Hands-On High Performance Programming with Qt 5

Build cross-platform applications using concurrency, parallel programming, and memory management

www.packt.com

Marek Krajewski

Hands-On High Performance Programming with Qt 5

Build cross-platform applications using concurrency, parallel programming, and memory management

Marek Krajewski

Packt

BIRMINGHAM - MUMBAI

Hands-On High Performance Programming with Qt 5

Commissioning Editor: Richa Tripathi
Acquisition Editor: Shriram Shekhar
Content Development Editor: Anugraha Arunagiri
Technical Editor: Abin Sebastian
Copy Editor: Safis Editing
Project Coordinator: Ulhas Kambali
Proofreader: Safis Editing
Indexer: Pratik Shirodkar
Graphics: Tom Scaria
Production Coordinator: Shraddha Falebhai

First edition: January 2019

Production reference: 1300119

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78953-124-4

`www.packtpub.com`

To my late father, who was a living example of the Polish intelligentsia tradition for me, and to my sister (now we are even, I wrote a book too!)
To my wife, who convinced me to write this book and supported me on this long journey.

–Marek Krajewski

Contributors

About the author

Marek Krajewski has been programming in C++ since the mid 90s, and in Qt since 2008. In his career, he has been involved with Unix and Windows system programming, client-server systems, UMTS network management, Enterprise Java, satellite protocol decoding, neural networks, image processing, DVB-T testing appliances, REST APIs, and embedded Linux. He holds a Ph.D. in computer science and is currently working as an independent programmer specializing in Qt, C++, GUIs, system programming, and communication protocols. His other interests are off-piste skiing and Aikido, where he holds the rank of second dan.

About the reviewer

Nibedit Dey is a techno-entrepreneur and innovator with over 8 years of experience in building complex software-based products using Qt and C++. Before starting his entrepreneurial journey, he worked for L&T and Tektronix in different research and development roles. Additionally, he has reviewed *The Modern C++ Challenge* and *Hands-on GUI programming with C++ and Qt5* books for Packt.

I would like to thank the online programming communities, bloggers, and my peers from earlier organizations, from whom I have learned a lot over the years.

Packt is searching for authors like you

Table of Contents

Preface

In today's world, programming knowledge appears to be scattered over a myriad of locations—blog posts, Wikipedia articles, conference presentations, Twitter threads, Stack Overflow questions, and the odd scientific paper to boot. Because of that, I was quite glad when the publishers approached me with an idea for a book about Qt performance—at last, a single place where program performance knowledge could be collected.

This book will try to give you an overview of program performance optimization techniques and apply them in the context of Qt programming. Qt is a cross-platform and cross-topic programming framework encompassing both GUI and system programming on desktop, mobile, and embedded platforms. Hence, while discussing Qt's performance, we will encounter a broad gamut of performance topics of every shade and color, and we will do this not in an abstract manner, but using real-life examples.

Since Qt has many different versions and releases, we will concentrate on the current (as of the time of writing) **long-term support** (**LTS**) version, namely, Qt 5.9, but will also provide information regarding later Qt versions up to Qt 5.12. Fitting the high-performance theme, we will mostly be using the epitome of high performance, that is Qt's underlying C++ language, directly.

Moreover, in order to make this book approachable, we will use open source development tools and a very widespread and relatively recent OS platform, namely, Microsoft Windows 10.

I wanted to write a book that I would enjoy reading when I first started to learn about performance years ago—we will start slowly, but then go deep, as I hope to provide you with most of the knowledge required to write performant Qt programs.

Who this book is for

This book is intended for developers who wish to build highly performant Qt applications for desktop and embedded devices using the C++ language. If you'd like to optimize the performance of a Qt application that has already been written, you could also find this book useful.

Furthermore, this book is aimed at intermediate-level Qt developers. While the *intermediate* designation is a rather broad one, we define it here as the skill level of a programmer who can implement simple Qt 5 programs using basic tooling. If you cannot do that, you should probably read an introductory Qt 5 book first.

So, let us have a look at the C++, Qt, and programming knowledge that is assumed in this book.

The Qt skills you will need include a basic knowledge of Qt Widgets, QML, and the signal-slot mechanism. You should be able to work with the Qt Creator IDE as the development environment. You should also be able to work on a Windows 10 computer.

As we will see, having a solid understanding of C++ is key to using Qt effectively. However, this book doesn't concentrate on C++ language features, so you definitively don't need to be an expert in C++ to read it.

You will, however, need to have some understanding of basic C++ features, such as object orientation, exceptions, basic templates, `auto` variables, and simple lambdas. Be aware that this book isn't about weird C++ language constructs, but rather tries to introduce you to a broader topic of performance, while using C++ as a sharp tool.

You also don't have to be that proficient in computer science or hardware architectures—we will always introduce the notions required as we go. The entry barrier isn't that high, so if you've already done some Qt programming in C++, you can embark on the journey.

What this book covers

`Chapter 1`, *Understanding Performant Programs*, introduces you to the world of performance, discusses modern processor architectures, and recounts some traditional performance wisdom.

`Chapter 2`, *Profiling to Find Bottlenecks*, looks at the tooling we use to diagnose performance problems, concentrating specifically on the tools available on Windows.

`Chapter 3`, *Deep Dive into C++ and Performance*, takes a closer look at the C++ 11 language, its general performance, the performance of its numerous features, and the role of compilers and linkers in the performance game.

`Chapter 4`, *Using Data Structures and Algorithms Efficiently*, discusses the performance impact of data structures and algorithms and then takes a closer look at the data structures used by Qt in its API and the associated performance gotchas.

`Chapter 5`, *An In-Depth Guide to Concurrency and Multithreading*, describes multithreading concepts, Qt multithreading classes and techniques, explains the **multithreading is hard** mantra, and offers some techniques for minimizing multithreading costs and pitfalls.

`Chapter 6`, *Performance Fails and How to Overcome Them*, looks back at some performance challenges I encountered during my work.

`Chapter 7`, *Understanding I/O Performance and Overcoming Related Problems*, turns our attention to the unglamorous theme of reading, writing, and parsing files, as well as unexpected interactions with a number of OS mechanisms.

`Chapter 8`, *Optimizing Graphical Performance*, looks at the main usage scenario of Qt—writing graphical UIs. We will learn about GPUs, and their performance and usage in Qt. Later, we will have a look at the main classes of Qt Widgets and Qt Quick and will discuss what their performance depends upon.

`Chapter 9`, *Optimizing Network Performance*, introduces basic networking concepts, the TCP and HTTP protocols, the Qt classes supporting them, and the network optimization techniques available in Qt.

`Chapter 10`, *Qt Performance on Embedded and Mobile Platforms*, looks at mobile and embedded usage of Qt. It builds on the knowledge we acquired in the previous chapters to examine the specific performance challenges faced by Qt on those platforms.

`Chapter 11`, *Testing and Deploying Qt Applications*, ends this book with a discussion of testing and deployment tools and techniques.

To get the most out of this book

As already stated, this book is aimed at intermediate Qt developers. You should be able to write small- to medium-sized Qt programs in C++ and QML using Qt Creator as an IDE on Windows 10.

You don't need to install any software before you start. Instructions will be provided at the point where specific software is needed. We will use exclusively open source programs so you won't have to purchase any licences.

Download the example code files

You can download the example code files for this book from your account at `www.packt.com`. If you purchased this book elsewhere, you can visit `www.packt.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at **`https://github.com/PacktPublishing/Hands-On High Performance Programming with Qt 5`**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://www.packtpub.com/sites/default/files/downloads/9781789531244_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "As could be seen in the previous example, Qt `Test` defines macros to specify pass/fail criteria for tests, namely, `QCOMPARE()` and `QVERIFY()`."

A block of code is set as follows:

```
QSignalSpy spy(tstPushBtn, SIGNAL(clicked()));
QTest::mouseClick(tstPushBtn, Qt::LeftButton);
QCOMPARE(spy.count(), 1);
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
QSignalSpy spy(tstPushBtn, SIGNAL(clicked()));
QTest::mouseClick(tstPushBtn, Qt::LeftButton);
QCOMPARE(spy.count(), 1);
```

Any command-line input or output is written as follows:

```
$ mkdir Qt
$ cd Qt
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Unfortunately, results displayed in the **Test Result** pane don't seem to work with QML tests with the Qt Creator version used in this book. We have to run a QML test project in the **Projects** view."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packt.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

1
Understanding Performant Programs

In this introductory chapter, we'll start this book with some general discussions about program performance: why it's important, what are the factors that determine it, and how programmers generally go about performance themes. We'll begin with a broad discussion of performances' relevance in programming, before looking at some traditional performance-related knowledge, and we'll finish this chapter with the impact modern CPU architectures made in this field.

This chapter will therefore cover the following topics:

- **Why performance is important**: To motivate ourselves before diving into technicalities
- **Traditional wisdom and basic guidelines**: Old and proven performance knowledge
- **Modern processor architectures**: At least the performance-relevant parts of it

Why performance is important

Maybe you just started reading this book out of curiosity and you're asking yourself this question: Why is performance important? Isn't that a thing of the past, when we didn't have enough CPU power and memory, and when our networks were grinding to a halt? In today's high-tech world, we have enough resources—the computers are so insanely fast!

Well, in principle, you're right to some degree, but consider the following:

- A faster program runs more quickly, consuming less power along the way. This is good for the planet (if you're running it in a big server farm) and good for your user (if you're running it on a desktop computer).
- A faster program means that it can serve more requests in the same time than a slower one. This is good for business, as you'll need to buy or lease fewer machines to serve your customers, and, again, it's good for the planet!
- Faster software in today's business world's cut-throat competition means an advantage in respect to your competitors. This is nowhere more evident than in the world of automated trading (which is, by the way, dominated by C++), but also the fact that sluggishly-loading websites and programs needing an eternity to start won't be used that much!
- And, lastly, especially on mobile devices, we still have to cope with constrained resources—network speed is finite (light speed) and battery life is finite too—and as faster programs use less resources, they're good for users!

Our quest for performance will hence pursue a three-pronged objective: to save the planet, to strengthen your business, and to make the life of users better—not a small feat I'd say!

The price of performance optimization

So, everything is rosy? Well, not quite, as there's a dark side to performance optimization too. If we'll try to squeeze the last drops of performance from our hardware, we can end up having unreadable, inflexible, unmaintainable, and hence outright ugly, code!

So, be aware that there are caveats and that there's a price to be paid. We must decide whether we want to pay it, and at what point we'll stop the optimization to save the clarity of our code.

Traditional wisdom and basic guidelines

When I started with programming (a long time ago), the pieces of advice about performance optimization traditionally given to a newbie were the following:

- Don't do it (yet)
- Premature optimization is the root of all evil
- First make it run, then make it right, then make it fast

The first advice contained the *yet* only in its variant for the experts; the second was (and still is) normally misquoted, leaving out the "*in say 97% of the cases*" part, and the third quote gives you the impression that merely writing a program is already so difficult that fretting about performance is a luxury. It's no wonder then that the normal approach to performance was to *fix it later*!

But all of the adages nonetheless highlight an important insight—performance isn't distributed evenly through your code. The 80-20, or maybe even the 90-10 rule, applies here, because there are some hotspots where extreme care is needed, but we shouldn't try to optimize every nook and cranny in our code. So, our first guideline will be premature optimization—we should forget about it in, say, 95% of cases.

But what exactly are the 20%, 10%, or 5% of code where we shouldn't forget about it? Another old-age programming wisdom states this—programmers are notoriously bad at guessing performance bottlenecks.

So, we shouldn't try to predict the tight spot and measure the performance of a ready program instead. This does sound a lot like the *fix it later* cowboy coder's approach. Well, this book takes the stance that though premature optimization should be avoided, nonetheless, **premature pessimization** should be avoided at all costs, as it's even worse! However, avoiding premature pessimizations requires much detailed knowledge about which language constructs, which framework use cases, and which architectural decisions come with what kind of performance price tags. This book will try to provide this knowledge in the context of the Qt framework.

But, first, let's talk about quite general principles that address the question of what should be avoided, lest the performance degrades. As I see it, we can distill from the traditional performance wisdom from the following basic common-sense advice:

- Don't do the same thing twice.
- Don't do slow things often.
- Don't copy data unnecessarily.

You'll agree that all that can't be good for performance? So, let's discuss these three simple but fundamental insights in some more detail.

Avoiding repeated computation

The techniques falling under the first point are concerned with unneeded repetition of work. The basic counter measure here is **caching**, that is, saving the results of computation for later use. A more extreme example of avoiding repletion of work is to precompute results even before their first usage. This is normally achieved by hand-coded (or generated by a script) **precomputed tables** or, if your programming language allows that, with **compile-time computation**. In the latter case, we sacrifice compilation times for better run-time performance. We'll have a look at C++ compile time techniques in `Chapter 3`, *Deep Dive into C++ and Performance*.

Choosing the **optimal algorithm and data structure** also falls into that realm, as different algorithms and data structures are optimized for different use cases, and you have to make your choice wisely. We'll have a look at some gotchas pertaining Qt's own data structures in `Chapter 4`, *Using Data Structures and Algorithms Efficiently*.

The very basic techniques such as **pulling code out of a loop**, such as the repeated computations or initializations of local variables, fall into that class as well, but I'm convinced you knew about this already.

Avoiding paying the high price

The techniques falling under the second point come into play if there's something we can't avoid doing, but it has a pretty high cost tagged on to it. An example of this is interaction with the operating system or hardware, such as writing data to a file or sending a packet over the network. In this case, we resort to **batching**, also known in I/O context as **buffering**—instead of writing or sending a couple of small chunks of data right away, we first gather them and then write or send them together to avoid paying the high cost each time.

On the other hand, we can apply techniques of this type too. In I/O or memory context, this would be the **prefetching** of data, also known as **read-ahead**. When reading data from a file, we read more than the user actually requested, hoping that the next portion of data will be needed soon. In the networking context, there are examples of speculative **pre-resolving** of **Domain Name System** (**DNS**) addresses when a user is hovering over a link in browsers or even **pre-connecting** to such addresses. However, such measures can turn into its counterpart when the prediction fails, and such techniques require very careful tuning!

Related techniques to be mentioned in this context are also **avoidance of system calls** and **avoidance of locking** to spare the costs of system call and switching to the kernel context.

We'll see some applications of such techniques in last chapters of the book when we discuss I/O, graphics , and networking.

Another example of when this rule can be used is memory management. General-purpose memory allocators tend to incur rather high costs on single allocations, so the remedy is to **preallocate** one big buffer at first and then use it for all needs of the program by managing it by ourselves using a **custom allocation** strategy. If we additionally know how big our objects are going to be, we can just allocate several buffer pools for different object sizes, making the custom allocation strategy rather simple. Preallocating memory at the start used to be a classic measure to improve the performance of memory intensive programs. We'll discuss these technical C++ details in `Chapter 3`, *Deep Dive into C++ and Performance*.

Avoiding copying data around

The techniques falling under the third point tend to be somehow of a lower-level nature. The first example is avoiding copying data when passing parameters to a function call. A suitable choice of data structure will avoid copying of data as well—just think about an automatically growing vector. In many cases, we can use **preallocation** techniques to prevent this (such as the `reserve()` method of `std::vector`) or choose a different data structure that will better match the intended use case.

Another common case when the copying of data can be a problem is string processing. Just adding two strings together will, in the naive implementation, allocate a new one and copy the contents of the two strings to be joined. And as much of programming contains some string manipulations, this can be a big problem indeed! The remedy for that could be using static **string literals** or just choosing **a better library** implementation for strings.

We'll discuss these themes in `Chapter 3`, *Deep Dive into C++ and Performance,* and `Chapter 4`, *Using Data Structures and Algorithms Efficiently*.

Another example of this optimization rule is the holy grail of network programming—the zero-copy sending and receiving of data. The idea is that data isn't copied between user buffers and network stack before sending it out. Most modern network hardware supports **scatter-gather** (also known as vectored I/O), where the data to be sent doesn't have to be provided in a single contiguous buffer but can be made available as a series of separate buffers.

In that way, a user's data doesn't have to be consolidated before sending, sparing us copying of data. The same principle can be applied to software APIs as well; for example, Facebook's recent TSL 1.3 implementation (codename Fizz, open sourced) supports scatter-gather API on library level!

General performance optimization approach

Up to now, we listed the following classic optimization techniques:

- Optimal algorithms
- Optimal data structures
- Caching
- Precomputed tables
- Preallocation and custom allocators
- Buffering and batching
- Read-ahead
- Copy avoidance
- Finding a better library

With our current stand of knowledge, we can formulate the following general-performance optimization procedure:

1. Write your code, avoiding unnecessary pessimizations where it doesn't cost much, as in the following examples:
    - Pass parameters by reference.
    - Use reasonably good, widely known algorithms and data structures.
    - Avoid copying data and unnecessary allocations.

   This alone should give you a pretty decent baseline performance.

2. Measure the performance, find the tight spots, and use some of the standard techniques listed. Then, measure again and iterate. This step must be done if the performance of our program isn't satisfactory despite our sound programming practices. Unfortunately, we can't know or anticipate everything that will happen in the complex interplay of hardware and software—there can always be surprises waiting for us.
3. If you still can't achieve good performance, then your hardware is probably too slow. Even with performance optimization techniques, we still can't do magic, sorry!

The preceding advice looks quite reasonable, and you might ask: *Are we done? That wasn't that scary!* Unfortunately, it's not the whole story. Enter the leaky abstraction of modern processor architectures.

Modern processor architectures

All the classic performance advice and algorithmic foo stems from the times of simple CPU setups, where processor and memory speeds were roughly equal. But then the processor speeds exploded by increasing quite faithfully to the Moore law by 60% per year where memory access times increased by only 10% and couldn't quite hold pace with them. The problem is that the main memory (**dynamic random-access memory** (**DRAM**), contains minuscule capacitors keeping an electrical charge to indicate the 1 bit and none to indicate the 0 bit. This results in an inexpensive circuitry that doesn't have to be kept under voltage but is working basically in the analog realm and can't profit that much from advances made in the digital components.

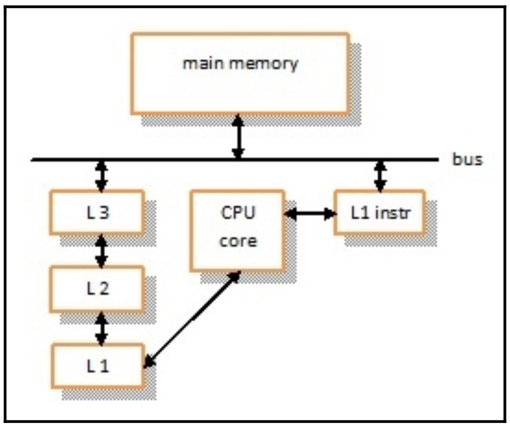
The second change that occurred since then was the demise of Moore's law in its simple form. Up to the early 2000s, CPU manufacturers steadily increased processor frequency rates, making CPUs run faster and faster. That was achieved by increasing the number of transistors packed on chips, and Moore's law predicted that number of transistors that can be packed on a chip will double every 18 months. In simple terms, it was understood as doubling the processor speed every two years.

This trend continued until processor manufacturers hit a physical barrier, the so-called *power wall*—at some point, the densely packed transistors produced so much heat that they couldn't be effectively cooled on consumer machines (high-end, expensive water-cooling systems are, too expensive for a laptop or a mobile device), so a different approach to increasing a CPU's performance had to be found.

Caches

The attempts to overcome these problems led to a slew of architectural innovations. First, the impedance between CPU and memory speeds was fought using a classic optimization technique we already know, namely, caching, on chip level. The on-chip **static RAM** (**SRAM**) memory requires six transistors (forming a flip-flop) per bit, and all of them must be kept under voltage. This means it's expensive and it drives the power consumption up (take care not to hit that power wall!). In exchange, the memory access times are at lightning speed, as all that is needed is to apply the current to the input and read the output.

So, the idea is to add a small caching stage of expensive but fast on-chip memory in front of big, slow but inexpensive main memory. Meanwhile, modern CPUs can command up to three levels of caches, commonly denoted with **L1**, **L2**, and **L3** acronyms, decreasing in density and speed but increasing in size as the cache level goes up. The figure below shows us an overview of the memory hierarchy typically found in modern CPUs:



As of the time of this writing, access times for **L1** caches are in the order of **3** cycles, **L2** of **12** cycles, **L3** of **38** cycles, and main memory is around **100-300** cycles. The main memory access time is that high because the analog nature of DRAM requires, among other things, periodic charge refreshing, pre-charging of the read line before reading, analog-digital conversion, communication through **memory controller unit** (**MCU**), and so on.

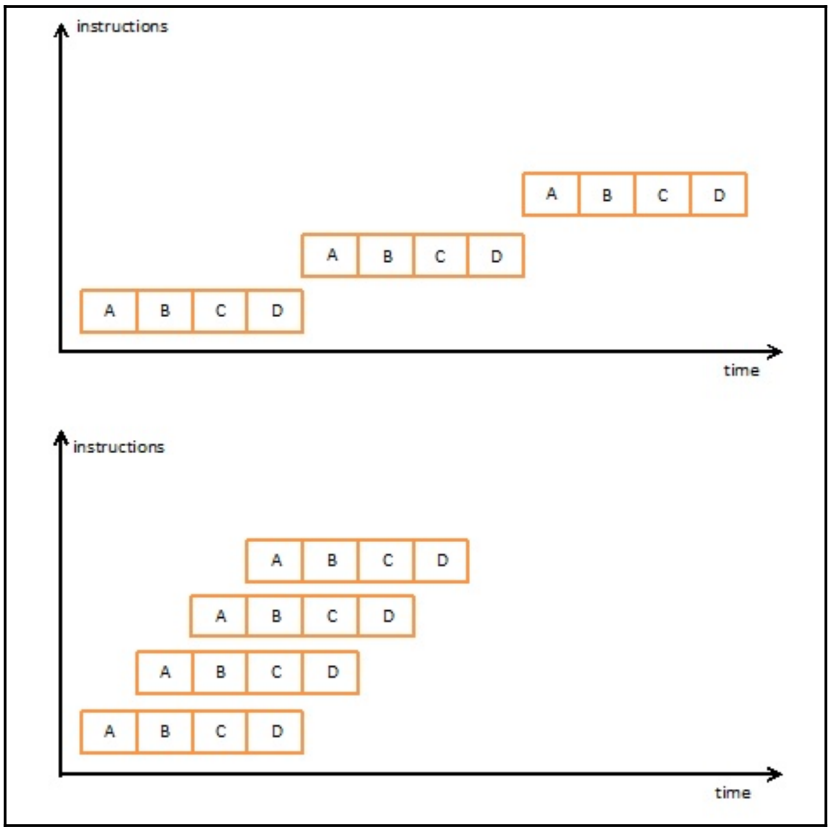
Caches are organized in cache lines, which on the current Intel architectures are 64 bytes long. Each cache update will hence fetch the entire cache line from **main memory**, doing a kind of prefetching already at that level. Speaking about prefetching, Intel processors have a special prefetch instruction we can invoke in assembler code for very low-level optimizations.

In addition to data caches, there's also an instruction cache, because in the *von Neumann* architecture, both are kept in the common memory. The instruction caches were added to **Intel Pentium Pro** (**P6**) as an experiment, but they were never removed since then.

Pipelining

Another possibility to increase a processor's speed is the **instruction level parallelism** (**ILP**), also known as **superscalar computation**.

The processing of a CPU instruction can internally be split into several stages, such as instruction fetch, decode, execute, and write-back. Before the Intel 486 processor, each instruction has to be finished before the next can be started. With pipelining, when the first stage of an instruction is ready, that instruction can be forwarded to the next stage, and the next instruction's processing can begin with its first stage. In that manner, several instructions can be in flight in parallel, keeping a processor's resources optimally utilized. The next screenshot illustrates this principle, graphically, using a hypothetical four-stages pipeline:



The original Intel 486 pipeline was five stages long, but on modern processors, it can be much longer. For example, the current Intel Atom processors command a pipeline of 16 stages.

That's all well and good, but, unfortunately, there are some problems lurking in the corners.

Speculative execution and branch prediction

As long as the pipeline is pumping, everything is OK. But what if we encounter a conditional branch? The pipeline has to wait till the result of the test is known, hence the next instruction can be started only when the current one has finished. Welcome to the pre-80,486 world! This is called a **pipeline stall** and defeats the whole purpose of pipelining. And because every program is literally dotted with `if-then-else` clauses, something must be done here.

The solution manufacturers came up with was **speculative execution**: instead of idling, we just start executing one of the branches speculatively. If we are lucky, we've just done the right thing, but, if not, we discard our speculative work, and we are on even ground with the pipeline stall case. As we decide randomly, we'll be right 50% of the time, and we just seriously increased the throughput of the pipeline!

The only problem is that the branches of the `if` clause are not equally probable! In most cases, they're even highly unevenly distributed: one of them is the error branch; the other is the normal case branch. But the processor doesn't know the meaning of the test, so what can we do? The solution to that is branch predicting—the processor is learning about branches in your code and can predict which branch will be taken on a given condition rather well.

This got **complicated quickly**, didn't it? If you're thinking it, you're not alone. Not so long ago, the programming world was shaken by disclosure of the Spectre and Meltdown vulnerabilities, which allowed the attacker to see contents of the memory regions where they don't have access rights. The first part of the exploit's to fool the branch predictor to take the false branch for speculative execution. After the processor sees a disallowed access, the instruction will be retired, but the protected data will be present in the cache, where they can be guessed with some complicated techniques we won't discuss here. These bugs basically put in question the processor optimizations of the last decade, as fixing them would incur meaningful performance losses.

Considering that, we are all rather curious about how CPU architectures evolve next time, aren't we?

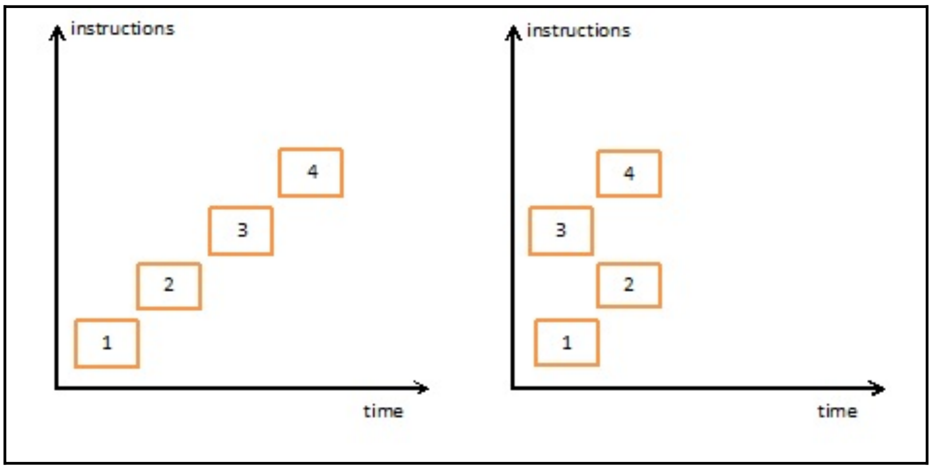
Out-of-order execution

There's another refinement to the pipeline concept allowing an even higher utilization of a CPU's resources. Namely, as processor manufacturers started to add redundant processing units (Intel P6 already had two integer and two floating-point execution units), it became possible to execute two instructions in parallel.

Up until Pentium Pro (P6), instructions were fed into the pipeline in their order of appearance. But if there's a data dependency between two consecutive instructions, then they can't be processed in parallel, leaving the additional execution unit idle:

```
a = b + 1;  // 1
c = a + 5;  // 2
d = e + 10; // 3
f = d + 15; // 4
```

The solution to this problem is to take the next independent instruction and execute it before the dependent one. See the next diagram for a visual explanation:



Here, on the left side, we see the traditional execution preserving the instruction order and, on the right, parallel execution with reordering, where instruction **3** will be executed before instruction **2**.

Multicore

The problem of the power wall was in the end overcome by freezing or even decreasing CPU frequencies but introducing parallelly working processor cores, adding more general registers, vector processing **single instruction multiple data** (**SIMD**) registers, and instructions. In a word, they either duplicated active processing units or added more elements that don't have to be always under voltage. In that way, the density of the transistors didn't have to be increased.

When all of the CPU cores are placed on one chip, we have a **symmetric multiprocessing** (**SMP**) CPU, because all cores can access their respective data within a local chip. The counterpart to that is a **non-uniform memory access** (**NUMA**) system, where we have several physically separate CPUs having their own internal caches. The memory access for internal CPUs will then be much cheaper than the memory access to an external CPU. Another problem is the cache coherence between the CPUs, which requires complicated cache-coherence protocols and can take down performance. In the context of Qt's application area, we normally encounter SMP machines, so we'll ignore NUMA in this book.

In a multicore chip, the processing resources can be classified in **core** and **uncore** ones—those which are duplicated for each core, and those that aren't and must be shared. For example, the top-level cache (L3 or L2, depending on the processor) is an uncore resource shared among processor cores.

One often-encountered notion is that of **hyperthreading**. This is another idea for increasing a CPU's parallelism, and hence its resource utilization. A processor with hyperthreading consists of two logical processors per core, each of which keeps its own internal state. The parts of the processor where it holds its architectural state (for example, running, interrupted, and halted) will be duplicated for each core, but the computation resources will be shared among logical cores. The intent of that is to increase the utilization of the processor's resources and prevent pipeline stalls, by borrowing resources from the stalled logical core. The operating system then uses these two logical cores as physical ones and has to be HTT-aware to optimally use such a system.

Strictly speaking, **graphical-processing units** (**GPUs**) are not a form of multicore, but we'll mention them in this context, because processors can ship with integrated GPUs on board. A GPU comprises many very simple processing units that can run massively parallel, although simple, computations. Normally, they're used to accelerate graphic processing, but they can be also used to speed up in general computations, such as the training of neural networks in deep-learning applications. In a Qt context, however, we use GPUs only for their graphical capabilities.

Additional instruction sets

As already mentioned, to increase processor performance chip, manufacturers started to add more sophisticated instructions that can either vectorize computations or execute algorithms that hitherto had to be implemented in application code.