

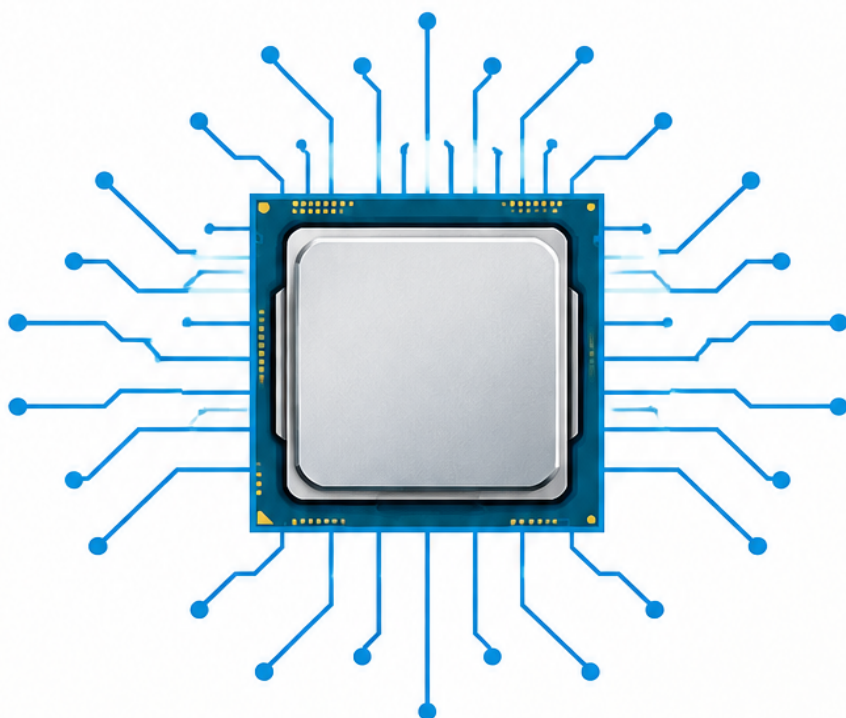
CPU

PERFORMANCE

ENGINEERING

From Architecture to Optimization

A Practical Guide to Making Software Faster



DANIEL HARROW

CPU Performance Engineering

From Architecture to Optimization - A Practical Guide to Making Software Faster

Steve Publications

This book is available at <https://leanpub.com/cpuperformanceengineering>

This version was published on 2026-10-06



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2026 Steve Publications

Contents

From Architecture to Optimization - A Practical Guide to Making Software Faster	1
Introduction: The Machine Beneath the Abstraction	2
Why Performance Still Matters	3
Why This Book Is Different	3
How to Read This Book	4
About the Repository Analysis	5
What This Book Will Not Do	5
Getting Started	6
Chapter Summary	6
Chapter 1: Why Performance Matters - and Why It Is Hard	7
The End of Free Lunch - Moore's Law and the Performance Wall	7
Latency vs. Throughput - Two Different Problems	8
The Cost of Performance Mistakes - Real-World Economics	9
Why Intuition Fails - Counterintuitive Behavior of Modern CPUs	10
A Scientific Approach - The Performance Engineering Methodology	11
What This Book Will and Will Not Cover	12
Chapter Summary	13
Chapter 2: Inside the Modern CPU - A Mental Model	14
The Von Neumann Architecture and Its Limitations	14
The Fetch-Decode-Execute Cycle - And Why It Is Not That Simple	14
Cores, Threads and SMT - Understanding Compute Resources	16
The Processor as a Factory - Pipeline, Buffer and Queue Analogies	17
Why Clock Speed Is Only Part of the Story	17
Instruction Set Architectures - x86-64 and ARM64 Compared	18
Chapter Summary	19
Chapter 3: Instructions and the Instruction Set	21

CONTENTS

What Is an Instruction - Anatomy and Encoding	21
RISC vs. CISC - History, Philosophy and Modern Reality	21
Micro-ops - How x86 Instructions Become Executable Work	21
Instruction Latency vs. Instruction Throughput - Two Different Num- bers	21
Addressing Modes and Their Performance Cost	21
Reading Agner Fog and Processor Reference Manuals	22
Chapter Summary	22
Chapter 4: Pipelines and Superscalar Execution	23
The Pipeline - Stages, Hazards and Throughput	23
Superscalar Design - Issuing Multiple Instructions Per Cycle	23
Execution Ports and Resource Constraints	23
Pipeline Depth - Benefits and the Cost of Mispredictions	23
Frontend vs. Backend Bottlenecks	23
Architectural Implications - How to Write Code for Superscalar CPUs	23
Chapter Summary	24
Chapter 5: Out-of-Order Execution and Instruction-Level Parallelism	25
The Problem - Dependencies and Stall Cycles	25
Out-of-Order Execution - Scoreboarding and Tomasulo's Algorithm	25
The Reorder Buffer and Commit Stage	25
True Dependencies, Anti-Dependencies and Output Dependencies	25
Instruction-Level Parallelism - Measuring and Exploiting It	25
What You Can Control - Software Techniques to Increase ILP	26
Chapter Summary	26
Chapter 6: Branch Prediction and Speculative Execution	27
The Branch Problem - Why Control Flow Kills Throughput	27
Branch Prediction Mechanisms - Static and Dynamic Predictors	27
Speculative Execution - Running Before You Know	27
Branch Misprediction Penalty - Quantifying the Cost	27
Writing Predictable Code - Data-Dependent Branches and Conditional Moves	27
Security Implications - Spectre, Meltdown and Their Microcode Fixes	28
Chapter Summary	28
Chapter 7: Instruction Scheduling and Code Layout	29
Instruction Fetch - Width, Alignment and Cache Lines	29
Code Alignment and Its Impact on Pipeline Performance	29

CONTENTS

Software Pipelining - Manual Instruction Scheduling	29
Compiler Scheduling - What Compilers Do and What They Cannot See	29
Loop Layout and Branch Placement	29
Reading and Optimizing Assembly Output	29
Chapter Summary	30
Chapter 8: Caches - The Memory Hierarchy	31
The Memory Wall - Why Main Memory Is Too Slow	31
Cache Levels - L1, L2, L3 and Their Roles	31
Cache Internals - Lines, Sets, Ways and Associativity	31
Cache Access Types - Hits, Misses and Conflict Misses	31
False Sharing - A Subtle Performance Killer	31
Cache-Friendly Programming - Locality, Stride and Data Structures	31
Chapter Summary	32
Chapter 9: Cache Coherency and Multi-Core Communication	33
The Cache Coherency Problem - Multiple Copies, One Truth	33
MESI and MOESI Protocols - How CPUs Keep Caches in Sync	33
The Snooping Bus and Directory-Based Coherency	33
Coherency Traffic - Measuring Its Impact	33
Writing Cache-Coherency-Aware Code	33
NUMA and Cache Coherency in Large Systems	34
Chapter Summary	34
Chapter 10: Virtual Memory and Translation Lookaside Buffers	35
Virtual Memory - Isolation, Abstraction and Performance Cost	35
Page Tables and Page Faults - Walking the Tree	35
TLBs - Purpose, Hierarchy and Miss Penalties	35
TLB Shootdowns - The Multi-Core Cost	35
Huge Pages - Reducing Translation Overhead	35
Memory Mapping Strategies for Performance	35
Chapter Summary	36
Chapter 11: Memory Access Patterns and Bandwidth	37
Sequential vs. Random Access - The Performance Gap	37
Spatial and Temporal Locality - Exploiting Hardware Prefetching	37
Memory Bandwidth - Measuring and Maximizing It	37
Prefetching - Hardware and Software Techniques	37
Data Layout Transformations - AOS vs. SOA and Beyond	37
Benchmarking Memory Performance - Real Numbers on Real Hardware	37

Chapter Summary	38
Chapter 12: SIMD and Vectorization	39
SIMD Fundamentals - Doing More Work Per Instruction	39
x86 SIMD Families - SSE, AVX, AVX2, AVX-512	39
ARM NEON and SVE - Vectorization on ARM	39
Auto-Vectorization - When Compilers Succeed and Fail	39
Manual Vectorization - Intrinsics and Assembly	39
Common Pitfalls - Alignment, Reductions and Control Flow	39
Chapter Summary	40
Chapter 13: Multithreading and Hardware Concurrency	41
Hardware Multithreading - Simultaneous Multithreading (SMT)	41
Software Threads - Mapping to Hardware	41
Scaling Performance - Amdahl's Law and Gustafson's Law	41
Thread Affinity and NUMA Awareness	41
Load Balancing - Dynamic and Static Scheduling	41
When Multithreading Hurts - Resource Contention and False Sharing	42
Chapter Summary	42
Chapter 14: Synchronization and Atomic Operations	43
Atomic Operations - The Hardware Foundation	43
Memory Ordering and Barriers - Sequential Consistency vs. Weaker Models	43
Locks - Mutexes, Spinlocks and Their Overhead	43
Lock-Free and Wait-Free Algorithms - Possibilities and Limits	43
Read-Write Locks and Fine-Grained Locking	43
Lock Coarsening, Elimination and False Sharing	44
Chapter Summary	44
Chapter 15: Performance Counters and Hardware Monitoring	45
What Are Performance Counters - Hardware Events and Counters . .	45
Intel PMU, AMD IBS and ARM PMU - Vendor-Specific Features	45
Key Metrics - IPC, Cache Miss Rates, Branch Mispredictions	45
Using perf - The Linux Performance Toolkit	45
Statistical Sampling vs. Precise Events	45
Designing Counter-Based Experiments	46
Chapter Summary	46
Chapter 16: Profiling - Understanding What Your Code Actually Does .	47

CONTENTS

Sampling vs. Instrumentation - Trade-Offs	47
Flame Graphs - Visualizing Execution Profiles	47
CPU Profiling - Finding Hot Paths	47
Cache Profiling - Identifying Memory Bottlenecks	47
Branch Profiling - Understanding Control Flow Performance	47
Profiling in Production - Sampling, Tracing and Overhead	47
Chapter Summary	48
Chapter 17: Benchmarking - Measuring What Matters	49
Microbenchmarks vs. Macrobenchmarks - When to Use Each	49
Experimental Design - Controls, Variables and Repetition	49
Statistical Variation - Warmup, Cooling and Outliers	49
Measuring Latency - Precision Timers and Statistical Analysis	49
Measuring Throughput - Sustained Performance and Sustaining It	49
Common Benchmarking Mistakes - And How to Avoid Them	50
Chapter Summary	50
Chapter 18: Compiler Optimization - Working With the Compiler	51
Compiler Optimization Pipeline - What Happens Under the Hood	51
Optimization Levels - What -O2 and -O3 Actually Do	51
Compiler Flags - Tuning for Your Target	51
Reading Optimization Reports - -Rpass and -Rpass-missed	51
LTO and PGO - Link-Time and Profile-Guided Optimization	51
When the Compiler Fails - Inline Assembly, Intrinsics and Hand-Tuning	52
Chapter Summary	52
Chapter 19: CPU Frequency, Power Management and Thermal Limits	53
Turbo Boost and Dynamic Frequency Scaling - How It Works	53
Power Limits - TDP, PL1, PL2 and Their Real-World Impact	53
Thermal Throttling - When Heat Becomes the Bottleneck	53
Measuring Real Frequency - Is Your CPU Running at Full Speed?	53
Disabling Power Management for Benchmarks	53
Performance per Watt - Optimizing for Efficiency, Not Just Speed	54
Chapter Summary	54
Chapter 20: Operating System Interactions and System Configuration	55
Scheduler Effects - Context Switches and Priority Inversion	55
Interrupt Handling - Softirqs, Hardirqs and Latency	55
CPU Isolation - Isolating Cores for Latency-Sensitive Workloads	55
Kernel Configuration - Tuning for Performance	55

CONTENTS

cgroups and Resource Controls - Containment and QoS	55
Virtualization Overhead - When Your VM Hurts Performance	56
Chapter Summary	56
Chapter 21: Performance Analysis Methodologies - A Scientific Approach	57
The Performance Engineering Workflow - Measure, Hypothesize, Optimize, Validate	57
Establishing a Baseline - Reproducibility and Regression Prevention .	57
Formulating Hypotheses - From Symptoms to Root Causes	57
The Bottleneck Taxonomy - CPU-Bound, Memory-Bound, I/O-Bound	57
Prioritizing Optimizations - Impact vs. Effort	57
Validating Results - Avoiding Premature Celebration	58
Chapter Summary	58
Chapter 22: Case Studies in Performance Engineering	59
Case Study 1 - Optimizing a Hash Table: Cache Locality and False Sharing	59
Case Study 2 - Vectorizing Image Processing: From Naive to AVX-512	59
Case Study 3 - Reducing Lock Contention in a High-Throughput Server	59
Case Study 4 - NUMA-Aware Memory Allocation in a Database	59
Lessons Learned - Common Patterns Across Cases	59
Chapter Summary	60
Chapter 23: Critical Analysis of the CPU-Performance-Engineering Repository	61
Repository Overview - Structure and Scope	61
Strengths - What It Gets Right	61
Weaknesses and Omissions - What Is Missing or Under-Explained . .	61
Accuracy Assessment - Comparing to Primary Sources and Reference Manuals	61
Outdated or Misleading Material - Specific Examples and Corrections	61
How to Use This Repository - Recommendations and Caveats	62
Chapter Summary	62
Conclusion: The Performance Engineering Mindset	63
The Key Principles - A Condensed Checklist	63
What Changes and What Stays the Same	63
Future Trends - Chiplets, AI Accelerators and New Memory Technologies	63
Continuing Your Journey - Resources and Communities	63
Final Thoughts	63

References 64

From Architecture to Optimization - A Practical Guide to Making Software Faster

This book explains how modern CPUs actually work, why software runs fast or slow and how to systematically measure, diagnose and improve performance. It covers the fundamentals of processor architecture and the advanced techniques used by systems programmers to extract every last cycle of performance. Every concept is grounded in real hardware behavior, measured with real tools and explained with real code. Whether you are writing a high-frequency trading engine, a database kernel, a game engine or simply code that refuses to be slow, this book will give you the knowledge to understand what your processor is doing with your instructions and how to make it do more, faster.

Introduction: The Machine Beneath the Abstraction

There is a gap between how programmers think about computers and how computers actually work.

You write code in terms of variables, loops, function calls and data structures. The compiler translates your code into machine instructions. The CPU executes those instructions and your program runs. Or so the story goes.

The reality is that modern processors execute instructions in ways that would seem bizarre if you did not know the design goals that led to them. They guess which branch your if-else will take and run code speculatively before knowing whether they are right. They execute instructions out of order and keep dozens of operations in flight at once. They cache data in tiny, meticulously structured pools of memory and invalidate those caches across cores whenever anyone writes to a shared line. All of this machinery exists to hide latency and keep the processor busy.

Understanding that machinery is the entire premise of this book.

Performance engineering is not guesswork. It is not premature optimization or hand-waving about making things faster. It is the practice of understanding what hardware does, measuring how your software interacts with that hardware and then making deliberate changes that improve the interaction. Done well, it can turn a sluggish system into a responsive one, a throughput bottleneck into a smooth pipeline, a tail latency problem into a solved one.

The challenge is that the modern CPU is one of the most complex pieces of engineered hardware in existence. A single core may contain billions of transistors arranged into pipelines, caches, branch predictors, execution units and speculative execution engines. The gap between programmer intent and hardware behavior can be enormous and the consequences of misunderstanding are severe. Code that looks simple and correct can run orders of magnitude slower than necessary due to cache misses, branch mispredictions, false sharing, or NUMA misalignment. Code that looks complex and clever can

run faster than naive code because it aligns with the actual behavior of the silicon.

Why Performance Still Matters

For decades, performance was automatic. Moore's Law delivered faster processors every year and software got away with whatever inefficiencies it introduced. By the time the inefficiencies compounded into a noticeable problem, the next generation of hardware had absorbed the cost.

That era ended. Single-thread performance growth has slowed dramatically since the mid-2000s as processors hit physical limits on clock speed and power consumption. Transistor counts continue to grow but they are increasingly spent on multiple cores, caches and specialized units rather than making individual cores faster. The free lunch is over, as Herb Sutter famously put it in 2005 and we have been paying for the bill ever since.

This has real consequences. A database query that takes one millisecond instead of two affects user experience at scale. A trading algorithm that executes microseconds faster captures profit others miss. A machine learning inference workload that achieves higher throughput reduces cloud costs. A game engine that maintains higher frame rates keeps users engaged. A compiler that produces faster code gives every program compiled with it a permanent advantage.

Performance is not just about speed. It is about resource efficiency, energy consumption, cost and user experience. And performance problems are often hidden, showing up only under load, under specific conditions or in the tail of latency distributions where averages hide the truth.

Why This Book Is Different

There are many resources on CPU performance: research papers, architecture manuals, blog posts, conference talks, online courses and tool documentation. Many of them are excellent. Most of them are not books. And most of them are not written for the reader who wants to understand the whole picture, from foundational concepts to advanced practice, without assuming a PhD in computer architecture.

This book is written for that reader. It assumes you know how to program and have some familiarity with systems concepts. It does not assume you understand Tomasulo's algorithm, cache set associativity, or the difference between a reorder buffer and a reservation station. It builds those concepts from the ground up and uses them to explain real-world performance behavior.

Every major concept in this book is explained with a consistent framework:

- What it is
- Why it matters
- How it works internally
- When it is useful
- What the performance implications are
- Common misconceptions
- Practical trade-offs
- How it interacts with other parts of the system

This framework ensures that you do not just memorize facts but develop a working mental model of the CPU that you can use to reason about new problems you have never seen before.

The book is also grounded in measurement. Performance without measurement is just opinion. Every optimization technique discussed is tied to ways of measuring whether it helped using tools you can actually run on real hardware. The tools include Linux perf, hardware performance counters, flame graphs, microbenchmarks and analysis frameworks like Intel's Top-down Microarchitecture Analysis. Understanding what these tools measure and how to interpret their output is as important as knowing what to optimize.

How to Read This Book

The book is organized in a deliberate progression. The first section establishes foundational concepts: how processors work at a high level, what instructions are and how they relate to the code you write. The second section dives into the heart of the CPU core, explaining superscalar execution, out-of-order execution, branch prediction and instruction-level parallelism. The third section covers the memory subsystem, where most real-world performance problems live. The fourth section addresses parallelism, covering SIMD, multithreading and synchronization. The fifth section focuses on tooling

and methodology, teaching you how to measure and diagnose performance problems systematically. The final sections cover advanced topics including power management, OS interactions and case studies.

You should read the book in order, at least through the foundational sections. Later chapters assume knowledge of earlier ones. Once you have the fundamentals, you can dip into specific chapters as needed. The case studies in the final chapters tie everything together and show how the concepts interact in real optimization scenarios.

Some sections include code examples in C, C++, Rust and assembly. These are chosen to illustrate specific points about how different implementations interact with CPU behavior. You do not need to be fluent in all of these languages to benefit from the examples but familiarity with at least one systems programming language is assumed.

About the Repository Analysis

A notable resource in the CPU performance community is the GitHub repository `cpu-performance-engineering` by `usamahz`, which provides a structured reading path covering instruction lifecycle, microarchitecture, memory hierarchy, tools and more. The repository has strengths: its emphasis on primary sources, its structured organization and its requirement for reproducible data points are all valuable contributions. The repository is referenced throughout this book where relevant. In a dedicated chapter near the end, the repository is analyzed in detail: its strengths, its gaps, its outdated or potentially misleading content and how it fits into the broader landscape of performance engineering knowledge.

The critical analysis is not meant to dismiss the repository. It is meant to show how to evaluate any resource on CPU performance by comparing claims against primary sources and real measurements. That practice of critical evaluation is one of the most important skills a performance engineer can develop.

What This Book Will Not Do

This book will not teach you how to program. If you need a primer on C++, Rust, or systems programming, there are many excellent books for that purpose.

This book will not cover every optimization technique that exists. The space is too large and too workload-specific. Instead, it teaches you the principles and methods you need to discover the right optimization for your specific situation.

This book will not give you magic tricks. There are no secret compiler flags or hidden CPU registers that make everything faster. Performance comes from understanding and alignment, not hacks.

Getting Started

The most important thing you can do as you read this book is develop the habit of questioning your assumptions about performance. If you think sorting an array is fast, measure it. If you think a branch is cheap, profile it. If you think two threads are running in parallel, check for false sharing. Performance engineering begins with curiosity and measurement and ends with understanding.

Let us begin by understanding the machine.

Chapter Summary

Key takeaways from this introduction:

- Modern CPUs execute instructions in complex ways that are very different from how programmers conceptualize execution. Understanding that hardware behavior is essential for writing fast code.
- The era of automatic performance improvement via Moore's Law is over. Performance engineering is now necessary rather than optional.
- Performance engineering is a measurement-driven discipline, not guesswork. Every optimization should be verified with real data from real hardware.
- This book builds understanding progressively from foundations to advanced topics and uses a consistent framework for explaining each concept.
- Critical evaluation of sources is essential. Claims about CPU performance should be verified against primary sources and measurement.

Chapter 1: Why Performance Matters - and Why It Is Hard

The End of Free Lunch - Moore's Law and the Performance Wall

For roughly fifty years, software engineers operated under an implicit guarantee: if you made your code a little less efficient, the next generation of hardware would make it fast again. This was Moore's Law in practice, the observation that transistor counts in integrated circuits approximately doubled every two years. More transistors meant faster processors, bigger caches and more capabilities, all at roughly the same cost.

The result was a culture of performance indifference. Engineers optimized for maintainability, correctness and developer velocity and let the hardware absorb the cost. If a program was slow, you upgraded the server. If a website was sluggish, you added more machines. Performance was a managed resource, not a design constraint.

That era ended in the mid-2000s. Clock speeds plateaued around 3 to 4 GHz as physical limits on power consumption and heat dissipation made further increases impractical. The solution was to add more cores, but that only helps if your software can actually use them. Single-threaded performance, the raw speed at which a single core executes a sequence of instructions, grew much more slowly than it had before.

Modern CPUs still get faster, but the improvement comes from microarchitectural refinements, wider execution pipelines, better branch predictors and more aggressive parallelization, not from simply running at higher clock speeds. Intel's IPC (instructions per cycle) improvements from Skylake to Golden Cove were roughly 15 to 20 percent. AMD's Zen 1 to Zen 4 progression showed more dramatic improvement, but even there, single-thread gains per generation are typically in the 10 to 25 percent range. These are real improvements, but they are nowhere near the factor-of-two every-two-years trajectory that Moore's Law once implied for single-thread performance.

For software that cannot use multiple cores, or for the critical single-threaded paths that bottleneck overall performance, this slowdown has real consequences. Performance problems that used to disappear with the next hardware refresh now persist and compound. Software that was fast five years ago may be unacceptable today, even on the latest hardware, simply because the workload grew larger while the per-thread execution speed improved more slowly than the data.

Latency vs. Throughput - Two Different Problems

The first distinction you must make when thinking about CPU performance is between latency and throughput. They are not the same thing and they often require different optimization approaches.

Latency is the time it takes for a single operation to complete, from start to finish. If a function takes two microseconds to execute, its latency is two microseconds. Latency is what matters when you care about how fast an individual request is processed: a web server responding to a client, a trading system executing an order, a game engine rendering a frame.

Throughput is the amount of work completed per unit of time. If a server processes ten thousand requests per second, its throughput is ten thousand requests per second. Throughput is what matters when you care about how much work the system can handle overall: a database processing queries, a video encoding pipeline converting footage, a machine learning model serving predictions.

The distinction matters because optimizing for one can hurt the other. Techniques that reduce latency for individual operations, such as minimizing queueing delays or avoiding batching, may reduce throughput by underutilizing hardware resources. Techniques that maximize throughput, such as batching operations or using deep pipelines, may increase latency for individual operations because they wait for the batch or pipeline to fill.

Consider a simple example. You have a function that processes one item at a time. Each processing takes one microsecond of CPU work, but there is also a memory access that adds five microseconds of latency. For latency, you want to minimize the time from input to output for a single item. You might optimize memory layout so the access is cached, reducing the effective latency to one microsecond. For throughput, you might batch multiple items together and

process them in a vectorized loop that keeps the execution units busy. Even though each item in the batch might wait for the batch to be filled, the overall rate at which items are processed could increase dramatically.

Many real-world systems need both low latency and high throughput. That is why performance engineering is hard: you often have to optimize for multiple objectives that pull in different directions. Understanding the distinction is the first step to making deliberate trade-offs.

The Cost of Performance Mistakes - Real-World Economics

Performance is not an abstract concern. It has direct economic consequences that scale with the size of your system.

A study of large-scale web services found that response time delays of just one hundred milliseconds had measurable effects on user behavior, including reduced page views, lower conversion rates and decreased customer satisfaction. At the scale of major platforms, a one-hundred-millisecond improvement in response time translates to millions of dollars in additional revenue per year.

In high-frequency trading, latency differences of microseconds determine profitability. Firms spend millions on hardware, network equipment and software optimization to reduce latency by fractions of a millisecond. The economic incentive is straightforward: being faster than competitors means seeing prices first and capturing trades before others can react.

In cloud computing, performance directly affects cost. A workload that runs in two seconds instead of four seconds uses half the compute resources, which means half the cloud bill. Organizations that optimize their code for efficiency can reduce infrastructure costs substantially, especially at scale.

Even in desktop and mobile applications, performance matters. Users abandon slow websites and apps. Battery life on mobile devices depends heavily on how efficiently software uses CPU cycles. A well-optimized app runs faster, uses less power and keeps users happier.

The economic argument for performance is clear, but the practical challenge is knowing where to invest effort. Not every optimization is worth it. A function that takes one percent of total execution time can be made ten times faster and the overall program will only be one percent faster. Performance

engineering is about focusing effort where it has the biggest impact, which requires measurement and understanding.

Why Intuition Fails - Counterintuitive Behavior of Modern CPUs

Modern CPUs are not intuitive. Their behavior can be surprising to programmers who think about execution in terms of the sequential logic of their source code. Here are some examples that illustrate the gap between intuition and reality.

Branches are expensive, but predictable branches are almost free. A naive model of execution says that an if-else branch is just a conditional jump: the processor checks the condition and goes to the appropriate place. In reality, modern CPUs use deep pipelines and branch prediction to keep the pipeline full. When a branch is predictable, the processor guesses the outcome and continues fetching instructions from the predicted path. If the guess is correct, there is no penalty. If the guess is wrong, the processor must discard all the work done along the wrong path and start over, losing 15 to 25 cycles or more on modern deep pipelines. Code with unpredictable branches can run dramatically slower than branchless code, even though a naive reading suggests they should be equivalent.

Memory access is not uniform. A naive model says that accessing any variable in memory takes the same time. In reality, memory is organized in a hierarchy with dramatically different latencies. A variable in the L1 cache takes roughly four cycles to access. A variable in L2 might take 12 to 15 cycles. L3 might take 30 to 50 cycles. Main memory takes 100 to 300 nanoseconds, which at a 4 GHz clock is hundreds of cycles. Two loops that iterate over arrays of the same size can run at very different speeds depending on whether the data fits in cache or must be fetched from main memory every time.

Parallel threads can slow each other down without sharing any data. Two threads writing to different variables might seem independent, but if those variables happen to be within the same cache line, a phenomenon called false sharing can serialize the threads. Each write invalidates the cache line in the other core, causing both cores to fetch the line repeatedly over the cache coherency protocol. The performance degradation can be severe, even though no actual data conflict exists.

Adding more cores does not always make your program faster. Amdahl's Law states that the speedup from parallelization is limited by the fraction of work that cannot be parallelized. If 10 percent of your program is inherently sequential, adding unlimited cores will give you at most a ten-fold speedup. In practice overhead from synchronization, communication and load imbalance further reduce the benefit. Some workloads actually run slower on more cores because the overhead exceeds the parallelism.

These examples illustrate why performance engineering requires a solid understanding of the underlying hardware. Intuition based on high-level programming constructs is not enough. You need to know how the CPU actually executes your code.

A Scientific Approach - The Performance Engineering Methodology

Performance engineering is not about guessing which optimization will help and hoping it does. It is a systematic, measurement-driven discipline that follows a clear methodology:

1. Define the goal. What are you trying to improve? Latency? Throughput? Tail latency? Power consumption? Performance without a clear objective is just noise. You cannot optimize everything at once and trying to do so often leads to trade-offs that make nothing better.

2. Establish a baseline. Measure the current performance using reliable, repeatable methods. You need a reference point to compare against. This means choosing appropriate metrics (latency percentiles, throughput, CPU utilization), using proper tools (profilers, performance counters, benchmarks) and ensuring measurement conditions are controlled and reproducible.

3. Profile and measure. Use tools to understand where the program spends its time and where it encounters bottlenecks. CPU profilers show hot paths. Hardware performance counters reveal cache miss rates and branch misprediction rates. Flame graphs visualize call stack distributions. The goal is to identify the specific code regions or patterns that are causing poor performance.

4. Form a hypothesis. Based on your measurements, form a hypothesis about what change will improve performance. This hypothesis should be

grounded in your understanding of the CPU architecture and the behavior you observed. For example, if profiling shows a high L3 cache miss rate in a particular loop, a hypothesis might be that restructuring the data layout for better cache locality will reduce cache misses and improve performance.

5. Implement and measure. Make the change and measure again using the same metrics and methods as the baseline. The only way to know if the change helped is to measure it. Do not rely on intuition or theory.

6. Iterate. If the change helped, look for the next bottleneck. If it did not, analyze why and form a new hypothesis. Performance optimization is typically iterative, addressing one bottleneck after another. Fixing one bottleneck often reveals the next one.

This methodology is not rocket science, but it is easy to violate. Common mistakes include optimizing without measuring, optimizing the wrong part of the code, making changes based on theory without verifying and failing to account for measurement noise and variability. The scientific approach avoids these mistakes by making measurement and evidence central to every decision.

What This Book Will and Will Not Cover

This book focuses on CPU performance engineering: understanding how processors execute instructions and how software can be written to exploit that execution model. The primary scope includes:

- CPU architecture and instruction execution
- Pipelines, superscalar execution, out-of-order execution and branch prediction
- Caches, memory hierarchy and memory access patterns
- Virtual memory and TLBs
- SIMD and vectorization
- Concurrency, multithreading and synchronization
- Performance measurement tools and methodologies
- Compiler optimization techniques
- System-level factors: NUMA, OS scheduling, power management

The book covers x86-64 as the primary architecture because it is the dominant server and desktop architecture and has the most detailed public

documentation. ARM64 is covered where relevant, especially for concepts that differ from x86 or where ARM-specific features matter.

The book does not focus on GPU programming, database internals, or networking, though it touches on those areas when they relate to CPU performance. It also does not cover JIT compilation and garbage-collected languages in depth, though many principles apply. The focus is on understanding the CPU itself, which is valuable regardless of the specific language or domain you work in.

Chapter Summary

Key takeaways from Chapter 1:

- Moore's Law no longer delivers automatic performance improvements for single-threaded work. Performance engineering is now essential.
- Latency and throughput are distinct objectives that often require different optimization strategies.
- Performance has real economic consequences that scale with system size.
- Modern CPU behavior is counterintuitive. Branch prediction, cache hierarchy and cache coherency can make seemingly equivalent code run at vastly different speeds.
- Performance engineering is a systematic, measurement-driven methodology, not guesswork. Define goals, establish baselines, profile, hypothesize, implement and iterate.
- This book focuses on CPU performance engineering with primary emphasis on x86-64, covering architecture, memory, concurrency, tools and optimization methodology.

Chapter 2: Inside the Modern CPU - A Mental Model

The Von Neumann Architecture and Its Limitations

To understand a modern CPU, you must understand the problem it is trying to solve. That problem originates from the Von Neumann architecture, the fundamental design that has governed computer architecture for nearly eighty years.

In the Von Neumann model, a computer has a processor that executes instructions stored in memory, operating on data also stored in memory. The processor fetches an instruction from memory, decodes it to understand what operation to perform, executes the operation on operands from memory or registers and writes the result back to memory or registers. Then it fetches the next instruction and repeats. This is the fetch-decode-execute cycle, the heartbeat of computation.

The Von Neumann architecture is elegant in its simplicity but has a fundamental bottleneck: the processor must communicate with memory for every instruction and every data access and memory is much slower than the processor. Modern CPUs execute billions of instructions per second, but main memory can only supply data at a fraction of that rate. This gap between processor speed and memory speed is known as the memory wall and it is the central problem that much of modern CPU design is trying to solve.

The Von Neumann bottleneck implies that a significant portion of processor cycles is spent waiting for data, rather than executing instructions. Various techniques have been developed to hide this latency: caches to keep frequently used data close to the processor, prefetching to fetch data before it is needed, pipelining to overlap instruction fetch with execution and speculative execution to execute instructions before knowing they are definitely needed. All of these techniques share the same goal: keep the processor busy by reducing the time it spends waiting.

The Fetch-Decode-Execute Cycle - And Why It Is Not That Simple

If you picture a CPU as a simple loop that fetches an instruction, decodes it, executes it and writes the result, you have the right intuition but the wrong level of detail. Modern processors transform that simple cycle into a massively parallel, speculative, out-of-order factory that is doing many things at once.

Let us trace what happens to a single instruction on a modern processor, keeping the description high-level for now. Later chapters will fill in the details.

Fetch: The instruction fetch unit retrieves instructions from memory, specifically from the instruction cache. Instructions are fetched in blocks called cache lines, typically 64 bytes on modern processors. A single cache line may contain several instructions, so fetching one instruction often brings neighboring instructions into the cache for free.

Decode: The fetched instructions must be translated into a format that the execution units can use. On x86 processors, this step is particularly important because the ISA has variable-length instructions (one to 15 bytes) that encode complex operations. Modern x86 processors decode instructions into fixed-size micro-operations, or uops, which represent simpler operations that map directly to execution units. A single x86 instruction may decode into multiple uops.

Rename: Before execution, the processor renames registers. Your source code and the assembly language use a small set of architectural registers (like `rax`, `rbx` on x86-64). The processor has many more internal physical registers and it maps architectural registers to physical registers dynamically. This technique, called register renaming, eliminates artificial dependencies between instructions that reuse the same register name but actually work on independent data.

Dispatch and issue: The uops are dispatched to a pool where they wait for their operands to be ready and for execution units to become available. The processor dynamically selects which uops to issue to execution units based on data availability and resource constraints. Instructions may execute out of their original program order.

Execute: The execution units perform the actual computation. Different types of execution units handle different operations: arithmetic-logic units

for integer operations, floating-point units for floating-point operations, load-store units for memory accesses and vector units for SIMD operations. A modern processor has multiple execution units of each type, allowing it to execute multiple operations simultaneously.

Retire: After execution, the results must be committed to the architectural state in the correct program order. Even though instructions execute out of order, they retire in order preserving the illusion of sequential execution. This step ensures that exceptions, interrupts and program semantics are correct.

Each of these steps happens in parallel for many instructions at once. While one instruction is being fetched, another is being decoded, another is being executed and another is retiring. This parallelism is the essence of modern CPU performance.

Cores, Threads and SMT - Understanding Compute Resources

Modern processors have multiple cores, each of which is essentially an independent execution engine with its own fetch, decode, execution units and private caches. Cores share some resources, such as the L3 cache and the connection to main memory, but they execute independently.

Within each core, there may be support for hardware multithreading, known as Simultaneous Multithreading (SMT) on AMD processors or Hyper-Threading on Intel processors. SMT allows a single core to execute instructions from two (or more, on some architectures) software threads simultaneously by sharing the execution units between threads.

SMT does not double the computational power of a core. The execution units, caches and other resources are shared. When both threads have work ready and the required resources are available, SMT can improve utilization by filling gaps that would otherwise be idle. For example if Thread A is waiting for a cache miss and Thread B has instructions ready to execute, the execution units can process Thread B's instructions while Thread A waits.

The performance benefit of SMT depends heavily on the workload. For mixed workloads where threads have complementary resource usage patterns, SMT can improve throughput by 15 to 30 percent. For workloads that fully utilize a core's resources, the second thread competes for resources and may

degrade the performance of the first thread. For cache-sensitive or latency-critical workloads, disabling SMT can sometimes improve performance by reducing resource contention and interference.

Understanding your processor's resources is fundamental to performance engineering. You need to know how many physical cores you have, whether SMT is enabled, how cores are organized across sockets and NUMA nodes and what resources are shared versus private. Tools like `lscpu` on Linux or `CPU-Z` on Windows can provide this information.

The Processor as a Factory - Pipeline, Buffer and Queue Analogies

An analogy that helps build intuition for modern CPU operation is to think of the processor as a factory assembly line.

The instruction stream is the input material. The fetch unit is the warehouse that retrieves raw materials. The decode unit breaks complex operations into simple steps. The execution units are the workers performing the actual assembly. The caches are shelves nearby that hold frequently used parts, avoiding trips to the distant warehouse. The reorder buffer is the quality control station that ensures everything comes out in the right order, regardless of the order in which workers completed their tasks.

Just like a factory, the goal is to keep workers busy and minimize idle time. If a worker is waiting for a part, the factory loses productivity. Modern CPUs use various techniques to avoid this: prefetching (anticipating which parts will be needed and fetching them in advance), pipelining (overlapping stages so multiple items are in progress at once) and out-of-order execution (letting workers do available work while waiting for dependencies to complete).

This analogy also helps explain why complexity exists. Every additional feature (more execution units, deeper pipelines, more aggressive branch prediction, larger caches) costs transistors and power. The design trade-offs in a CPU are all about whether the performance benefit justifies the cost. Understanding these trade-offs helps you understand why CPUs behave the way they do and what kinds of code run well on them.

Why Clock Speed Is Only Part of the Story

Clock speed, measured in gigahertz, is often used as a measure of processor speed in marketing materials. A 5 GHz processor sounds faster than a 3 GHz processor and it often is, but the picture is incomplete without understanding what actually happens within each clock cycle.

Two processors with the same clock speed can have very different performance because they may execute different amounts of work per cycle. This is measured as instructions per cycle, or IPC. A processor with an IPC of 2 executes twice as many instructions per cycle as a processor with an IPC of 1, even at the same clock speed. Modern high-performance processors typically achieve IPCs between 1 and 4, depending on the workload and microarchitecture.

Clock speed also has diminishing returns. A faster clock means each cycle is shorter, which limits the amount of work that can be done within that cycle. Deep pipelines, which are needed to achieve high clock speeds, have their own costs: longer recovery from branch mispredictions and more power consumption from switching transistors at higher frequencies.

Performance is the product of clock speed and IPC:

Performance $\approx$ Clock Speed $\times$ IPC

This formula simplifies reality but captures an important point: optimizing for performance means considering both. A slower clock with higher IPC can outperform a faster clock with lower IPC. And software behavior affects IPC dramatically. Code that aligns with the processor's execution model achieves high IPC. Code that causes cache misses, branch mispredictions, or resource conflicts achieves low IPC.

Instruction Set Architectures - x86-64 and ARM64 Compared

The instruction set architecture, or ISA, defines the programming model that software sees: the registers, instructions, addressing modes and memory model. Two dominant ISAs for modern servers and desktops are x86-64 (from Intel and AMD) and ARM64 (also known as AArch64).

x86-64 evolved from the original 16-bit x86 architecture developed by Intel in the 1970s. It is a complex instruction set architecture, or CISC, with variable-length instructions, many addressing modes and instructions that perform complex operations in a single step. This complexity has benefits: compact code and rich operations, but it also means the decode stage is more complicated. Modern x86 processors handle this complexity by decoding instructions into micro-operations internally, effectively creating a simpler RISC-like execution model underneath.

ARM64 is a reduced instruction set architecture, or RISC, with fixed-length instructions, simpler addressing modes and a cleaner design philosophy. Each instruction performs a simpler operation, which makes decode faster and more predictable. ARM64 has gained significant market share in servers, especially with companies like AWS offering ARM-based instances and with Apple's transition to ARM-based Mac processors.

The RISC vs. CISC debate is largely academic on modern hardware because both approaches converge in practice. x86 processors decode to micro-ops that look like RISC instructions. ARM processors have added complexity over generations, including out-of-order execution and aggressive speculative execution. The practical differences are mostly in details: instruction encoding, register file size, SIMD capabilities and specific microarchitectural implementations.

For performance engineering, understanding the ISA matters because it determines what instructions are available and what their costs are. x86-64 has more instructions, which can mean more opportunities for optimization but also more complexity. ARM64 has a simpler instruction set, which can make performance analysis more straightforward. Both have rich SIMD capabilities for vectorization.

The specific microarchitecture matters more than the ISA family. An Intel processor and an AMD processor both use x86-64 but have different cache sizes, execution units, branch predictors and performance characteristics. A performance engineer needs to understand the specific processor they are targeting, not just the ISA.

Chapter Summary

Key takeaways from Chapter 2:

- The Von Neumann architecture and the memory wall define the fundamental problem CPUs try to solve: keeping the processor busy despite slow memory.
- Modern processors transform the simple fetch-decode-execute cycle into a massively parallel, out-of-order, speculative execution pipeline.
- Cores provide parallel execution engines. SMT/Hyper-Threading allows a core to share resources between threads, improving utilization for some workloads but potentially degrading performance for others.
- Thinking of the CPU as a factory helps build intuition: the goal is to keep execution units busy and minimize idle time.
- Clock speed alone does not determine performance. IPC (instructions per cycle) is equally important and software behavior dramatically affects achievable IPC.
- x86-64 and ARM64 are the dominant ISAs, with different design philosophies that converge in practice. The specific microarchitecture matters more than the ISA family for performance.

Chapter 3: Instructions and the Instruction Set

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

What Is an Instruction - Anatomy and Encoding

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

RISC vs. CISC - History, Philosophy and Modern Reality

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Micro-ops - How x86 Instructions Become Executable Work

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Instruction Latency vs. Instruction Throughput - Two Different Numbers

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Addressing Modes and Their Performance Cost

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Reading Agner Fog and Processor Reference Manuals

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter 4: Pipelines and Superscalar Execution

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

The Pipeline - Stages, Hazards and Throughput

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Superscalar Design - Issuing Multiple Instructions Per Cycle

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Execution Ports and Resource Constraints

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Pipeline Depth - Benefits and the Cost of Mispredictions

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Frontend vs. Backend Bottlenecks

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Architectural Implications - How to Write Code for Superscalar CPUs

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter 5: Out-of-Order Execution and Instruction-Level Parallelism

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

The Problem - Dependencies and Stall Cycles

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Out-of-Order Execution - Scoreboarding and Tomasulo's Algorithm

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

The Reorder Buffer and Commit Stage

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

True Dependencies, Anti-Dependencies and Output Dependencies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Instruction-Level Parallelism - Measuring and Exploiting It

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

What You Can Control - Software Techniques to Increase ILP

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter 6: Branch Prediction and Speculative Execution

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

The Branch Problem - Why Control Flow Kills Throughput

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Branch Prediction Mechanisms - Static and Dynamic Predictors

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Speculative Execution - Running Before You Know

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Branch Misprediction Penalty - Quantifying the Cost

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Writing Predictable Code - Data-Dependent Branches and Conditional Moves

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Security Implications - Spectre, Meltdown and Their Microcode Fixes

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter 7: Instruction Scheduling and Code Layout

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Instruction Fetch - Width, Alignment and Cache Lines

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Code Alignment and Its Impact on Pipeline Performance

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Software Pipelining - Manual Instruction Scheduling

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Compiler Scheduling - What Compilers Do and What They Cannot See

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Loop Layout and Branch Placement

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Reading and Optimizing Assembly Output

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cperformanceengineering>.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cperformanceengineering>.

Chapter 8: Caches - The Memory Hierarchy

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

The Memory Wall - Why Main Memory Is Too Slow

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Cache Levels - L1, L2, L3 and Their Roles

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Cache Internals - Lines, Sets, Ways and Associativity

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Cache Access Types - Hits, Misses and Conflict Misses

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

False Sharing - A Subtle Performance Killer

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Cache-Friendly Programming - Locality, Stride and Data Structures

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter 9: Cache Coherency and Multi-Core Communication

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

The Cache Coherency Problem - Multiple Copies, One Truth

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

MESI and MOESI Protocols - How CPUs Keep Caches in Sync

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

The Snooping Bus and Directory-Based Coherency

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Coherency Traffic - Measuring Its Impact

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Writing Cache-Coherency-Aware Code

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

NUMA and Cache Coherency in Large Systems

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter 10: Virtual Memory and Translation Lookaside Buffers

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Virtual Memory - Isolation, Abstraction and Performance Cost

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Page Tables and Page Faults - Walking the Tree

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

TLBs - Purpose, Hierarchy and Miss Penalties

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

TLB Shootdowns - The Multi-Core Cost

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Huge Pages - Reducing Translation Overhead

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Memory Mapping Strategies for Performance

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter 11: Memory Access Patterns and Bandwidth

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Sequential vs. Random Access - The Performance Gap

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Spatial and Temporal Locality - Exploiting Hardware Prefetching

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Memory Bandwidth - Measuring and Maximizing It

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Prefetching - Hardware and Software Techniques

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Data Layout Transformations - AOS vs. SOA and Beyond

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Benchmarking Memory Performance - Real Numbers on Real Hardware

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter 12: SIMD and Vectorization

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

SIMD Fundamentals - Doing More Work Per Instruction

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

x86 SIMD Families - SSE, AVX, AVX2, AVX-512

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

ARM NEON and SVE - Vectorization on ARM

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Auto-Vectorization - When Compilers Succeed and Fail

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Manual Vectorization - Intrinsics and Assembly

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Common Pitfalls - Alignment, Reductions and Control Flow

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter 13: Multithreading and Hardware Concurrency

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Hardware Multithreading - Simultaneous Multithreading (SMT)

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Software Threads - Mapping to Hardware

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Scaling Performance - Amdahl's Law and Gustafson's Law

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Thread Affinity and NUMA Awareness

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Load Balancing - Dynamic and Static Scheduling

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

When Multithreading Hurts - Resource Contention and False Sharing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter 14: Synchronization and Atomic Operations

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Atomic Operations - The Hardware Foundation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Memory Ordering and Barriers - Sequential Consistency vs. Weaker Models

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Locks - Mutexes, Spinlocks and Their Overhead

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Lock-Free and Wait-Free Algorithms - Possibilities and Limits

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Read-Write Locks and Fine-Grained Locking

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Lock Coarsening, Elimination and False Sharing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter 15: Performance Counters and Hardware Monitoring

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

What Are Performance Counters - Hardware Events and Counters

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Intel PMU, AMD IBS and ARM PMU - Vendor-Specific Features

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Key Metrics - IPC, Cache Miss Rates, Branch Mispredictions

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Using perf - The Linux Performance Toolkit

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Statistical Sampling vs. Precise Events

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Designing Counter-Based Experiments

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter 16: Profiling - Understanding What Your Code Actually Does

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Sampling vs. Instrumentation - Trade-Offs

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Flame Graphs - Visualizing Execution Profiles

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

CPU Profiling - Finding Hot Paths

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Cache Profiling - Identifying Memory Bottlenecks

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Branch Profiling - Understanding Control Flow Performance

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Profiling in Production - Sampling, Tracing and Overhead

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cperformanceengineering>.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cperformanceengineering>.

Chapter 17: Benchmarking - Measuring What Matters

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Microbenchmarks vs. Macrobenchmarks - When to Use Each

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Experimental Design - Controls, Variables and Repetition

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Statistical Variation - Warmup, Cooling and Outliers

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Measuring Latency - Precision Timers and Statistical Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Measuring Throughput - Sustained Performance and Sustaining It

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Common Benchmarking Mistakes - And How to Avoid Them

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter 18: Compiler Optimization - Working With the Compiler

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Compiler Optimization Pipeline - What Happens Under the Hood

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Optimization Levels - What -O2 and -O3 Actually Do

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Compiler Flags - Tuning for Your Target

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Reading Optimization Reports - -Rpass and -Rpass-missed

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

LTO and PGO - Link-Time and Profile-Guided Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

When the Compiler Fails - Inline Assembly, Intrinsic and Hand-Tuning

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter 19: CPU Frequency, Power Management and Thermal Limits

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Turbo Boost and Dynamic Frequency Scaling - How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Power Limits - TDP, PL1, PL2 and Their Real-World Impact

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Thermal Throttling - When Heat Becomes the Bottleneck

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Measuring Real Frequency - Is Your CPU Running at Full Speed?

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Disabling Power Management for Benchmarks

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Performance per Watt - Optimizing for Efficiency, Not Just Speed

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter 20: Operating System Interactions and System Configuration

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Scheduler Effects - Context Switches and Priority Inversion

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Interrupt Handling - Softirqs, Hardirqs and Latency

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

CPU Isolation - Isolating Cores for Latency-Sensitive Workloads

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Kernel Configuration - Tuning for Performance

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

cgroups and Resource Controls - Containment and QoS

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Virtualization Overhead - When Your VM Hurts Performance

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter 21: Performance Analysis Methodologies - A Scientific Approach

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

The Performance Engineering Workflow - Measure, Hypothesize, Optimize, Validate

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Establishing a Baseline - Reproducibility and Regression Prevention

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Formulating Hypotheses - From Symptoms to Root Causes

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

The Bottleneck Taxonomy - CPU-Bound, Memory-Bound, I/O-Bound

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Prioritizing Optimizations - Impact vs. Effort

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Validating Results - Avoiding Premature Celebration

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter 22: Case Studies in Performance Engineering

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Case Study 1 - Optimizing a Hash Table: Cache Locality and False Sharing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Case Study 2 - Vectorizing Image Processing: From Naive to AVX-512

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Case Study 3 - Reducing Lock Contention in a High-Throughput Server

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Case Study 4 - NUMA-Aware Memory Allocation in a Database

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Lessons Learned - Common Patterns Across Cases

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cperformanceengineering>.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cperformanceengineering>.

Chapter 23: Critical Analysis of the CPU-Performance-Engineering Repository

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Repository Overview - Structure and Scope

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Strengths - What It Gets Right

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Weaknesses and Omissions - What Is Missing or Under-Explained

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Accuracy Assessment - Comparing to Primary Sources and Reference Manuals

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Outdated or Misleading Material - Specific Examples and Corrections

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

How to Use This Repository - Recommendations and Caveats

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Chapter Summary

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Conclusion: The Performance Engineering Mindset

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

The Key Principles - A Condensed Checklist

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

What Changes and What Stays the Same

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Future Trends - Chiplets, AI Accelerators and New Memory Technologies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Continuing Your Journey - Resources and Communities

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

Final Thoughts

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.

References

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/cpuperformanceengineering>.