High-Performance .NET

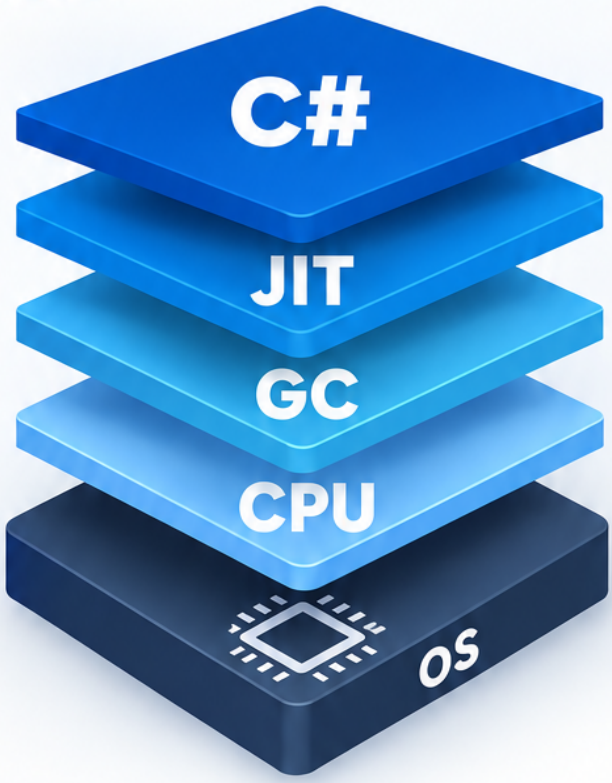
From JIT to Hardware

A Systems-Level Guide to Writing Fast .NET Applications

Understand how .NET executes your code

Measure and diagnose real bottlenecks

Use professional tooling to improve performance



JOON-HO KANG

High-Performance .NET: From JIT to Hardware

A Systems-Level Guide to Writing Fast .NET Applications

Steve Publications

This book is available at
https://leanpub.com/high-performancenetfromjittohardware

This version was published on 2026-09-16

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

A Systems-Level Guide to Writing Fast .NET Applications

This book teaches experienced .NET developers how to understand, measure, diagnose, and systematically improve the performance of modern .NET applications from source code down to the CPU and operating system. You will learn how the .NET runtime executes your code, how the JIT compiler translates C# into machine instructions, how the garbage collector manages memory, how modern CPUs execute those instructions, and how to use professional tooling to find and fix real bottlenecks. The book is built around .NET 8 LTS and C# 12, with version-specific behavior clearly identified where it matters.

Introduction

A production service you maintain receives a sudden spike in traffic. Response times double. The on-call engineer notices a steady rise in memory usage and frequent pauses that feel like garbage collection thrashing. Another incident: a CPU-intensive batch job runs noticeably slower on the new ARM64 virtual machines than on the old x64 instances, even though the ARM chips have twice the core count. Yet another: your team added caching to a hot path and saw throughput improve, but only until a week later when latency started climbing again and you discovered the cache was promoting temporary objects into generation two, forcing expensive full GC cycles every few minutes.

These are the kinds of problems that performance engineering exists to solve. But performance engineering is not about memorizing a list of tricks or applying random micro-optimizations. It is a disciplined practice that requires understanding what your code actually does when it runs, measuring real bottlenecks rather than guessing, and making changes based on evidence rather than intuition.

This book teaches that practice for modern .NET applications.

When you write C# code, you are not writing instructions that the CPU understands directly. Your code goes through a complex pipeline before it ever reaches the processor: compilation to IL, loading by the runtime, Just-In-Time compilation into machine code, execution by the garbage-collected runtime, scheduling by the operating system, and finally execution on a multi-core processor with caches, pipelines, and memory hierarchies. Every layer in that stack can introduce overhead, every layer can create bottlenecks, and every layer can be optimized.

The problem is that most .NET developers stop thinking at the C# level. They know what a loop looks like in C#, but not what machine instructions the JIT generates for that loop. They know that objects live on the heap, but not exactly how the allocator places them in memory, how the GC scans them, or why allocating even a small object millions of times can cause serious performance problems. They know that async/await helps with scalability, but not that each awaited task that actually yields can allocate hundreds of

bytes and consume several microseconds of overhead. They know that locking coordinates threads, but not that two counters on the same cache line can serialize their updates through false sharing even though they look completely independent in C#.

This book fills those gaps. It takes you through the complete performance stack: how the runtime works, how the JIT optimizes your code, how the garbage collector manages memory, how your data is laid out, how concurrency works under the hood, how modern CPUs execute instructions, and how to measure everything correctly. You will see the actual disassembly the JIT produces, you will understand the allocation patterns of your code, and you will know how to use profiling and tracing tools to find real problems in production systems.

By the end of this book you will be able to look at a slow .NET application and form a hypothesis about why it is slow based on evidence, not guesswork. You will know how to design code that the JIT can optimize aggressively. You will understand when memory allocations actually matter and when they do not. You will be able to tune the garbage collector for your workload. You will understand the performance implications of async code, collections, and concurrent data structures. You will know when SIMD vectorization is worthwhile and how to apply it. You will understand why data layout matters and how CPU caches affect your code. And you will have a systematic methodology for performance work that separates real improvements from measurement noise.

This book assumes you already know C# and general .NET development. It does not teach basic programming. Instead, it focuses on the deep questions: what happens when this code runs, why does it run at the speed it does, and how can you make it faster in a way that is measurable and repeatable. The code examples are complete, compilable, and runnable. They show inefficient implementations first, then progressively optimized versions, with clear explanations of what each change does at the machine level and what trade-offs it introduces.

Performance work is one of the most valuable skills a software engineer can develop. Fast applications handle more users with the same infrastructure, respond more quickly to requests, cost less to run, and provide better experiences. The techniques you learn in this book will pay for themselves many times over in operational savings and user satisfaction. Let us begin.

Chapter 1: Why Performance Matters and How to Think About It

The Hidden Costs of Slow Software

Performance problems are expensive. They are also subtle. A single inefficient loop running inside a hot request path may add only twenty microseconds of delay per request. That number is so small it feels irrelevant. But at one thousand requests per second that delay costs two seconds of CPU time every second. At ten thousand requests per second it costs twenty seconds per second. At that point you are not optimizing microseconds anymore, you are buying additional server capacity.

Consider a realistic scenario. A high-throughput API endpoint processes JSON payloads and stores them in a database. The original implementation parses the JSON with a general-purpose library that creates many intermediate objects: string allocations for each field, temporary dictionaries for parsing, boxed value types when generic interfaces are used. The code works correctly and the developer considers it clean because it uses well-known patterns and avoids low-level complexity.

During load testing the endpoint handles 500 requests per second before latency exceeds the SLA. Each request allocates approximately 15 kilobytes of short-lived objects. The garbage collector runs generation zero collections about every 50 milliseconds, keeping pauses under one millisecond most of the time. The application looks healthy in monitoring dashboards. CPU usage is moderate. Memory is stable.

But when production traffic grows to 3,000 requests per second, problems emerge. The allocation rate reaches 45 megabytes per second. The GC is constantly collecting generation zero, but some objects survive long enough to be promoted to generation one and then to generation two. Generation two collections start occurring every minute, each taking 50 to 100 milliseconds and blocking all requests during the pause. The developer who wrote the original code sees the GC metrics and adds more memory, which postpones

the generation two collections but does not solve the underlying problem. The application is now consuming expensive compute resources to run code that could be significantly more efficient.

This scenario is common enough that it has a pattern: applications that are "good enough" at small scale become painful at large scale because hidden costs accumulate. Memory allocations compound into GC pressure. GC pressure compounds into latency. Latency compounds into queueing and thread pool exhaustion. Thread pool exhaustion compounds into timeouts and cascading failures. None of the individual problems look catastrophic on their own. The cumulative effect is.

Performance matters not just for systems that handle massive scale. Even moderate traffic can expose performance problems. A service that takes 200 milliseconds to respond instead of 50 milliseconds is consuming four times as many server resources to do the same work. That directly affects operational costs in cloud environments where you pay for compute time. A batch job that runs in ten minutes instead of two minutes blocks other work, delays data availability, and may miss scheduled windows.

There is also a human cost. Slow applications frustrate users. Users who wait more than a second for a response notice. Users who wait more than three seconds often leave. In internal tools, slow performance compounds over thousands of employees working every day, reducing productivity in ways that are real but hard to measure.

The good news is that performance is not magic. Every performance problem has a cause that you can identify, measure, and fix. The question is whether you have the right mental model and the right tools to find it.

Throughput Versus Latency: What Are You Optimizing?

Before optimizing anything, you must know what you are optimizing for. Performance is not a single metric. Two of the most important metrics are throughput and latency, and they often pull in different directions.

Throughput is the amount of work completed per unit of time. A web service that handles 5,000 requests per second has higher throughput than one that handles 2,000 requests per second. A batch job that processes one million records in five minutes has higher throughput than one that processes the same records in twenty minutes.

Latency is the time it takes for a single unit of work to complete. A web service that responds in 10 milliseconds has lower latency than one that responds in 100 milliseconds. A database query that returns in 2 milliseconds has lower latency than one that returns in 20 milliseconds.

These two metrics are related but not identical. You can sometimes increase throughput while increasing latency. If you add more threads to process requests, you can complete more requests per second even if each individual request takes slightly longer due to context switching and cache pressure. You can also sometimes decrease latency while decreasing throughput. If you optimize the critical path for individual requests, each request finishes faster, but you may be doing less batched work overall and completing fewer total requests.

Understanding which metric matters for your workload is essential. A real-time trading system cares primarily about latency. Every microsecond of delay means worse prices and lost trades. A data pipeline cares primarily about throughput. Moving more data per hour matters more than any individual record completing quickly. A web API typically cares about both: you want to handle many requests (throughput) while keeping response times low (latency). But even here you need to decide which takes priority when they conflict.

There is also tail latency, which is often more important than average latency. If your API has an average response time of 20 milliseconds but the 99th percentile (the slowest one percent of requests) takes 500 milliseconds, users will experience that 500-millisecond latency often enough to notice. Optimization work that reduces average latency by 10 percent while increasing tail latency by 50 percent may make your dashboard look better while making users unhappy. Always measure percentiles, not just averages.

The Performance Stack: From C# to Silicon

When a .NET application runs, your C# code does not execute directly. It passes through multiple layers before reaching the CPU. Understanding this stack is the foundation for performance engineering.

At the top is your C# source code. You write it, the C# compiler (Roslyn) translates it into Intermediate Language, and that IL is stored in an assembly. The IL is a platform-independent bytecode format. It is not machine code. It must be translated into machine code before it can run.

The translation happens in the .NET runtime. The CoreCLR execution engine loads your assemblies, manages types and metadata, and invokes the Just-In-Time compiler to translate IL into native machine code as methods are called. The JIT compiler, RyuJIT, is a sophisticated optimizing compiler. It does not just translate IL one-to-one into instructions. It performs many optimizations: method inlining, dead code elimination, loop unrolling, value numbering, register allocation, and many others. The quality of the code the JIT produces has a massive effect on your application's performance.

The generated machine code executes in managed memory. The garbage collector allocates objects on the managed heap, tracks which objects are reachable, and reclaims memory from objects that are no longer in use. The GC algorithm affects performance in several ways: allocation speed, collection pause times, CPU overhead for tracking, and memory footprint. Different GC modes (workstation versus server, background versus foreground) make different trade-offs.

The operating system schedules your process and its threads on CPU cores. The OS manages virtual memory, handles system calls for I/O, and mediates access to hardware resources. The way your application interacts with the OS affects performance: thread pool sizing, I/O patterns, memory pressure, and CPU affinity all matter.

Finally, the machine code executes on physical CPU cores. Modern CPUs are complex machines with multiple execution pipelines, branch prediction units, cache hierarchies, and memory controllers. The actual speed of your code depends on how well the instructions flow through these pipelines, how often the CPU predicts branches correctly, how often it finds data in cache versus fetching from main memory, and how many instructions it can execute in parallel.

Every layer in this stack can be a bottleneck. A slow C# algorithm is obvious. A suboptimal JIT compilation that fails to inline a hot method is subtle. A GC pause caused by an accidental promotion pattern is easy to miss. A cache miss pattern that serializes multi-threaded updates through false sharing is invisible in C#. All of these problems can dominate your application's performance.

The key insight is that you cannot optimize effectively if you only think at the C# level. You need to understand what each layer does and how your high-level code maps to low-level behavior. That is what this book teaches.

Evidence-Based Optimization: The Scientific Method Applied

Good performance engineering is a scientific process. It relies on observation, measurement, hypothesis, experimentation, and validation. It does not rely on rules of thumb, folklore, or premature assumptions.

Here is the methodology you should follow:

First, establish a baseline. Measure the current performance using appropriate tools. Know what "slow" actually means in numbers. If you are optimizing latency, measure percentiles at specific load levels. If you are optimizing throughput, measure requests per second at acceptable latency. If you are optimizing memory, measure allocation rate and GC pressure. The baseline must be measured under realistic conditions, not artificial microbenchmarks that do not match production behavior.

Second, identify the bottleneck. Use profiling, tracing, and analysis tools to find where time is actually being spent. The bottleneck is not necessarily where you think it is. It is not necessarily the most complex-looking code. It is the part of the system that limits overall performance. Profiling shows you the real hot paths. If 80 percent of execution time is in 5 percent of the code, optimizing that 5 percent will have far more impact than optimizing everything else.

Third, form a hypothesis. Based on what profiling tells you, make an educated guess about why that code is slow. Is it doing unnecessary work? Is it allocating too much memory? Is it contending on locks? Is it missing from cache? Is it not vectorized? Your hypothesis should be specific enough to test.

Fourth, make one change. Implement a single modification that addresses your hypothesis. Do not make multiple changes at once. If you change three things and the performance improves, you do not know which change caused the improvement. You might be carrying around a change that hurts readability or correctness for no reason.

Fifth, measure again. Run the same benchmark or profiling session as the baseline. Compare the numbers. Did the performance actually improve? By how much? Is the improvement statistically significant, or could it be measurement noise? Are there any regressions elsewhere?

Sixth, validate correctness. Faster code is worthless if it is wrong. Ensure

the change does not introduce bugs. Run your test suite. In performance work, the temptation to skip tests because "it just works" is strong, but correctness is non-negotiable.

Seventh, decide whether to keep the change. If the change improves performance significantly, does not hurt correctness, and does not introduce unacceptable trade-offs in readability or maintainability, keep it. If the improvement is marginal or the trade-offs are too high, revert it. Not every optimization is worth keeping.

This process sounds obvious, but many teams skip steps. They optimize based on guesses instead of measurements. They make many changes at once and cannot attribute improvements. They measure only averages instead of distributions. They skip correctness validation because they are in a hurry. They keep micro-optimizations that save microseconds but make code unreadable for no reason.

The methodology above prevents these mistakes. It keeps your optimization work focused, evidence-based, and repeatable.

Common Mistakes and False Intuitions

Performance work attracts bad advice. Here are some common mistakes to avoid.

The most famous mistake is premature optimization. The phrase "premature optimization is the root of all evil" is often misused to justify ignoring performance entirely. The original intent was different: do not optimize before you know where the bottleneck is. Write clear, correct code first. Then measure. Then optimize what matters. Ignoring performance until it is too late is equally wrong, but the solution is not to micro-optimize everything from day one.

Another common mistake is optimizing for microbenchmarks instead of real workloads. A code change might improve a microbenchmark by 30 percent while making no difference or even hurting performance in production. Microbenchmarks isolate individual operations and run them in a vacuum. Production workloads involve I/O, concurrency, cache effects, and many interacting components. Always validate that microbenchmark improvements translate to real-world improvements.

A third mistake is assuming that faster hardware solves everything. Upgrading from four cores to sixteen cores does not help if your code is single-threaded. Adding more memory does not help if your bottleneck is CPU-bound computation. Increasing network bandwidth does not help if your application is allocating too many objects and spending its time in GC. Hardware helps, but it does not fix bad algorithms or poor design.

A fourth mistake is ignoring trade-offs. Every optimization has trade-offs. Inlining a method makes calls faster but increases code size, which can hurt instruction cache. Using structs instead of classes avoids heap allocation but can increase copying. Lock-free algorithms avoid lock contention but are harder to get correct and may not be faster in practice. Always understand the trade-off before applying an optimization.

A fifth mistake is over-optimizing rare paths. If a code path runs once per hour, it does not matter if it takes one millisecond or one microsecond. Focus your optimization effort on hot paths that run frequently and account for a significant portion of execution time.

A sixth mistake is measuring with low-precision timers. Using DateTime.Now or other wall-clock timers for performance measurement introduces too much noise. Use Stopwatch for high-resolution timing in benchmarks. Use proper benchmarking frameworks like BenchmarkDotNet that handle warmup, iterations, outlier removal, and statistical analysis automatically.

A seventh mistake is assuming that what you know about one version of .NET applies to another. The runtime, JIT, and libraries change between versions. An optimization that was valuable in .NET Framework 4.8 might be irrelevant in .NET 8 because the JIT now does that optimization automatically. Always verify assumptions against the version you are targeting.

Avoiding these mistakes requires discipline and knowledge. The rest of this book provides both.

Chapter 2: The .NET Runtime Architecture

Inside the Execution Engine: CorRuntime and CoreCLR

When you run a .NET application with the dotnet command, you are starting a process that loads the .NET runtime and executes your managed code within it. The runtime is a complex system responsible for loading assemblies, managing memory, executing code, handling exceptions, enforcing security, and providing core library functionality. Understanding how the runtime works is essential for performance engineering because the runtime's behavior directly affects your application's speed, memory usage, and responsiveness.

The modern .NET runtime, known as CoreCLR, is built on a modular architecture. At its core is the execution engine, often referred to internally as corruntime. This component is responsible for the fundamental mechanics of running managed code: loading assemblies and their types, invoking the JIT compiler to generate native code, executing that code, managing the garbage-collected heap, and coordinating with the operating system for system calls and resources.

When you start a .NET application, the runtime performs several initialization steps. It loads core libraries that provide fundamental types and services. It sets up the managed heap where objects will be allocated. It initializes the garbage collector. It prepares the JIT compiler for code generation. It creates the thread pool for background work. All of this happens before your entry point method runs, and it contributes to startup time. Understanding what happens during this initialization helps you understand startup performance, which is especially important for serverless functions, containerized applications, and command-line tools.

The execution engine manages a hierarchy of components. At the highest level is the AppDomain subsystem, which in modern .NET has been largely replaced by AssemblyLoadContext for assembly loading isolation. The engine coordinates between assembly loaders, type loaders, the JIT compiler, and the

garbage collector. When your code references a type for the first time, the type loader resolves the type from metadata, ensures its containing assembly is loaded, and invokes the type initializer if necessary. When your code calls a method for the first time, the engine checks whether native code exists for that method and invokes the JIT compiler if not.

This coordination introduces overhead. Type resolution, assembly loading, and JIT compilation all consume time. In performance-sensitive applications, especially those with strict latency requirements, these overheads can be significant. The runtime provides several mechanisms to reduce them: precompilation with ReadyToRun, tiered compilation to balance startup and steady-state performance, and Native AOT to eliminate JIT compilation entirely. Each mechanism works by shifting work from runtime to compile time, and understanding how they work requires understanding the baseline behavior they are optimizing.

The Type System: Types, Metadata, and Runtime Representation

The .NET type system is at the heart of the runtime. Every type you define in C# corresponds to metadata that the runtime uses to understand the type's structure, methods, fields, and relationships to other types. This metadata is stored in PE (Portable Executable) files alongside the IL code. When the runtime loads an assembly, it reads this metadata to build internal data structures that describe the types.

Each type in the runtime has a method table, which is a critical performance structure. The method table contains pointers to the method implementations, information about the type's size and layout, synchronization information, and the type's base type and interfaces. When you allocate an instance of a class, the object header contains a pointer to the method table. This pointer allows the runtime to determine the object's actual type at runtime, which is essential for virtual method dispatch, casting, and reflection.

The method table is performance-relevant for several reasons. First, virtual method calls use the method table to find the actual implementation to invoke. When you call a virtual method on an object, the CPU dereferences the method table pointer in the object header, looks up the method's entry in the table, and jumps to the native code at that address. This indirection has a cost, especially

if the CPU cannot predict which implementation will be called. Second, the JIT compiler uses information in the method table to determine whether it can devirtualize a call, replacing the indirect call with a direct call that the CPU can predict. Third, the method table structure affects how types are laid out in memory and how casting operations work.

Value types and reference types are represented differently at runtime. Reference types (classes, arrays, delegates, strings) always live on the managed heap. Variables that hold reference types are pointers to objects on the heap. When you assign a reference variable to another variable, you are copying the pointer, not the object. Both variables refer to the same object on the heap.

Value types (structs, enums, primitive numeric types) are represented differently. A value type variable contains the actual data, not a pointer. When you assign a value type to another variable, the data is copied. When a value type is a field of a reference type, the value type's data is stored inline within the containing object. This difference in representation has significant performance implications. Value types avoid heap allocation when used as local variables or inline fields, but copying them can be expensive if they are large. Reference types avoid copying the data when passed around, but require heap allocation and GC management.

Understanding these representation details matters because they directly affect allocation patterns, memory usage, and CPU cache behavior. A large struct passed by value is copied on every method call. A small class allocated in a tight loop creates thousands of heap objects that the GC must track. The right choice depends on the specific scenario, and knowing the underlying mechanics helps you choose correctly.

Assembly Loading and Isolation: AssemblyLoadContext and Performance

Assemblies are the units of deployment in .NET. When your application references another assembly, that assembly must be loaded into the runtime before your code can use the types it defines. Assembly loading involves several steps: locating the assembly file (or bytes), verifying its format, reading its metadata, and preparing the types for use. This process can be slow, especially for large assemblies or assemblies with complex dependency graphs.

In modern .NET, assembly loading is managed by AssemblyLoadContext instances. An AssemblyLoadContext defines an isolation boundary for assemblies. The default context loads all assemblies that your application depends on. You can create custom contexts to load assemblies in isolation, which is useful for plugin systems or scenarios where you need to load multiple versions of the same assembly.

Assembly loading performance matters for startup time and for dynamic loading scenarios. When an assembly is first loaded, the runtime must parse its metadata and prepare types. If the assembly contains types with static constructors, those constructors run during loading. If the assembly depends on other assemblies, those dependencies must be loaded first. In applications that load many assemblies, the cumulative loading time can be noticeable.

The runtime caches loaded assemblies within their AssemblyLoadContext. Once an assembly is loaded, subsequent references to the same assembly use the cached version. This caching is fast and efficient. However, if you create multiple AssemblyLoadContext instances and load the same assembly into each, the assembly is loaded multiple times, consuming additional memory and time.

For performance-sensitive applications, reducing the number of loaded assemblies and their total size is beneficial. Native AOT compilation, trimming, and single-file publishing all reduce assembly loading overhead by consolidating code into fewer files or eliminating metadata that is not needed at runtime.

The Execution Pipeline: From Source to Machine Code

When you write C# code and compile it, the C# compiler (Roslyn) translates your source code into IL (Intermediate Language) and stores it in an assembly. The IL is a bytecode format that describes what your program does without specifying how it executes on any particular hardware. The IL includes type definitions, method bodies, field declarations, and metadata about types and their relationships.

When you run the application, the .NET runtime loads the assembly and the execution engine coordinates the process of turning IL into running code. The key component in this process is the Just-In-Time compiler, which translates IL into native machine code that runs directly on the CPU.

The JIT compilation process does not happen all at once when the application starts. By default, the runtime uses on-demand compilation: methods are compiled when they are first called. When your code calls a method for the first time, the runtime invokes the JIT compiler to translate that method's IL into native code. The resulting native code is cached in memory and reused for subsequent calls. This approach reduces startup time because only the methods that are actually used get compiled, and only when they are needed.

On-demand compilation has a cost: the first call to any method includes the overhead of JIT compilation. This overhead can be significant for methods that are complex or rarely called. For frequently called methods, the compilation cost is amortized over many invocations and becomes negligible. But for methods that run once at startup, the compilation cost can dominate the execution time.

The runtime provides several mechanisms to manage this trade-off. ReadyToRun compilation pre-generates native code during publishing, reducing JIT overhead at runtime. Tiered compilation uses a fast initial compilation for startup and then recompiles hot methods with more aggressive optimizations. Native AOT compiles all code ahead of time, eliminating JIT compilation entirely. Each approach shifts the compilation work to a different phase and makes different trade-offs between startup time, steady-state performance, deployment size, and flexibility.

Understanding Runtime Overhead: What the CLR Actually Does

The .NET runtime is not free. Every managed operation has some overhead compared to running equivalent code in a language like C. Understanding this overhead helps you understand when it matters and when you can ignore it.

Managed method calls have overhead from the calling convention and runtime checks. The runtime must handle things like stack overflow detection, exception dispatching, and security checks. Virtual method calls add indirection through the method table. The JIT compiler optimizes many of these costs away through inlining and devirtualization when it can prove they are safe, but some overhead remains.

Memory allocation in the managed heap is extremely fast compared to native malloc, but it is not free. The GC must track every allocation, and every

reference from one object to another creates a dependency that the GC must trace during collection. High allocation rates increase GC pressure, which increases CPU usage and can cause pauses.

Garbage collection is the most visible runtime overhead. When the GC runs, it may pause application threads (stop-the-world pauses), consuming time that could be spent executing your code. The GC also uses CPU cycles for tracing objects, managing card tables for write barriers, and compacting the heap. Well-designed applications that allocate short-lived objects and reuse long-lived objects minimize this overhead. Poorly designed applications that allocate excessively or create long-lived references to temporary objects force the GC to work harder and cause more pauses.

Type checking and casting have overhead. When you cast an object to a specific type, the runtime must verify that the object is actually of that type or a derived type. If the cast fails, the runtime throws an InvalidCastException. The JIT can sometimes eliminate these checks when it can prove the type is known, but many casts require runtime verification.

Reflection is expensive. When you use reflection to access types, methods, fields, or properties at runtime, the runtime must look up metadata, create wrapper objects, and perform security checks. Repeated reflection calls in a hot path can dominate execution time. Source generators and compiled expression trees can eliminate much of this overhead by generating equivalent code at compile time.

Understanding runtime overhead does not mean you should avoid managed features. It means you should understand when the overhead matters. For most applications, the convenience and safety of the managed runtime far outweigh the small overhead it introduces. But in performance-critical code paths, understanding where that overhead comes from allows you to make informed decisions about when to use more efficient alternatives.

Chapter 3: JIT Compilation and Code Generation

The RyuJIT Compiler: Architecture and Optimization Pipeline

When you write C# code, you write high-level abstractions: objects, methods, properties, generics, exceptions, async flows. None of these concepts exist in CPU instructions. The CPU understands only machine code: load this value from memory, add two registers, branch if zero, store to an address. The bridge between your C# code and the CPU is the Just-In-Time compiler.

In modern .NET, that compiler is RyuJIT. RyuJIT is a sophisticated optimizing compiler that translates IL into native machine code for x64, x86, and ARM64 processors. It is not a simple bytecode interpreter or a naive translator. It performs many of the same optimizations that traditional offline compilers like GCC or MSVC perform: dead code elimination, constant propagation, loop optimization, register allocation, instruction selection, and more. The difference is that RyuJIT runs at runtime, which gives it access to information that offline compilers do not have, such as the actual CPU features of the machine it is running on and runtime profiling data about which methods are called most often.

RyuJIT's architecture is built around a well-defined pipeline. When the runtime asks RyuJIT to compile a method, RyuJIT creates a Compiler object that represents the compilation of that method. The compilation proceeds through a series of phases, each transforming the method's representation.

The input to RyuJIT is IL bytecode. IL is a stack-based instruction set. Operations push and pop values from a virtual stack. This is convenient for a platform-independent intermediate format but inefficient for actual execution because real CPUs use registers, not stacks, for computation. RyuJIT's first job is to convert from this stack-based representation to a register-based representation optimized for the target CPU.

The compilation pipeline starts with Importation. In this phase, RyuJIT reads the IL bytecode and converts it into an internal representation based on a tree structure called GenTree. Each node in the GenTree represents an operation, a value, or a variable. The Importation phase also performs initial analysis to identify basic blocks (straight-line sequences of code with no branches in the middle) and sets up the control flow graph for the method.

During Importation, RyuJIT also performs an initial round of inlining. Inlining is one of the most important optimizations: it replaces a method call with the body of the called method, eliminating call overhead and enabling further optimizations across what were previously separate method boundaries. RyuJIT evaluates inlining candidates using a set of heuristics based on method size, call frequency, and whether inlining would enable additional optimizations. We will cover inlining in detail in Chapter 6.

After Importation comes Morphing. In this phase, RyuJIT normalizes the IR, performs local optimizations, and prepares the code for the main optimization phases. Morphing includes operations like struct promotion (determining whether a value type can be allocated in registers instead of on the stack), local variable optimization, and instruction normalization.

The Loop Analysis phase detects loops in the control flow graph, identifies loop headers, and classifies loops. This information is essential for subsequent optimizations like loop unrolling, loop-invariant code motion, and induction variable optimization.

The main optimization phases build on this foundation. RyuJIT puts the code into SSA (Static Single Assignment) form, which is a representation where each variable is assigned exactly once. SSA form makes data flow analysis much easier and enables powerful optimizations. Value numbering identifies equivalent expressions and eliminates redundant computations. Loop-invariant code motion (LICM) moves computations that produce the same result on every loop iteration outside the loop. Common subexpression elimination (CSE) removes duplicate computations. Copy propagation replaces uses of a variable with the value it was copied from when that value is known. Range analysis determines the possible values of variables, which enables elimination of unnecessary bounds checks.

After the optimization phases, RyuJIT performs Rationalization, which converts the high-level tree-based IR into a lower-level linear IR. This phase removes high-level constructs and prepares the code for machine-specific

code generation.

Lowering transforms the IR further to expose register requirements and control flow in a form that the code generator can use. This phase is architecture-specific and adapts the code for the target CPU's instruction set.

Register Allocation assigns variables to CPU registers using a linear scan algorithm (LSRA). CPUs have a limited number of registers, so not all variables can be kept in registers at once. The register allocator decides which variables to keep in registers and which to spill to the stack. Good register allocation is critical for performance because accessing registers is much faster than accessing memory.

Finally, Code Generation emits the actual machine code bytes using the emitter component. This phase generates the prolog and epilog that set up and tear down the stack frame, emits the instructions for the method body, generates GC information so the garbage collector knows where to find managed references, and produces unwind information for exception handling.

The output is a contiguous block of native machine code that the CPU can execute directly. This code is cached in memory and reused for every call to the method. The first call to a method pays the cost of compilation, but all subsequent calls run at native speed.

On-Demand Compilation: When Methods Get Compiled

By default, the .NET runtime compiles methods on demand, also known as lazy compilation. A method is not compiled until it is called for the first time. This approach reduces startup time because the application can begin executing before all methods are compiled. It also reduces memory usage because methods that are never called never consume memory for their compiled code.

The mechanism works like this: when the runtime first encounters a call to a method that has not been compiled, it pauses the calling thread and invokes the JIT compiler. The JIT compiler reads the method's IL from the assembly, performs its optimization passes, and produces native machine code. The runtime stores this code in a code heap (a region of memory reserved for JIT-compiled code) and updates an internal stub so that subsequent calls jump directly to the native code. The calling thread then resumes and executes the method at native speed.

This process is transparent to your code. From the perspective of your application, methods just execute. The JIT compilation overhead is hidden inside the first call. But this overhead is real and measurable. For small methods, compilation might take a few microseconds. For large or complex methods, it can take milliseconds. If many methods are compiled during startup, the cumulative overhead can be significant.

Consider a web API that handles many different routes. Each route handler method, each middleware component, and many library methods must be compiled before they can run. The first request to each route includes the compilation overhead for all methods that are called for the first time. This can result in slow initial response times until the "warmup" period is complete.

On-demand compilation also has implications for profiling and benchmarking. If you measure performance before all hot methods have been compiled, your measurements include JIT overhead that will not be present in steady-state operation. Proper benchmarking requires warmup to ensure all relevant methods are compiled before measuring.

The runtime provides several mechanisms to manage JIT compilation overhead. ReadyToRun precompiles methods during publishing, so the runtime can use pre-compiled code instead of invoking the JIT. Tiered compilation uses a fast initial compilation for startup and then recompiles hot methods with more aggressive optimizations. Native AOT compiles everything ahead of time. Each mechanism addresses different aspects of the JIT overhead problem.

Optimization Passes: From IL to Optimized Machine Code

The quality of the code that RyuJIT generates has a massive impact on your application's performance. RyuJIT performs many optimizations, and understanding these optimizations helps you write code that the JIT can optimize effectively.

Dead code elimination removes code that can never execute or whose results are never used. If a method contains a branch that is always taken in one direction, the code in the other branch is dead and can be removed. If a variable is computed but never read, the computation is dead and can be removed. This optimization is powerful because it removes unnecessary work automatically.

Constant propagation tracks constants through the code. If a variable is assigned a constant value and never modified, the JIT replaces uses of that variable with the constant itself. This enables further optimizations: if a branch condition is a constant, the unreachable branch is eliminated. If an array index is a constant, bounds checking may be simplified or eliminated.

Method inlining is perhaps the most impactful optimization. When a method call is inlined, the body of the called method is inserted at the call site. This eliminates the overhead of the call instruction, parameter passing, stack frame setup, and return. More importantly, inlining enables other optimizations by making the callee's code visible to the caller's context. A loop that spans two methods cannot be optimized across method boundaries unless one method is inlined into the other. RyuJIT uses heuristics to decide which methods to inline, based on factors like method size, inlining depth, and whether inlining would enable devirtualization.

Loop optimizations are critical for performance because loops are where most computation happens. RyuJIT performs several loop optimizations. Loop unrolling duplicates the loop body multiple times to reduce the overhead of loop control (branching and counter updates) and expose more instruction-level parallelism. Loop-invariant code motion moves computations that produce the same result on every iteration outside the loop. Induction variable optimization simplifies loop counters and index calculations. Range analysis eliminates redundant bounds checks when the JIT can prove the index is always in range.

For example, consider a simple loop that sums elements of an array:

```
int Sum(int[] array)
{
    int sum = 0;
    for (int i = 0; i < array.Length; i++)
    {
        sum += array[i];
    }
    return sum;
}
```

A naive implementation would check the array bounds on every iteration. But RyuJIT's range analysis can prove that if the loop starts at 0 and increments by 1 until it reaches array.Length, every access is within bounds. The bounds checks can be eliminated, leaving just the load and add operations. On modern

CPUs, this loop can execute at several elements per clock cycle when combined with vectorization and out-of-order execution.

Value numbering identifies equivalent expressions and eliminates redundant computations. If the same expression is computed multiple times and its operands have not changed, value numbering replaces later computations with the result of the first computation. This is a generalization of common subexpression elimination that works across basic blocks.

Copy propagation replaces uses of a variable with the value it was copied from when the original value is known and has not changed. This optimization simplifies the code and enables further optimizations by exposing the actual values being used.

Register allocation is one of the most complex phases. RyuJIT uses linear scan register allocation, which processes live intervals of variables and assigns registers based on availability. Good register allocation minimizes memory accesses by keeping frequently used values in registers. Poor register allocation forces values to spill to the stack, which adds load and store instructions that slow execution.

Instruction selection chooses the best CPU instructions for each operation. Modern CPUs have many instruction variants with different trade-offs. RyuJIT selects instructions based on the target CPU's capabilities, choosing faster instructions when available. For example, on CPUs that support BMI2 (Bit Manipulation Instruction Set 2), RyuJIT can use specialized bit manipulation instructions instead of sequences of generic instructions.

Reading Disassembly: Understanding What the JIT Produced

To understand performance at a deep level, you need to be able to read the machine code that the JIT produces. You do not need to be an assembly language expert, but you should be able to look at disassembly and identify key patterns: where allocations happen, where branches occur, where methods are inlined or called, where loops are vectorized, and where unnecessary work is being done.

The .NET runtime provides built-in support for dumping JIT disassembly. You can use environment variables to enable disassembly output for specific

methods or for all methods. The simplest approach is to use DOTNET_JitDisasm:

```
set DOTNET_JitDisasm=*
dotnet yourapp.dll
```

This dumps the disassembly for every JIT-compiled method. To focus on a specific method, use a more targeted pattern:

```
set DOTNET_JitDisasm=*YourNamespace!YourClass*
dotnet yourapp.dll
```

The output shows the IL input and the generated assembly side by side, making it easier to correlate high-level code with low-level instructions. For even more detail, use DOTNET_JitDump to capture full compilation logs including all intermediate representations and optimization decisions.

When reading disassembly, look for these patterns:

Method calls appear as call or callq instructions (on x64). If you see a call where you expected inlining, the method may be too large to inline, may be virtual, or may have other characteristics that prevent inlining. Inlined code appears as the instructions of the called method inserted directly at the call site with no call instruction.

Allocations often appear as calls to GC helper functions. On x64, heap allocation typically involves a call to the runtime's allocation stub or a specialized allocation instruction. If you see allocation calls in a tight loop, that is a strong signal that you are creating many short-lived objects and potentially causing GC pressure.

Branches appear as conditional jump instructions (je, jne, jl, jge, etc.). Too many branches in a hot loop can hurt performance because they interfere with instruction pipelining and branch prediction. Unpredictable branches that are taken roughly half the time are especially costly.

Loop unrolling appears when the loop body is repeated multiple times with different index values. If you see a compact loop with a single iteration body, the JIT did not unroll. If you see multiple copies of the body, it did.

SIMD vectorization appears when you see SSE, AVX, or NEON instructions operating on vector registers (xmm, ymm, zmm on x64; v registers on ARM64).

Vector instructions process multiple data elements in parallel. If you have a loop that could benefit from vectorization but do not see vector instructions, there may be something preventing the JIT from vectorizing.

You do not need to understand every instruction. Focus on the patterns that matter for your optimization goals: are methods being inlined, are allocations happening where you did not expect, are there unnecessary branches, is the loop vectorized? These patterns tell you whether the JIT is generating good code and whether your C# code structure enables the optimizations you want.

How Your C# Constructs Translate to Machine Code

Different C# constructs have different costs at the machine level. Understanding these costs helps you write code that compiles efficiently.

A simple value type arithmetic operation compiles to a few machine instructions with no overhead. Adding two integers is a single add instruction. Comparing two values is a cmp instruction followed by a conditional branch. These are fast and the CPU executes them in a single cycle or less.

A reference type field access requires a memory load. The CPU must read the object pointer, add the field offset, and load the value from memory. If the object is in cache, this is fast. If it is not, the CPU must fetch it from higher-level memory, which is slower. Multiple field accesses on different objects can cause cache misses if the objects are scattered in memory.

A virtual method call requires dereferencing the method table. The CPU loads the method table pointer from the object header, adds an offset to find the specific method slot, loads the method address, and jumps to it. This indirection prevents the CPU from predicting the target address reliably, which can hurt branch prediction. If the JIT can prove there is only one possible implementation (for example, the class is sealed or there are no other overrides), it can devirtualize the call to a direct jump, eliminating this cost.

A property access in C# is syntactic sugar for a method call. Simple auto-properties (private backing field, public getter/setter) compile to direct field access when the JIT can inline the getter. Complex properties with custom logic compile to actual method calls, with all the overhead that implies. If you access a property in a tight loop, ensure it is a simple auto-property or inline the getter by marking it with the [MethodImpl(MethodImplOptions.AggressiveInlining)] attribute.

A for loop compiles to a sequence of compare and branch instructions that control iteration. The JIT optimizes simple loops aggressively: it eliminates redundant bounds checks, unrolls when beneficial, and may vectorize the loop body. Complex loops with many branches, dynamic conditions, or side effects may not optimize as well.

A foreach loop over a collection may allocate an enumerator if the collection does not support foreach efficiently. Arrays and Span do not allocate because the JIT can use a simple index. List allocates a struct enumerator on the stack, which is cheap. Collections that implement IEnumerable by returning a class enumerator cause a heap allocation per foreach. If you use foreach in a hot path over such collections, consider iterating with a for loop instead.

A lambda expression may or may not allocate. A lambda that does not capture variables from its enclosing scope compiles to a static method and a delegate that is instantiated once. A lambda that captures variables compiles to a hidden class (the closure) that stores the captured variables, plus a delegate that references an instance of that class. Every time the lambda is created, a new closure instance is allocated. If you create capturing lambdas in a tight loop, you are allocating heap objects on every iteration.

Understanding how C# constructs translate to machine code does not mean you should write assembly-like C#. It means you should be aware of the performance implications of the abstractions you use, especially in hot paths where the overhead of each operation is multiplied by millions of invocations.

Chapter 4: Tiered Compilation and Dynamic Optimization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

ReadyToRun and Precompilation: Reducing JIT Overhead

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Tiered Compilation: JIT Tiering for Startup and Steady-State

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Crossgen2 and Full AOT Compilation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Profile-Guided Optimization: Using Runtime Data to Improve Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Choosing Compilation Modes for Different Scenarios

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Chapter 5: Native AOT

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

How Native AOT Works: The Compilation Story

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Startup Performance and Memory Footprint Gains

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Feature Limitations: What Native AOT Cannot Do

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Trim Mode and Dependency Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

When to Use Native AOT: Real-World Tradeoffs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Chapter 6: Method Inlining and Devirtualization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Method Inlining: How It Works and Why It Matters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Inlining Decisions: Size Limits, Heuristics, and Exceptions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Devirtualization: Turning Virtual Calls into Direct Calls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

When Inlining and Devirtualization Fail

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Designing Code the JIT Can Optimize Aggressively

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Chapter 7: Value Types, Reference Types, and Struct Design

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Stack Versus Heap: Understanding Where Types Live

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Value Type Semantics: Copies, Moves, and Escape Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Struct Design: When Structs Help and When They Hurt

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Boxing and Unboxing: The Hidden Performance Cost

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Escaping and In-Struct Storage of Value Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Chapter 8: Object Layout, Memory, and Allocation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Object Headers and Layout: Understanding Managed Object Structure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Allocation: The Allocator and the Thread-Local Allocation Buffer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Padding, Alignment, and Memory Wastage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Designing Memory-Efficient Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Field Ordering and Cache-Friendly Layout

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Chapter 9: Boxing, Generics, and Reflection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Boxing and Unboxing: Detailed Mechanics and Alternatives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Generic Code Generation: Code Sharing Across Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Generic Virtual Calls and Interface Constraints

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Reflection Performance: Costs and Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Source Generation and Compiled Reflection as Alternatives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Chapter 10: Spans, Memory, and Zero-Allocation Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Span and Memory: The Abstraction Story

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

How Spans Work: Pointers, Handles, and Safety

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Stackalloc and Native Arrays

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Zero-Allocation Parsing and Processing Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

When Zero-Allocation Does Not Help

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Chapter 11: Object Pooling and ArrayPool

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Why Object Pooling Works: Avoiding GC Pressure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

ArrayPool: Built-In Array Reuse

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Building Custom Object Pools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Concurrency and Pooling: Thread Safety Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

When Pooling Makes Things Worse

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Chapter 12: Collections and Data Structures Performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

List and Array: Allocation Patterns and Growth

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Dictionary Performance: Hashing, Collisions, and Load Factors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Concurrent Collections: Overhead and Alternatives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

LINQ and Allocation-Free Query Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Choosing the Right Structure for Your Access Pattern

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Chapter 13: The Garbage Collector: Generations and Algorithms

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

How Generational GC Works: The Theory and Practice

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

The Nursery: Short-Lived Object Collection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Promotion: How Objects Survive and Age

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Old Generation Collection: Compacting Long-Lived Objects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Tracing Algorithms and Concurrent Marking

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Chapter 14: GC Modes: Workstation, Server, Background, and Latency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Workstation GC Versus Server GC: Fundamental Differences

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Background GC: Reducing Stop-the-World Pauses

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Low-Latency and High-Throughput Modes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Large Object Heap: Behavior and Pitfalls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Pinned Object Heap and Pinning Costs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Chapter 15: Finalization, IDisposable, and Cleanup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

The Finalization Queue: How Finalizers Work

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Why Finalizers Are Expensive

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

The Dispose Pattern: Managed and Unmanaged Resources

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

SuppressFinalize and When to Use It

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Alternatives to Finalization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Chapter 16: GC Tuning, Diagnostics, and Troubleshooting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

GC Configuration Options and Environment Variables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Reading GC Counters and Metrics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Diagnosing Excessive Allocations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

LOH Fragmentation: Detection and Remediation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Tuning GC for Different Workloads

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Chapter 17: Async State Machines, Tasks, and ValueTask

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

How async and await Work: The Generated State Machine

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Task Allocation and Lifecycle

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

ValueTask: Benefits and Pitfalls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Async Overhead: Measuring and Reducing It

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Async Patterns for High-Performance Scenarios

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Chapter 18: Delegates, Closures, and Event Performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Delegates: How They Work Under the Hood

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Closures and Captured Variables: The Hidden Allocation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Multicast Delegates and Event Invocation Performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Avoiding Closure Allocations in Hot Paths

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Delegate Performance Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Chapter 19: Thread Pools, Synchronization, and Concurrency

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

The Thread Pool: Architecture and Behavior

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Lock Types: Monitor, Mutex, Semaphore, and SpinLock

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Lock Contention: Detection and Mitigation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Reader-Writer Locks and Partitioning Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

High-Level Concurrency Patterns in .NET

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Chapter 20: Lock-Free Programming and Memory Ordering

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Interlocked Operations: Atomic Updates

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Memory Ordering and the Memory Barrier

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Volatile Semantics in C# and .NET

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Building Lock-Free Data Structures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

When Lock-Free Is Not Actually Faster

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Chapter 21: I/O, Networking, and Serialization Performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Synchronous Versus Asynchronous I/O: Mechanisms and Performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Socket and Networking Performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Span-Based I/O and Zero-Copy Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Serialization Formats: Performance Comparison

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Binary Serialization and Protocol Buffers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Chapter 22: SIMD, Vectorization, and Hardware Intrinsics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

SIMD Architecture: Single Instruction Multiple Data

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

System.Numerics.Vectors and Portable SIMD

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Hardware Intrinsics: Direct CPU Instruction Access

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

AutoSIMD and Compiler Vectorization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Practical SIMD in .NET: Real Performance Gains

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Chapter 23: CPU Architecture, Caches, and Branch Prediction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

CPU Execution Pipelines: How Instructions Really Run

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Branch Prediction: Cost of Wrong Predictions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

CPU Caches: L1, L2, L3, and Cache Lines

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Cache Locality and Data Layout

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

False Sharing and Cache Line Contention

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Chapter 24: Memory Subsystem, NUMA, and Hardware Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

The Memory Hierarchy: From Registers to Main Memory

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Memory Bandwidth: When It Becomes the Bottleneck

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

NUMA: Non-Uniform Memory Access and Awareness

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

x64 Versus ARM64: Architectural Differences

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Modern Hardware Effects on .NET Performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Chapter 25: Benchmarking: Methodology and Tools

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

The Scientific Method for Performance Work

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

BenchmarkDotNet: Architecture and Features

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Controlling Variables and Isolation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Statistical Significance and Noise

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Common Benchmarking Mistakes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Chapter 26: Profiling, Tracing, and Diagnostics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Profiling Approaches: Sampling Versus Instrumentation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

EventPipe and ETW: The Tracing Infrastructure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

PerfView: Deep Runtime Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

dotnet-trace, dotnet-counters, dotnet-dump

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Production Profiling and Diagnostics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Chapter 27: Disassembly Analysis and Flame Graphs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Reading JIT Disassembly Output

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Understanding Assembly: Registers, Instructions, and Calls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Generating Flame Graphs from Profiling Data

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Interpreting Flame Graphs: Finding the Real Bottleneck

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Combining Disassembly and Flame Graphs for Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Chapter 28: Production Observability and Performance Monitoring

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Runtime Counters for Production Monitoring

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Tracing in Production: Overhead and Safety

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Metrics and Alerting for Performance Degradation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Correlating Application and Infrastructure Metrics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Continuous Performance Regressions and Guardrails

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Chapter 29: Case Studies and Real-World Performance Engineering

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Case Study: Optimizing a CPU-Bound Computational Workload

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Case Study: Reducing Allocations in a High-Throughput Service

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Case Study: Tuning for Low Latency in a Trading System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Case Study: Optimizing Startup and Deployment with Native AOT

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

Case Study: Concurrency Optimization in a Database-Backed API

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.

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/high-performancenetfromjittohardware.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/high-performancenetfromjittohardware.