c)	x	-	-	x	-	-	-	-	-
a+b+c+d = (a+b)+(c+d)	-	-	-	x	-	-	-	-	-
a*b+a*c = a*(b+c)	x	-	-	-	x	-	-	-	x
a*x*x*x + b*x*x + c*x + d = ((a*x+b)*x+c)*x+d	x	-	x	x	x	-	-	-	-
x*x*x*x*x*x*x*x = $((x^2)^2)^2$	-	-	-	x	-	-	-	-	-
a+a+a+a = a*4	x	-	-	x	x	-	-	-	-
-(-a) = a	-	-	x	x	x	x	x	x	-
a-(-b) = a+b	-	-	-	x	x	x	-	x	-
a+0 = a	x	-	x	x	x	x	x	x	-
a*0 = 0	-	-	x	x	x	x	-	x	x
a*1 = a	x	-	x	x	x	x	x	-	x
(-a)*(-b) = a*b	-	-	-	x	x	x	-	-	-
a/a = 1	-	-	-	-	-	-	-	-	x
a/1 = a	x	-	x	x	x	-	x	-	-
0/a = 0	-	-	-	x	x	-	-	x	x
(-a == -b) = (a == b)	-	-	-	x	x	-	-	-	-
(-a > -b) = (a < b)	-	-	-	x	x	-	-	-	x
Divide by constant = multiply by reciprocal	x	x	-	x	x	-	-	x	-
Boolean algebra reductions:									
!(!a) = a	x	-	x	x	x	x	x	x	x
(a&&b) || (a&&c) = a&&(b||c)	x	-	x	x	x	-	-	-	-
!a && !b = !(a || b)	x	x	x	x	x	x	x	x	x
a && !a = false, a || !a = true	x	-	x	x	x	x	-	-	-
a && true = a, a || false = a	x	x	x	x	x	x	x	x	-
a && false = false, a || true = true	x	-	x	x	x	x	x	x	-
a && a = a	x	-	x	x	x	x	-	-	-
(a&&b) || (a&&!b) = a	x	-	-	x	x	-	-	-	-
(a&&b) || (!a&&c) = a ? b : c	x	-	x	x	-	-	-	-	-
(a&&b) || (!a&&c) || (b&&c) = a ? b : c	x	-	-	x	-	-	-	-	-
(a&&b) || (a&&b&&c) = a&&b	x	-	-	x	x	-	-	-	-
(a&&b) || (a&&c) || (a&&b&&c) = a&&(b||c)	x	-	-	x	x	-	-	-	-
(a&&!b) || (!a&&b) = a XOR b	-	-	-	-	-	-	-	-	-
Bit vector algebra reductions:									
~(~a) = a	x	-	x	x	x	x	x	-	-
(a&b)|(a&c) = a&(b|c)	x	-	x	x	x	x	-	-	x
(a|b)&(a|c) = a|(b&c)	x	-	x	x	x	x	-	-	x
~a & ~b = ~(a | b)	-	-	x	x	x	x	-	-	-
a & a = a	x	-	-	x	x	x	-	-	x
a & ~a = 0	-	-	x	x	x	x	-	-	-
a & -1 = a, a | 0 = a	x	-	x	x	x	x	x	x	x
a & 0 = 0, a | -1 = -1	x	-	x	x	x	x	x	x	x
(a&b) | (~a&c) | (b&c) = (a&b) | (~a&c)	-	-	-	-	-	-	-	-	-
a&b&c&d = (a&b)&(c&d)	-	-	-	x	-	-	-	-	-
a ^ 0 = a	x	x	x	x	x	-	x	x	x

a ^ -1 = ~a	x	-	x	x	x	-	x	x	-
a ^ a = 0	x	-	x	x	x	x	-	x	x
a ^ ~a = -1	-	-	-	x	x	x	-	-	-
(a&~b) | (~a&b) = a ^ b	-	-	-	-	-	-	-	-	-
~a ^ ~b = a ^ b	-	-	-	x	x	-	-	-	-
a<<b<<c = a<<(b+c)	x	-	x	x	x	-	-	x	x
Integer XMM (vector) reductions:									
Common subexpression elimination	x	n.a.	x	x	x	-	n.a.	n.a.	x
Constant folding	-	n.a.	-	x	-	-	n.a.	n.a.	-
a+b = b+a, a*b = b*a	-	n.a.	-	x	-	-	n.a.	n.a.	x
(a+b)+c = a+(b+c)	-	n.a.	-	-	-	-	n.a.	n.a.	-
a*b+a*c = a*(b+c)	-	n.a.	-	-	-	-	n.a.	n.a.	-
$x*x*x*x*x*x*x*x = ((x^2)^2)^2$	-	n.a.	-	-	-	-	n.a.	n.a.	-
a+a+a+a = a*4	-	n.a.	-	-	-	-	n.a.	n.a.	-
-(-a) = a	-	n.a.	-	-	-	-	n.a.	n.a.	-
a-a = 0	-	n.a.	x	-	-	-	n.a.	n.a.	-
a+0 = a	-	n.a.	-	-	-	-	n.a.	n.a.	-
a*0 = 0	-	n.a.	-	x	-	-	n.a.	n.a.	-
a*1 = a	-	n.a.	-	x	-	-	n.a.	n.a.	-
(-a)*(-b) = a*b	-	n.a.	-	-	-	-	n.a.	n.a.	-
!(a < b) = (a >= b)	-	n.a.	-	-	-	-	n.a.	n.a.	-
Floating point XMM (vector) reductions:									
a+b = b+a, a*b = b*a	x	n.a.	-	x	-	-	n.a.	n.a.	x
a+b+c = a+(b+c)	-	n.a.	-	-	-	-	n.a.	n.a.	-
a*b+a*c = a*(b+c)	-	n.a.	-	-	-	-	n.a.	n.a.	-
-(-a) = a	-	n.a.	-	-	-	-	n.a.	n.a.	-
a-a = 0	-	n.a.	-	x	-	-	n.a.	n.a.	-
a+0 = a	-	n.a.	x	-	-	-	n.a.	n.a.	-
a*0 = 0	-	n.a.	x	-	-	-	n.a.	n.a.	-
a*1 = a	-	n.a.	-	x	-	-	n.a.	n.a.	-
a/1 = a	-	n.a.	-	x	-	-	n.a.	n.a.	-
0/a = 0	-	n.a.	x	x	-	-	n.a.	n.a.	-
Divide by constant = multiply by reciprocal	-	n.a.	-	-	-	-	n.a.	n.a.	-
Boolean XMM (vector) reductions:									
~(~a) = a	-	n.a.	-	-	-	-	n.a.	n.a.	-
(a&b)|(a&c) = a&(b|c)	-	n.a.	-	-	-	-	n.a.	n.a.	-
a & a = a, a | a = a	-	n.a.	x	x	-	-	n.a.	n.a.	-
a & ~a = 0	-	n.a.	-	x	-	-	n.a.	n.a.	-
a & -1 = a, a | 0 = a	-	n.a.	-	-	-	-	n.a.	n.a.	-
a & 0 = 0	-	n.a.	-	x	-	-	n.a.	n.a.	-
a | -1 = -1	-	n.a.	-	-	-	-	n.a.	n.a.	-
a ^ a = 0	-	n.a.	x	x	-	-	n.a.	n.a.	-
andnot(a,a) = 0	-	n.a.	-	x	-	-	n.a.	n.a.	-
a<<b<<c = a<<(b+c)	-	n.a.	-	-	-	-	n.a.	n.a.	-

Table 8.1. Comparison of optimizations in different C++ compilers

The tests were carried out with all relevant optimization options turned on, including relaxed floating point precision. The following compiler versions were tested:
Microsoft C++ Compiler v. 14.00 for 80x86 / x64 (Visual Studio 2005).
Borland C++ 5.82 (Embarcadero/CodeGear/Borland C++ Builder 5, 2009).
Intel C++ Compiler v. 11.1 for IA-32/Intel64, 2009.
Gnu C++ v. 4.1.0, 2006 (Red Hat).
PathScale C++ v. 3.1, 2007.
PGI C++ v. 7.1-4, 2008.
Digital Mars Compiler v. 8.42n, 2004.
Open Watcom C/C++ v. 1.4, 2005.
Codeplay VectorC v. 2.1.7, 2004.
No differences were observed between the optimization capabilities for 32-bit and 64-bit code for the Microsoft, Intel, Gnu and PathScale compilers.

8.3 Obstacles to optimization by compiler

There are several factors that can prevent the compiler from doing the optimizations that we want it to do. It is important for the programmer to be aware of these obstacles and to know how to avoid them. Some important obstacles to optimization are discussed below.

Cannot optimize across modules

The compiler doesn't have information about functions in other modules than the one it is compiling. This prevents it from making optimizations across function calls. Example:

```
// Example 8.20
module1.cpp
int Func1(int x) {
   return x*x + 1;
}

module2.cpp
int Func2() {
   int a = Func1(2);
   ...
}
```

If `Func1` and `Func2` were in the same module then the compiler would be able do function inlining and constant propagation and reduce `a` to the constant 5. But the compiler doesn't have the necessary information about `Func1` when compiling `module2.cpp`.

The simplest way to solve this problem is to combine the multiple `.cpp` modules into one by means of `#include` directives. This is sure to work on all compilers. Some compilers have a feature called whole program optimization, which will enable optimizations across modules (See page 83).

Pointer aliasing

When accessing a variable through a pointer or reference, the compiler may not be able to completely rule out the possibility that the variable pointed to is identical to some other variable in the code. Example:

```
// Example 8.21
void Func1 (int a[], int * p) {
   int i;
   for (i = 0; i < 100; i++) {
      a[i] = *p + 2;
   }
}
```

```
void Func2() {
   int list[100];
   Func1(list, &list[8]);
}
```

Here, it is necessary to reload `*p` and calculate `*p+2` a hundred times because the value pointed to by `p` is identical to one of the elements in `a[]` which will change during the loop. It is not permissible to assume that `*p+2` is a loop-invariant code that can be moved out of the loop. Example 8.21 is indeed a very contrived example, but the point is that the compiler cannot rule out the theoretical possibility that such contrived examples exist. Therefore the compiler is prevented from assuming that `*p+2` is a loop-invariant expression that it can move outside the loop.

Most compilers have an option for assuming no pointer aliasing (`/Oa`). The easiest way to overcome the obstacle of possible pointer aliasing is to turn on this option. This requires that you analyze all pointers and references in the code carefully to make sure that no variable or object is accessed in more than one way in the same part of the code. It is also possible to tell the compiler that a specific pointer does not alias anything by using the keyword `__restrict` or `__restrict__`, if supported by the compiler.

We can never be sure that the compiler takes the hint about no pointer aliasing. The only way to make sure that the code is optimized is to do it explicitly. In example 8.21, you could calculate `*p+2` and store it in a temporary variable outside the loop if you are sure that the pointer does not alias any elements in the array. This method requires that you can predict where the obstacles to optimization are.

Dynamic memory allocation

Any array or object that is allocated dynamically (with `new` or `malloc`) is necessarily accessed through a pointer. It may be obvious to the programmer that pointers to different dynamically allocated objects are not overlapping or aliasing, but the compiler is usually not able to see this. It also prevents the compiler from aligning the data optimally, or from knowing that the objects are aligned. It is preferred to declare objects and fixed size arrays inside the function that needs them.

Pure functions

A pure function is a function that has no side-effects and its return value depends only on the values of its arguments. This closely follows the mathematical notion of a "function".

Multiple calls to a pure function with the same arguments are sure to produce the same result. A compiler can eliminate common subexpressions that contain pure function calls and it can move out loop-invariant code containing pure function calls. Unfortunately, the compiler cannot know that a function is pure if the function is defined in a different module or a function library.

Therefore, it is necessary to do optimizations such as common subexpression elimination, constant propagation, and loop-invariant code motion manually when it involves pure function calls.

The Gnu compiler and the Intel compiler for Linux have an attribute which can be applied to a function prototype to tell the compiler that this is a pure function. Example:

```
// Example 8.22
#ifdef __GNUC__
#define pure_function  __attribute__((const))
#else
#define pure_function
#endif
```

```
double Func1(double) pure_function ;

double Func2(double x) {
   return Func1(x) * Func1(x) + 1.;
}
```

Here, the Gnu compiler will make only one call to `Func1`, while other compilers will make two.

Some other compilers (Microsoft, Intel) know that standard library functions like `sqrt`, `pow` and `log` are pure functions, but unfortunately there is no way to tell these compilers that a user-defined function is pure.

Virtual functions and function pointers

It is rarely possible for the compiler to predict with certainty which version of a virtual function will be called, or what a function pointer points to. Therefore, it cannot inline the function or otherwise optimize across the function call.

Algebraic reduction

Most compilers can do simple algebraic reductions such as -(-a) = a, but they are not able to do more complicated reductions. Algebraic reduction is a complicated process which is difficult to implement in a compiler.

Many algebraic reductions are not permissible for reasons of mathematical purity. In many cases it is possible to construct obscure examples where the reduction would cause overflow or loss of precision, especially in floating point expressions (see page 74). The compiler cannot rule out the possibility that a particular reduction would be invalid in a particular situation, but the programmer can. It is therefore necessary to do the algebraic reductions explicitly in many cases.

Integer expressions are less susceptible to problems of overflow and loss of precision for reasons explained on page 74. It is therefore possible for the compiler to do more reductions on integer expressions than on floating point expressions. Most reductions involving integer addition, subtraction and multiplication are permissible in all cases, while many reductions involving division and relational operators (e.g. '>') are not permissible for reasons of mathematical purity. For example, compilers cannot reduce the integer expression `-a > -b` to `a < b` because of a very obscure possibility of overflow.

Table 8.1 (page 79) shows which reductions the compilers are able to do, at least in some situations, and which reductions they cannot do. All the reductions that the compilers cannot do must be done manually by the programmer.

Floating point induction variables

Compilers cannot make floating point induction variables for the same reason that they cannot make algebraic reductions on floating point expressions. It is therefore necessary to do this manually. This principle is useful whenever a function of a loop counter can be calculated more efficiently from the previous value than from the loop counter. Any expression that is an n'th degree polynomial of the loop counter can be calculated by n additions and no multiplications. The following example shows the principle for a 2'nd order polynomial:

```
// Example 8.23a. Loop to make table of polynomial
const double A = 1.1, B = 2.2, C = 3.3; // Polynomial coefficients
double Table[100];                      // Table
int x;                                  // Loop counter
for (x = 0; x < 100; x++) {
   Table[x] = A*x*x + B*x + C;          // Calculate polynomial
```

```
}
```

The calculation of this polynomial can be done with just two additions by the use of two induction variables:

```
// Example 8.23b. Calculate polynomial with induction variables
const double A = 1.1, B = 2.2, C = 3.3; // Polynomial coefficients
double Table[100];                      // Table
int x;                                  // Loop counter
const double A2 = A + A;                // = 2*A
double Y = C;                           // = A*x*x + B*x + C
double Z = A + B;                       // = Delta Y
for (x = 0; x < 100; x++) {
   Table[x] = Y;                        // Store result
   Y += Z;                              // Update induction variable Y
   Z += A2;                             // Update induction variable Z
}
```

The loop in example 8.23b has two loop-carried dependency chains, namely the two induction variables `Y` and `Z`. Each dependency chain has a latency which is the same as the latency of a floating point addition. This is small enough to justify the method. A longer loop-carried dependency chain would make the induction variable method unfavorable, unless the value is calculated from a value that is two or more iterations back.

The method of induction variables can also be vectorized if you take into account that each value is calculated from the value that lies r places back in the sequence, where r is the number of elements in a vector or the loop unroll factor. A little math is required for finding the right formula in each case.

Inlined functions have a non-inlined copy

Function inlining has the complication that the same function may be called from another module. The compiler has to make a non-inlined copy of the inlined function for the sake of the possibility that the function is also called from another module. This non-inlined copy is dead code if no other modules call the function. This fragmentation of the code makes caching less efficient.

There are various ways around this problem. If a function is not referenced from any other module then add the keyword `static` to the function definition. This tells the compiler that the function cannot be called from any other module. The `static` declaration makes it easier for the compiler to evaluate whether it is optimal to inline the function, and it prevents the compiler from making an unused copy of an inlined function. The `static` keyword also makes various other optimizations possible because the compiler doesn't have to obey any specific calling conventions for functions that are not accessible from other modules. You may add the `static` keyword to all local non-member functions.

Unfortunately, this method doesn't work for class member functions because the `static` keyword has a different meaning for member functions. You can force a member function to be inlined by declaring the function body inside the class definition. This will prevent the compiler from making a non-inlined copy of the function, but it has the disadvantage that the function is always inlined even when it is not optimal to do so (i.e. if the member function is big and is called from many different places).

Some compilers have an option (Windows: `/Gy`, Linux: `-ffunction-sections`) which allows the linker to remove unreferenced functions. It is recommended to turn on this option.

8.4 Obstacles to optimization by CPU

Modern CPUs can do a lot of optimization by executing instructions out of order. Long dependency chains in the code prevent the CPU from doing out-of-order execution, as explained on page 22. Avoid long dependency chains, especially loop-carried dependency chains with long latencies.

8.5 Compiler optimization options

All C++ compilers have various optimization options that you can turn on and off. It is important to study the available options for the compiler you are using and turn on all relevant options.

Many optimization options are incompatible with debugging. A debugger can execute a code one line at a time and show the values of all variables. Obviously, this is not possible when parts of the code have been reordered, inlined, or optimized away. It is common to make two versions of a program executable: a debug version with full debugging support which is used during program development, and a release version with all relevant optimization options turned on. Most IDE's (Integrated Development Environments) have facilities for making a debug version and a release version of object files and executables. Make sure to distinguish these two versions and turn off debugging and profiling support in the optimized version of the executable.

Most compilers offer the choice between optimizing for size and optimizing for speed. Optimizing for size is relevant when the code is fast anyway and you want the executable to be as small as possible or when code caching is critical. Optimizing for speed is relevant when CPU access and memory access are critical time consumers. Choose the strongest optimization option available.

Some compilers offer profile-guided optimization. This works in the following way. First you compile the program with profiling support. Then you make a test run with a profiler which determines the program flow and the number of times each function and branch is executed. The compiler can then use this information to optimize the code and put the different functions in the optimal order.

Some compilers have support for whole program optimization. This works by compiling in two steps. All source files are first compiled to an intermediate file format instead of the usual object file format. The intermediate files are then linked together in the second step where the compilation is finished. Register allocation and function inlining is done at the second step. The intermediate file format is not standardized. It is not even compatible with different versions of the same compiler. It is therefore not possible to distribute function libraries in this format.

Other compilers offer the possibility of compiling multiple `.cpp` files into a single object file. This enables the compiler to do cross-module optimizations when interprocedural optimization is enabled. A more primitive, but efficient, way of doing whole program optimization is to join all source files into one by means of `#include` directives and declare all functions static or inline. This will enable the compiler to do interprocedural optimizations of the whole program.

During the history of CPU development, each new generation of CPUs increased the available instruction set. The newer instruction sets enable the compiler to make more efficient code, but this makes the code incompatible with old CPUs. The Pentium Pro instruction set makes floating point comparisons more efficient. This instruction set is supported by all modern CPUs. The SSE2 instruction set is particularly interesting because it makes floating point code more efficient in some cases and it makes it possible to use vector instructions (see page 107). Using the SSE2 instruction set is not always optimal, though. In some cases the SSE2 instruction set makes floating point code slower, especially

when the code mixes float and double (see page 143). The SSE2 instruction set is supported by most CPUs and operating systems available today.

You may choose a newer instruction set when compatibility with old CPUs is not needed. Even better, you may make multiple versions of the most critical part of the code to support different CPUs. This method is explained on page 125.

The code becomes more efficient when there is no exception handling. It is recommended to turn off support for exception handling unless the code relies on structured exception handling and you want the code to be able to recover from exceptions. See page 62.

It is recommended to turn off support for runtime type identification (RTTI). See page 55.

It is recommended to enable fast floating point calculations or turn off requirements for strict floating point calculations unless the strictness is required. See page 74 and 74 for discussions.

Turn on the option for "function level linking" if available. See page 82 for an explanation of this option.

Use the option for "assume no pointer aliasing" if you are sure the code has no pointer aliasing. See page 79 for an explanation. (The Microsoft compiler supports this option only in the Professional and Enterprise editions).

Do not turn on correction for the "FDIV bug". The FDIV bug is a minor error in the oldest Pentium CPUs which may cause slight imprecision in some rare cases of floating point division. Correction for the FDIV bug causes floating point division to be slower.

Many compilers have an option for "standard stack frame" or "frame pointer". The standard stack frame is used for debugging and exception handling. Omitting the standard stack frame makes function calls faster and makes an extra register available for other purposes. This is advantageous because registers is a scarce resource. Do not use a stack frame unless your program relies on exception handling.

8.6 Optimization directives

Some compilers have many keywords and directives which are used for giving specific optimization instructions at specific places in the code. Many of these directives are compiler-specific. You cannot expect a directive for a Windows compiler to work on a Linux compiler, or vice versa. But most of the Microsoft directives work on the Intel compiler for Windows and the Gnu compiler for Windows, while most of the Gnu directives work on the PathScale and Intel compilers for Linux.

Keywords that work on all C++ compilers

The `register` keyword can be added to a variable declaration to tell the compiler that you want this to be a register variable. The register keyword is only a hint and the compiler may not take the hint, but it can be useful in situations where the compiler is unable to predict which variables will be used most.

The opposite of `register` is `volatile`. The `volatile` keyword makes sure that a variable is never stored in a register, not even temporarily. This is intended for variables that are shared between multiple threads, but it can also be used for turning off all optimizations of a variable for test purposes.

The `const` keyword tells that a variable is never changed. This will allow the compiler to optimize away the variable in many cases. For example:

```
// Example 8.24. Integer constant
const int ArraySize = 1000;
int List[ArraySize];
...
for (int i = 0; i < ArraySize; i++) List[i]++;
```

Here, the compiler can replace all occurrences of `ArraySize` by the value `1000`. The loop in example 8.24 can be implemented in a more efficient way if the value of the loop count (`ArraySize`) is constant and known to the compiler at compile time. No memory will be allocated for an integer constant, unless the address of it (`&ArraySize`) is taken.

A `const` pointer or `const` reference cannot change what it points to. A `const` member function cannot modify data members. It is recommended to use the `const` keyword wherever appropriate to give the compiler additional information about a variable, pointer or member function because this may improve the possibilities for optimization. For example, the compiler can safely assume that the value of a class data member is unchanged across a call to a `const` function that is member of the same class.

The `static` keyword has several meanings depending on the context. The keyword `static`, when applied to a non-member function, means that the function is not accessed by any other modules. This makes inlining more efficient and enables interprocedural optimizations. See page 82.

The keyword `static`, when applied to a global variable means that it is not accessed by any other modules. This enables interprocedural optimizations.

The keyword `static`, when applied to a local variable inside a function means that the variable will be preserved when the function returns and remain unchanged the next time the function is called. This may be inefficient because some compilers will insert extra code to guard the variable against access from multiple threads simultaneously. This may apply even if the variable is `const`.

There may, nevertheless, be a reason to make a local variable `static` and `const` to make sure it is initialized only the first time the function is called. Example:

```
// Example 8.25
void Func () {
   static const double log2 = log(2.0);
   ...
}
```

Here, `log(2.0)` is only calculated the first time `Func` is executed. Without `static`, the logarithm would be re-calculated every time `Func` is executed. This has the disadvantage that the function must check if it has been called before. This is faster than calculating the logarithm again, but it would be even faster to make `log2` a global `const` variable or replace it with the calculated value.

The keyword `static`, when applied to a class member function means that it cannot access any non-static data members or member functions. A static member function is called faster than a non-static member function because it doesn't need a '`this`' pointer. It is recommended to make member functions `static` where appropriate.

Compiler-specific keywords

Fast function calling. `__fastcall` or `__attribute__((fastcall))`. The fastcall modifier can make function calls faster in 32-bit mode. The first two integer parameters are transferred in registers rather than on the stack (three parameters on CodeGear compiler).

Fastcall functions are not compatible across compilers. Fastcall is not needed in 64-bit mode where the parameters are transferred in registers anyway.

Pure function. `__attribute__((const))` (Linux only). Specifies a function to be pure. This allows common subexpression elimination and loop-invariant code motion. See page 80.

Assume no pointer aliasing. `__declspec(noalias)` or `__restrict` or `#pragma optimize("a",on)`. Specifies that pointer aliasing does not occur. See page 79 for an explanation. Note that these directives do not always work.

Data alignment. `__declspec(align(16))` or `__attribute__((aligned(16)))`. Specifies alignment of arrays and structures. Useful for vector operations, see page 107.

8.7 Checking what the compiler does

It can be very useful to study the code that a compiler generates to see how well it optimizes the code. Sometimes the compiler does quite ingenious things to make the code more efficient, and sometimes it does incredibly stupid things. Looking at the compiler output can often reveal things that can be improved by modifications of the source code, as the example below shows.

The best way to check the code that the compiler generates is to use a compiler option for assembly language output. On most compilers you can do this by invoking the compiler from the command line with all the relevant optimization options and the options `-S` or `/Fa` for assembly output. The assembly output option is also available from the IDE on some systems. If the compiler doesn't have an assembly output option then use an object file disassembler.

Note that the Intel compiler has an option for source annotation in the assembly output (`/FAs` or `-fsource-asm`). This option makes the assembly output more readable but unfortunately it prevents certain optimizations. Do not use the source annotation option if you want to see the result of full optimization.

It is also possible to see the compiler-generated code in the disassembly window of a debugger. However, the code that you see in the debugger is not the optimized version because the debugging options prevent optimization. The debugger cannot set a breakpoint in the fully optimized code because it doesn't have the line number information. It is often possible to insert a fixed breakpoint in the code with an inline assembly instruction for interrupt 3. The code is `__asm int 3;` or `__asm ("int 3");` or `__debugbreak();`. If you run the optimized code (release version) in the debugger then it will break at the interrupt 3 breakpoint and show a disassembly, probably without information about function names and variable names. Remember to remove the interrupt 3 breakpoint again.

The following example shows what the assembly output of a compiler can look like and how you can use it for improving the code.

```
// Example 8.26a
void Func(int a[], int & r) {
   int i;
   for (i = 0; i < 100; i++) {
      a[i] = r + i/2;
   }
}
```

The Intel compiler generates the following assembly code from example 8.26a (32-bit mode):

```
; Example 8.26a compiled to assembly:
ALIGN     4                               ; align by 4
PUBLIC ?Func@@YAXQAHAAH@Z                 ; mangled function name
?Func@@YAXQAHAAH@Z  PROC NEAR             ; start of Func
; parameter 1: 8 + esp                    ; a
; parameter 2: 12 + esp                   ; r
$B1$1:                                    ; unused label
        push      ebx                     ; save ebx on stack
        mov       ecx, DWORD PTR [esp+8]  ; ecx = a
        xor       eax, eax                ; eax = i = 0
        mov       edx, DWORD PTR [esp+12] ; edx = r
$B1$2:                                    ; top of loop
        mov       ebx, eax                ; compute i/2 in ebx
        shr       ebx, 31                 ; shift down sign bit of i
        add       ebx, eax                ; i + sign(i)
        sar       ebx, 1                  ; shift right = divide by 2
        add       ebx, DWORD PTR [edx]    ; add what r points to
        mov       DWORD PTR[ecx+eax*4],ebx ; store result in array
        add       eax, 1                  ; i++
        cmp       eax, 100                ; check if i < 100
        jl        $B1$2                   ; repeat loop if true
$B1$3:                                    ; unused label
        pop       ebx                     ; restore ebx from stack
        ret                               ; return
        ALIGN     4                       ; align
?Func@@YAXQAHAAH@Z ENDP                   ; mark end of procedure
```

Most of the comments generated by the compiler have been replaced by my comments, in green. It takes some experience to get used to read and understand compiler-generated assembly code. Let me explain the above code in details. The funny looking name `?Func@@YAXQAHAAH@Z` is the name of `Func` with a lot of added information about the function type and its parameters. This is called name mangling. The characters '`?`', '`@`' and '`$`' are allowed in assembly names. The details about name mangling are explained in manual 5: "Calling conventions for different C++ compilers and operating systems". The parameters `a` and `r` are transferred on the stack at address `esp+8` and `esp+12` and loaded into `ecx` and `edx`, respectively. (In 64-bit mode, the parameters would be transferred in registers rather than on the stack). `ecx` now contains the address of the first element of the array `a` and `edx` contains the address of the variable that `r` points to. A reference is the same as a pointer in assembly code. Register `ebx` is pushed on the stack before it is used and popped from the stack before the function returns. This is because the register usage convention says that a function is not allowed to change the value of `ebx`. Only the registers `eax`, `ecx` and `edx` can be changed freely. The loop counter `i` is stored as a register variable in `eax`. The loop initialisation `i=0;` has been translated to the instruction `xor eax,eax`. This is a common way of setting a register to zero that is more efficient than `mov eax,0`. The loop body begins at the label `$B1$2:`. This is just an arbitrary name that the compiler has chosen for the label. It uses `ebx` as a temporary register for computing `i/2+r`. The instructions `mov ebx,eax` / `shr ebx,31` copies the sign bit of `i` into the least significant bit of `ebx`. The next two instructions `add ebx, eax` / `sar ebx,1` adds this to `i` and shifts one place to the right in order to divide `i` by 2. The instruction `add ebx, DWORD PTR [edx]` adds, not `edx` but the variable whose address is in `edx`, to `ebx`. The square brackets mean use the value in `edx` as a memory pointer. This is the variable that `r` points to. Now `ebx` contains `i/2+r`. The next instruction `mov DWORD PTR [ecx+eax*4],ebx` stores this result in `a[i]`. Note how efficient the calculation of the array address is. `ecx` contains the address of the beginning of the array. `eax` holds the index, `i`. This index must be multiplied by the size (in bytes) of each array element in order to calculate the address of element number `i`. The size of an `int` is 4. So the address of array element `a[i]` is `ecx+eax*4`. The result `ebx` is then stored at address `[ecx+eax*4]`. This is all done in a single instruction. The CPU supports this kind of

instructions for fast access to array elements. The instruction `add eax,1` is the loop increment `i++`. `cmp eax, 100` / `jl $B1$2` is the loop condition `i < 100`. It compares `eax` with `100` and jumps back to the `$B1$2` label if `i < 100`. `pop ebx` restores the value of `ebx` that was saved in the beginning. `ret` returns from the function.

The assembly listing reveals three things that can be optimized further. The first thing we notice is that it does some funny things with the sign bit of `i` in order to divide `i` by `2`. The compiler has not noticed that `i` can never be negative so that we don't have to care about the sign bit. We can tell it this by making `i` an `unsigned int` or by type-casting `i` to `unsigned int` before dividing by `2` (See page 141).

The second thing we notice is that the value pointed to by `r` is re-loaded from memory a hundred times. This is because we forgot to tell the compiler to assume no pointer aliasing (see page 79). Adding the compiler option "assume no pointer aliasing" (if valid) can possibly improve the code.

The third thing that can be improved is that `r+i/2` could be calculated by an induction variable because it is a staircase function of the loop index. The integer division prevents the compiler from making an induction variable unless the loop is rolled out by 2. (See page 72).

The conclusion is that we can help the compiler optimize example 8.26a by rolling out the loop by two and making an explicit induction variable. (This eliminates the need for the first two suggested improvements).

```
// Example 8.26b
void Func(int a[], int & r) {
   int i;
   int Induction = r;
   for (i = 0; i < 100; i += 2) {
      a[i] = Induction;
      a[i+1] = Induction;
      Induction++;
   }
}
```

The compiler generates the following assembly code from example 8.26b:

```
; Example 8.26b compiled to assembly:
ALIGN     4                                 ; align by 4
PUBLIC ?Func@@YAXQAHAAH@Z                   ; mangled function name
?Func@@YAXQAHAAH@Z  PROC NEAR               ; start of Func
; parameter 1: 4 + esp                      ; a
; parameter 2: 8 + esp                      ; r
$B1$1:                                      ; unused label
        mov       eax, DWORD PTR [esp+4]    ; eax = address of a
        mov       edx, DWORD PTR [esp+8]    ; edx = address in r
        mov       ecx, DWORD PTR [edx]      ; ecx = Induction
        lea       edx, DWORD PTR [eax+400]  ; edx = point to end of a
$B2$2:                                      ; top of loop
        mov       DWORD PTR [eax], ecx      ; a[i] = Induction;
        mov       DWORD PTR [eax+4], ecx    ; a[i+1] = Induction;
        add       ecx, 1                    ; Induction++;
        add       eax, 8                    ; point to a[i+2]
        cmp       edx, eax                  ; compare with end of array
        ja        $B2$2                     ; jump to top of loop
$B2$3:                                      ; unused label
        ret                                 ; return from Func
        ALIGN     4
; mark_end;
```

```
?Func2@@YAXQAHAAH@Z ENDP
```

This solution is clearly better. The loop body now contains only six instructions rather than nine, even though it is doing two iterations in one. The compiler has replaced `i` by a second induction variable (`eax`) which contains the address of the current array element. Rather than comparing `i` with `100` in the loop control it compares the array pointer `eax` to the address of the end of the array, which it has calculated in advance and stored in `edx`. Furthermore, this solution is using one register less so that it doesn't have to push and pop `ebx`.

9 Optimizing memory access

9.1 Caching of code and data

A cache is a proxy for the main memory in a computer. The proxy is smaller and closer to the CPU than the main memory and therefore it is accessed much faster. There may be two or three levels of cache for the sake of fastest possible access to the most used data.

The speed of CPUs is increasing faster than the speed of RAM memory. Efficient caching is therefore becoming more and more important.

9.2 Cache organization

It is useful to know how a cache is organized if you are making programs that have big data structures with non-sequential access and you want to prevent cache contention. You may skip this section if you are satisfied with more heuristic guidelines.

Most caches are organized into lines and sets. Let me explain this with an example. My example is a cache of 8 kb size with a line size of 64 bytes. Each line covers 64 consecutive bytes of memory. One kilobyte is 1024 bytes, so we can calculate that the number of lines is 8*1024/64 = 128. These lines are organized as 32 sets × 4 ways. This means that a particular memory address cannot be loaded into an arbitrary cache line. Only one of the 32 sets can be used, but any of the 4 lines in the set can be used. We can calculate which set of cache lines to use for a particular memory address by the formula: (*set*) = (*memory address*) / (*line size*) % (*number of sets*). Here, / means integer division with truncation, and % means modulo. For example, if we want to read from memory address a = 10000, then we have (*set*) = (10000 / 64) % 32 = 28. This means that a must be read into one of the four cache lines in set number 28. The calculation becomes easier if we use hexadecimal numbers because all the numbers are powers of 2. Using hexadecimal numbers, we have a = 0x2710 and (*set*) = (0x2710 / 0x40) % 0x20 = 0x1C. Reading or writing a variable from address 0x2710 will cause the cache to load the entire 64 or 0x40 bytes from address 0x2700 to 0x273F into one of the four cache lines from set 0x1C. If the program afterwards reads or writes to any other address in this range then the value is already in the cache so we don't have to wait for another memory access.

Assume that a program reads from address 0x2710 and later reads from addresses 0x2F00, 0x3700, 0x3F00 and 0x4700. These addresses all belong to set number 0x1C. There are only four cache lines in each set. If the cache always chooses the least recently used cache line then the line that covered the address range from 0x2700 to 0x273F will be evicted when we read from 0x4700. Reading again from address 0x2710 will cause a cache miss. But if the program had read from different addresses with different set values then the line containing the address range from 0x2700 to 0x273F would still be in the cache. The problem only occurs because the addresses are spaced a multiple of 0x800 apart. I will call this distance the critical stride. Variables whose distance in memory is a multiple of the

critical stride will contend for the same cache lines. The critical stride can be calculated as (*critical stride*) = (*number of sets*) × (*line size*) = (*total cache size*) / (*number of ways*).

If a program contains many variables and objects that are scattered around in memory then there is a risk that several variables happen to be spaced by a multiple of the critical stride and cause contentions in the data cache. The same can happen in the code cache if there are many functions scattered around in program memory. If several functions that are used in the same part of the program happen to be spaced by a multiple of the critical stride then this can cause contentions in the code cache. The subsequent sections describe various ways to avoid these problems.

More details about how caches work can be found in Wikipedia under CPU cache (en.wikipedia.org/wiki/L2_cache).

The details of cache organization for different processors are covered in manual 3: "The microarchitecture of Intel, AMD and VIA CPUs".

9.3 Functions that are used together should be stored together

The code cache works most efficiently if functions that are used near each other are also stored near each other in the code memory. The functions are usually stored in the order in which they appear in the source code. It is therefore a good idea to collect the functions that are used in the most critical part of the code together near each other in the same source file. Keep often used functions separate from seldom used functions, and put seldom used branches such as error handling in the end of a function or in a separate function.

Sometimes, functions are kept in different source files for the sake of modularity. For example, it may be convenient to have the member functions of a parent class in one source file and the derived class in another source file. If the member functions of parent class and derived class are called from the same critical part of the program then it can be advantageous to keep the two modules contiguous in program memory. This can be done by controlling the order in which the modules are linked together. The link order is usually the order in which the modules appear in the project window or makefile. You can check the order of functions in memory by requesting a map file from the linker. The map file tells the address of each function relative to the beginning of the program. The map file includes the addresses of library functions linked from static libraries (`.lib` or `.a`), but not dynamic libraries (`.dll` or `.so`). There is no easy way to control the addresses of dynamically linked library functions.

9.4 Variables that are used together should be stored together

Cache misses are very expensive. A variable can be fetched from the cache in just a few clock cycles, but it can take more than a hundred clock cycles to fetch the variable from RAM memory if it is not in the cache.

The cache works most efficiently if pieces of data that are used together are stored near each other in memory. Variables and objects should preferably be declared in the function in which they are used. Such variables and objects will be stored on the stack, which is very likely to be in the level-1 cache. The different kinds of variable storage are explained on page 26. Avoid global and static variables if possible, and avoid dynamic memory allocation (`new` and `delete`).

Object oriented programming can be an efficient way of keeping data together. Data members of a class (also called properties) are always stored together in an object of the class. Data members of a parent class and a derived class are stored together in an object of the derived class (see page 52).

The order in which data are stored can be important if you have big data structures. For example, if a program has two arrays, `a` and `b`, and the elements are accessed in the order `a[0]`, `b[0]`, `a[1]`, `b[1]`, ... then you may improve the performance by organizing the data as an array of structures:

```
// Example 9.1a
int Func(int);
const int size = 1024;
int a[size], b[size], i;
...
for (i = 0; i < size; i++) {
   b[i] = Func(a[i]);
}
```

The data in this example can be accessed sequentially in memory if organized as follows:

```
// Example 9.1b
int Func(int);
const int size = 1024;
struct Sab {int a; int b;};
Sab ab[size];
int i;
...
for (i = 0; i < size; i++) {
   ab[i].b = Func(ab[i].a);
}
```

There will be no extra overhead in the program code for making the structure in example 9.1b. On the contrary, the code becomes simpler because it needs only calculate element addresses for one array rather than two.

Some compilers will use different memory spaces for different arrays even if they are never used at the same time. Example:

```
// Example 9.2a
void F1(int   x[]);
void F2(float x[]);

void F3(bool y) {
   if (y) {
      int a[1000];
      F1(a);
   }
   else {
      float b[1000];
      F2(b);
   }
}
```

Here it is possible to use the same memory area for `a` and `b` because their live ranges do not overlap. You can save a lot of cache space by joining `a` and `b` in a union:

```
// Example 9.2b
void F3(bool y) {
   union {
      int   a[1000];
      float b[1000];
   };
   if (y) {
      F1(a);
   }
   else {
```

```
            F2(b);
        }
    }
```

Using a union is not a safe programming practice, of course, because you will get no warning from the compiler if the uses of `a` and `b` overlap. You should use this method only for big objects that take a lot of cache space. Putting simple variables into a union is not optimal because it prevents the use of register variables.

9.5 Alignment of data

A variable is accessed most efficiently if it is stored at a memory address which is divisible by the size of the variable. For example, a `double` takes 8 bytes of storage space. It should therefore preferably be stored at an address divisible by 8. The size should always be a power of 2. Objects bigger than 16 bytes should be stored at an address divisible by 16. You can generally assume that the compiler takes care of this alignment automatically.

The alignment of structure and class members may cause a waste of cache space, as explained in example 7.39 page 53.

You may choose to align large objects and arrays by the cache line size, which is typically 64 bytes. This makes sure that the beginning of the object or array coincides with the beginning of a cache line. Some compilers will align large static arrays automatically but you may as well specify the alignment explicitly by writing:

```
__declspec(align(64)) int BigArray[1024]; // Windows syntax
```

or

```
int BigArray[1024] __attribute__((aligned(64))); // Linux syntax
```

See page 97 and 123 for discussion of aligning dynamically allocated memory.

9.6 Dynamic memory allocation

Objects and arrays can be allocated dynamically with `new` and `delete`, or `malloc` and `free`. This can be useful when the amount of memory required is not known at compile time. Four typical uses of dynamic memory allocation can be mentioned here:

- A large array can be allocated dynamically when the size of the array is not known at compile time.
- A variable number of objects can be allocated dynamically when the total number of objects is not known at compile time.
- Text strings and similar objects of variable size can be allocated dynamically.
- Arrays that are too large for the stack can be allocated dynamically.

The advantages of dynamic memory allocation are:

- Gives a more clear program structure in some cases.
- Does not allocate more space than needed. This makes data caching more efficient than when a fixed-size array is made very big in order to cover the worst case situation of the maximum possible memory requirement.

- Useful when no reasonable upper limit to the required amount of memory space can be given in advance.

The disadvantages of dynamic memory allocation are:

- The process of dynamic allocation and deallocation of memory takes much more time than other kinds of storage. See page 26.

- The heap space becomes fragmented when objects of different sizes are allocated and deallocated in random order. This makes data caching inefficient.

- An allocated array may need to be resized in the event that it becomes full. This may require that a new bigger memory block is allocated and the entire contents copied to the new block. Any pointers to data in the old block then become invalid.

- The heap manager will start garbage collection when the heap space has become too fragmented. This garbage collection may start at unpredictable times and cause delays in the program flow at inconvenient times when a user is waiting for response.

- It is the responsibility of the programmer to make sure that everything that has been allocated is also deallocated. Failure to do so will cause the heap to be filled up. This is a common programming error known as memory leaks.

- It is the responsibility of the programmer to make sure that no object is accessed after it has been deallocated. Failure to do so is also a common programming error.

- The allocated memory may not be optimally aligned. See page 123 for how to align dynamically allocated memory.

- It is difficult for the compiler to optimize code that uses pointers because it cannot rule out aliasing (see page 79).

- A matrix or multidimensional array is less efficient when the row length is not known at compile time because of the extra work needed for calculating row addresses at each access. The compiler may not be able to optimize this with induction variables.

It is important to weigh the advantages over the disadvantages when deciding whether to use dynamic memory allocation. There is no reason to use dynamic memory allocation when the size of an array or the number of objects is known at compile time or a reasonable upper limit can be defined.

The cost of dynamic memory allocation is negligible when the number of allocations is limited. Dynamic memory allocation can therefore be advantageous when a program has one or a few arrays of variable size. The alternative solution of making the arrays very big to cover the worst case situation is a waste of cache space. A situation where a program has several large arrays and where the size of each array is a multiple of the critical stride (see above, page 89) is likely to cause contentions in the data cache.

If the number of elements in an array grows during program execution then it is preferable to allocate the final array size right from the beginning rather than allocating more space step by step. In most systems, you cannot increase the size of a memory block that has already been allocated. If the final size cannot be predicted or if the prediction turns out to be too small, then it is necessary to allocate a new bigger memory block and copy the contents of the old memory block into the beginning of the new bigger memory block. This is inefficient, of course, and causes the heap space to become fragmented. An alternative is to keep multiple memory blocks, either in the form of a linked list or with an index of memory

blocks. A method with multiple memory blocks makes the access to individual array elements more complicated and time consuming.

A collection of a variable number of objects is often implemented as a linked list. Each element in a linked list has its own memory block and a pointer to the next block. A linked list is less efficient than a linear array for the following reasons:

- Each object is allocated separately. The allocation, deallocation and garbage collection takes a considerable amount of time.
- The objects are not stored contiguously in the memory. This makes data caching less efficient.
- Extra memory space is used for the link pointers and for information stored by the heap manager for each allocated block.
- Walking through a linked list takes more time than looping through a linear array. No link pointer can be loaded until the previous link pointer has been loaded. This makes a critical dependency chain which prevents out-of-order execution.

It is often more efficient to allocate one big block of memory for all the objects (memory pooling) than to allocate a small block for each object.

A little-known alternative to using `new` and `delete` is to allocate variable-size arrays with `alloca`. This is a function that allocates memory on the stack rather than the heap. The space is automatically deallocated when returning from the function in which `alloca` was called. There is no need to deallocate the space explicitly when `alloca` is used. The advantages of `alloca` over `new` and `delete` or `malloc` and `free` are:

- There is very little overhead to the allocation process because the microprocessor has hardware support for the stack.
- The memory space never becomes fragmented thanks to the first-in-last-out nature of the stack.
- Deallocation has no cost because it goes automatically when the function returns. There is no need for garbage collection.
- The allocated memory is contiguous with other objects on the stack, which makes data caching very efficient.

The following example shows how to make a variable-size array with `alloca`:

```
// Example 9.3
#include <malloc.h>

void SomeFunction (int n) {
   if (n > 0) {
      // Make dynamic array of n floats:
      float * DynamicArray = (float *)alloca(n * sizeof(float));
      // (Some compilers use the name _alloca)
      for (int i = 0; i < n; i++) {
         DynamicArray[i] = WhateverFunction(i);
         // ...
      }
   }
}
```

Obviously, a function should never return any pointer or reference to anything it has allocated with `alloca`, because it is deallocated when the function returns. `alloca` may not be compatible with structured exception handling. See the manual for your compiler for restrictions on using `alloca`.

The C99 extension supports variable-size arrays. This feature is controversial and is only available in C, not in C++. You may use `alloca` instead of variable-size arrays since it provides the same functionality.

9.7 Container classes

Whenever dynamic memory allocation is used, it is recommended to wrap the allocated memory into a container class. The container class must have a destructor to make sure everything that is allocated is also de-allocated. This is the best way to prevent memory leaks and other common programming errors associated with dynamic memory allocation.

Container classes can also be convenient for adding bounds-checking to an array and for more advanced data structures with First-In-First-Out or First-In-Last-Out access, sort and search facilities, binary trees, hash maps etc.

It is common to make container classes in the form of templates where the type of objects they contain is provided as a template parameter. There is no performance cost to using templates.

Ready made container class templates are available for many different purposes. The most commonly used set of containers is the Standard Template Library (STL) which comes with most modern C++ compilers. The advantage of using ready made containers is that you don't have to reinvent the wheel. The containers in the STL are universal, flexible, well tested, and very useful for many different purposes.

However, the STL is designed for generality and flexibility, while execution speed, memory economy, cache efficiency and code size have got low priority. Especially the memory allocation is unnecessarily wasteful in the STL. Some STL templates, such as `list`, `set` and `map` are prone to even allocate more memory blocks than there are objects in the container. STL `deque` (doubly ended queue) allocates one memory block for every four objects. STL `vector` stores all the objects in the same memory block, but this memory block is re-allocated every time it is filled up, which happens quite often because the block size grows by only 50% or less each time. An experiment where 10 elements were inserted, one by one, into an STL `vector` turned up to cause seven memory allocations of sizes 1, 2, 3, 4, 6, 9 and 13 objects, respectively (MS Visual Studio 2008 version). This wasteful behavior can be prevented by calling `vector::reserve` with a prediction or estimate of the final size needed before adding the first object to the `vector`. The other STL containers do not have such a feature for reserving memory in advance.

The frequent allocation and de-allocation of memory with `new` and `delete` (or `malloc` and `free`) causes the memory to become fragmented and caching becomes inefficient. There is a large overhead cost to memory management and garbage collection, as mentioned above.

The generality of the STL also costs in terms of code size. In fact, the STL has been criticized for code bloat and complexity (en.wikipedia.org/wiki/Standard_Template_Library). The objects stored in an STL container are allowed to have constructors and destructors. The copy constructors and destructors of each object are called every time an object is moved, which may happen quite often. This is necessary if the objects stored are containers themselves. But implementing a matrix in STL as a `vector` of `vector`s, as is often seen, is certainly a very inefficient solution.

Many containers use linked lists. A linked list is a convenient way of making the container expandable, but it is very inefficient. Linear arrays are faster than linked lists in most cases.

The so-called iterators that are used in STL for accessing container elements are cumbersome to use for many programmers and they are not necessary if you can use a linear list with a simple index. A good compiler can optimize away the extra overhead of the iterator in some cases, but not all.

Fortunately, there are more efficient alternatives that can be used where execution speed, memory economy and small code size has higher priority than code generality. The most important remedy is memory pooling. It is more efficient to store many objects together in one big memory block than to store each object in its own allocated memory block. A large block containing many objects can be copied or moved with a single call to `memcpy` rather than moving each object separately if there are no copy constructors and destructors to call.

I have implemented a collection of example container classes that use these methods to improve efficiency. These are available as an appendix to this manual at www.agner.org/optimize/cppexamples.zip containing container classes and templates for several different purposes. All these examples are optimized for execution speed and for minimizing memory fragmentation. Bounds checking is included for the sake of security, but may be removed after debugging if required for performance reasons. Use these example containers in cases where the performance of the STL is not satisfactory.

The following considerations should be taken into account when choosing a container for a specific purpose:

- Contain one or multiple elements? If the container is to hold only one element then use a smart pointer (see page 38).

- Is the size known at compile time? If the number of elements is known at compile time or a not-too-big upper limit can be set then the optimal solution is a fixed size array or container without dynamic memory allocation. Dynamic memory allocation may be needed, however, if the array or container is too big for the stack.

- Is the size known before the first element is stored? If the total number of elements to store is known before the first element is stored (or if a reasonable estimate can be made) then it is preferred to use a container that allows you to reserve the amount of memory needed in advance rather than allocating piecewise or re-allocating when a memory block turns out to be too small.

- Are objects numbered consecutively? If objects are identified by consecutive indices or by keys within a limited range then a simple array is the most efficient solution.

- Is a multidimensional structure needed? A matrix or multidimensional array should be stored in one contiguous memory block. Do not use one container for each row or column. The access is faster if the number of elements per row is a constant known at compile time.

- Are objects accessed in a FIFO manner? If objects are accessed on a First-In-First-Out (FIFO) basis then use a queue. It is more efficient to implement a queue as a circular buffer than as a linked list.

- Are objects accessed in a FILO manner? If objects are accessed on a First-In-Last-Out (FILO) basis then use a linear array with a top-of-stack index.

- Are objects identified by a key? If the key values are confined to a narrow range then a simple array can be used. If the number of objects is high then the most efficient

solution may be a binary tree or a hash map.

- Do objects have a natural ordering? If you need to do searches of the kind: "what is the nearest element to x?" or "how many elements are there between x and y?" then you may use a sorted list or a binary tree.

- Is searching needed after all objects have been added? If search facilities are needed, but only after all objects have been stored in the container, then a linear array will be an efficient solution. Sort the array after all elements have been added and then use binary search for finding elements. A hash map may also be an efficient solution.

- Is searching needed before all objects have been added? If search facilities are needed, and new objects can be added at any time, then the solution is more complicated. If the total number of elements is small then a sorted list is the most efficient solution because of its simplicity. But a sorted list can be very inefficient if the list is large because the insertion of a new element in the list causes all subsequent elements in the sequence to be moved. A binary tree or a hash map is needed in this case. A binary tree may be used if elements have a natural order and there are search requests for elements in a specific interval. A hash map can be used if elements have no specific order but are identified by a unique key.

- Do objects have mixed types or sizes? It is possible to store objects of different types or strings of different lengths in the same memory pool. See www.agner.org/optimize/cppexamples.zip. If the number and types of elements is known at compile time then there is no need to use a container or memory pool.

- Alignment? Some applications require that data are aligned at round addresses. Especially the use of intrinsic vectors requires alignment to addresses divisible by 16. Alignment of data structures to addresses divisible by the cache line size (typically 64) can improve performance in some cases.

- Multiple threads? Container classes are generally not thread safe if multiple threads can add, remove or modify objects simultaneously. In multithreaded applications it is much more efficient to have separate containers for each thread than to temporarily lock a container for exclusive access by each thread.

- Pointers to contained objects? It may not be safe to make a pointer to a contained object because the container may move the object in case memory re-allocation is needed. Objects inside containers should be identified by their index or key in the container rather than by pointers or references. It is OK, however, to pass a pointer or reference to such an object to a function that doesn't add or remove any objects if no other threads have access to the container.

- Can the container be recycled? There is a large cost to creating and deleting containers. If the program logic allows it, it may be more efficient to re-use a container than to delete it and create a new one.

I have provided several examples of suitable containers class templates in www.agner.org/optimize/cppexamples.zip. These may be used as alternatives to the standard template library (STL) if the full generality and flexibility of the STL containers is not needed. You may write your own container classes or modify the ones that are available to fit specific needs.

9.8 Strings

Text strings typically have variable lengths that are not known at compile time. The storage of text strings in classes like `string`, `wstring` or `CString` uses `new` and `delete` to allocate a new memory block every time a string is created or modified. This can be quite inefficient if a program creates or modifies many strings.

In most cases, the fastest way to handle strings is the old fashioned C style with character arrays. Strings can be manipulated with C functions such as `strcpy`, `strcat`, `strlen`, `sprintf`, etc. But beware that these functions have no check for overflow of the arrays. An array overflow can cause unpredictable errors elsewhere in the program that are very difficult to diagnose. It is the responsibility of the programmer to make sure the arrays are sufficiently large to handle the strings including the terminating zero and to make overflow checks where necessary. Fast versions of common string functions as well as efficient functions for string searching and parsing are provided in the asmlib library at www.agner.org/optimize/asmlib.zip.

If you want to improve speed without jeopardizing safety, you may store all strings in a memory pool, as explained above. Examples are provided in an appendix to this manual at www.agner.org/optimize/cppexamples.zip.

9.9 Access data sequentially

A cache works most efficiently when the data are accessed sequentially. It works somewhat less efficiently when data are accessed backwards and much less efficiently when data are accessed in a random manner. This applies to reading as well as writing data.

Multidimensional arrays should be accessed with the last index changing in the innermost loop. This reflects the order in which the elements are stored in memory. Example:

```
// Example 9.4
const int NUMROWS = 100, NUMCOLUMNS = 100;
int matrix[NUMROWS][NUMCOLUMNS];
int row, column;
for (row = 0; row < NUMROWS; row++)
   for (column = 0; column < NUMCOLUMNS; column++)
      matrix[row][column] = row + column;
```

Do not swap the order of the two loops (except in Fortran where the storage order is opposite).

9.10 Cache contentions in large data structures

It is not always possible to access a multidimensional array sequentially. Some applications (e.g. in linear algebra) require other access patterns. This can cause severe delays if the distance between rows in a big matrix happen to be equal to the critical stride, as explained on page 89. This will happen if the size of a matrix line (in bytes) is a high power of 2.

The following example illustrates this. My example is a function which transposes a quadratic matrix, i.e. each element `matrix[r][c]` is swapped with element `matrix[c][r]`.

```
// Example 9.5a
const int SIZE = 64;          // number of rows/columns in matrix

void transpose(double a[SIZE][SIZE]) { // function to transpose matrix
   // define a macro to swap two array elements:
   #define swapd(x,y) {temp=x; x=y; y=temp;}
```

```
   int r, c;  double temp;
   for (r = 1; r < SIZE; r++) {          // loop through rows
      for (c = 0; c < r; c++) {          // loop columns below diagonal
         swapd(a[r][c], a[c][r]);        // swap elements
      }
   }
}

void test () {
   __declspec(__align(64))          // align by cache line size
   double matrix[SIZE][SIZE];       // define matrix
   transpose(matrix);               // call transpose function
}
```

Transposing a matrix is the same as reflecting it at the diagonal. Each element `matrix[r][c]` below the diagonal is swapped with element `matrix[c][r]` at its mirror position above the diagonal. The `c` loop in example 9.5a goes from the leftmost column to the diagonal. The elements at the diagonal remain unchanged.

The problem with this code is that if the elements `matrix[r][c]` below the diagonal are accessed row-wise, then the mirror elements `matrix[c][r]` above the diagonal are accessed column-wise.

Assume now that we are running this code with a 64×64 matrix on a Pentium 4 computer where the level-1 data cache is 8 kb = 8192 bytes, 4 ways, with a line size of 64. Each cache line can hold 8 `double`'s of 8 bytes each. The critical stride is 8192 / 4 = 2048 bytes = 4 rows.

Let's look at what happens inside the loop, for example when `r` = 28. We take the elements from row 28 below the diagonal and swap these elements with column 28 above the diagonal. The first eight elements in row 28 share the same cache line. But these eight elements will go into eight different cache lines in column 28 because the cache lines follow the rows, not the columns. Every fourth of these cache lines belong to the same set in the cache. When we reach element number 16 in column 28, the cache will evict the cache line that was used by element 0 in this column. Number 17 will evict number 1. Number 18 will evict number 2, etc. This means that all the cache lines we used above the diagonal have been lost at the time we are swapping column 29 with line 29. Each cache line has to be reloaded eight times because it is evicted before we need the next element. I have confirmed this by measuring the time it takes to transpose a matrix using example 9.5a on a Pentium 4 with different matrix sizes. The results of my experiment are given below. The time unit is clock cycles per array element.

Matrix size	Total kilobytes	Time per element
63×63	31	11.6
64×64	32	16.4
65×65	33	11.8
127×127	126	12.2
128×128	128	17.4
129×129	130	14.4
511×511	2040	38.7
512×512	2048	230.7
513×513	2056	38.1

Table 9.1. Time for transposition of different size matrices, clock cycles per element.

The table shows that it takes 40% more time to transpose the matrix when the size of the matrix is a multiple of the level-1 cache size. This is because the critical stride is a multiple of the size of a matrix line. The delay is less than the time it takes to reload the level-1 cache from the level-2 cache because the out-of-order execution mechanism can prefetch the data.

The effect is much more dramatic when contentions occur in the level-2 cache. The level-2 cache is 512 kb, 8 ways. The critical stride for the level-2 cache is 512 kb / 8 = 64 kb. This corresponds to 16 lines in a 512×512 matrix. My experimental results in table 9.1 show that it takes six times as long time to transpose a matrix when contentions occur in the level-2 cache as when contentions do not occur. The reason why this effect is so much stronger for level-2 cache contentions than for level-1 cache contentions is that the level-2 cache cannot prefetch more than one line at a time.

A simple way of solving the problem is to make the rows in the matrix longer than needed in order to avoid that the critical stride is a multiple of the matrix line size. I tried to make the matrix 512×520 and leave the last 8 columns unused. This removed the contentions and the time consumption was down to 36.

There may be cases where it is not possible to add unused columns to a matrix. For example, a library of math functions should work efficiently on all sizes of matrices. An efficient solution in this case is to divide the matrix into smaller squares and handle one square at a time. This is called square blocking or tiling. This technique is illustrated in example 9.5b.

```
// Example 9.5b
void transpose(double a[SIZE][SIZE]) {
   // Define macro to swap two elements:
   #define swapd(x,y) {temp=x; x=y; y=temp;}
   // Check if level-2 cache contentions will occur:
   if (SIZE > 256 && SIZE % 128 == 0) {
      // Cache contentions expected. Use square blocking:
      int r1, r2, c1, c2; double temp;
      // Define size of squares:
      const int TILESIZE = 8;   // SIZE must be divisible by TILESIZE
      // Loop r1 and c1 for all squares:
      for (r1 = 0; r1 < SIZE; r1 += TILESIZE) {
         for (c1 = 0; c1 < r1; c1 += TILESIZE) {
            // Loop r2 and c2 for elements inside sqaure:
            for (r2 = r1; r2 < r1+TILESIZE; r2++) {
               for (c2 = c1; c2 < c1+TILESIZE; c2++) {
                  swapd(a[r2][c2],a[c2][r2]);
               }
            }
         }
         // At the diagonal there is only half a square.
         // This triangle is handled separately:
         for (r2 = r1+1; r2 < r1+TILESIZE; r2++) {
            for (c2 = r1; c2 < r2; c2++) {
               swapd(a[r2][c2],a[c2][r2]);
            }
         }
      }
   }
   else {
      // No cache contentions. Use simple method.
      // This is the code from example 9.5a:
      int r, c;  double temp;
      for (r = 1; r < SIZE; r++) {     // loop through rows
         for (c = 0; c < r; c++) {     // loop columns below diagonal
            swapd(a[r][c], a[c][r]);  // swap elements
         }
```

```
            }
        }
    }
```

This code took 50 clock cycles per element for a 512×512 matrix in my experiments.

Contentions in the level-2 cache are so expensive that it is very important to do something about them. You should therefore be aware of situations where the number of columns in a matrix is a high power of 2. Contentions in the level-1 cache are less expensive. Using complicated techniques like square blocking for the level-1 cache may not be worth the effort.

Square blocking and similar methods are further described in the book "Performance Optimization of Numerically Intensive Codes", by S. Goedecker and A. Hoisie, SIAM 2001.

9.11 Explicit cache control

Microprocessors with the SSE and SSE2 instruction sets have certain instructions that allow you to manipulate the data cache. These instructions are accessible from compilers that have support for intrinsic functions (i.e. Microsoft, Intel and Gnu). Other compilers need assembly code to access these instructions.

Function	Assembly name	Intrinsic function name	Instruction set
Prefetch	PREFETCH	_mm_prefetch	SSE
Store 4 bytes without cache	MOVNTI	_mm_stream_si32	SSE2
Store 8 bytes without cache	MOVNTQ	_mm_stream_pi	SSE
Store 16 bytes without cache	MOVNTPS	_mm_stream_ps	SSE
Store 16 bytes without cache	MOVNTPD	_mm_stream_pd	SSE2
Store 16 bytes without cache	MOVNTDQ	_mm_stream_si128	SSE2

Table 9.2. Cache control instructions.

There are other cache control instructions than the ones mentioned in table 9.2, such as flush and fence instructions, but these are hardly relevant to optimization.

Prefetching data

The prefetch instruction can be used for fetching a cache line that we expect to use later in the program flow. However, this did not improve the execution speed in any of the examples I have tested. The reason is that modern processors prefetch data automatically thanks to out-of-order execution and advanced prediction mechanisms. Modern microprocessors are able to automatically prefetch data for regular access patterns containing multiple streams with different strides. Therefore, you don't have to prefetch data explicitly if data access can be arranged in regular patterns with fixed strides.

Uncached memory store

An uncached write is more expensive than an uncached read because the write causes an entire cache line to be read and written back.

The so-called nontemporal write instructions (MOVNT) are designed to solve this problem. These instructions write directly to memory without loading a cache line. This is advantageous in cases where we are writing to uncached memory and we do not expect to read from the same or a nearby address again before the cache line would be evicted. Don't mix nontemporal writes with normal writes or reads to the same memory area.

The nontemporal write instructions are not suitable for example 9.5 because we are reading and writing from the same address so a cache line will be loaded anyway. If we modify example 9.5 so that it writes only, then the effect of nontemporal write instructions becomes noticeable. The following example transposes a matrix and stores the result in a different array.

```
// Example 9.6a
const int SIZE = 512; // number of rows and columns in matrix

// function to transpose and copy matrix
void TransposeCopy(double a[SIZE][SIZE], double b[SIZE][SIZE]) {
   int r, c;
   for (r = 0; r < SIZE; r++) {
      for (c = 0; c < SIZE; c++) {
         a[c][r] = b[r][c];
      }
   }
}
```

This function writes to matrix `a` in a column-wise manner where the critical stride causes all writes to load a new cache line in both the level-1 and the level-2 cache. Using the nontemporal write instruction prevents the level-2 cache from loading any cache lines for matrix `a`:

```
// Example 9.6b.
#include "xmmintrin.h"  // header for intrinsic functions

// This function stores a double without loading a cache line:
static inline void StoreNTD(double * dest, double const & source) {
   _mm_stream_pi((__m64*)dest, *(__m64*)&source);  // MOVNTQ
   _mm_empty();                                    // EMMS
}

const int SIZE = 512; // number of rows and columns in matrix
// function to transpose and copy matrix
void TransposeCopy(double a[SIZE][SIZE], double b[SIZE][SIZE]) {
   int r, c;
   for (r = 0; r < SIZE; r++) {
      for (c = 0; c < SIZE; c++) {
         StoreNTD(&a[c][r], b[r][c]);
      }
   }
}
```

The execution times per matrix cell for different matrix sizes were measured on a Pentium 4 computer. The measured results were as follows:

Matrix size	Time per element Example 9.6a	Time per element Example 9.6b
64×64	14.0	80.8
65×65	13.6	80.9
512×512	378.7	168.5
513×513	58.7	168.3

Table 9.3. Time for transposing and copying different size matrices, clock cycles per element.

As table 9.3 shows, the method of storing data without caching is advantageous if, and only if, a level-2 cache miss can be expected. The 64×64 matrix size causes misses in the level-1 cache. This has hardly any effect on the total execution time because the cache miss on a

store operation doesn't delay the subsequent instructions. The 512×512 matrix size causes misses in the level-2 cache. This has a very dramatic effect on the execution time because the memory bus is saturated. This can be ameliorated by using nontemporal writes. If the cache contentions can be prevented in other ways, as explained in chapter 9.10, then the nontemporal write instructions are not optimal.

There are certain restrictions on using the instructions listed in table 9.2. All these instructions require that the microprocessor has the SSE or SSE2 instruction set, as listed in the table. The 16-byte instructions `MOVNTPS`, `MOVNTPD` and `MOVNTDQ` require that the operating system has support for XMM registers; see page 125.

The Intel compiler can insert nontemporal writes automatically in vectorized code when the `#pragma vector nontemporal` is used. However, this does not work in example 9.6b.

The `MOVNTQ` instruction must be followed by an `EMMS` instruction before any floating point instructions. This is coded as `_mm_empty()` as shown in example 9.6b. The `MOVNTQ` instruction cannot be used in 64-bit device drivers for Windows.

10 Multithreading

The clock frequency of the CPU is limited by physical factors. The way to increase the throughput of CPU-intensive programs when the clock frequency is limited is to do multiple things at the same time. There are three ways to do things in parallel:

- Using multiple CPUs or multi-core CPUs, as described in this chapter.
- Using the out-of-order capabilities of modern CPUs, as described in chapter 11.
- Using the vector operations of modern CPUs, as described in chapter 12.

Most modern CPUs have two or more cores, and it can be expected that the number of cores will grow in the future. To use multiple CPUs or CPU cores, we need to divide the work into multiple threads. There are two main principles here: *functional decomposition* and *data decomposition*. Functional decomposition here means that different threads are doing different kinds of jobs. For example, one tread can take care of the user interface, another thread can take care of communication with a remote database, and a third thread can do mathematical calculations. It is important that the user interface is not in the same thread as very time-consuming tasks because this would give annoyingly long and irregular response times. It is often useful to put time-consuming tasks into separate threads with low priority.

In many cases, however, there is a single task that consumes most of the resources. In this case we need to split up the data into multiple blocks in order to utilize the multiple processor cores. Each thread should then handle its own block of data. This is data decomposition.

It is important to distinguish between coarse-grained parallelism and fine-grained parallelism when deciding whether it is advantageous to do things in parallel. Coarse-grained parallelism refers to the situation where a long sequence of operations can be carried out independently of other tasks that are running in parallel. Fine-grained parallelism is the situation where a task is divided into many small subtasks, but it is impossible to work for very long on a particular subtask before coordination with other subtasks is necessary.

Multithreading works more efficiently with coarse-grained parallelism than with fine-grained parallelism because communication and synchronization between the different cores is slow. If the granularity is too fine then it is not advantageous to split the tasks into multiple

threads. Out-of-order execution (chapter 11) and vector operations (chapter 12) are more useful methods for exploiting fine-grained parallelism.

The way to use multiple CPU cores is to divide the work into multiple threads. The use of threads is discussed on page 61. In the case of data decomposition, we should preferably have no more threads with the same priority than the number of cores or logical processors available in the system. The number of logical processors available can be determined by a system call (e.g. `GetProcessAffinityMask` in Windows).

There are several ways to divide the workload between multiple CPU cores:

- Define multiple threads and put an equal amount of work into each thread. This method works with all compilers.

- Use automatic parallelization. The Gnu, Intel and PathScale compilers can automatically detect opportunities for parallelization in the code and divide it into multiple threads, but the compiler may not be able to find the optimal decomposition of the data.

- Use OpenMP directives. OpenMP is a standard for specifying parallel processing in C++ and Fortran. These directives are supported by Microsoft, Intel, PathScale and Gnu compilers. See www.openmp.org and the compiler manual for details.

- Use function libraries with internal multi-threading, e.g. Intel Math Kernel Library.

The multiple CPU cores or logical processors usually share the same cache, at least at the last cache level, and in some cases even the same level-1 cache. The advantage of sharing the same cache is that communication between threads becomes faster and that threads can share the same code and read-only data. The disadvantage is that the cache will be filled up if the threads use different memory areas, and there will be cache contentions if the threads write to the same memory areas.

Data that are read-only can be shared between multiple threads, while data that are modified should be separate for each thread. It is not good to have two or more threads writing to the same cache line, because the threads will invalidate each other's caches and cause large delays. The easiest way to make thread-specific data is to declare it locally in the thread function so that it is stored on the stack. Each thread has its own stack. Alternatively, you may define a structure or class for containing thread-specific data and make one instance for each thread. This structure or class should be aligned by at least the cache line size in order to avoid multiple threads writing to the same cache line. The cache line size is typically 64 bytes on contemporary processors. The cache line size may possibly be more (128 or 256 bytes) on future processors.

There are various methods for communication and synchronization between threads, such as semaphores, mutexes and message systems. All of these methods are time consuming. Therefore, the data and resources should be organized so that the amount of necessary communication between threads is minimized. For example, if multiple threads are sharing the same queue, list, database, or other data structure then you may consider if it is possible to give each thread its own data structure and then merge the multiple data structures in the end when all threads have finished the time-consuming data processing.

Running multiple threads on a system with only one logical processor is not an advantage if the threads are competing for the same resources. But it can be a good idea to put time-consuming calculations into a separate thread with lower priority than the user interface. It is also useful to put file access and network access in separate threads so that one thread can do calculations while another thread is waiting for response from a hard disk or network.

Various development tools for supporting multi-threaded software are available from Intel. See Intel Technology Journal Vol. 11, Iss. 4, 2007 (www.intel.com/technology/itj/).

10.1 Simultaneous multithreading

Many microprocessors are able to run two threads in each core. For example, a processor with four cores can run eight threads simultaneously. This processor has four physical processors but eight logical processors.

"Hyperthreading" is Intel's term for simultaneous multithreading. Two threads running in the same core will always compete for the same resources, such as cache and execution units. If any of the shared resources are limiting factors for the performance then there is no advantage to using simultaneous multithreading. On the contrary, each thread may run at less than half speed because of cache evictions and other resource conflicts. But if a large fraction of the time goes to cache misses, branch misprediction, or long dependency chains, then each thread will run at more than half the single-thread speed. In this case there is an advantage to using simultaneous multithreading, but the performance is not doubled. A thread that shares the resources of the core with another thread will always run slower than a thread that runs alone in the core.

It is often necessary to do experiments in order to determine whether it is advantageous to use simultaneous multithreading or not in a particular application.

If simultaneous multithreading is not advantageous then it is necessary to query certain operating system functions (e.g. `GetLogicalProcessorInformation` in Windows) to determine if the processor has simultaneous multithreading. If so, then you can avoid simultaneous multithreading by using only the even-numbered logical processors (0, 2, 4, etc.). Older operating systems lack the necessary functions for distinguishing between the number of physical processors and the number of logical processors.

There is no way to tell the processor to give higher priority to one thread than another. Therefore, it can often happen that a low-priority thread steals resources from a higher-priority thread running in the same core. It is the responsibility of the operating system to avoid running two threads with widely different priority in the same processor core. Unfortunately, contemporary operating are not able to handle this problem adequately.

The Intel compiler is capable of making two threads where one thread is used for prefetching data for the other thread. However, in most cases automatic hardware prefetching is more efficient than software prefetching.

11 Out of order execution

All modern x86 CPUs can execute instructions out of order or do more than one thing at the same time, except for some small low-power CPUs (Intel Atom). The following example shows how to take advantage of this capability:

```
// Example 11.1a
float a, b, c, d, y;
y = a + b + c + d;
```

This expression is calculated as `((a+b)+c)+d`. This is a dependency chain where each addition has to wait for the result of the preceding one. You can improve this by writing:

```
// Example 11.1b
float a, b, c, d, y;
y = (a + b) + (c + d);
```

Now the two parentheses can be calculated independently. The CPU will start to calculate `(c+d)` before it has finished the calculation of `(a+b)`. This can save several clock cycles. You cannot assume that an optimizing compiler will change the code in example 11.1a to 11.1b automatically, although it appears to be an obvious thing to do. The reason why compilers do not make this kind of optimizations on floating point expressions is that it may cause a loss of precision, as explained on page 74. You have to set the parentheses manually.

The effect of dependency chains is stronger when they are long. This is often the case in loops. Consider the following example, which calculates the sum of 100 numbers:

```
// Example 11.2a
const int size = 100;
float list[size], sum = 0;  int i;
for (i = 0; i < size; i++) sum += list[i];
```

This has a long dependency chain. If a floating point addition takes 5 clock cycles, then this loop will take approximately 500 clock cycles. You can improve the performance dramatically by unrolling the loop and splitting the dependency chain in two:

```
// Example 11.2b
const int size = 100;
float list[size], sum1 = 0, sum2 = 0;  int i;
for (i = 0; i < size; i += 2) {
   sum1 += list[i];
   sum2 += list[i+1];}
sum1 += sum2;
```

If the microprocessor is doing an addition to `sum1` from time T to T+5, then it can do another addition to `sum2` from time T+1 to T+6, and the whole loop will take only 256 clock cycles.

Calculations in a loop where each iteration needs the result of the preceding one is called a loop-carried dependency chain. Such dependency chains can be very long and very time-consuming. There is a lot to gain if such dependency chains can be broken up. The two summation variables `sum1` and `sum2` are called accumulators. Current CPUs have only one floating point addition unit, but this unit is pipelined, as explained above, so that it can start a new addition before the preceding addition is finished.

The optimal number of accumulators for floating point addition and multiplication may be three or four, depending on the CPU.

Unrolling a loop becomes a little more complicated if the number of iterations is not divisible by the unroll factor. For example, if the number of elements in `list` in example 11.2b was an odd number then we would have to add the last element outside the loop or add an extra dummy element to `list` and make this extra element zero.

It is not necessary to unroll a loop and use multiple accumulators if there is no loop-carried dependency chain. A microprocessor with out-of-order capabilities can overlap the iterations and start the calculation of one iteration before the preceding iteration is finished. Example:

```
// Example 11.3
const int size = 100;  int i;
float a[size], b[size], c[size];
float register temp;
for (i = 0; i < size; i++) {
   temp = a[i] + b[i];
   c[i] = temp * temp;
}
```

Microprocessors with out-of-order capabilities are very smart. They can detect that the value of register `temp` in one iteration of the loop in example 11.3 is independent of the value in the previous iteration. This allows it to begin calculating a new value of `temp` before it is finished using the previous value. It does this by assigning a new physical register to `temp` even though the logical register that appears in the machine code is the same. This is called register renaming. The CPU can hold many renamed instances of the same logical register.

This advantage comes automatically. There is no reason to unroll the loop and have a `temp1` and `temp2`. Modern CPUs are capable of register renaming and doing multiple calculations in parallel if certain conditions are satisfied. The conditions that make it possible for the CPU to overlap the calculations of loop iterations are:

- No loop-carried dependency chain. Nothing in the calculation of one iteration should depend on the result of the previous iteration (except for the loop counter, which is calculated fast if it is an integer).

- All intermediate results should be saved in registers, not in memory. The renaming mechanism works only on registers, not on variables in memory or cache. Most compilers will make `temp` a register variable in example 11.3 even without the `register` keyword. The CodeGear compiler cannot make floating point register variables, but will save `temp` in memory. This prevents the CPU from overlapping calculations.

- The loop branch should be predicted. This is no problem if the repeat count is large or constant. If the loop count is small and changing then the CPU may occasionally predict that the loop exits, when in fact it does not, and therefore fail to start the next calculation. However, the out-of-order mechanism allows the CPU to increment the loop counter ahead of time so that it may detect the misprediction before it is too late. You should therefore not be too worried about this condition.

In general, the out-of-order execution mechanism works automatically. However, there are a couple of things that the programmer can do to take maximum advantage of out-of-order execution. The most important thing is to avoid long dependency chains. Another thing that you can do is to mix different kinds of operations in order to divide the work evenly between the different execution units in the CPU. It can be advantageous to mix integer and floating point calculations as long as you don't need conversions between integers and floating point numbers. It can also be advantageous to mix floating point addition with floating point multiplication, to mix simple integer with vector integer operations, and to mix mathematical calculations with memory access.

Very long dependency chains put a strain on the out-of-order resources of the CPU, even if they do not carry into the next iteration of a loop. A modern CPU can typically handle more than a hundred pending operations (see manual 3: "The microarchitecture of Intel, AMD and VIA CPUs"). It may be useful to split up a loop and store intermediate results in order to break an extremely long dependency chain.

12 Using vector operations

Today's microprocessors have vector instructions that make it possible to do operations on all elements of a vector simultaneously. This is also called Single-Instruction-Multiple-Data (SIMD) operations. The total size of each vector can be 64 bits (MMX), 128 bits (XMM), 256 bits (YMM), and 512 bits (ZMM).

Vector operations are useful when doing calculations on large data sets where the same operation is performed on multiple data elements and the program logic allows parallel calculations. Examples are image processing, sound processing, and mathematical operations on vectors and matrixes. Algorithms that are inherently serial, such as most sorting algorithms, are not well suited for vector operations. Algorithms that rely heavily on table lookup or require a lot of data shuffling, such as many encryption algorithms, are perhaps less suited for vector operations.

The vector operations use a set of special vector registers. The maximum size of each vector register is 128 bits (XMM) if the SSE2 instruction set is available, 256 bits (YMM) if the AVX instruction set is supported by the microprocessor and the operating system, and 512 bits when the AVX512 instruction set is available. The number of elements in each vector depends on the size and type of data elements, as follows:

Type of elements	Size of each element, bits	Number of elements	Total size of vector, bits	Instruction set
`char`	8	8	64	MMX
`short int`	16	4	64	MMX
`int`	32	2	64	MMX
`int64_t`	64	1	64	MMX
`char`	8	16	128	SSE2
`short int`	16	8	128	SSE2
`int`	32	4	128	SSE2
`int64_t`	64	2	128	SSE2
`float`	32	4	128	SSE
`double`	64	2	128	SSE2
`char`	8	32	256	AVX2
`short int`	16	16	256	AVX2
`int`	32	8	256	AVX2
`int64_t`	64	4	256	AVX2
`float`	32	8	256	AVX
`double`	64	4	256	AVX
`char`	8	64	512	AVX512BW
`short int`	16	32	512	AVX512BW
`int`	32	16	512	AVX512
`int64_t`	64	8	512	AVX512
`float`	32	16	512	AVX512
`double`	64	8	512	AVX512

Table 12.1. Vector sizes available in different instruction set extensions

For example, a 128-bit XMM register can be organized as a vector of eight 16-bit integers or four `float`'s when the SSE2 instruction set is available. The older MMX registers, which are 64 bits wide, should be avoided because they cannot be mixed with x87 style floating point code.

The 128-bit XMM vectors must be aligned by 16, i.e. stored at a memory address that is divisible by 16 (see below). The 256-bit YMM vectors are preferably aligned by 32 and the 512-bit ZMM registers by 64, but the alignment requirements are less strict when compiling for the AVX and later instruction sets.

Vector operations are particularly fast on newer processors. Many processors can calculate a vector just as fast as a scalar (Scalar means a single element). The first generation of processors that support a new vector size often have execution units, memory ports, etc. of only half the size of the largest vector. These units are used twice for handling a full size vector.

The use of vector operations is more advantageous the smaller the data elements are. For example, you get four `float` additions in the same time that it takes to do two additions with `double`'s. It is almost always advantageous to use vector operations on contemporary CPUs if the data fit nicely into the vector registers. It may not be advantageous if a lot of data manipulation is required for putting the right data into the right vector elements.

12.1 AVX instruction set and YMM registers

The 128-bit XMM registers are extended to 256-bit registers named YMM in the AVX instruction set. The main advantage of the AVX instruction set is that it allows larger floating point vectors. There are also other advantages that may improve the performance somewhat. The AVX2 instruction set also allows 256-bit integer vectors.

Code that is compiled for the AVX instruction set can run only if AVX is supported by both the CPU and the operating system. AVX is supported in Windows 7 and Windows Server 2008 R2 as well as in Linux kernel version 2.6.30 and later. The AVX instruction set is supported in the latest compilers from Microsoft, Intel, Gnu and Clang.

There is a problem when mixing code compiled with and without AVX support on some Intel processors. There is a performance penalty when going from AVX code to non-AVX code because of a change in the YMM register state. This penalty should be avoided by calling the intrinsic function `_mm256_zeroupper()` before any transition from AVX code to non-AVX code. This can be necessary in the following cases:

- If part of a program is compiled with AVX support and another part of the program is compiled without AVX support then call `_mm256_zeroupper()` before leaving the AVX part.

- If a function is compiled in multiple versions with and without AVX using CPU dispatching then call `_mm256_zeroupper()` before leaving the AVX part.

- If a piece of code compiled with AVX support calls a function in a library other than the library that comes with the compiler, and the library has no AVX support, then call `_mm256_zeroupper()` before calling the library function.

12.2 AVX512 instruction set and ZMM registers

The 256-bit YMM registers are extended to 512-bit registers named ZMM in the AVX512 instruction set. The number of vector registers is extended from 16 to 32 in 64 bit mode. There are only 8 vector registers in 32-bit mode. Therefore, AVX512 code should preferably be compiled for 64-bit mode.

The AVX512 instruction set also adds a set of mask registers. These are used as Boolean vectors. Almost any vector instruction can be masked with a mask register so that each vector element is calculated only if the corresponding bit in the mask register is 1. This makes vectorization of code with branches more efficient.

There are several additional extensions to AVX512. All processors with AVX512 have some of these extensions, but no processor so far has them all (writing in 2016). The known and planned extensions to AVX512 are the following:

- AVX512F. Foundation. All AVX512 processors have this. Includes operations on 32-bit and 64-bit integers, float and double in 512-bit vectors, including masked operations.
- AVX512VL. Includes the same operations on 128-bit and 256-bit vectors, including masked operations and 32 vector registers.
- AVX512BW. Operations on 8-bit and 16-bit integers in 512-bit vectors.
- AVX512DQ. Multiplication and conversion instructions with 64-bit integers. Various other instructions on float and double.
- AVX512ER. Fast reciprocal, reciprocal square root, and exponential function. Precise on float; approximate on double.
- AVX512CD. Conflict detection. Find duplicate elements in a vector.
- AVX512PF. Prefetch instructions with gather/scatter logic.
- AVX512VBMI. Permutation and shift with 8-bit granularity.
- AVX512IFMA. Fused multiply-and-add on 52-bit integers.
- AVX512_4VNNIW. Iterated dot product on 16-bit integers.
- AVX512_4FMAPS. Iterated fused multiply-and-add, single precision.

This makes CPU dispatching more complicated. You may pick the extensions that are useful for a particular task and make a code branch for processors that have this extension.

The use of `_mm256_zeroupper()` is less important in AVX512 code, but still recommended. See manual 2: "Optimizing subroutines in assembly language" chapter 13.2 and manual 5: "Calling conventions" chapter 6.3.

12.3 Automatic vectorization

Good compilers such as the Gnu, Clang and Intel compilers can use vector operations automatically in cases where the parallelism is obvious. See the compiler documentation for detailed instructions. Example:

```
// Example 12.1a. Automatic vectorization
const int size = 1024;
int a[size], b[size];
// ...
for (int i = 0; i < size; i++) {
   a[i] = b[i] + 2;
}
```

A good compiler will optimize this loop by using vector operations when the SSE2 or later instruction set is specified. The code will read four, eight, or sixteen elements of `b` into a vector register depending on the instruction set, and do an addition with another vector register containing (2,2,2,...), and store the four results in `a`. This operation will then be repeated as many times as the array size divided by the number of elements per vector. The speed is improved accordingly. It is best when the loop count is divisible by the number of elements per vector. You may even add dummy elements at the end of the array to make the array size a multiple of the vector size.

There is a disadvantage when the arrays are accessed through pointers, e.g.:

```
// Example 12.1b. Vectorization with alignment problem
void AddTwo(int * __restrict aa, int * __restrict bb) {
   for (int i = 0; i < size; i++) {
      aa[i] = bb[i] + 2;
   }
}
```

The performance is best if the arrays are aligned by the vector size, i.e. 16, 32 or 64 for XMM, YMM, and ZMM registers respectively. Efficient vector operations under instruction

sets prior to AVX require that the arrays are aligned by addresses divisible by 16. In example 12.1a, the compiler can align the arrays as required, but in example 12.1b, the compiler cannot know for sure whether the arrays are properly aligned or not. The loop can still be vectorized, but the code will be less efficient because the compiler has to take extra precautions to account for unaligned arrays. There are various things you can do to make the code more efficient when arrays are accessed through pointers or references:

- If the Intel compiler is used, then use `#pragma vector aligned` or the `__assume_aligned` directive to tell the compiler that the arrays are aligned, and make sure that they are.

- Declare the function `inline`. This may enable the compiler to reduce example 12.1b to 12.1a.

- Enable the instruction set with the largest vector size if possible. The AVX and later instruction sets have very few restrictions on alignment and the resultant code will be efficient whether the arrays are aligned or not.

The automatic vectorization works best if the following conditions are satisfied:

1. Use a compiler that supports automatic vectorization, such as Gnu, Clang, Intel or PathScale.

2. Use the latest version of the compiler. The compilers are becoming better and better at vectorization.

3. Use appropriate compiler options to enable the desired instruction set (`/arch:SSE2`, `/arch:AVX` etc. for Windows, `-msse2`, `-mavx`, etc. for Linux)

4. Use the less restrictive floating point options. For the Gnu compiler, use -O3 -fno-trapping-math -fno-math-errno.

5. Align arrays and big structures by 16 for SSE2, preferably 32 for AVX and preferably 64 for AVX512.

6. The loop count should preferably be a constant that is divisible by the number of elements in a vector.

7. If arrays are accessed through pointers so that the alignment is not visible in the scope of the function where you want vectorization then follow the advice given above.

8. If the arrays or structures are accessed through pointers or references then tell the compiler explicitly that pointers do not alias, if appropriate. See the compiler documentation for how to do this.

9. Minimize the use of branches at the vector element level.

10. Avoid table lookup at the vector element level.

You may look at the assembly output listing to see if the code is indeed vectorized as intended (see page 86).

The compiler can also use vector operations where there is no loop if the same operation is performed on a sequence of consecutive variables. Example:

```
// Example 12.2
__declspec(align(16))      // Make all instances of S1 aligned
```

```
struct S1 {                // Structure of 4 floats
   float a, b, c, d;
};

void Func() {
   S1 x, y;
   ...
   x.a = y.a + 1.;
   x.b = y.b + 2.;
   x.c = y.c + 3.;
   x.d = y.d + 4.;
};
```

A structure of four `float`'s fits into a 128-bit XMM register. In example 12.2, the optimized code will load the structure `y` into a vector register, add the constant vector `(1,2,3,4)`, and store the result in `x`.

The compiler is not always able to predict correctly whether vectorization will be advantageous or not. The Intel compiler allows you to use the `#pragma vector always` to tell the compiler to vectorize, or `#pragma novector` to tell the compiler not to vectorize. The pragmas must be placed immediately before the loop or the series of statements that you want them to apply to.

It is advantageous to use the smallest data size that fits the application. In example 12.3a, for example, you can double the speed by using `short int` instead of `int`. A `short int` is 16 bits wide, while an `int` is 32 bits, so you can have eight numbers of type `short int` in one vector, while you can only have four numbers of type `int`. Therefore, it is advantageous to use the smallest integer size that is big enough to hold the numbers in question without generating overflow. Likewise, it is advantageous to use `float` rather than `double` if the code can be vectorized, because a `float` uses 32 bits while a `double` uses 64 bits.

The SSE2 vector instruction set cannot multiply integers of any size other than `short int` (16 bits). There are no instructions for integer division in vectors, but the vector class library and the asmlib function library have functions for integer vector division.

12.4 Using intrinsic functions

It is difficult to predict whether the compiler will vectorize a loop or not. The following example shows a code that the compiler may or may not vectorize automatically. The code has a branch that chooses between two expressions for every element in the arrays:

```
// Example 12.4a. Loop with branch

// Loop with branch
void SelectAddMul(short int aa[], short int bb[], short int cc[]) {

   for (int i = 0; i < 256; i++) {
      aa[i] = (bb[i] > 0) ? (cc[i] + 2) : (bb[i] * cc[i]);
   }
}
```

It is possible to vectorize code explicitly by using the so-called intrinsic functions. This is useful in situations like example 12.4a where current compilers don't vectorize the code automatically. It is also useful in situations where automatic vectorization leads to suboptimal code.

Intrinsic functions are primitive operations in the sense that each intrinsic function call is translated to just one or a few machine instructions. Intrinsic functions are supported by the

Gnu, Clang, Intel, Microsoft and PathScale compilers. (The PGI compiler supports intrinsic functions, but in a very inefficient way. The Codeplay compiler has some support for intrinsic functions, but the function names are not compatible with the other compilers). The best performance is obtained with the Gnu, Clang and Intel compilers.

We want to vectorize the loop in example 12.4a so that we can handle eight elements at a time in vectors of eight 16-bit integers. The branch inside the loop can be implemented in various ways depending on the available instruction set. The most compatible way is to make a bit-mask which is all 1's when `bb[i] > 0` is true, and all 0's when false. The value of `cc[i]+2` is AND'ed with this mask, and `bb[i]*cc[i]` is AND'ed with the inverted mask. The expression that is AND'ed with all 1's is unchanged, while the expression that is AND'ed with all 0's gives zero. An OR combination of these two gives the chosen expression.

Example 12.4b shows how this can be implemented with intrinsic functions for the SSE2 instruction set:

```
// Example 12.4b. Vectorized with SSE2
#include <emmintrin.h>       // Define SSE2 intrinsic functions

// Function to load unaligned integer vector from array
static inline __m128i LoadVector(void const * p) {
   return _mm_loadu_si128((__m128i const*)p);
}

// Function to store unaligned integer vector into array
static inline void StoreVector(void * d, __m128i const & x) {
   _mm_storeu_si128((__m128i *)d, x);
}

// Branch/loop function vectorized:
void SelectAddMul(short int aa[], short int bb[], short int cc[]) {

   // Make a vector of (0,0,0,0,0,0,0,0)
   __m128i zero = _mm_set1_epi16(0);
   // Make a vector of (2,2,2,2,2,2,2,2)
   __m128i two  = _mm_set1_epi16(2);

   // Roll out loop by eight to fit the eight-element vectors:
   for (int i = 0; i < 256; i += 8) {
      // Load eight consecutive elements from bb into vector b:
      __m128i b = LoadVector(bb + i);
      // Load eight consecutive elements from cc into vector c:
      __m128i c = LoadVector(cc + i);
      // Add 2 to each element in vector c
      __m128i c2 = _mm_add_epi16(c, two);
      // Multiply b and c
      __m128i bc = _mm_mullo_epi16 (b, c);
      // Compare each element in b to 0 and generate a bit-mask:
      __m128i mask = _mm_cmpgt_epi16(b, zero);
      // AND each element in vector c2 with the bit-mask:
      c2 = _mm_and_si128(c2, mask);
      // AND each element in vector bc with the inverted bit-mask:
      bc = _mm_andnot_si128(mask, bc);
      // OR the results of the two AND operations:
      __m128i a = _mm_or_si128(c2, bc);
      // Store the result vector in eight consecutive elements in aa:
      StoreVector(aa + i, a);
   }
}
```

The resulting code will be very efficient because it handles eight elements at a time and it avoids the branch inside the loop. Example 12.4b executes three to seven times faster than example 12.4a, depending on how predictable the branch inside the loop is.

The type `__m128i` defines a 128 bit vector containing integers. It can contain either sixteen integers of 8 bits each, eight integers of 16 bits each, four integers of 32 bits each, or two integers of 64 bits each. The type `__m128` defines a 128 bit vector of four `float`. The type `__m128d` defines a 128 bit vector of two `double`.

The intrinsic vector functions have names that begin with `_mm`. These functions are listed in the compiler manual or in the programming manuals from Intel: "IA-32 Intel Architecture Software Developer's Manual", Volume 2A and 2B. There are hundreds of different intrinsic functions and it can be difficult to find the right function for a particular purpose.

The clumsy AND-OR construction in example 12.4b can be replaced by a blend instruction if the SSE4.1 instruction set is available:

```
// Example 12.4c. Same example, vectorized with SSE4.1

// Function to load unaligned integer vector from array
static inline __m128i LoadVector(void const * p) {
   return _mm_loadu_si128((__m128i const*)p);
}

// Function to store unaligned integer vector into array
static inline void StoreVector(void * d, __m128i const & x) {
   _mm_storeu_si128((__m128i *)d, x);
}

void SelectAddMul(short int aa[], short int bb[], short int cc[]) {

   // Make a vector of (0,0,0,0,0,0,0,0)
   __m128i zero = _mm_set1_epi16(0);
   // Make a vector of (2,2,2,2,2,2,2,2)
   __m128i two  = _mm_set1_epi16(2);

   // Roll out loop by eight to fit the eight-element vectors:
   for (int i = 0; i < 256; i += 8) {
      // Load eight consecutive elements from bb into vector b:
      __m128i b = LoadVector(bb + i);
      // Load eight consecutive elements from cc into vector c:
      __m128i c = LoadVector(cc + i);
      // Add 2 to each element in vector c
      __m128i c2 = _mm_add_epi16(c, two);
      // Multiply b and c
      __m128i bc = _mm_mullo_epi16 (b, c);
      // Compare each element in b to 0 and generate a bit-mask:
      __m128i mask = _mm_cmpgt_epi16(b, zero);
      // Use mask to choose between c2 and bc for each element
      __m128i a = _mm_blendv_epi8(bc, c2, mask);
      // Store the result vector in eight consecutive elements in aa:
      StoreVector(aa + i, a);
   }
}
```

You have to include the appropriate header file for the instruction set that you want to compile for. The names of the header files are as follows:

Instruction set	Header file
MMX	mmintrin.h
SSE	xmmintrin.h

SSE2	emmintrin.h
SSE3	pmmintrin.h
Suppl. SSE3	tmmintrin.h
SSE4.1	smmintrin.h
SSE4.2	nmmintrin.h (MS) smmintrin.h (Gnu)
AES, PCLMUL	wmmintrin.h
AVX	immintrin.h
AMD SSE4A	ammintrin.h
AMD XOP	ammintrin.h (MS) xopintrin.h (Gnu)
AMD FMA4	fma4intrin.h (Gnu)
all	intrin.h (MS) x86intrin.h (Gnu)

Table 12.2. Header files for intrinsic functions

You have to make sure that the CPU supports the corresponding instruction set. If you are including a header file for a higher instruction set than the CPU supports then you are risking to insert an instruction that the CPU doesn't support, and the program will crash. See page 125 for how to check for the supported instruction set.

Aligning data

Loading data into a vector goes faster if the data are aligned to an address divisible by the vector size (16 or 32 bytes). This has a significant effect on older processors and on Intel Atom processors, but is less important on most newer processors. The following example shows how to align arrays.

```
// Example 12.5. Aligned arrays

// Define macro for aligning data
#ifdef _MSC_VER              // If Microsoft compiler
#define Alignd(X) __declspec(align(16)) X
#else                        // Gnu compiler, etc.
#define Alignd(X) X __attribute__((aligned(16)))
#endif

const int size = 256;           // Array size

Alignd ( short int aa[size] );  // Make three aligned arrays
Alignd ( short int bb[size] );
Alignd ( short int cc[size] );

// Function to load aligned integer vector from array
static inline __m128i LoadVectorA(void const * p) {
   return _mm_load_si128((__m128i const*)p);
}

// Function to store aligned integer vector into array
static inline void StoreVectorA(void * d, __m128i const & x) {
   _mm_store_si128((__m128i *)d, x);
}
```

Vectorized table lookup

Lookup tables can be useful for optimizing code, as explained on page 135. Unfortunately, table lookup is often an obstacle to vectorization. The newest instruction sets include a few instructions that may be used for vectorized table lookup. These instructions are summarized below.

Intrinsic function	Max. number of elements in table	Size of each table element	Number of simultaneous lookups	Instruction set needed
_mm_shuffle_epi8	16	1 byte = char	16	SSSE3
_mm_perm_epi8	32	1 byte = char	16	XOP, AMD only
_mm_permutevar_ps	4	4 bytes = float or int	4	AVX
_mm256_permutevar_ps	4	4 bytes = float or int	8	AVX2
_mm_i32gather_epi32	unlimited	4 bytes = int	4	AVX2
_mm256_i32gather_epi32	unlimited	4 bytes = int	8	AVX2
_mm_i64gather_epi32	unlimited	8 bytes = int64_t	2	AVX2
_mm256_i64gather_epi32	unlimited	8 bytes = int64_t	4	AVX2
_mm_i32gather_ps	unlimited	4 bytes = float	4	AVX2
_mm256_i32gather_ps	unlimited	4 bytes = float	8	AVX2
_mm_i64gather_pd	unlimited	8 bytes = double	2	AVX2
_mm256_i64gather_pd	unlimited	8 bytes = double	4	AVX2

Table 12.3. Intrinsic functions for vectorized table lookup

Using intrinsic functions can be quite tedious and the code becomes bulky and difficult to read. It is often easier to use vector classes, as explained in the next section.

12.5 Using vector classes

Programming in the way of example 12.4b and 12.4c is quite tedious indeed. It is possible to write the same in a more clear and intelligible way by wrapping the vectors into C++ classes and using overloaded operators for things like adding vectors. The operators are inlined so that the resulting machine code becomes the same as if you had used intrinsic functions. It is just easier to write `a + b` than to write `_mm_add_epi16(a,b)`.

Various libraries of predefined vector classes are currently available, including one from Intel and one from me. My vector class library (VCL) has many features, see www.agner.org/optimize/#vectorclass. The Intel vector class library has not been updated lately and may be considered obsolete.

Vector class library	Intel	VCL (Agner)
Available from	Intel and Microsoft C++ compilers	www.agner.org/ optimize/#vectorclass
Include file	`dvec.h`	`vectorclass.h`
Supported compilers	Intel, Microsoft	Intel, Microsoft, Gnu, Clang
Supported operating systems	Windows, Linux, Mac	Windows, Linux, Mac, BSD
Instruction set control	no	yes
License	license included in compiler price	GNU General Public License, optional commercial license

Table 12.4. Vector class libraries

The following table lists the available vector classes. Including the appropriate header file will give you access to all of these classes.

Size of each element, bits	Number of elements in vector	Type of elements	Total size of vector, bits	Vector class, Intel	Vector class, VCL
8	8	char	64	Is8vec8	
8	8	unsigned char	64	Iu8vec8	
16	4	short int	64	Is16vec4	
16	4	unsigned short int	64	Iu16vec4	
32	2	int	64	Is32vec2	
32	2	unsigned int	64	Iu32vec2	
64	1	int64_t	64	I64vec1	
8	16	char	128	Is8vec16	Vec16c
8	16	unsigned char	128	Iu8vec16	Vec16uc
16	8	short int	128	Is16vec8	Vec8s
16	8	unsigned short int	128	Iu16vec8	Vec8us
32	4	int	128	Is32vec4	Vec4i
32	4	unsigned int	128	Iu32vec4	Vec4ui
64	2	int64_t	128	I64vec2	Vec2q
64	2	uint64_t	128		Vec2uq
8	32	char	256		Vec32c
8	32	unsigned char	256		Vec32uc
16	16	short int	256		Vec16s
16	16	unsigned short int	256		Vec16us
32	8	int	256		Vec8i
32	8	unsigned int	256		Vec8ui
64	4	int64_t	256		Vec4q
64	4	uint64_t	256		Vec4uq
32	16	int	512		Vec16i
32	16	unsigned int	512		Vec16ui
64	8	int64_t	512		Vec8q
64	8	uint64_t	512		Vec8uq
32	4	float	128	F32vec4	Vec4f
64	2	double	128	F64vec2	Vec2d
32	8	float	256	F32vec8	Vec8f
64	4	double	256	F64vec4	Vec4d
32	16	float	512		Vec16f
64	8	double	512		Vec8d

Table 12.5. Vector classes defined in two libraries

It is not recommended to use the vectors of 64 bits total size, because these are incompatible with floating point code. If you do use the 64-bit vectors then you have to execute `_mm_empty()` after the 64-bit vector operations and before any floating point code. The bigger vectors do not have this problem.

Vectors of 256 and 512 bits size are only available if supported by the CPU and the operating system (see page 109). My VCL vector class library can emulate a 256-bit vector as two 128-bit vectors or a 512-bit vector as two 256-bit vectors or four 128-bit vectors.

The following example shows the same code as example 12.4b, rewritten with the use of Intel vector classes:

```
// Example 12.4d. Same example, using Intel vector classes
#include <dvec.h>  // Define vector classes

// Function to load unaligned integer vector from array
static inline __m128i LoadVector(void const * p) {
   return _mm_loadu_si128((__m128i const*)p);}

// Function to store unaligned integer vector into array
static inline void StoreVector(void * d, __m128i const & x) {
   _mm_storeu_si128((__m128i *)d, x);}

void SelectAddMul(short int aa[], short int bb[], short int cc[]) {

   // Make a vector of (0,0,0,0,0,0,0,0)
   Is16vec8 zero(0,0,0,0,0,0,0,0);
   // Make a vector of (2,2,2,2,2,2,2,2)
   Is16vec8 two(2,2,2,2,2,2,2,2);

   // Roll out loop by eight to fit the eight-element vectors:
   for (int i = 0; i < 256; i += 8) {
      // Load eight consecutive elements from bb into vector b:
      Is16vec8 b = LoadVector(bb + i);
      // Load eight consecutive elements from cc into vector c:
      Is16vec8 c = LoadVector(cc + i);
      // result = b > 0 ? c + 2 : b * c;
      Is16vec8 a = select_gt(b, zero, c + two, b * c);
      // Store the result vector in eight consecutive elements in aa:
      StoreVector(aa + i, a);
   }
}
```

The same example using my VCL vector classes looks like this:

```
// Example 12.4e. Same example, using VCL
#include "vectorclass.h"     // Define vector classes

void SelectAddMul(short int aa[], short int bb[], short int cc[]) {
   // Define vector objects
   Vec16s a, b, c;

   // Roll out loop by eight to fit the eight-element vectors:
   for (int i = 0; i < 256; i += 16) {
      // Load eight consecutive elements from bb into vector b:
      b.load(bb+i);
      // Load eight consecutive elements from cc into vector c:
      c.load(cc+i);
      // result = b > 0 ? c + 2 : b * c;
      a = select(b > 0, c + 2, b * c);
      // Store the result vector in eight consecutive elements in aa:
      a.store(aa+i);
   }
}
```

The Microsoft compiler does not allow vector objects as function parameters because of alignment problems. It is recommended to use a constant reference instead:

```
// Example 12.6. Function with vector parameters
Vec4f polynomial (Vec4f const & x) {
   // polynomial(x) = 2.5*x^2 - 8*x + 2
   return (2.5f * x - 8.0f) * x + 2.0f;
```

```
}
```

CPU dispatching with vector classes

The VCL vector class library makes it possible to compile for different instruction sets from the same source code. The library has preprocessing directives that select the best implementation for a given instruction set.

The following example shows how to make the `SelectAddMul` example (12.4e) with automatic CPU dispatching. The code in this example should be compiled three times, one for the SSE2 instruction set, one for SSE4.1 and one for AVX2 and all three versions should be linked into the same executable. SSE2 is the minimum supported instruction set for the vector class library, SSE4.1 gives an advantage in the select function, and the AVX2 instruction set gives the advantage of bigger vector registers. The vector class library will use one 256-bit vector register for the class `Vec16s` when compiling for AVX2, or two 128-bit vector registers when compiling for a lower instruction set. The preprocessing macro `INSTRSET` is used for giving the function a different name for each instruction set. A CPU dispatcher then sets a function pointer to the best possible version. See the vectorclass manual for details.

```
// Example 12.7. Vector class code with automatic CPU dispatching
#include "vectorclass.h"  // vector class library
#include <stdio.h>        // define fprintf

// define function type
typedef void FuncType(short int aa[], short int bb[], short int cc[]);

// function prototypes for each version
FuncType SelectAddMul, SelectAddMul_SSE2, SelectAddMul_SSE41,
SelectAddMul_AVX2, SelectAddMul_dispatch;

// Define function name depending on instruction set
#if   INSTRSET == 2                    // SSE2
#define FUNCNAME SelectAddMul_SSE2
#elif INSTRSET == 5                    // SSE4.1
#define FUNCNAME SelectAddMul_SSE41
#elif INSTRSET == 8                    // AVX2
#define FUNCNAME SelectAddMul_AVX2
#endif

// specific version of the function. Compile once for each version
void FUNCNAME(short int aa[], short int bb[], short int cc[]) {
   Vec16s a, b, c; // Define biggest possible vector objects
   // Roll out loop by 16 to fit the biggest vectors:
   for (int i = 0; i < 256; i += 16) {
      b.load(bb+i);
      c.load(cc+i);
      a = select(b > 0, c + 2, b * c);
      a.store(aa+i);
   }
}

#if INSTRSET == 2
// make dispatcher in only the lowest of the compiled versions
#include "instrset_detect.cpp" // instrset_detect function

// Function pointer initially points to the dispatcher.
// After first call it points to the selected version
FuncType * SelectAddMul_pointer = &SelectAddMul_dispatch;

// Dispatcher
void SelectAddMul_dispatch(short int aa[], short int bb[],
```

```
   short int cc[]) {
   // Detect supported instruction set
   int iset = instrset_detect();
   // Set function pointer
   if      (iset >= 8) SelectAddMul_pointer = &SelectAddMul_AVX2;
   else if (iset >= 5) SelectAddMul_pointer = &SelectAddMul_SSE41;
   else if (iset >= 2) SelectAddMul_pointer = &SelectAddMul_SSE2;
   else {
      // Error: lowest instruction set not supported
      fprintf(stderr, "\nError: Instruction set SSE2 not supported");
      return;
    }
   // continue in dispatched version
   return (*SelectAddMul_pointer)(aa, bb, cc);
}

// Entry to dispatched function call
inline void SelectAddMul(short int aa[], short int bb[],
   short int cc[]) {
   // go to dispatched version
   return (*SelectAddMul_pointer)(aa, bb, cc);
}

#endif  // INSTRSET == 2
```

12.6 Transforming serial code for vectorization

Not all code has a parallel structure that can easily be organized into vectors. A lot of code is serial in the sense that each calculation depends on the previous one. It may nevertheless be possible to organize the code in a way that can be vectorized if the code is repetitive. The simplest case is a sum of a long list of numbers:

```
// Example 12.8a. Sum of a list
float a[100];
float sum = 0;
for (int i = 0; i < 100; i++) sum += a[i];
```

The above code is serial because each value of `sum` depends on the preceding value of `sum`. The trick is to roll out the loop by n and reorganize the code so that each value depends on the value that is n places back, where n is the number of elements in a vector. If $n = 4$, we have:

```
// Example 12.8b. Sum of a list, rolled out by 4
float a[100];
float s0 = 0, s1 = 0, s2 = 0, s3 = 0, sum;
for (int i = 0; i < 100; i += 4) {
   s0 += a[i];
   s1 += a[i+1];
   s2 += a[i+2];
   s3 += a[i+3];
}
sum = (s0+s1)+(s2+s3);
```

Now `s0`, `s1`, `s2` and `s3` can be combined into a 128-bit vector so that we can do four additions in one operation. A good compiler will convert example 12.8a to 12.8b automatically and vectorize the code if we specify the options for fast math and the SSE or higher instruction set.

More complicated cases cannot be vectorized automatically. For example, let's look at the example of a Taylor series. The exponential function can be calculated by the series:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

A C++ implementation may look like this:

```
// Example 12.9a. Taylor series
float Exp(float x) {       // Approximate exp(x) for small x
   float xn = x;           // x^n
   float sum = 1.f;        // sum, initialize to x^0/0!
   float nfac = 1.f;       // n factorial
   for (int n = 1; n <= 16; n++) {
      sum += xn / nfac;
      xn *= x;
      nfac *= n+1;
   }
   return sum;
}
```

Here, each value x^n is calculated from the previous value as $x^n = x \cdot x^{n-1}$, and each value of n! is calculated from the previous value as $n! = n \cdot (n-1)!$. If we want to roll out the loop by four, we will have to calculate each value from the value that is four places back. Thus, we will calculate x^n as $x^4 \cdot x^{n-4}$. There is no easy way to roll out the calculation of the factorials, but this is not necessary because the factorials don't depend on x so we can store the values in a pre-calculated table. Even better: store the reciprocal factorials so that we don't have to do the divisions (Division is slow, you know). The code can now be vectorized as follows (using Intel vector classes):

```
// Example 12.9b. Taylor series, vectorized
#include <dvec.h>          // Define vector classes (Intel)
#include <pmmintrin.h>     // SSE3 required

// This function adds the elements of a vector, uses SSE3.
// (This is faster than the function add_horizontal)
static inline float add_elements(__m128 const & x) {
   __m128 s;
   s = _mm_hadd_ps(x, x);
   s = _mm_hadd_ps(s, s);
   return _mm_cvtss_f32(s);
}

float Exp(float x) {       // Approximate exp(x) for small x
   __declspec(align(16))   // align table by 16
   const float coef[16] = {      // table of 1/n!
      1., 1./2., 1./6., 1./24., 1./120., 1./720., 1./5040.,
      1./40320., 1./362880., 1./3628800., 1./39916800.,
      1./4.790016E8, 1./6.22702E9, 1./8.71782E10,
      1./1.30767E12, 1./2.09227E13};
   float x2 = x * x;              // x^2
   float x4 = x2 * x2;            // x^4
   // Define vectors of four floats
   F32vec4 xxn(x4, x2*x, x2, x);  // x^1, x^2, x^3, x^4
   F32vec4 xx4(x4);               // x^4
   F32vec4 s(0.f, 0.f, 0.f, 1.f); // initialize sum
   for (int i = 0; i < 16; i += 4) {  // Loop by 4
      s += xxn * _mm_load_ps(coef+i); // s += x^n/n!
      xxn *= xx4;                    // next four x^n
   }
   return add_elements(s);           // add the four sums
}
```

This loop calculates four consecutive terms in one vector. It may be worthwhile to unroll the loop further if the loop is long because the speed here is likely to be limited by the latency of the multiplication of `xxn` rather than the throughput (see p. 106). The table of coefficients is

calculated at compile time here. It may be more convenient to calculate the table at runtime, if only you make sure it is only calculated once, rather than each time the function is called.

12.7 Mathematical functions for vectors

There are various function libraries for computing mathematical functions such as logarithms, exponential functions, trigonometric functions, etc. in vectors. These function libraries are useful for vectorizing mathematical code.

There are two different kinds of vector math libraries: long vector libraries and short vector libraries. To explain the difference, let's say that you want to calculate the same function on a thousand numbers. With a long vector library, you are feeding an array of thousand numbers as a parameter to the library function, and the function stores the thousand results in another array. The disadvantage of using a long vector library is that if you are doing a sequence of calculations then you have to store the intermediate result of each step of the sequence in a temporary array before calling the function for the next step. With a short vector library, you divide the data set into sub-vectors that fit the size of the vector registers in the CPU. If the vector registers can hold e.g. four numbers, then you have to call the library function 250 times with four numbers at a time packed into a vector register. The library function will return the result in a vector register which can be fed directly to the next step in the sequence of calculations without the need to store intermediate results in RAM memory. This may be faster despite the extra function calls because the CPU can do calculations while simultaneously prefetching the code of the next function. However, the short vector method may be at a disadvantage if the sequence of calculations forms a long dependency chain. We want the CPU to start calculations on the second sub-vector before it has finished the calculations on the first sub-vector. A long dependency chain may fill up the queue of pending instructions in the CPU and prevent it from fully utilizing its out-of-order calculation capabilities.

Here is a list of some long vector math libraries:

- Intel vector math library (VML, MKL). Works with all x86 platforms. This library has reduced performance on non-Intel CPUs unless you are overriding Intel's CPU dispatcher. See page 134.
- Intel Performance Primitives (IPP). Works with all x86 platforms. Works well with non-Intel CPUs. Includes many functions for statistics, signal processing and image processing.
- Yeppp. Open source library. Supports x86 and ARM platforms and various programming languages. www.yeppp.info

And here is a list of short vector math libraries:

- Sleef library. Supports many different platforms. Open source. www.sleef.org
- Intel short vector math library (SVML). This is supplied with Intel's compilers and invoked with automatic vectorization. The Gnu compiler can use this library with the option `-mveclibabi=svml`. This library usually works well with non-Intel CPUs if you are not using an Intel compiler. See page 134.
- AMD LIBM library. Only available for 64-bit Linux and Windows platforms. This library has reduced performance on CPUs without the FMA4 instruction set. (This instruction set was originally designed by Intel but is currently only supported on AMD CPUs). The Gnu compiler can use this library with the option `-mveclibabi=acml`.
- VCL vector class library. Open source. Supports all x86 platforms. Microsoft, Intel, Gnu and Clang compilers. The code is inlined - no need to link with external libraries. www.agner.org/optimize/#vectorclass

All these libraries have very good performance and precision. The speed is many times faster than any non-vector library.

The function names in the SVML and LIBM libraries are not well documented. The examples in this table may be of some help if you want to call the library functions directly:

Library	exp function of 4 floats	exp function of 2 double
Intel SVML v.10.2 & earlier	`vmlsExp4`	`vmldExp2`
Intel SVML v.10.3 & later	`__svml_expf4`	`__svml_exp2`
Intel SVML + ia32intrin.h	`_mm_exp_ps`	`_mm_exp_pd`
AMD Core Math Library	`__vrs4_expf`	`__vrd2_exp`
AMD LIBM Library	`amd_vrs4_expf`	`amd_vrd2_exp`
VCL vector class library	`exp`	`exp`

12.8 Aligning dynamically allocated memory

Memory allocated with `new` or `malloc` is typically aligned by 8 rather than by 16. This is a problem with vector operations when alignment by 16 is required. The Intel compiler has solved this problem by defining `_mm_malloc` and `_mm_free`.

A more general method is to wrap the allocated array into a container class that takes care of the alignment. See www.agner.org/optimize/cppexamples.zip for examples of how to make aligned arrays with vector access.

12.9 Aligning RGB video or 3-dimensional vectors

RGB image data have three values per point. This does not fit into a vector of e.g. four floats. The same applies to 3-dimensional geometry and other odd-sized vector data. The data have to be aligned by the vector size for the sake of efficiency. Using unaligned reads and writes may slow down the execution to the point where it is less advantageous to use vector operations. You may choose one of the following solutions, depending on what fits best into the algorithm in question:

- Put in an unused fourth value to make the data fit into the vector. This is a simple solution, but it increases the amount of memory used. You may avoid this method if memory access is a bottleneck.

- Organize the data into groups of four (or eight) points with the four R value in one vector, the four G values in the next vector, and the four B value in the last vector.

- Organize the data with all the R values first, then all the G values, and last all the B values.

The choice of which method to use depends on what fits best into the algorithm in question. You may choose the method that gives the simplest code.

If the number of points is not divisible by the vector size then add a few unused points in the end in order to get an integral number of vectors.

12.10 Conclusion

There is a lot to gain in speed by using vectors if the code contains natural parallelism. The gain depends on the number of elements per vector. The simplest and most clean solution is to rely on automatic vectorization by the compiler. The compiler will vectorize the code automatically in simple cases where the parallelism is obvious and the code contains only

simple standard operations. All you have to do is to enable the appropriate instruction set and a non-restrictive floating point option.

However, there are many cases where the compiler is unable to vectorize the code automatically or does so in a suboptimal way. Here you have to vectorize the code explicitly. There are various ways to do this:

- Use assembly language
- Use intrinsic functions
- Use predefined vector classes

The easiest way to vectorize code explicitly is by using a vector class library. You may combine this with intrinsic functions if you need things that are not defined in the vector class library. Whether you choose to use intrinsic functions or vector classes is just a matter of convenience - there is no difference in performance. A good optimizing compiler should produce the same code in either case. Intrinsic functions look clumsy and tedious. The code becomes more readable when you are using vector classes and overloaded operators.

A good compiler is often able to optimize the code further after you have vectorized it manually. The compiler can use optimization techniques such as function inlining, common subexpression elimination, constant propagation, loop optimization, etc. These techniques are rarely used in manual assembly coding because it makes the code unwieldy, error prone, and almost impossible to maintain. The combination of manual vectorization with further optimization by the compiler can therefore give the best result in many cases. Current compilers are not always good at constant propagation and certain other optimization techniques on vector code. Therefore, it is sometimes better to rely on automatic vectorization by the compiler in cases where the compiler can do so without problems. Some experimentation may be needed to find the best solution.

Vectorized code often contains a lot of extra instructions for converting the data to the right format and getting them into the right positions in the vectors. This data conversion and shuffling can sometimes take more time than the actual calculations. This should be taken into account when deciding whether it is profitable to use vectorized code or not.

I will conclude this section by summing up the factors that decide how advantageous vectorization is.

Factors that make vectorization favorable:

- Small data types: `char`, `short int`, `float`.
- Similar operations on all data in large arrays.
- Array size divisible by vector size.
- Unpredictable branches that select between two simple expressions.
- Operations that are only available with vector operands: minimum, maximum, saturated addition, fast approximate reciprocal, fast approximate reciprocal square root, RGB color difference.
- Vector instruction set available, e.g. AVX, AVX2, AVX-512
- Mathematical vector function libraries.
- Use Gnu, Clang or Intel compiler.

Factors that make vectorization less favorable:

- Larger data types: `int64_t`, `double`.
- Misaligned data.
- Extra data conversion, shuffling, packing, unpacking needed.

- Predictable branches that can skip large expressions when not selected.
- Compiler has insufficient information about pointer alignment and aliasing.
- Operations that are missing in the instruction set for the appropriate type of vector, such as 32-bit integer multiplication prior to SSE4.1 and integer division.
- Older CPUs with execution units smaller than the vector register size.

Vectorized code is more difficult for the programmer to make and therefore more error prone. The vectorized code should therefore preferably be put away in reusable and well-tested library modules and header files.

13 Making critical code in multiple versions for different instruction sets

Microprocessor producers keep adding new instructions to the instruction set. These new instructions can make certain kinds of code execute faster. The most important addition to the instruction set is the vector operations mentioned in chapter 12.

If the code is compiled for a particular instruction set then it will be compatible with all CPUs that support this instruction set or any higher instruction set, but possibly not with earlier CPUs. The sequence of backwards compatible instruction sets is as follows:

Instruction set	Important features
80386	32 bit mode
SSE	128 bit float vectors
SSE2	128 bit integer and double vectors
SSE3	horizontal add, etc.
SSSE3	a few more integer vector instructions
SSE4.1	some more vector instructions
SSE4.2	string search instructions
AVX	256 bit float and double vectors
AVX2	256 bit integer vectors
FMA3	floating point multiply-and-add
AVX-512	512 bit integer and floating point vectors

Table 13.1. Instruction sets

A more detailed explanation of the instruction sets is provided in manual 4: "Instruction tables". There are certain restrictions on mixing code compiled for AVX or later with code compiled without AVX, as explained on page 109.

A disadvantage of using the newest instruction set is that the compatibility with older microprocessors is lost. This dilemma can be solved by making the most critical parts of the code in multiple versions for different CPUs. This is called CPU dispatching. For example, you may want to make one version that takes advantage of the AVX2 instruction set, another version for CPUs with only the SSE2 instruction set, and a generic version that is compatible with old microprocessors without any of these instruction sets. The program should automatically detect which instruction set is supported by the CPU and the operating system and choose the appropriate version of the subroutine for the critical innermost loops.

13.1 CPU dispatch strategies

It is quite expensive - in terms of development, testing and maintenance - to make a piece of code in multiple versions, each carefully optimized and fine-tuned for a particular set of CPUs. These costs can be justified for general function libraries that are used in multiple applications, but not always for application-specific code. If you consider making highly

optimized code with CPU dispatching, then it is advisable to make it in the form of a re-usable library if possible. This also makes testing and maintenance easier.

I have done a good deal of research on CPU dispatching and discovered that many common programs use inappropriate CPU dispatch methods.

The most common pitfalls of CPU dispatching are:

- Optimizing for present processors rather than future processors. Consider the time it takes to develop and publish a function library with CPU dispatching. Add to this the time it takes before the application programmer gets the new version of the library. Add to this the time it takes to develop and market the application program. Add to this the time before the end user gets the latest version of the application program. All in all, it will often take several years before your code is running in the majority of end user's computers. At this time, any processor that you optimized for is likely to be obsolete. Programmers very often underestimate this time lag.

- Thinking in terms of specific processor models rather than processor features. The programmer typically thinks "what works best on processor X?" rather than "what works best on processors with this instruction set?". A list of which code branch to use for each processor model is going to be very long and difficult to maintain. And it is unlikely that the end user will have an up-to-date version. The CPU dispatcher should not look at CPU brand names and model numbers, but on what instruction sets and other features it has.

- Assuming that processor model numbers form a logical sequence. If you know that processor model N supports a particular instruction set, then you cannot assume that model N+1 supports at least the same instruction set. Neither can you assume that model N-1 is inferior. A model with a higher number is not necessarily newer. The CPU family and model numbers are not always sequential, and you cannot make any assumption about an unknown CPU based on its family and model number.

- Failure to handle unknown processors properly. Many CPU dispatchers are designed to handle only known processors. Other brands or models that were unknown at the time of programming will typically get the generic branch, which is the one that gives the worst possible performance. We must bear in mind, that many users will prefer to run a speed-critical program on the newest CPU model, which quite likely is a model that was unknown at the time of programming. The CPU dispatcher should give a CPU of unknown brand or model the best possible branch if it has an instruction set that is compatible with that branch. The common excuse that *"we don't support processor X"* is simply not appropriate here. It reveals a fundamentally flawed approach to CPU dispatching.

- Underestimating the cost of keeping a CPU dispatcher updated. It is tempting to fine-tune the code to a specific CPU model and then think that you can make an update when the next new model comes on the market. But the cost of fine-tuning, testing, verifying and maintaining a new branch of code is so high that it is unrealistic that you can do this every time a new processor enters the market for many years to come. Even big software companies often fail to keep their CPU dispatchers up to date. A more realistic goal is to make a new branch only when a new instruction set opens the possibility for significant improvements.

- Making too many branches. If you are making branches that are fine-tuned for specific CPU brands or specific models then you will soon get a lot of branches that take up cache space and are difficult to maintain. Any specific bottleneck or any particularly slow instruction that you are dealing with in a particular CPU model is

likely to be irrelevant within a year or two. Often, it is sufficient to have just two branches: one for the latest instruction set and one that is compatible with CPUs that are up to five or ten years old. The CPU market is developing so fast that what is brand new today will be mainstream next year.

- Ignoring virtualization. The time when the CPUID instruction was certain to truly represent a known CPU model is over. Virtualization is becoming increasingly important. A virtual processor may have a reduced number of cores in order to reserve resources for other virtual processors on the same machine. The virtual processor may be given a false model number to reflect this or for the sake of compatibility with some legacy software. It may even have a false vendor string. In the future we may also see emulated processors and FPGA soft cores that do not correspond to any known hardware CPU. These virtual processors can have any brand name and model number. The only CPUID information that we can surely rely on is the feature information, such as supported instruction sets and cache sizes.

Fortunately, the solution to these problems is quite simple in most cases: The CPU dispatcher should have as few branches as possible, and the dispatching should be based on which instruction sets the CPU supports, rather than its brand, family and model number.

I have seen many examples of poor CPU dispatching. For example, the latest version of Mathcad (v. 15.0) is using a six years old version of Intel's Math Kernel Library (MKL v. 7.2). This library has a CPU dispatcher that doesn't handle current CPUs optimally. The speed for certain tasks on current Intel CPUs can be increased by more than 33% when the CPUID is artificially changed to the old Pentium 4. The reason is that the CPU dispatcher in the MKL relies on the CPU family number, which is 15 on the old Pentium 4, while all newer Intel CPUs have family number 6! The speed on non-Intel CPUs was more than doubled for this task when the CPUID was manipulated to fake an Intel Pentium 4. Even worse, many software products fail to recognize VIA processors because this brand was less popular at the time the software was developed.

A CPU dispatch mechanism that treats different brands of CPUs unequally can become a serious legal issue, as you can read about in my blog. Here, you can also find more examples of bad CPU dispatching.

Obviously, you should apply CPU dispatching only to the most critical part of the program - preferably isolated into a separate function library. The radical solution of making the entire program in multiple versions should be used only when instruction sets are mutually incompatible. A function library with a well-defined functionality and a well-defined interface to the calling program is more manageable and easier to test, maintain and verify than a program where the dispatch branches are scattered everywhere in the source files.

13.2 Model-specific dispatching

There may be cases where a particular code implementation works particularly bad on a particular processor model. You may ignore the problem and assume that the next processor model will work better. If the problem is too important to ignore, then the solution is to make a *negative list* of processor models on which this code version performs poorly. It is not a good idea to make a *positive list* of processor models on which a code version performs well. The reason is that a positive list needs to be updated every time a new and better processor appears on the market. Such a list is almost certain to become obsolete within the lifetime of your software. A negative list, on the other hand, does not need updating in the likely case that the next generation of processors is better. Whenever a processor has a particular weakness or bottleneck, it is likely that the producer will try to fix the problem and make the next model work better.

Remember again, that most software runs most of the time on processors that were unknown at the time the software was coded. If the software contains a positive list of which processor models to run the most advanced code version on, then it will run an inferior version on the processors that were unknown at the time it was programmed. But if the software contains a negative list of which processor models to avoid running the advanced version on, then it will run the advanced version on all newer models that were unknown at the time of programming.

13.3 Difficult cases

In most cases, the optimal branch can be chosen based on the CPUID information about supported instruction sets, cache size, etc. There are a few cases, however, where there are different ways of doing the same thing and the CPUID instruction doesn't give the necessary information about which implementation is best. These cases are sometimes dealt with in assembly language. Here are some examples:

- `strlen` function. The string length function scans a string of bytes to find the first byte of zero. A good implementation uses XMM registers to test 16 bytes at a time and afterwards a `BSF` (bit scan forward) instruction to localize the first byte of zero within a block of 16 bytes. Some CPUs have particularly slow implementations of this bit scan instruction. Programmers that have tested the `strlen` function in isolation have been unsatisfied with the performance of this function on CPUs with a slow bit scan instruction and have implemented a separate version for specific CPU models. However, we must consider that the bit scan instruction is executed only once for each function call so that you have to call the function billions of times before the performance even matters, which few programs do. Hence, it is hardly worth the effort to make special versions of the `strlen` function for CPUs with slow bit scan instructions. My recommendation is to use the bit scan instruction and expect this to be the fastest solution on future CPUs.

- Half size execution units. The size of vector registers has been increased from 64-bit MMX to 128-bit XMM and 256-bit YMM registers. The first processors that supported 128-bit vector registers had in fact only 64-bit execution units. Each 128-bit operation was split into two 64-bit operations so that there was hardly any speed advantage in using the larger vector size. Later models had the full 128-bit execution units and hence higher speed. In the same way, the first processors that supported 256-bit instructions were splitting 256-bit read operations into two 128-bit reads. The same can be expected for the forthcoming 512-bit instruction set and for further expansions of the register size in the future. Typically, the full advantage of a new register size comes only in the second generation of processors that support it. There are situations where a vector implementation is optimal only on CPUs with full-size execution units. The problem is that it is difficult for the CPU dispatcher to know whether the largest vector register size is handled at half speed or full speed. A simple solution to this problem is to use a new register size only when the next higher instruction set is supported. For example, use AVX only when AVX2 is supported in such applications. Alternatively, use a negative list of processors on which it is not advantageous to use the newest instruction set.

- High precision math. Libraries for high precision math allow addition of integers with a very large number of bits. This is usually done in a loop of `ADC` (add with carry) instructions where the carry bit must be saved from one iteration to the next. The carry bit can be saved either in the carry flag or in a register. If the carry bit is kept in the carry flag then the loop branch must rely on instructions that use the zero flag and don't modify the carry flag (e.g. `DEC`, `JNZ`). This solution can incur a large delay due to the so-called partial flags stall on some processors because the CPU has problems separating the flags register into the carry and zero flags on older Intel CPUs, but not on AMD CPUs (See manual 3: "The microarchitecture of Intel, AMD

and VIA CPUs"). This is one of the few cases where it makes sense to dispatch by CPU brand. The version that works best on the newest CPU of a particular brand is likely to be the optimal choice for future models of the same brand. Newer processors support the ADX instructions for high precision math.

Memory copying. There are several different ways of copying blocks of memory. These methods are discussed in manual 2: "Optimizing subroutines in assembly language", section 17.9: "Moving blocks of data", where it is also discussed which method is fastest on different processors. In a C++ program, you should choose an up-to-date function library that has a good implementation of the `memcpy` function. There are so many different cases for different microprocessors, different alignments and different sizes of the data block to copy that the only reasonable solution is to have a standard function library that takes care of the CPU dispatching. This function is so important and generally used that most function libraries have CPU dispatching for this function, though not all libraries have the best and most up-to-date solution. The compiler is likely to use the `memcpy` function implicitly when copying a large object, unless there is a copy constructor specifying otherwise.

In difficult cases like these, it is important to remember that your code is likely to run most of its time on processors that were unknown at the time it was programmed. Therefore, it is important to consider which method is likely to work best on future processors, and choose this method for all unknown processors that support the necessary instruction set. It is rarely worth the effort to make a CPU dispatcher based on complicated criteria or lists of specific CPU models if the problem is likely to go away in the future due to general improvements in microprocessor hardware design.

The ultimate solution would be to include a performance test that measures the speed of each version of the critical code to see which solution is optimal on the actual processor. However, this involves the problems that the clock frequency may vary dynamically and that measurements are unstable due to interrupts and task switches; so that it is necessary to test the different versions alternatingly several times in order to make a reliable decision.

13.4 Test and maintenance

There are two things to test when software uses CPU dispatching:

1. How much you gain in speed by using a particular code version.

2. Check that all code versions work correctly.

The speed test should preferably be done on the type of CPU that each particular branch of code is intended for. In other words, you need to test on several different CPUs if you want to optimize for several different CPUs.

On the other hand, it is not necessary to have many different CPUs to verify that all code branches works correctly. A code branch for a low instruction set can still run on a CPU with a higher instruction set. Therefore, you only need a CPU with the highest instruction set in order to test all branches for correctness. It is therefore recommended to put a test feature into the code that allows you to override the CPU dispatching and run any code branch for test purposes.

If the code is implemented as a function library or a separate module then it is convenient to make a test program that can call all code branches separately and test their functionality. This will be very helpful for later maintenance. However, this is not a textbook on test theory. Advice on how to test a software module for correctness must be found elsewhere.

13.5 Implementation

The CPU dispatch mechanism can be implemented in different places making the dispatch decision at different times:

- Dispatch on every call. A branch tree or switch statement leads to the appropriate version of the critical function. The branching is done every time the critical function is called. This has the disadvantage that the branching takes time.

- Dispatch on first call. The function is called through a function pointer which initially points to a dispatcher. The dispatcher changes the function pointer and makes it point to the right version of the function. This has the advantage that it does not spend time on deciding which version to use in case the function is never called. This method is illustrated in example 13.1 below.

- Make pointer at initialization. The program or library has an initialization routine that is called before the first call to the critical function. The initialization routine sets a function pointer to the right version of the function. This has the advantage that the response time is consistent for the function call.

- Load library at initialization. Each code version is implemented in a separate dynamic link library (`*.dll` or `*.so`). The program has an initialization routine that loads the appropriate version of the library. This method is useful if the library is very large or if different versions have to be compiled with different compilers.

- Dispatch at load time. The program uses a procedure linkage table (PLT) that is initialized when the program is loaded. This method requires OS support and is available in newer versions of Linux and perhaps Mac OS. See page 132 below.

- Dispatch at installation time. Each code version is implemented in a separate dynamic link library (`*.dll` or `*.so`). The installation program makes a symbolic link to the appropriate version of the library. The application program loads the library through the symbolic link.

- Use different executables. This method can be used if instruction sets are mutually incompatible. You may make separate executables for 32-bit and 64-bit systems. The appropriate version of the program may be selected during the installation process or by an executable file stub.

If different versions of the critical code are compiled with different compilers then it is recommended to specify static linking for any library functions called by the critical code so that you don't have to distribute all the dynamic libraries (`*.dll` or `*.so`) that belong to each compiler with the application.

The availability of various instruction sets can be determined with system calls (e.g. `IsProcessorFeaturePresent` in Windows). Alternatively, you may call the CPUID instruction directly, or use the CPU detection function that I have supplied in the library www.agner.org/optimize/asmlib.zip. The name of this function is `InstructionSet()`.The following example shows how to implement the *dispatch on first call* method using `InstructionSet()`:

```
// Example 13.1
// CPU dispatching on first call

// Header file for InstructionSet()
#include "asmlib.h"

// Define function type with desired parameters
```

```
typedef int CriticalFunctionType(int parm1, int parm2);

// Function prototype
CriticalFunctionType CriticalFunction_Dispatch;

// Function pointer serves as entry point.
// After first call it will point to the appropriate function version
CriticalFunctionType * CriticalFunction = &CriticalFunction_Dispatch;

// Lowest version
int CriticalFunction_386(int parm1, int parm2) {...}

// SSE2 version
int CriticalFunction_SSE2(int parm1, int parm2) {...}

// AVX version
int CriticalFunction_AVX(int parm1, int parm2) {...}

// Dispatcher. Will be called only first time
int CriticalFunction_Dispatch(int parm1, int parm2)
{
   // Get supported instruction set, using asmlib library
   int level = InstructionSet();

   // Set pointer to the appropriate version (May use a table
   // of function pointers if there are many branches):
   if (level >= 11)
   {  // AVX supported
      CriticalFunction = &CriticalFunction_AVX;
   }
   else if (level >= 4)
   {  // SSE2 supported
      CriticalFunction = &CriticalFunction_SSE2;
   }
   else
   {  // Generic version
      CriticalFunction = &CriticalFunction_386;
   }

   // Now call the chosen version
   return (*CriticalFunction)(parm1, parm2);
}

int main()
{
   int a, b, c;
   ...

   // Call critical function through function pointer
   a = (*CriticalFunction)(b, c);

   ...
   return 0;
}
```

The `InstructionSet()` function is available in the function library asmlib, which is available in different versions for different compilers. This function is OS independent and checks both the CPU and the operating system for support of the different instructions sets. The different versions of `CriticalFunction` in example 13.1 can be placed in separate modules if necessary, each compiled for the specific instruction set.

13.6 CPU dispatching in Gnu compiler

A feature called "Gnu indirect function" has been introduced in Linux and supported by the Gnu utilities in 2010. This feature is intended for CPU dispatching and is used in the Gnu C library. It requires support from both compiler, linker and loader (requires binutils version 2.20, glibc version 2.11 ifunc branch).

This feature uses an ordinary procedure linkage table (PLT) in the following way: There are two or more versions of the same function, each optimized for a particular CPU or other hardware conditions. A dispatcher function decides which function to use and returns a pointer to the desired function. The PLT entry initially points to the dispatcher function. When the program is loaded, the loader calls the dispatcher function and replaces the PLT entry with the pointer it gets from the dispatcher function. This will make any call to the function go to the desired version. Note that the dispatcher function is usually called before the program starts running and before any constructors are called. Therefore, the dispatcher function cannot rely on anything else being initialized. The dispatcher function will most likely be called, even if the dispatched function is never called.

Unfortunately, the syntax described in the Gnu manual currently doesn't works (gcc v. 4.5.2, July 2011). Instead, the following work-around can be used:

```
// Example 13.2. CPU dispatching in Gnu compiler
// Same as example 13.1, Requires binutils version 2.20 or later

// Header file for InstructionSet()
#include "asmlib.h"

// Lowest version
int CriticalFunction_386(int parm1, int parm2) {...}

// SSE2 version
int CriticalFunction_SSE2(int parm1, int parm2) {...}

// AVX version
int CriticalFunction_AVX(int parm1, int parm2) {...}

// Prototype for the common entry point
extern "C" int CriticalFunction ();
__asm__ (".type CriticalFunction, @gnu_indirect_function");

// Make the dispatcher function.
typeof(CriticalFunction) * CriticalFunctionDispatch(void)
   __asm__ ("CriticalFunction");
typeof(CriticalFunction) * CriticalFunctionDispatch(void)
{
   // Returns a pointer to the desired function version

   // Get supported instruction set, using asmlib library
   int level = InstructionSet();

   // Set pointer to the appropriate version (May use a table
   // of function pointers if there are many branches):
   if (level >= 11)
   {  // AVX supported
      return &CriticalFunction_AVX;
   }
   if (level >= 4)
   {  // SSE2 supported
      return &CriticalFunction_SSE2;
   }
   // Default version
   return &CriticalFunction_386;
}
```

```
int main()
{
   int a, b, c;
   ...

   // Call critical function
   a = CriticalFunction(b, c);

   ...
   return 0;
}
```

The indirect function feature is used in the Gnu C function library for a few functions that are particularly critical.

13.7 CPU dispatching in Intel compiler

Intel compilers have a feature for making multiple versions of a function for different Intel CPUs. It uses the *dispatch on every call* method. When the function is called, a dispatch is made to the desired version of the function. The automatic dispatching can be made for all suitable functions in a module by compiling the module with, e.g. the option `/QaxAVX` or `-axAVX`. This will make multiple versions even of functions that are not critical. It is possible to do the dispatching only for speed-critical functions by using the directive `__declspec(cpu_dispatch(...))`. See the Intel C++ Compiler Documentation for details. Note that the CPU dispatch mechanism in the Intel compiler works only for Intel CPUs, not for other brands of CPUs such as AMD and VIA. The next section (page 134) shows a way to work around this limitation and other flaws in the CPU detection mechanism.

The CPU dispatch mechanism in the Intel compiler is less efficient than the Gnu compiler mechanism because it makes dispatching on every call of the critical function. In some cases, the Intel mechanism executes a series of branches every time the function is called, while the Gnu mechanism stores a pointer to the desired version in a procedure linkage table. If a dispatched function calls another dispatched function then the dispatch branch of the latter is executed even though the CPU-type is already known at this place. This can be avoided by inlining the latter function, but it may be better to do the CPU dispatching explicitly as in example 13.1 page 130.

The Intel compilers and function libraries have features for automatic CPU dispatching. Many Intel library functions have several versions for different processors and instruction sets. Likewise, the compiler can automatically generate multiple versions of the user-written code with automatic CPU dispatching.

Unfortunately, the CPU detection mechanism in Intel compilers has several flaws:

- The best possible version of the code is chosen only when running on an Intel processor. The CPU dispatcher checks whether the processor is an Intel before it checks which instruction set it supports. An inferior version of the code is selected if the processor is not an Intel, even if the processor is compatible with a better version of the code. This can lead to a dramatic degradation of performance on AMD and VIA processors.

- Explicit CPU dispatching works only with Intel processors. A non-Intel processor makes the dispatcher signal an error simply by performing an illegal operation that crashes the program.

- The CPU dispatcher does not check if XMM registers are supported by the operating system. It will crash on old operating systems that do not support SSE.

Several function libraries published by Intel have similar CPU dispatch mechanisms, and some of these also treat non-Intel CPUs in a suboptimal way.

The fact that the Intel CPU dispatcher treats non-Intel CPUs in a suboptimal way has become a serious legal issue. See my blog for details.

The behavior of the Intel compiler puts the programmer in a bad dilemma. You may prefer to use the Intel compiler because it has many advanced optimizing features, and you may want to use the well optimized Intel function libraries, but who would like to put a tag on a program saying that it doesn't work well on non-Intel machines?

Possible solutions to this problem are the following:

- Compile for a specific instruction set, e.g. `/arch:SSE2`. The compiler will produce the optimal code for this instruction set and insert only the SSE2 version of most library functions without CPU dispatching. Test if the program runs satisfactorily on a non-Intel CPU. If not, then it may be necessary to replace the CPU detection function as described below. The program will not be compatible with old microprocessors that do not have the selected instruction set.

- Make two or more versions of the most critical part of the code and compile them separately with the appropriate instruction set specified. Insert an explicit CPU dispatching in the code to call the version that fits the microprocessor it is running on.

- Replace or bypass the CPU detection function of the Intel compiler. This method is described below.

- Make calls directly to the CPU-specific versions of the library functions. The CPU-specific functions have names with suffixes such as e.g. `.R`. for AVX. These suffixes are listed in table 19 in manual 5: calling conventions. The dot in the function name is not allowed in C++ so you need to use assembly code or use objconv or a similar utility for modifying the name in the object file.

- Use another function library that works well on all brands of CPUs.

The performance on non-Intel processors can be improved by using one or more of the above methods if the most time-consuming part of the program contains automatic CPU dispatching or memory-intensive functions such as `memcpy`, `memmove`, `memset`, or mathematical functions such as `pow`, `log`, `exp`, `sin`, etc.

Overriding the Intel CPU detection function

In some cases, there are two versions of the CPU detection function, one that discriminates between CPU brands, and one that doesn't.

The undocumented Intel library function `__intel_cpu_features_init()` sets the variable `__intel_cpu_feature_indicator` where each bit indicates a specific CPU feature on Intel CPU's. Another function `__intel_cpu_features_init_x()` does the same without discriminating between CPU brands and similarly sets the variable `__intel_cpu_feature_indicator_x`. You can bypass the check for CPU brand simply by setting these variables to zero and then call `__intel_cpu_features_init_x()`.

In other cases, it is possible to replace the CPU detection function in Intel function libraries and compiler-generated code by making another function with the same name. In the Windows operating system, this requires static linking (e.g. option `/MT`). In Linux and Mac systems, this may work with both static and dynamic linking.

The file http://www.agner.org/optimize/asmlib.zip contains complete code examples for these methods.

If you are using an Intel compiler, then make sure the startup code and `main()` are compiled without any option that limits the CPU brand. Critical parts of the code can then be placed in a separate C or C++ file and compiled for the desired instruction set. If the CPU brand check is bypassed by any of these methods then the critical part can run optimally on any brand of CPU.

These methods also work when Intel libraries are used with other compilers. This includes the libraries named MKL, VML and SVML. The IPP library does not need any patch.

Note that these methods are based on my own research, not on publicly available information. They have worked well in tests on Intel compiler versions 7 through 14, with some changes for each version. The examples are intended to work in both Windows and Linux, 32-bit and 64-bit. They have not been tested in Mac systems.

14 Specific optimization topics

14.1 Use lookup tables

Reading a value from a table of constants is very fast if the table is cached. Usually it takes only a few clock cycles to read from a table in the level-1 cache. We can take advantage of this fact by replacing a function call with a table lookup if the function has only a limited number of possible inputs.

Let's take the integer factorial function ($n!$) as an example. The only allowed inputs are the integers from 0 to 12. Higher inputs give overflow and negative inputs give infinity. A typical implementation of the factorial function looks like this:

```
// Example 14.1a
int factorial (int n) {          // n!
   int i, f = 1;
   for (i = 2; i <= n; i++) f *= i;
   return f;
}
```

This calculation requires `n-1` multiplications, which can take quite a long time. It is more efficient to use a lookup table:

```
// Example 14.1b
int factorial (int n) {          // n!
   // Table of factorials:
   const int FactorialTable[13] = {1, 1, 2, 6, 24, 120, 720,
      5040, 40320, 362880, 3628800, 39916800, 479001600};
   if ((unsigned int)n < 13) {   // Bounds checking (see page 137)
      return FactorialTable[n];  // Table lookup
   }
   else {
      return 0;                  // return 0 if out of range
   }
}
```

This implementation uses a lookup table instead of calculating the value each time the function is called. I have added a bounds check on `n` here because the consequence of `n` being out of range is possibly more serious when `n` is an array index than when `n` is a loop count. The method of bounds checking is explained below on page 137.

The table should be declared `const` in order to enable constant propagation and other optimizations. You may declare the function `inline`.

Replacing a function with a lookup table is advantageous in most cases where the number of possible inputs is limited and there are no cache problems. It is not advantageous to use a lookup table if you expect the table to be evicted from the cache between each call, and the time it takes to calculate the function is less than the time it takes to reload the value from memory plus the costs to other parts of the program of occupying a cache line.

Table lookup cannot be vectorized with the current instruction set. Do not use lookup tables if this prevents a faster vectorized code.

Storing something in static memory can cause caching problems because static data are likely to be scattered around at different memory addresses. If caching is a problem then it may be useful to copy the table from static memory to stack memory outside the innermost loop. This is done by declaring the table inside a function but outside the innermost loop and without the `static` keyword:

```
// Example 14.1c
void CriticalInnerFunction () {
   // Table of factorials:
   const int FactorialTable[13] = {1, 1, 2, 6, 24, 120, 720,
      5040, 40320, 362880, 3628800, 39916800, 479001600};
   ...
   int i, a, b;
   // Critical innermost loop:
   for (i = 0; i < 1000; i++) {
      ...
      a = FactorialTable[b];
      ...
   }
}
```

The `FactorialTable` in example 14.1c is copied from static memory to the stack when `CriticalInnerFunction` is called. The compiler will store the table in static memory and insert a code that copies the table to stack memory at the start of the function. Copying the table takes extra time, of course, but this is permissible when it is outside the critical innermost loop. The loop will use the copy of the table that is stored in stack memory which is contiguous with other local variables and therefore likely to be cached more efficiently than static memory.

If you don't care to calculate the table values by hand and insert the values in the code then you may of course make the program do the calculations. The time it takes to calculate the table is not significant as long as it is done only once. One may argue that it is safer to calculate the table in the program than to type in the values because a typo in a hand-written table may go undetected.

The principle of table lookup can be used in any situation where a program chooses between two or more constants. For example, a branch that chooses between two constants can be replaced by a table with two entries. This may improve the performance if the branch is poorly predictable. For example:

```
// Example 14.2a
float a;  int b;
```

```
a = (b == 0) ? 1.0f : 2.5f;
```

If we assume that `b` is always 0 or 1 and that the value is poorly predictable, then it is advantageous to replace the branch by a table lookup:

```
// Example 14.2b
float a;  int b;
const float OneOrTwo5[2] = {1.0f, 2.5f};
a = OneOrTwo5[b & 1];
```

Here, I have AND'ed `b` with `1` for the sake of security. `b & 1` is certain to have no other value than `0` or `1` (see page 138). This extra check on `b` can be omitted, of course, if the value of `b` is guaranteed to be `0` or `1`. Writing `a = OneOrTwo5[b!=0];` will also work, although slightly less efficiently. This method is inefficient, however, when `b` is a `float` or `double` because all the compilers I have tested implement `OneOrTwo5[b!=0]` as `OneOrTwo5[(b!=0) ? 1 : 0]` in this case so we don't get rid of the branch. It may seem illogical that the compiler uses a different implementation when `b` is floating point. The reason is, I guess, that compiler makers assume that floating point comparisons are more predictable than integer comparisons. The solution `a = 1.0f + b * 1.5f;` is efficient when `b` is a `float`, but not if `b` is an integer because the integer-to-float conversion takes more time than the table lookup.

Lookup tables are particular advantageous as replacements for `switch` statements because `switch` statements often suffer from poor branch prediction. Example:

```
// Example 14.3a
int n;
switch (n) {
case 0:
   printf("Alpha");  break;
case 1:
   printf("Beta");   break;
case 2:
   printf("Gamma");  break;
case 3:
   printf("Delta");  break;
}
```

This can be improved by using a lookup table:

```
// Example 14.3b
int n;
char const * const Greek[4] = {
   "Alpha", "Beta", "Gamma", "Delta"
};
if ((unsigned int)n < 4) { // Check that index is not out of range
   printf(Greek[n]);
}
```

The declaration of the table has `const` twice because both the pointers and the texts they point to are constant.

14.2 Bounds checking

In C++, it is often necessary to check if an array index is out of range. This may typically look like this:

```
// Example 14.4a
const int size = 16; int i;
```

```
float list[size];
...
if (i < 0 || i >= size) {
   cout << "Error: Index out of range";
}
else {
   list[i] += 1.0f;
}
```

The two comparisons `i < 0` and `i >= size` can be replaced by a single comparison:

```
// Example 14.4b
if ((unsigned int)i >= (unsigned int)size) {
   cout << "Error: Index out of range";
}
else {
   list[i] += 1.0f;
}
```

A possible negative value of `i` will appear as a large positive number when `i` is interpreted as an unsigned integer and this will trigger the error condition. Replacing two comparisons by one makes the code faster because testing a condition is relatively expensive, while the type conversion generates no extra code at all.

This method can be extended to the general case where you want to check whether an integer is within a certain interval:

```
// Example 14.5a
const int min = 100, max = 110;  int i;
...
if (i >= min && i <= max) { ...
```

can be changed to:

```
// Example 14.5b
if ((unsigned int)(i - min) <= (unsigned int)(max - min)) { ...
```

There is an even faster way to limit the range of an integer if the length of the desired interval is a power of 2. Example:

```
// Example 14.6
float list[16]; int i;
...
list[i & 15] += 1.0f;
```

This needs a little explanation. The value of `i&15` is guaranteed to be in the interval from 0 to 15. If `i` is outside this interval, for example `i` = 18, then the `&` operator (bitwise and) will cut off the binary value of `i` to four bits, and the result will be 2. The result is the same as `i` modulo 16. This method is useful for preventing program errors in case the array index is out of range and we don't need an error message if it is. It is important to note that this method works only for powers of 2 (i.e. 2, 4, 8, 16, 32, 64, ...). We can make sure that a value is less than 2^n and not negative by AND'ing it with 2^n-1. The bitwise AND operation isolates the least significant *n* bits of the number and sets all other bits to zero.

14.3 Use bitwise operators for checking multiple values at once

The bitwise operators `&`, `|`, `^`, `~`, `<<`, `>>` can test or manipulate all the bits of an integer in one operation. For example, if each bit of a 32-bit integer has a particular meaning, then you can set multiple bits in a single operation using the `|` operator; you can clear or mask out multiple bits with the `&` operator; and you can toggle multiple bits with the `^` operator.

The & operator is also useful for testing multiple conditions in a single operation. Example:

```
// Example 14.7a. Testing multiple conditions
enum Weekdays {
   Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
};
Weekdays Day;
if (Day == Tuesday || Day == Wednesday || Day == Friday) {
   DoThisThreeTimesAWeek();
}
```

The if statement in this example has three conditions which are implemented as three branches. They can be joined into a single branch if the constants Sunday, Monday, etc. are defined as powers of 2:

```
// Example 14.7b. Testing multiple conditions using &
enum Weekdays {
   Sunday = 1, Monday = 2, Tuesday = 4, Wednesday = 8,
   Thursday = 0x10, Friday = 0x20, Saturday = 0x40
};
Weekdays Day;
if (Day & (Tuesday | Wednesday | Friday)) {
   DoThisThreeTimesAWeek();
}
```

By giving each constant a value that is a power of 2 in example 14.7b, we are in fact using each bit in Day for signifying one of the weekdays. The maximum number of constants we can define in this way is equal to the number of bits in an integer, usually 32. In 64-bit systems we can use 64-bit integers with hardly any loss of efficiency.

The expression (Tuesday | Wednesday | Friday) in example 14.7b is converted by the compiler to the value 0x2C so that the if condition can be calculated by a single & operation, which is very fast. The result of the & operation will be non-zero, and therefore count as true, if any of the bits for Tuesday, Wednesday or Friday is set in the variable Day.

Note the difference between the Boolean operators &&, ||, ! and the corresponding bitwise operators &, |, ~. The Boolean operators produce a single result, true (1) or false (0); and the second operand is evaluated only when needed. The bitwise operators produce 32 results when applied to 32-bit integers, and they always evaluate both operands. Nevertheless, the bitwise operators are calculated much faster than the Boolean operators because they do not use branches, provided that the operands are integer expressions rather than Boolean expressions.

There are lots of things you can do with bitwise operators using integers as Boolean vectors, and these operations are very fast. This can be useful in programs with many Boolean expressions. Whether the constants are defined with enum, const, or #define makes no difference for the performance.

14.4 Integer multiplication

Integer multiplication takes longer time than addition and subtraction (3 - 10 clock cycles, depending on the processor). Optimizing compilers will often replace integer multiplication by a constant with a combination of additions and shift operations. Multiplying by a power of 2 is faster than multiplying by other constants because it can be done as a shift operation. For example, a * 16 is calculated as a << 4, and a * 17 is calculated as (a << 4) + a.

You can take advantage of this by preferably using powers of 2 when multiplying with a constant. The compilers also have fast ways of multiplying by 3, 5 and 9.

Multiplications are done implicitly when calculating the address of an array element. In some cases this multiplication will be faster when the factor is a power of 2. Example:

```
// Example 14.8
const int rows = 10, columns = 8;
float matrix[rows][columns];
int i, j;
int order(int x);
...
for (i = 0; i < rows; i++) {
   j = order(i);
   matrix[j][0] = i;
}
```

Here, the address of `matrix[j][0]` is calculated internally as `(int)&matrix[0][0] + j * (columns * sizeof(float))`. Now, the factor to multiply `j` by is `(columns * sizeof(float)) = 8 * 4 = 32`. This is a power of 2, so the compiler can replace `j * 32` with `j << 5`. If `columns` had not been a power of 2 then the multiplication would take longer time. It can therefore be advantageous to make the number of columns in a matrix a power of 2 if the rows are accessed in a non-sequential order.

The same applies to an array of structure or class elements. The size of each object should preferably be a power of 2 if the objects are accessed in a non-sequential order. Example:

```
// Example 14.9
struct S1 {
   int a;
   int b;
   int c;
   int UnusedFiller;
};
int order(int x);
const int size = 100;
S1 list[size];  int i, j;
...
for (i = 0; i < size; i++) {
   j = order(i);
   list[j].a = list[j].b + list[j].c;
}
```

Here, we have inserted `UnusedFiller` in the structure to make sure its size is a power of 2 in order to make the address calculation faster.

The advantage of using powers of 2 applies only when elements are accessed in non-sequential order. If the code in example 14.8 and 14.9 is changed so that it has `i` instead of `j` as index then the compiler can see that the addresses are accessed in sequential order and it can calculate each address by adding a constant to the preceding one (see page 72). In this case it doesn't matter if the size is a power of 2 or not.

The advise of using powers of 2 does not apply to very big data structures. On the contrary, you should by all means avoid powers of 2 if a matrix is so big that caching becomes a problem. If the number of columns in a matrix is a power of 2 and the matrix is bigger than the cache then you can get very expensive cache contentions, as explained on page 98.

14.5 Integer division

Integer division takes much longer time than addition, subtraction and multiplication (27 - 80 clock cycles for 32-bit integers, depending on the processor).

Integer division by a power of 2 can be done with a shift operation, which is much faster.

Division by a constant is faster than division by a variable because optimizing compilers can compute a / b as a * (2^n / b) >> n with a suitable choice of n. The constant (2^n / b) is calculated in advance and the multiplication is done with an extended number of bits. The method is somewhat more complicated because various corrections for sign and rounding errors must be added. This method is described in more detail in manual 2: "Optimizing subroutines in assembly language". The method is faster if the dividend is unsigned.

The following guidelines can be used for improving code that contains integer division:

- Integer division by a constant is faster than division by a variable. Make sure the value of the divisor is known at compile time.
- Integer division by a constant is faster if the constant is a power of 2
- Integer division by a constant is faster if the dividend is unsigned

Examples:

```
// Example 14.10
int a, b, c;
a = b / c;                // This is slow
a = b / 10;               // Division by a constant is faster
a = (unsigned int)b / 10; // Still faster if unsigned
a = b / 16;               // Faster if divisor is a power of 2
a = (unsigned int)b / 16; // Still faster if unsigned
```

The same rules apply to modulo calculations:

```
// Example 14.11
int a, b, c;
a = b % c;                // This is slow
a = b % 10;               // Modulo by a constant is faster
a = (unsigned int)b % 10; // Still faster if unsigned
a = b % 16;               // Faster if divisor is a power of 2
a = (unsigned int)b % 16; // Still faster if unsigned
```

You can take advantage of these guidelines by using a constant divisor that is a power of 2 if possible and by changing the dividend to unsigned if you are sure that it will not be negative.

The method described above can still be used if the value of the divisor is not known at compile time, but the program is dividing repeatedly with the same divisor. In this case you have to do the necessary calculations of (2^n / b) etc. at compile time. The function library at www.agner.org/optimize/asmlib.zip contains various functions for these calculations.

Division of a loop counter by a constant can be avoided by rolling out the loop by the same constant. Example:

```
// Example 14.12a
int list[300];
int i;
for (i = 0;  i < 300;  i++) {
   list[i] += i / 3;
```

```
}
```

This can be replaced with:

```
// Example 14.12b
int list[300];
int i, i_div_3;
for (i = i_div_3 = 0;  i < 300;  i += 3, i_div_3++) {
   list[i]   += i_div_3;
   list[i+1] += i_div_3;
   list[i+2] += i_div_3;
}
```

A similar method can be used to avoid modulo operations:

```
// Example 14.13a
int list[300];
int i;
for (i = 0;  i < 300;  i++) {
   list[i] = i % 3;
}
```

This can be replaced with:

```
// Example 14.13b
int list[300];
int i;
for (i = 0;  i < 300;  i += 3) {
   list[i]   = 0;
   list[i+1] = 1;
   list[i+2] = 2;
}
```

The loop unrolling in example 14.12b and 14.13b works only if the loop count is divisible by the unroll factor. If not, then you must do the extra operations outside the loop:

```
// Example 14.13c
int list[301];
int i;
for (i = 0;  i < 301;  i += 3) {
   list[i]   = 0;
   list[i+1] = 1;
   list[i+2] = 2;
}
list[300] = 0;
```

14.6 Floating point division

Floating point division takes much longer time than addition, subtraction and multiplication (20 - 45 clock cycles).

Floating point division by a constant should be done by multiplying with the reciprocal:

```
// Example 14.14a
double a, b;
a = b / 1.2345;
```

Change this to:

```
// Example 14.14b
double a, b;
```

```
a = b * (1. / 1.2345);
```

The compiler will calculate `(1./1.2345)` at compile time and insert the reciprocal in the code, so you will never spend time doing the division. Some compilers will replace the code in example 14.14a with 14.14b automatically but only if certain options are set to relax floating point precision (see page 74). It is therefore safer to do this optimization explicitly.

Divisions can sometimes be eliminated completely. For example:

```
// Example 14.15a
if (a > b / c)
```

can sometimes be replaced by

```
// Example 14.15b
if (a * c > b)
```

But beware of the pitfalls here: The inequality sign must be reversed if `c` < 0. The division is inexact if `b` and `c` are integers, while the multiplication is exact.

Multiple divisions can be combined. For example:

```
// Example 14.16a
double y, a1, a2, b1, b2;
y = a1/b1 + a2/b2;
```

Here we can eliminate one division by making a common denominator:

```
// Example 14.16b
double y, a1, a2, b1, b2;
y = (a1*b2 + a2*b1) / (b1*b2);
```

The trick of using a common denominator can even be used on completely independent divisions. Example:

```
// Example 14.17a
double a1, a2, b1, b2, y1, y2;
y1 = a1 / b1;
y2 = a2 / b2;
```

This can be changed to:

```
// Example 14.17b
double a1, a2, b1, b2, y1, y2, reciprocal_divisor;
reciprocal_divisor = 1. / (b1 * b2);
y1 = a1 * b2 * reciprocal_divisor;
y2 = a2 * b1 * reciprocal_divisor;
```

14.7 Don't mix float and double

Floating point calculations usually take the same time regardless of whether you are using single precision or double precision, but there is a penalty for mixing single and double precision in programs compiled for 64-bit operating systems and programs compiled for the instruction set SSE2 or later. Example:

```
// Example 14.18a
float a, b;
a = b * 1.2;       // Mixing float and double is bad
```

The C/C++ standard specifies that all floating point constants are double precision by default, so `1.2` in this example is a double precision constant. It is therefore necessary to convert `b` from single precision to double precision before multiplying with the double precision constant and then convert the result back to single precision. These conversions take a lot of time. You can avoid the conversions and make the code up to 5 times faster either by making the constant single precision or by making `a` and `b` double precision:

```
// Example 14.18b
float a, b;
a = b * 1.2f;       // everything is float

// Example 14.18c
double a, b;
a = b * 1.2;        // everything is double
```

There is no penalty for mixing different floating point precisions when the code is compiled for old processors without the SSE2 instruction set, but it may be preferable to keep the same precision in all operands in case the code is later ported to another platform.

14.8 Conversions between floating point numbers and integers

Conversion from floating point to integer

According to the standards for the C++ language, all conversions from floating point numbers to integers use truncation towards zero, rather than rounding. This is unfortunate because truncation takes much longer time than rounding unless the SSE2 instruction set is used. It is recommended to enable the SSE2 instruction set if possible. SSE2 is always enabled in 64-bit mode.

A conversion from floating point to integer without SSE2 typically takes 40 clock cycles. If you cannot avoid conversions from `float` or `double` to `int` in the critical part of the code, then you may improve efficiency by using rounding instead of truncation. This is approximately three times faster. The logic of the program may need modification to compensate for the difference between rounding and truncation.

Efficient conversion from `float` or `double` to integer can be done with the functions `lrintf` and `lrint`. Unfortunately, these functions are missing in many commercial compilers due to controversies over the C99 standard. An implementation of the `lrint` function is given in example 14.19 below. The function rounds a floating point number to the nearest integer. If two integers are equally near then the even integer is returned. There is no check for overflow. This function is intended for 32-bit Windows and 32-bit Linux with Microsoft, Intel and Gnu compilers.

```
// Example 14.19
static inline int lrint (double const x) { // Round to nearest integer
   int n;
#if defined(__unix__) || defined(__GNUC__)
   // 32-bit Linux, Gnu/AT&T syntax:
   __asm ("fldl %1 \n fistpl %0 " : "=m"(n) : "m"(x) : "memory" );
#else
   // 32-bit Windows, Intel/MASM syntax:
   __asm fld qword ptr x;
   __asm fistp dword ptr n;
#endif
   return n;}
```

This code will work only on Intel/x86-compatible microprocessors. The function is also available in the function library at www.agner.org/optimize/asmlib.zip.

The following example shows how to use the `lrint` function:

```
// Example 14.20
double d = 1.6;
int a, b;
a = (int)d;       // Truncation is slow. Value of a will be 1
b = lrint(d);     // Rounding is fast. Value of b will be 2
```

In 64-bit mode or when the SSE2 instruction set is enabled there is no difference in speed between rounding and truncation. The missing functions can be implemented as follows in 64-bit mode or when the SSE2 instruction set is enabled:

```
// Example 14.21.  // Only for SSE2 or x64
#include <emmintrin.h>

static inline int lrintf (float const x) {
   return _mm_cvtss_si32(_mm_load_ss(&x));}

static inline int lrint (double const x) {
   return _mm_cvtsd_si32(_mm_load_sd(&x));}
```

The code in example 14.21 is faster than other methods of rounding, but neither faster nor slower than truncation when the SSE2 instruction set is enabled.

Conversion from integer to floating point

Conversion of integers to floating point is faster than from floating point to integer. The conversion time is typically between 5 and 20 clock cycles. It may in some cases be advantageous to do simple integer calculations in floating point variables in order to avoid conversions from integer to floating point.

Conversion of unsigned integers to floating point numbers is less efficient than signed integers. It is more efficient to convert unsigned integers to signed integers before conversion to floating point if the conversion to signed integer doesn't cause overflow. Example:

```
// Example 14.22a
unsigned int u;  double d;
d = u;
```

If you are certain that `u` < 2^{31} then convert it to signed before converting to floating point:

```
// Example 14.22b
unsigned int u;  double d;
d = (double)(signed int)u;
```

14.9 Using integer operations for manipulating floating point variables

Floating point numbers are stored in a binary representation according to the IEEE standard 754 (1985). This standard is used in almost all modern microprocessors and operating systems (but not in some very old DOS compilers).

The representation of `float`, `double` and `long double` reflects the floating point value written as $\pm 2^{eee} \cdot 1.fffff$, where $\pm$ is the sign, `eee` is the exponent, and `fffff` is the binary decimals of the fraction. The sign is stored as a single bit which is 0 for positive and 1 for negative numbers. The exponent is stored as a biased binary integer, and the fraction is stored as the binary digits. The exponent is always normalized, if possible, so that the value before the decimal point is 1. This '1' is not included in the representation, except in the `long double` format. The formats can be expressed as follows:

```
struct Sfloat {
   unsigned int fraction : 23; // fractional part
   unsigned int exponent :  8; // exponent + 0x7F
   unsigned int sign     :  1; // sign bit
};

struct Sdouble {
   unsigned int fraction : 52; // fractional part
   unsigned int exponent : 11; // exponent + 0x3FF
   unsigned int sign     :  1; // sign bit
};

struct Slongdouble {
   unsigned int fraction : 63; // fractional part
   unsigned int one      :  1; // always 1 if nonzero and normal
   unsigned int exponent : 15; // exponent + 0x3FFF
   unsigned int sign     :  1; // sign bit
};
```

The values of nonzero floating point numbers can be calculated as follows:

$$floatvalue = (-1)^{sign} \cdot 2^{exponent-127} \cdot \left(1 + fraction \cdot 2^{-23}\right),$$
$$doublevalue = (-1)^{sign} \cdot 2^{exponent-1023} \cdot \left(1 + fraction \cdot 2^{-52}\right),$$
$$longdoublevalue = (-1)^{sign} \cdot 2^{exponent-16383} \cdot \left(one + fraction \cdot 2^{-63}\right).$$

The value is zero if all bits except the sign bit are zero. Zero can be represented with or without the sign bit.

The fact that the floating point format is standardized allows us to manipulate the different parts of the floating point representation directly with the use of integer operations. This can be an advantage because integer operations are faster than floating point operations. You should use such methods only if you are sure you know what you are doing. See the end of this section for some caveats.

We can change the sign of a floating point number simply by inverting the sign bit:

```
// Example 14.23
union {
   float f;
   int i;
} u;
u.i ^= 0x80000000; // flip sign bit of u.f
```

We can take the absolute value by setting the sign bit to zero:

```
// Example 14.24
union {
   float f;
   int i;
} u;
u.i &= 0x7FFFFFFF; // set sign bit to zero
```

We can check if a floating point number is zero by testing all bits except the sign bit:

```
// Example 14.25
union {
   float f;
   int i;
} u;
if (u.i & 0x7FFFFFFF) { // test bits 0 - 30
```

```
   // f is nonzero
}
else {
   // f is zero
}
```

We can multiply a nonzero floating point number by 2^n by adding `n` to the exponent:

```
// Example 14.26
union {
   float f;
   int i;
} u;
int n;
if (u.i & 0x7FFFFFFF) { // check if nonzero
   u.i += n << 23;      // add n to exponent
}
```

Example 14.26 does not check for overflow and works only for positive `n`. You can divide by 2^n by subtracting `n` from the exponent if there is no risk of underflow.

The fact that the representation of the exponent is biased allows us to compare two positive floating point numbers simply by comparing them as integers:

```
// Example 14.27
union {
   float f;
   int i;
} u, v;
if (u.i > v.i) {
   // u.f > v.f if both positive
}
```

Example 14.27 assumes that we know that `u.f` and `v.f` are both positive. It will fail if both are negative or if one is 0 and the other is -0 (zero with sign bit set).

We can shift out the sign bit to compare absolute values:

```
// Example 14.28
union {
   float f;
   unsigned int i;
} u, v;
if (u.i * 2 > v.i * 2) {
   // abs(u.f) > abs(v.f)
}
```

The multiplication by 2 in example 14.28 will shift out the sign bit so that the remaining bits represent a monotonically increasing function of the absolute value of the floating point number.

We can convert an integer in the interval $0 <= n < 2^{23}$ to a floating point number in the interval [1.0, 2.0) by setting the fraction bits:

```
// Example 14.29
union {
   float f;
   int i;
} u;
int n;
u.i = (n & 0x7FFFFF) | 0x3F800000; // Now 1.0 <= u.f < 2.0
```

This method is useful for random number generators.

In general, it is faster to access a floating point variable as an integer if it is stored in memory, but not if it is a register variable. The union forces the variable to be stored in memory, at least temporarily. Using the methods in the above examples will therefore be a disadvantage if other nearby parts of the code could benefit from using registers for the same variables.

In these examples we are using unions rather than type casting of pointers because this method is safer. Type casting of pointers may not work on compilers that rely on the strict aliasing rule of standard C, specifying that pointers of different types cannot point to the same object, except for `char` pointers.

The above examples all use single precision. Using double precision in 32-bit systems gives rise to some extra complications. A double is represented with 64 bits, but 32-bit systems do not have inherent support for 64-bit integers. Many 32-bit systems allow you to define 64-bit integers, but they are in fact represented as two 32-bit integers, which is less efficient. You may use the upper 32 bits of a `double` which gives access to the sign bit, the exponent, and the most significant part of the fraction. For example, to test the sign of a double:

```
// Example 14.23b
union {
   double d;
   int i[2];
} u;
if (u.i[1] < 0) {  // test sign bit
   // u.d is negative or -0
}
```

It is not recommended to modify a double by modifying only half of it, for example if you want to flip the sign bit in the above example with `u.i[1] ^= 0x80000000;` because this is likely to generate a store forwarding delay in the CPU (See manual 3: "The microarchitecture of Intel, AMD and VIA CPUs"). This can be avoided in 64-bit systems by using a 64-bit integer rather than two 32-bit integers to alias upon the double.

Another problem with accessing 32 bits of a 64-bit double is that it is not portable to systems with big-endian storage. Example 14.23b and 14.30 will therefore need modification if implemented on other platforms with big-endian storage. All x86 platforms (Windows, Linux, BSD, Intel-based Mac OS, etc.) have little-endian storage, but other systems may have big endian storage (e.g. PowerPC).

We can make an approximate comparison of doubles by comparing bits 32-62. This can be useful for finding the numerically largest element in a matrix for use as pivot in a Gauss elimination. The method in example 14.28 can be implemented like this in a pivot search:

```
// Example 14.30
const int size = 100;
// Array of 100 doubles:
union {double d; unsigned int u[2]} a[size];
unsigned int absvalue, largest_abs = 0;
int i, largest_index = 0;
for (i = 0; i < size; i++) {
   // Get upper 32 bits of a[i] and shift out sign bit:
   absvalue = a[i].u[1] * 2;
   // Find numerically largest element (approximately):
   if (absvalue > largest_abs) {
      largest_abs = absvalue;
      largest_index = i;
   }
```

```
}
```

Example 14.30 finds the numerically largest element in an array, or approximately so. It may fail to distinguish elements with a relative difference less than 2^{-20}, but this is sufficiently accurate for the purpose of finding a suitable pivot element. The integer comparison is likely to be faster than a floating point comparison. On big endian systems you have to replace `u[1]` by `u[0]`.

14.10 Mathematical functions

The most common mathematical functions such as logarithms, exponential functions, trigonometric functions, etc. are implemented in hardware in the x86 CPUs. However, a software implementation is faster than the hardware implementation in most cases when the SSE2 instruction set is available. The best compilers use the software implementation if the SSE2 instruction set is enabled.

The advantage of using a software implementation rather than a hardware implementation of these functions is higher for single precision than for double precision. But the software implementation is faster than the hardware implementation in most cases, even for double precision.

You may use the Intel math function library with a different compiler by including the library `libmmt.lib` and the header file `mathimf.h` that come with the Intel C++ compiler. This library contains many useful mathematical functions. A lot of advanced mathematical functions are supplied in Intel's Math Kernel Library, available from www.intel.com. (See also page 122). The AMD math core library contains similar functions, but less optimized.

Note that the Intel function libraries do not use the best possible instruction set when running on non-Intel processors (see page 133 for how to overcome this limitation).

14.11 Static versus dynamic libraries

Function libraries can be implemented either as static link libraries (`*.lib`, `*.a`) or dynamic link libraries, also called shared objects (`*.dll`, `*.so`). The mechanism of static linking is that the linker extracts the functions that are needed from the library file and copies them into the executable file. Only the executable file needs to be distributed to the end user.

Dynamic linking works differently. The link to a function in a dynamic library is resolved when the library is loaded or at run time. Therefore, both the executable file and one or more dynamic libraries are loaded into memory when the program is run. Both the executable file and all the dynamic libraries need to be distributed to the end user.

The advantages of using static linking rather than dynamic linking are:

- Static linking includes only the part of the library that is actually needed by the application, while dynamic linking makes the entire library (or at least a large part of it) load into memory even when just a single function from the library is needed.

- All the code is included in a single executable file when static linking is used. Dynamic linking makes it necessary to load several files when the program is started.

- It takes longer time to call a function in a dynamic library than in a static link library because it needs an extra jump through a pointer in an import table and possibly also a lookup in a procedure linkage table (PLT).

- The memory space becomes more fragmented when the code is distributed between multiple dynamic libraries. The dynamic libraries are loaded at round memory addresses

divisible by the memory page size (4096). This will make all dynamic libraries contend for the same cache lines. This makes code caching and data caching less efficient.

- Dynamic libraries are less efficient in some systems because of the needs of position-independent code, see below.

- Installing a second application that uses a newer version of the same dynamic library can change the behavior of the first application if dynamic linking is used, but not if static linking is used.

The advantages of dynamic linking are:

- Multiple applications running simultaneously can share the same dynamic libraries without the need to load more than one instance of the library into memory. This is useful on servers that run many processes simultaneously. Actually, only the code section and read-only data sections can be shared. Any writable data section needs one instance for each process.

- A dynamic library can be updated to a new version without the need to update the program that calls it.

- A dynamic library can be called from programming languages that do not support static linking.

- A dynamic library can be useful for making plug-ins that add functionality to an existing program.

Weighing the above advantages of each method, it is clear that static linking is preferable for speed-critical functions. Many function libraries are available in both static and dynamic versions. It is recommended to use the static version if speed is important.

Some systems allow lazy binding of function calls. The principle of lazy binding is that the address of a linked function is not resolved when the program is loaded, but waits until the first time the function is called. Lazy binding can be useful for large libraries where only few of the functions are actually called in a single session. But lazy binding definitely degrades performance for the functions that are called. A considerable delay comes when a function is called for the first time because it needs to load the dynamic linker.

The delay on lazy binding leads to a usability problem in interactive programs because the response time to e.g. a menu click becomes inconsistent and sometimes unacceptably long. Lazy binding should therefore be used only for very large libraries.

The memory address at which a dynamic library is loaded cannot be determined in advance, because a fixed address might clash with another dynamic library requiring the same address. There are two commonly used methods for dealing with this problem:

1. Relocation. All pointers and addresses in the code are modified, if necessary, to fit the actual load address. Relocation is done by the linker and the loader.

2. Position-independent code. All addresses in the code are relative to the current position.

Windows DLLs use relocation. The DLLs are relocated by the linker to a specific load address. If this address is not vacant then the DLL is relocated (rebased) once more by the loader to a different address. A call from the main executable to a function in a DLL goes through an import table or a pointer. A variable in a DLL can be accessed from main through an imported pointer, but this feature is seldom used. It is more common to

exchange data or pointers to data through function calls. Internal references to data within the DLL use absolute references in 32 bit mode and mostly relative references in 64 bit mode. The latter is slightly more efficient because relative references do not need relocation at load time.

Shared objects in Unix-like systems use position-independent code by default. This is less efficient than relocation, especially in 32-bit mode. The next chapter describes how this works and suggests methods for avoiding the costs of position-independent code.

14.12 Position-independent code

Shared objects in Linux, BSD and Mac systems normally use the so-called position-independent code. The name "position-independent code" actually implies more than it says. A code that is compiled as position-independent has the following features:

- The code section contains no absolute addresses that need relocation, but only self-relative addresses. Therefore, the code section can be loaded at an arbitrary memory address and shared between multiple processes.

- The data section is not shared between multiple processes because it often contains writeable data. Therefore, the data section may contain pointers or addresses that need relocation.

- All public functions and public data can be overridden in Linux and BSD. If a function in the main executable has the same name as a function in a shared object, then the version in main will take precedence, not only when called from main, but also when called from the shared object. Likewise, when a global variable in main has the same name as a global variable in the shared object, then the instance in main will be used, even when accessed from the shared object. This so-called symbol interposition is intended to mimic the behavior of static libraries. A shared object has a table of pointers to its functions, called procedure linkage table (PLT) and a table of pointers to its variables called global offset table (GOT) in order to implement this "override" feature. All accesses to functions and public variables go through the PLT and GOT.

The symbol interposition feature that allows overriding of public functions and data in Linux and BSD comes at a high price, and in most libraries it is never used. Whenever a function in a shared object is called, it is necessary to look up the function address in the procedure linkage table (PLT). And whenever a public variable in a shared object is accessed, it is necessary to first look up the address of the variable in the global offset table (GOT). These table lookups are needed even when the function or variable is accessed from within the same shared object. Obviously, all these table lookup operations slow down the execution considerably. A more detailed discussion can be found at http://www.macieira.org/blog/2012/01/sorry-state-of-dynamic-libraries-on-linux/

Another serious burden is the calculation of self-relative references in 32-bit mode. The 32-bit x86 instruction set has no instruction for self-relative addressing of data. The code goes through the following steps to access a public data object: (1) get its own address through a function call. (2) find the GOT through a self-relative address. (3) look up the address of the data object in the GOT, and finally (4) access the data object through this address. Step (1) is not needed in 64-bit mode because the x86-64 instruction set supports self-relative addressing.

In 32-bit Linux and BSD, the slow GOT lookup process is used for all static data, including local data that don't need the "override" feature. This includes static variables, floating point constants, string constants, and initialized arrays. I have no explanation why this delaying process is used when it is not needed.

Obviously, the best way to avoid the burdensome position-independent code and table lookup is to use static linking, as explained in the previous chapter (page 149). In the cases where dynamic linking cannot be avoided, there are various ways to avoid the time-consuming features of the position-independent code. These workaround methods depend on the system, as explained below.

Shared objects in 32 bit Linux

Shared objects are normally compiled with the option `-fpic` according to the Gnu compiler manual. This option makes the code section position-independent, makes a PLT for all functions and a GOT for all public and static data.

It is possible to compile a shared object without the `-fpic` option. Then we get rid of all the problems mentioned above. Now the code will run faster because we can access internal variables and internal functions in a single step rather than the complicated address calculation and table lookup mechanisms explained above. A shared object compiled without `-fpic` is much faster, except perhaps for a very large shared object where most of the functions are never called. The disadvantage of compiling without `-fpic` in 32-bit Linux is that the loader will have more references to relocate, but these address calculations are done only once, while the runtime address calculations have to be done at every access. The code section needs one instance for each process when compiled without `-fpic` because the relocations in the code section will be different for each process. Obviously, we lose the ability to override public symbols, but this feature is rarely needed anyway.

You may preferably avoid global variables or hide them for the sake of portability to 64-bit mode, as explained below.

Shared objects in 64 bit Linux

The procedure to calculate self-relative addresses is much simpler in 64-bit mode because the 64-bit instruction set has support for relative addressing of data. The need for special position-independent code is smaller because relative addresses are often used by default anyway in 64-bit code. However, we still want to get rid of the GOT and PLT lookups for local references.

If we compile the shared object without `-fpic` in 64 bit mode, we encounter another problem. The compiler sometimes uses 32-bit absolute addresses, mainly for static arrays. This works in the main executable because it is sure to be loaded at an address below 2 GB, but not in a shared object which is typically loaded at a higher address which can't be reached with a 32-bit (signed) address. The linker will generate an error message in this case. The best solution is to compile with the option `-fpie` instead of `-fpic`. This will generate relative addresses in the code section, but it will not use GOT and PLT for internal references. Therefore, it will run faster than when compiled with `-fpic` and it will not have the disadvantages mentioned above for the 32-bit case. The `-fpie` option is less useful in 32-bit mode, where it still uses a GOT.

Another possibility is to compile with `-mcmodel=large`, but this will use full 64-bit addresses for everything, which is quite inefficient, and it will generate relocations in the code section so that it cannot be shared.

You can't have public variables in a 64-bit shared object made with option `-fpie` because the linker makes an error message when it sees a relative reference to a public variable where it expects a GOT entry. You can avoid this error by avoiding any public variables. All global variables (i.e. variables defined outside any function) should be hidden by using the declaration `"static"` or `"__attribute__((visibility ("hidden")))"`.

The gnu compiler version 5.1 and later has an option `-fno-semantic-interposition`, which makes it avoid the use of PLT and GOT look, but only for references within the same file. The same effect can be obtained by using inline assembly code to give the variable two names, one global and one local, and use the local name for local references.

Despite these tricks, you may still get the error message: "relocation R_X86_64_PC32 against symbol `functionname' can not be used when making a shared object; recompile with -fPIC", when the shared object is made from multiple modules (source files) and there is a call from one module to another. I have not yet found a solution to this problem.

Shared objects in BSD

Shared objects in BSD work the same way as in Linux.

32-bit Mac OS X

Compilers for 32-bit Mac OS X make position-independent code and lazy binding by default, even when shared objects are not used. The method currently used for calculating self-relative addresses in 32-bit Mac code uses an unfortunate method that delays execution by causing return addresses to be mispredicted (See manual 3: "The microarchitecture of Intel, AMD and VIA CPUs" for an explanation of return prediction).

All code that is not part of a shared object can be speeded up significantly just by turning off the position-independent code flag in the compiler. Remember, therefore, always to specify the compiler option `-fno-pic` when compiling for 32-bit Mac OS X, unless you are making a shared object.

It is possible to make shared objects without position-independent code when you compile with the option `-fno-pic` and link with the option `-read_only_relocs suppress`.

GOT and PLT tables are not used for internal references.

64-bit Mac OS X

The code section is always position-independent because this is the most efficient solution for the memory model used here. The compiler option `-fno-pic` apparently has no effect.

GOT and PLT tables are not used for internal references.

There is no need to take special precautions for speeding up 64-bit shared objects in Mac OS X.

14.13 System programming

Device drivers, interrupt service routines, system core and high-priority threads are areas where speed is particularly critical. A very time-consuming function in system code or in a high-priority thread can possibly block the execution of everything else.

System code has to obey certain rules about register use, as explained in the chapter "Register usage in kernel code" in manual 5: "Calling conventions for different C++ compilers and operating systems". For this reason, you can use only compilers and function libraries that are intended for system code. System code should be written in C, C++ or assembly language.

It is important to economize resource use in system code. Dynamic memory allocation is particularly risky because it involves the risk of activating the very time-consuming garbage collector at inconvenient times. A queue should be implemented as a circular buffer with fixed size, not as a linked list. Do not use STL containers. See page 92.

15 Metaprogramming

Metaprogramming means to make code that makes code. For example, in interpreted script languages, it is often possible to make a piece of code that produces a string and then interpret that string as code.

Metaprogramming can be useful in compiled languages such as C++ for doing some calculations at compile time rather than at runtime if all the inputs to the calculations are available at compile time. (Of course there is no such advantage in interpreted languages where everything happens at runtime).

The following techniques can be considered metaprogramming in C++:

- Preprocessor directives. For example use `#if` instead of `if`. This is a very efficient way of removing superfluous code, but there are serious limitations to what the preprocessor can do because it comes before the compiler and it understands only the simplest expressions and operators.

- Make a C++ program that produces another C++ program (or part of it). This can be useful in some cases, for example to produce tables of mathematical functions that you want as static arrays in the final program. This requires, of course, that you compile the output of the first program.

- An optimizing compiler may try to do as much as possible at compile time. For example, all good compilers will reduce `int x = 2 * 5;` to `int x = 10;`

- Templates are instantiated at compile time. A template instance has its parameters replaced by their actual values before it is compiled. This is the reason why there is virtually no cost to using templates (see p. 58). It is possible to express any algorithm with template metaprogramming, but this method is extremely complicated and clumsy, as you will see shortly.

The following examples explain how metaprogramming can be used to speed up the calculation of the power function when the exponent is an integer known at compile time.

```
// Example 15.1a. Calculate x to the power of 10
double xpow10(double x) {
   return pow(x,10);
}
```

The `pow` function uses logarithms in the general case, but in this case it will recognize that 10 is an integer, so that the result can be calculated using multiplications only. The following algorithm is used inside the `pow` function when the exponent is a positive integer:

```
// Example 15.1b. Calculate integer power using loop
double ipow (double x, unsigned int n) {
   double y = 1.0;               // used for multiplication
   while (n != 0) {              // loop for each bit in nn
      if (n & 1) y *= x;         // multiply if bit = 1
      x *= x;                    // square x
      n >>= 1;                   // get next bit of n
   }
   return y;                     // return y = pow(x,n)
}

double xpow10(double x) {
   return ipow(x,10);            // ipow faster than pow
```

```
}
```

The method used in example 15.1b is easier to understand when we roll out the loop and reorganize:

```
// Example 15.1c. Calculate integer power, loop unrolled
double xpow10(double x) {
   double x2  = x *x;            // x^2
   double x4  = x2*x2;           // x^4
   double x8  = x4*x4;           // x^8
   double x10 = x8*x2;           // x^10
   return x10;                   // return x^10
}
```

As we can see, it is possible to calculate `pow(x,10)` with only four multiplications. How was it possible to come from example 15.1b to 15.1c? We took advantage of the fact that `n` is known at compile time to eliminate everything that depends only on `n`, including the `while` loop, the `if` statement and all the integer calculations. The code in example 15.1c is faster than 15.1b, and in this case it may be smaller as well.

The conversion from example 15.1b to 15.1c was done by me manually, but if we want to generate a piece of code that works for any compile-time constant `n`, then we need metaprogramming. None of the compilers I have tested can convert example 15.1a to 15.1c automatically, and only the Gnu compiler will convert example 15.1b to 15.1c. We can only hope that future compilers will do such optimizations automatically, but as long as this is not the case we may need metaprogramming.

The next example shows this calculation implemented with template metaprogramming. Don't panic if you don't understand it. I am giving this example only to show how tortuous and convoluted template metaprogramming is.

```
// Example 15.1d. Integer power using template metaprogramming

// Template for pow(x,N) where N is a positive integer constant.
// General case, N is not a power of 2:
template <bool IsPowerOf2, int N>
class powN {
public:
   static double p(double x) {
   // Remove right-most 1-bit in binary representation of N:
   #define N1 (N & (N-1))
      return powN<(N1&(N1-1))==0,N1>::p(x) * powN<true,N-N1>::p(x);
   #undef N1
   }
};

// Partial template specialization for N a power of 2
template <int N>
class powN<true,N> {
public:
   static double p(double x) {
      return powN<true,N/2>::p(x) * powN<true,N/2>::p(x);
   }
};

// Full template specialization for N = 1. This ends the recursion
template<>
class powN<true,1> {
public:
   static double p(double x) {
      return x;
   }
```

```
};

// Full template specialization for N = 0
// This is used only for avoiding infinite loop if powN is
// erroneously called with IsPowerOf2 = false where it should be true.
template<>
class powN<true,0> {
public:
   static double p(double x) {
      return 1.0;
   }
};

// Function template for x to the power of N
template <int N>
static inline double IntegerPower (double x) {
   // (N & N-1)==0 if N is a power of 2
   return powN<(N & N-1)==0,N>::p(x);
}

// Use template to get x to the power of 10
double xpow10(double x) {
   return IntegerPower<10>(x);
}
```

If you want to know how this works, here's an explanation. Please skip the following explanation if you are not sure you need it.

In C++ template metaprogramming, loops are implemented as recursive templates. The `powN` template is calling itself in order to emulate the `while` loop in example 15.1b. Branches are implemented by (partial) template specialization. This is how the `if` branch in example 15.1b is implemented. The recursion must always end with a non-recursing template specialization, not with a branch inside the template.

The `powN` template is a class template rather than a function template because partial template specialization is allowed only for classes. The splitting of `N` into the individual bits of its binary representation is particularly tricky. I have used the trick that `N1 = N&(N-1)` gives the value of `N` with the rightmost 1-bit removed. If `N` is a power of 2 then `N&(N-1)` is 0. The constant `N1` could have been defined in other ways than by a macro, but the method used here is the only one that works on all the compilers I have tried.

The Microsoft, Intel and Gnu compilers are actually reducing example 15.1d to 15.1c as intended, while the Borland and Digital Mars compilers produce less optimal code because they fail to eliminate common sub-expressions.

Why is template metaprogramming so complicated? Because the C++ template feature was never designed for this purpose. It just happened to be possible. Template meta-programming is so complicated that I consider it unwise to use it. Complicated code is a risk factor in itself, and the cost of verifying, debugging and maintaining such code is so high that it rarely justifies the relatively small gain in performance.

There are cases, however, where template metaprogramming is the only way to make sure that certain calculations are done at compile time. (Examples can be found in my vector class library).

The D language allows compile-time `if` statements (called `static if`), but no compile-time loops or compile-time generation of identifier names. We can only hope that such feature will become available in the future. If a future version of C++ should allow compile-time `if` and compile-time `while` loops, then the transformation of example 15.1b to metaprogramming would be straightforward. The MASM assembly language has full

metaprogramming features, including the ability to define function names and variable names from string functions. A metaprogramming implementation analogous to example 15.1b and d in assembly language is provided as an example in the "Macro loops" chapter in manual 2: "Optimizing subroutines in assembly language".

While we are waiting for better metaprogramming tools to be available, we may choose the compilers that are best at doing equivalent reductions at their own initiative whenever it is possible. A compiler that automatically reduces example 15.1a to 15.1c would of course be the easiest and the most reliable solution. (In my tests, the Intel compiler reduced 15.1a to an inlined 15.1b and the Gnu compiler reduced 15.1b to 15.1c, but none of the compilers reduced 15.1a to 15.1c).

16 Testing speed

Testing the speed of a program is an important part of the optimization job. You have to check if your modifications actually increase the speed or not.

There are various profilers available which are useful for finding the hot spots and measuring the overall performance of a program. The profilers are not always accurate, however, and it may be difficult to measure exactly what you want when the program spends most of its time waiting for user input or reading disk files. See page 16 for a discussion of profiling.

When a hot spot has been identified, then it may be useful to isolate the hot spot and make measurements on this part of the code only. This can be done with the resolution of the CPU clock by using the so-called time stamp counter. This is a counter that measures the number of clock pulses since the CPU was started. The length of a clock cycle is the reciprocal of the clock frequency, as explained on page 16. If you read the value of the time stamp counter before and after executing a critical piece of code then you can get the exact time consumption as the difference between the two clock counts.

The value of the time stamp counter can be obtained with the function `ReadTSC` listed below in example 16.1. This code works only for compilers that support intrinsic functions. Alternatively, you can use the header file `timingtest.h` from www.agner.org/optimize/testp.zip or get `ReadTSC` as a library function from www.agner.org/optimize/asmlib.zip.

```
// Example 16.1
#include <intrin.h>          // Or #include <ia32intrin.h> etc.

long long ReadTSC() {        // Returns time stamp counter
   int dummy[4];             // For unused returns
   volatile int DontSkip;    // Volatile to prevent optimizing
   long long clock;          // Time
   __cpuid(dummy, 0);        // Serialize
   DontSkip = dummy[0];      // Prevent optimizing away cpuid
   clock = __rdtsc();        // Read time
   return clock;
}
```

You can use this function to measure the clock count before and after executing the critical code. A test setup may look like this:

```
// Example 16.2

#include <stdio.h>
#include <asmlib.h>          // Use ReadTSC() from library asmlib..
                             // or from example 16.1
```

```
void CriticalFunction();   // This is the function we want to measure

...

const int NumberOfTests = 10;          // Number of times to test
int i; long long time1;
long long timediff[NumberOfTests];     // Time difference for each test
for (i = 0; i < NumberOfTests; i++) { // Repeat NumberOfTests times

   time1 = ReadTSC();                  // Time before test

   CriticalFunction();                 // Critical function to test

   timediff[i] = ReadTSC() - time1;    // (time after) - (time before)
}
printf("\nResults:");                  // Print heading
for (i = 0; i < NumberOfTests; i++) { // Loop to print out results
   printf("\n%2i  %10I64i", i, timediff[i]);
}
```

The code in example 16.2 calls the critical function ten times and stores the time consumption of each run in an array. The values are then output after the test loop. The time that is measured in this way includes the time it takes to call the `ReadTSC` function. You can subtract this value from the counts. It is measured simply by removing the call to `CriticalFunction` in example 16.2.

The measured time is interpreted in the following way. The first count is usually higher than the subsequent counts. This is the time it takes to execute `CriticalFunction` when code and data are not cached. The subsequent counts give the execution time when code and data are cached as good as possible. The first count and the subsequent counts represent the "worst case" and "best case" values. Which of these two values is closest to the truth depends on whether `CriticalFunction` is called once or multiple times in the final program and whether there is other code that uses the cache in between the calls to `CriticalFunction`. If your optimization effort is concentrated on CPU efficiency then it is the "best case" counts that you should look at to see if a certain modification is profitable. On the other hand, if your optimization effort is concentrated on arranging data in order to improve cache efficiency, then you may also look at the "worst case" counts. In any event, the clock counts should be multiplied by the clock period and by the number of times `CriticalFunction` is called in a typical application to calculate the time delay that the end user is likely to experience.

Occasionally, the clock counts that you measure are much higher than normal. This happens when a task switch occurs during execution of `CriticalFunction`. You cannot avoid this in a protected operating system, but you can reduce the problem by increasing the thread priority before the test and setting the priority back to normal afterwards.

The clock counts are often fluctuating and it may be difficult to get reproducible results. This is because modern CPUs can change their clock frequency dynamically depending on the work load. The clock frequency is increased when the work load is high and decreased when the work load is low in order to save power. There are various ways to get more reproducible time measurements:

- warm up the CPU by giving it some heavy work to do immediately before the code to test.
- disable power-save options in the BIOS setup.
- on Intel CPUs: use the core clock cycle counter (see below)

16.1 Using performance monitor counters

Many CPUs have a built-in test feature called performance monitor counters. A performance monitor counter is a counter inside the CPU which can be set up to count certain events, such as the number of machine instructions executed, cache misses, branch mispredictions, etc. These counters can be very useful for investigating performance problems. The performance monitor counters are CPU-specific and each CPU model has its own set of performance monitoring options.

CPU vendors are offering profiling tools that fit their CPUs. Intel's profiler is called VTune; AMD's profiler is called CodeAnalyst. These profilers are useful for identifying hot spots in the code.

For my own research, I have developed a test tool for using the performance monitor counters. My test tool supports both Intel, AMD and VIA processors, and it is available from www.agner.org/optimize/testp.zip. This tool is not a profiler. It is not intended for finding hot spots, but for studying a piece of code once the hot spots have been identified.

My test tool can be used in two ways. The first way is to insert the piece of code to test in the test program itself and recompile it. I am using this for testing single assembly instructions or small sequences of code. The second way is to set up the performance monitor counters before running a program you want to optimize, and reading the performance counters inside your program before and after the piece of code you want to test. You can use the same principle as in example 16.2 above, but read one or more performance monitor counters instead of (or in addition to) the time stamp counter. The test tool can set up and enable one or more performance monitor counters in all the CPU cores and leave them enabled (there is one set of counters in each CPU core). The counters will stay on until you turn them off or until the computer is reset or goes into sleep mode. See the manual for my test tool for details (www.agner.org/optimize/testp.zip).

A particularly useful performance monitor counter in Intel processors is called *core clock cycles*. The core clock cycles counter is counting clock cycles at the actual clock frequency that the CPU core is running at, rather than the external clock. This gives a measure that is almost independent of changes in the clock frequency. The core clock cycle counter is very useful when testing which version of a piece of code is fastest because you can avoid the problem that the clock frequency goes up and down.

Remember to insert a switch in your program to turn off the reading of the counters when you are not testing. Trying to read the performance monitor counters when they are disabled will crash the program.

16.2 The pitfalls of unit-testing

It is common practice to test each function or class separately in software development. This unit-testing is necessary for verifying the functionality of an optimized function, but unfortunately the unit-test does not give the full information about the performance of the function in terms of speed.

Assume that you have two different versions of a critical function and you want to find out which one is fastest. The typical way to test this is to make a small test program that calls the critical function many times with a suitable set of test data and measure how long time it takes. The version that performs best under this unit-test may have a larger memory footprint than the alternative version. The penalty of cache misses is not seen in the unit-test because the total amount of code and data memory used by the test program is likely to be less than the cache size.

When the critical function is inserted in the final program, it is very likely that code cache and data cache are critical resources. Modern CPUs are so fast that the clock cycles spent

on executing instructions are less likely to be a bottleneck than memory access and cache size. If this is the case then the optimal version of the critical function may be the one that takes longer time in the unit-test but has a smaller memory footprint.

If, for example, you want to find out whether it is advantageous to roll out a big loop then you cannot rely on a unit-test without taking cache effects into account.

You can calculate how much memory a function uses by looking at a link map or an assembly listing. Use the "generate map file" option for the linker. Both code cache use and data cache use can be critical. The branch target buffer is also a cache that can be critical. Therefore, the number of jumps, calls and branches in a function should also be considered.

A realistic performance test should include not only a single function or hot spot but also the innermost loop that includes the critical functions and hot spots. The test should be performed with a realistic set of data in order to get reliable results for branch mispredictions. The performance measurement should not include any part of the program that waits for user input. The time used for file input and output should be measured separately.

The fallacy of measuring performance by unit-testing is unfortunately very common. Even some of the best optimized function libraries available use excessive loop unrolling so that the memory footprint is unreasonably large.

16.3 Worst-case testing

Most performance tests are done under the best-case conditions. All disturbing influences are removed, all resources are sufficient, and the caching conditions are optimal. Best-case testing is useful because it gives more reliable and reproducible results. If you want to compare the performance of two different implementations of the same algorithm, then you need to remove all disturbing influences in order to make the measurements as accurate and reproducible as possible.

However, there are cases where it is more relevant to test the performance under the worst-case conditions. For example, if you want to make sure that the response time to user input never exceeds an acceptable limit, then you should test the response time under worst-case conditions.

Programs that produce streaming audio or video should also be tested under worst-case conditions in order to make sure that they always keep up with the expected real-time speed. Delays or glitches in the output are unacceptable.

Each of the following methods could possibly be relevant when testing worst-case performance:

- The first time you activate a particular part of the program, it is likely to be slower than the subsequent times because of lazy loading of the code, cache misses and branch mispredictions.

- Test the whole software package, including all runtime libraries and frameworks, rather than isolating a single function. Switch between different parts of the software package in order to increase the likelihood that certain parts of the program code are uncached or even swapped to disk.

- Software that relies on network resources and servers should be tested on a network with heavy traffic and a server in full use rather than a dedicated test server.

- Use large data files and databases with lots of data.
- Use an old computer with a slow CPU, an insufficient amount of RAM, a lot of irrelevant software installed, a lot of background processes running, and a slow and fragmented hard disk.
- Test with different brands of CPUs, different types of graphics cards, etc.
- Use an antivirus program that scans all files on access.
- Run multiple processes or threads simultaneously. If the microprocessor has hyperthreading, then try to run two threads in the same processor core.
- Try to allocate more RAM than there is, in order to force the swapping of memory to disk.
- Provoke cache misses by making the code size or data used in the innermost loop bigger than the cache size. Alternatively, you may actively invalidate the cache. The operating system may have a function for this purpose, or you may use the `_mm_clflush` intrinsic function.
- Provoke branch mispredictions by making the data more random than normal.

17 Optimization in embedded systems

Microcontrollers used in small embedded applications have less computing resources than standard PCs. The clock frequency may be a hundred or even a thousand times lower; and the amount of RAM memory may even be a million times less than in a PC. Nevertheless, it is possible to make software that runs quite fast on such small devices if you avoid the large graphics frameworks, interpreters, just-in-time compilers, system database, and other extra software layers and frameworks typically used on bigger systems.

The smaller the system, the more important it is to choose a software framework that uses few resources. On the smallest devices, you don't even have an operating system.

The best performance is obtained by choosing a programming language that can be cross-compiled on a PC and then transferred as machine code to the device. Any language that requires compilation or interpretation on the device itself is a big waste of resources. For these reasons, the preferred language will often be C or C++. Critical device drivers may need assembly language.

C++ takes only slightly more resources than C if you follow the guidelines below. You may choose either C or C++ based on what is most appropriate for the desired program structure.

It is important to economize the use of RAM memory. Big arrays should be declared inside the function they are used in so that they are deallocated when the function returns. Alternatively, you may reuse the same array for multiple purposes.

All dynamic memory allocation using `new`/`delete` or `malloc`/`free` should be avoided because of the large overhead of managing a memory heap. The heap manager has a garbage collector which is likely to consume time at unpredictable intervals which may interfere with real time applications.

Remember that container classes in the STL (Standard Template Library) and other container class libraries use dynamic memory allocation with `new` and `delete`, and often excessively so. These containers should definitely be avoided unless you have ample resources. For example, a FIFO queue should be implemented as a circular buffer with fixed size to avoid dynamic memory allocation. Do not use a linked list (see page 95).

All common implementations of string classes use dynamic memory allocation. You should avoid these and handle text strings in the old fashioned C style as character arrays. Note that the C style string functions have no check for overflow of the arrays. It is the responsibility of the programmer to make sure the arrays are sufficiently large to handle the strings including the terminating zero and to make overflow checks where necessary (see page 98).

Virtual functions in C++ take more resources than non-virtual functions. Avoid virtual functions if possible.

Smaller microprocessors have no native floating point execution units. Any floating point operation on such processors requires a big floating point library which is very time consuming. Therefore, you should avoid any use of floating point expressions. For example, `a = b * 2.5` may be changed to `a = b * 5 / 2` (be aware of possible overflow on the intermediate expression `b * 5`). As soon as you have even a single constant with a decimal point in your program, you will be loading the entire floating point library. If you want a number to be calculated with two decimals, for example, you should multiply it by 100 so that it can be represented as an integer.

Integer variables can be 8, 16 or 32 bits (rarely 64). You may save RAM space, if necessary, by using the smallest integer size that doesn't cause overflow in the particular application. The integer size is not standardized across platforms. See the compiler documentation for the size of each integer type.

Interrupt service routines and device drivers are particularly critical because they can block the execution of everything else. This normally belongs to the area of system programming, but in applications without an operating system this is the job of the application programmer. There is a higher risk that the programmer forgets that the system code is critical when there is no operating system, and therefore the system code is not separated from the application code. An interrupt service routine should do as little work as possible. Typically it should save one unit of received data in a static buffer or send data from a buffer. It should never respond to a command or do other input/output than the specific event it is servicing. A command received by an interrupt should preferably be responded to at a lower priority level, typically in a message loop in the main program. See page 153 for further discussion of system code.

In this chapter, I have described some of the considerations that are particularly important on small devices with limited resources. Most of the advice in the rest of the present manual is also relevant to small devices, but there are some differences due to the design of small microcontrollers:

- Smaller microcontrollers have no branch prediction (see p. 43). There is no need to take branch prediction into account in the software.

- Smaller microcontrollers have no cache (see p. 89). There is no need to organize data to optimize caching.

- Smaller microcontrollers have no out-of-order execution. There is no need to break down dependency chains (see p. 22).

18 Overview of compiler options

Table 18.1. Command line options relevant to optimization

	MS compiler Windows	Gnu compiler Linux	Intel compiler Windows	Intel compiler Linux
Optimize for speed	/O2 or /Ox	-O3 or -Ofast	/O3	-O3
Interprocedural optimization	/Og			
Whole program optimization	/GL	--combine -fwhole-program	/Qipo	-ipo
No exception handling	/EHs-			
No stack frame	/Oy	-fomit-frame-pointer		-fomit-frame-pointer
No runtime type identification (RTTI)	/GR-	-fno-rtti	/GR-	-fno-rtti
Assume no pointer aliasing	/Oa			-fno-alias
Non-strict floating point		-ffast-math	/fp:fast /fp:fast=2	-fp-model fast, -fp-model fast=2
Simple member pointers	/vms			
Fastcall functions	/Gr			
Function level linking (remove unreferenced functions)	/Gy	-ffunction-sections	/Gy	-ffunction-sections
SSE instruction set (128 bit float vectors)	/arch:SSE	-msse	/arch:SSE	-msse
SSE2 instruction set (128 vectors of integer or double)	/arch:SSE2	-msse2	/arch:SSE2	-msse2
SSE3 instruction set		-msse3	/arch:SSE3	-msse3
Suppl. SSE3 instr. set		-mssse3	/arch:SSSE2	-mssse3
SSE4.1 instr. set		-msse4.1	/arch:SSE4.1	-msse4.1
AVX instr. set	/arch:AVX	-mAVX	/arch:AVX	-mAVX
Automatic CPU dispatch			/QaxSSE3, etc. (Intel CPU only)	-axSSE3, etc. (Intel CPU only)
Automatic vectorization		-O3 -fno-trapping-math -fno-math-errno -mveclibabi		
Automatic parallelization by multiple threads			/Qparallel	-parallel
Parallelization by OpenMP directives	/openmp	-fopenmp	/Qopenmp	-openmp
32 bit code		-m32		
64 bit code		-m64		
Static linking	/MT	-static	/MT	-static

(multithreaded)				
Generate assembly listing	/FA	-S -masm=intel	/FA	-S
Generate map file	/Fm			
Generate optimization report			/Qopt-report	-opt-report

Table 18.2. Compiler directives and keywords relevant to optimization

	MS compiler Windows	Gnu compiler Linux	Intel compiler Windows	Intel compiler Linux
Align by 16	__declspec(align(16))	__attribute((aligned(16)))	__declspec(align(16))	__attribute((aligned(16)))
Assume pointer is aligned			#pragma vector aligned	#pragma vector aligned
Assume pointer not aliased	#pragma optimize("a", on) __restrict	__restrict	__declspec(noalias) __restrict #pragma ivdep	__restrict #pragma ivdep
Assume function is pure		__attribute((const))		__attribute((const))
Assume function does not throw exceptions	throw()	throw()	throw()	throw()
Assume function called only from same module	static	static	static	static
Assume member function called only from same module		__attribute__((visibility("internal")))		__attribute__((visibility("internal")))
Vectorize			#pragma vector always	#pragma vector always
Optimize function	#pragma optimize(...)			
Fastcall function	__fastcall	__attribute((fastcall))	__fastcall	
Noncached write			#pragma vector nontemporal	#pragma vector nontemporal

Table 18.3. Predefined macros

	MS compiler Windows	Gnu compiler Linux	Intel compiler Windows	Intel compiler Linux
Compiler identification	_MSC_VER and not __INTEL_COMPILER	__GNUC__ and not __INTEL_COMPILER	__INTEL_COMPILER	__INTEL_COMPILER
16 bit	not _WIN32	n.a.	n.a.	n.a.

platform				
32 bit platform	not `_WIN64`		not `_WIN64`	
64 bit platform	`_WIN64`	`_LP64`	`_WIN64`	`_LP64`
Windows platform	`_WIN32`		`_WIN32`	
Linux platform	n.a.	`__unix__` `__linux__`		`__unix__` `__linux__`
x86 platform	`_M_IX86`		`_M_IX86`	
x86-64 platform	`_M_IX86` and `_WIN64`		`_M_X64`	`_M_X64`

19 Literature

Other manuals by Agner Fog

The present manual is number one in a series of five manuals. See page 3 for a list of titles.

Literature on code optimization

Intel: "Intel 64 and IA-32 Architectures Optimization Reference Manual". developer.intel.com.

Many advices on optimization of C++ and assembly code for Intel CPUs. New versions are produced regularly.

AMD: "Software Optimization Guide for AMD Family 15h Processors". www.amd.com.

Advices on optimization of C++ and assembly code for AMD CPUs. New versions are produced regularly.

Intel: "Intel® C++ Compiler Documentation". Included with Intel C++ compiler, which is available from www.intel.com.

Manual on using the optimization features of Intel C++ compilers.

Wikipedia article on compiler optimization. en.wikipedia.org/wiki/Compiler_optimization.

ISO/IEC TR 18015, "Technical Report on C++ Performance". www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf.

OpenMP. www.openmp.org. Documentation of the OpenMP directives for parallel processing.

Scott Meyers: "Effective C++". Addison-Wesley. Third Edition, 2005; and "More Effective C++". Addison-Wesley, 1996.

These two books contain many tips on advanced C++ programming, how to avoid hard-to-find errors, and some tips on improving performance.

Stefan Goedecker and Adolfy Hoisie: "Performance Optimization of Numerically Intensive Codes", SIAM 2001.

Advanced book on optimization of C++ and Fortran code. The main focus is on mathematical applications with large data sets. Covers PC's, workstations and scientific vector processors.

Henry S. Warren, Jr.: "Hacker's Delight". Addison-Wesley, 2003.

Contains many bit manipulation tricks

Michael Abrash: "Zen of code optimization", Coriolis group books 1994.

Mostly obsolete.

Rick Booth: "Inner Loops: A sourcebook for fast 32-bit software development", Addison-Wesley 1997.

Mostly obsolete.

Microprocessor documentation

Intel: "IA-32 Intel Architecture Software Developer's Manual", Volume 1, 2A, 2B, and 3A and 3B. developer.intel.com.

AMD: "AMD64 Architecture Programmer's Manual", Volume 1 - 5. www.amd.com.

Internet forums

Several internet forums and newsgroups contain useful discussions about code optimization. See www.agner.org/optimize and the FAQ for the newsgroup comp.lang.asm.x86 for some links.

20 Copyright notice