Microoptimizations in High-Level Languages: A Case Study

Hawkynt*

*Address correspondence to: hawkynt@hawkynt.de

November, 2024

Abstract

In performance-critical applications, such as real-time systems and large-scale data processing, even small inefficiencies in code execution can lead to significant slowdowns or resource wastage. This paper explores microoptimizations—fine-grained code adjustments that maximize CPU efficiency—focusing on cases where compiler optimizations alone are insufficient. By examining various techniques the study demonstrates how careful application of these strategies can yield measurable performance gains in C# applications. The findings highlight the importance of bridging high-level language abstractions with low-level hardware execution to unlock the full potential of modern CPUs.

1 Introduction

In an era where software applications power everything from real-time financial trading to autonomous vehicles and artificial intelligence, performance is not merely a luxury—it is a necessity. Many of these systems require deterministic responses within strict time constraints or must process vast amounts of data with minimal latency. Even in more relaxed environments, inefficient code execution can lead to increased energy consumption leading to more carbonization, higher operational costs, and user dissatisfaction. Despite the abstraction provided by programming languages like C# , developers working on performance-critical applications when the stakes are high cannot rely solely on the smartness of compilers to produce optimal code.

This paper investigates how specific microoptimization techniques can bridge the gap between the high-level abstractions and the bare-metal realities of CPU execution. The problem centers on scenarios where traditional compiler optimizations fall short, particularly when dealing with data-intensive operations, computational bottlenecks, and situations where parallelism must be maximized. The research focuses on low-level aspects of performance optimization, including memory access patterns, and modern CPU design.

By exploring different code mechanics this study addresses the question: How can developers push the limits of a state-of-the-art programming language to meet the demands of modern performance-critical systems? Using empirical testing, we analyze the effectiveness of these methods and highlight their implications for software design.

The findings presented in this paper provide a roadmap for developers looking to maximize CPU efficiency in libraries and applications. These insights are vital not only for high-performance computing but also for understanding how modern CPUs interact with high-level languages. By bridging this gap, the research contributes to the broader goal of creating efficient and reliable software solutions in an increasingly performance-driven world.

2 Background and Theoretical Framework

This section provides an overview of modern CPU features and mechanisms relevant to the microoptimizations explored in this study.

2.1 Instruction Pipelining

Instruction pipelining is a fundamental feature of modern CPUs that enhances performance by overlapping the execution of multiple instructions. The CPU divides the execution of instructions into sequential stages, such as fetch, decode, execute, memory access, and write-back. Each stage processes a different instruction simultaneously, allowing multiple instructions to be in various stages of execution at any given time. This parallelism significantly increases throughput by keeping the processor's resources actively utilized.

However, pipelining introduces challenges such as data hazards (e.g., when an instruction depends on the result of a previous one), control hazards (e.g., caused by branches or jumps), and structural hazards (e.g., contention for resources). These hazards can stall the pipeline, leading to inefficiencies and reduced performance.

In this study, we reorganize code to optimize pipelining efficiency. Techniques such as loop unrolling, treewise result combination, and branch minimization are employed to reduce dependencies and eliminate unnecessary stalls. By ensuring that independent instructions are scheduled in a manner that avoids data hazards, we maximize the utilization of the CPU pipeline and improve overall execution performance.

2.2 Out-of-Order Execution (OOE)

OOE is a key feature of modern CPUs designed to maximize resource utilization and minimize stalls caused by instruction dependencies. Instead of executing instructions strictly in the order they appear in the program, the CPU dynamically reorders them to take advantage of available execution units, provided the reordering does not violate program correctness.

Modern CPUs feature a larger number of physical registers compared to logical registers, enabling the use of techniques like register renaming to resolve dependencies and avoid pipeline stalls. Additionally, CPUs are equipped with multiple Arithmetic Logic Units (ALUs) and execution ports, allowing them to process multiple instructions in parallel. However, the specifics of register renaming and resource allocation can vary significantly between CPU models and manufacturers, introducing non-deterministic behavior in low-level execution.

Due to these variations, this paper does not focus on register renaming or other internal mechanisms specific to particular CPU architectures. Instead, it emphasizes optimizations that align with general OOE principles, such as minimizing data dependencies and leveraging instruction-level parallelism effectively.

2.3 Branch Prediction and Speculative Execution

Branch prediction is a critical CPU mechanism for mitigating delays caused by control hazards, such as conditional branches or loops. It allows the processor to predict the outcome of a branch and speculatively execute subsequent instructions. If the prediction is correct, the speculative execution results are committed, maintaining a smooth pipeline flow. However, incorrect predictions trigger pipeline flushes, discarding speculative results and introducing performance penalties.

Optimizing for branch prediction involves structuring code to align with predictable patterns, thereby reducing mispredictions. Techniques include reordering instructions to favor common branch outcomes, employing jump tables to replace complex conditional chains, and utilizing conditional move (`cmov`) instructions and bit-manipulations to eliminate branches entirely. By minimizing the presence and impact of branches, developers can enhance pipeline efficiency and reduce the risk of performance degradation caused by control hazards.

2.4 Caching and Memory Hierarchy

The memory hierarchy in modern CPUs plays a critical role in performance optimization by reducing the latency of memory operations. At its core, the hierarchy consists of registers, caches (L1, L2, L3), main memory (RAM), and secondary storage. Each level provides a trade-off between speed, size, and accessibility.

Caches serve as high-speed intermediate storage, holding frequently accessed data to reduce the latency of fetching from slower main memory. Optimizations such as cache-friendly access patterns, which ensure data is accessed sequentially or in predictable strides, can significantly enhance performance by maximizing cache utilization and reducing cache misses.

Most CPU caches implement mechanisms like prefetching and look-ahead, where the processor predicts future memory accesses and loads data into the cache proactively. Developers can exploit this behavior by aligning their memory access patterns with these predictions, effectively minimizing delays caused by waiting for data from main memory.

Understanding the interplay between registers, stack, heap, and cache is essential for efficient memory management:

- **Registers**: The fastest memory, directly on the CPU. Frequently used variables and temporary results are stored here. However, when too many registers are in use simultaneously, some contents may be "spilled" onto the stack. These spills may remain in the cache, but in deeply nested function calls or memory-intensive operations, the stack may need to be read from main memory, introducing latency.

- **Stack**: Used for function calls and local variables, benefiting from cache proximity due to its sequential access pattern. Stack memory is generally faster than heap memory but may incur additional latency when overflows occur or when data spills need to be retrieved from main memory.
- **Heap**: Dynamically allocated chunks of memory in RAM, which is less cache-efficient due to its unpredictable access patterns. Data stored in here is most likely to provoke a cache-miss, leading to longer delays until the data is available at the CPU for processing.
- **Cache Levels**:
  - **L1 Cache**: Closest to the CPU, very fast but small in size.
  - **L2 and L3 Caches**: Larger and slower but still significantly faster than main memory.

Effective optimizations include:

- Minimizing heap allocations to reduce unpredictable cache access.
- Preferring stack allocations (`stackalloc` in C#) for small, short-lived data structures.
- Ensuring linear or contiguous data access to align with cache prefetching.
- Reducing register pressure by limiting the number of simultaneously active variables in performance-critical sections to avoid register spills.

By leveraging the memory hierarchy effectively and understanding the trade-offs between registers, stack, heap, and cache behavior, developers can reduce latency, increase throughput, and fully utilize the capabilities of modern CPUs.

2.5 Vectorization and Single-Instruction-Multiple-Data (SIMD)

Vectorization is a performance optimization technique that processes multiple data elements in parallel by applying the same operation to all elements simultaneously. This is achieved using SIMD-instructions, which are supported by modern CPUs through extensions like SSE (128 bits), AVX2 (256 bits), and AVX-512 (512 bits). These extensions enable efficient computation on vectors, which are data structures containing multiple elements of the same type (e.g., bytes, integers or floating-point numbers) thus allowing increased computational throughput.

In addition to hardware-supported SIMD, a primitive form of vectorization can be achieved by processing data in larger chunks, such as words (16 bits), double words (32 bits) or quad words (64 bits), instead of single bytes. This approach leverages the CPU's natural ability to handle wider data types without requiring specialized SIMD hardware, offering a basic level of parallelism in operations like addition or bitwise manipulation.

3 Materials and Methods

This section outlines the hardware, software, and methodologies used to conduct the experiments in this study. The detailed setup and tools ensure replicability of the results, and where applicable, additional resources are referenced.

3.1 Hardware Specifications

The tests were performed on two different systems to evaluate performance across varying hardware configurations:

- **System 1 (Dell PC):** Intel Core i9-11950H (2.6 GHz, AVX-512 support), 128 GB DDR4 RAM (3200 MHz), running Windows 10 Enterprise (22H2).
- **System 2 (Lenovo Laptop):** Intel Core i7-12700H (AVX2 support), 16 GB DDR4 RAM, running Windows 10 Professional (22H2).

3.2 Software Environment

All experiments were conducted using the following tools and software:

- **Integrated Development Environment:** Microsoft Visual Studio 2022 (version 17.10.0 Preview 6.0).
- **Compiler:** .NET SDK 9.0.100-preview.3.24204.13, targeting C# 8.0.
- **Disassembly Inspection:** Disasmo extension (version 5.9.2) for generating assembly listings.
- **Performance Benchmarking:** BenchmarkDotNet (version 0.14.0) NuGet Package.
- **Assembly Analysis Tools:** Additional assembly inspection was performed using the Web-services http://SharpLab.io and http://GodBolt.org.
- **Framework Compilation Testing:** .NET Framework 4.8.2 compilation results were also inspected for comparative purposes.

3.3 Project Setup

A standard console project was created using the Visual Studio IDE with the following settings:

- **Configuration:** Release mode.
- **Unsafe Code:** The `allow unsafe` option was enabled to facilitate low-level operations.
- **Target Framework:** Experiments were run on both .NET 8.0 and .NET Framework 4.8.2.

3.4 Methodology

To ensure accuracy and replicability:

- Benchmarks were written and executed using BenchmarkDotNet. Results were averaged over multiple iterations to account for environmental variations.
- Assembly code was inspected using Disasmo, SharpLab.io, and Godbolt.org to verify the compiler's code generation.
- AVX-512 and AVX2 support were explicitly tested using appropriate vectorization and SIMD instructions, dependent on the hardware capabilities of each system.

This methodology ensures consistent benchmarking across different systems and validates the low-level optimizations applied to the code.

3.5 Experimental Design

The primary objective of this study was to systematically evaluate the impact of various microoptimization techniques on the performance of critical C# applications. Specifically, the study aimed to measure how these optimizations influence execution time, instruction throughput, total instruction count, and memory utilization. The experiments were designed to account for differences in hardware capabilities, such as SIMD support, memory latency, and branch prediction mechanisms, ensuring that the results are generalizable across modern CPU architectures.

Each optimization was tested under controlled conditions on the hardware platforms. This setup allowed a comparison of performance gains across differing levels of hardware vectorization support. For consistency, benchmarks were executed multiple times, and results were averaged to minimize the impact of environmental variations such as background processes or thermal throttling.

Branch prediction behavior, which significantly affects pipeline efficiency, was explicitly analyzed. The study categorized and quantified the following jump types:

- **Unconditional Jumps and Method Calls:** Considered predictable and used to assess the baseline cost of branching.
- **Jump Tables:** Associated with switch-case constructs, often challenging for branch predictors due to dynamic target variability.
- **Conditional Forward Jumps:** Typically predicted as not taken by CPUs, providing insights into misprediction penalties.
- **Conditional Backward Jumps:** Often associated with loops and assumed to be always taken, allowing a measure of prediction efficiency in iterative constructs.

The study evaluated the following optimization techniques:

- Built-in functions like `ReadOnlySpan.SequenceEqual` for comparison
- Naïve implementations without further optimization as a baseline

- No-bounds-check implementations using unsafe pointers
- n-times Loop unrolling with unsafe pointers
- Branch minimization using bitwise operations
- Treewise result combination to minimize data dependencies
- Chunked data processing
- A combination of the above
- Branch tables
- Vectorization
- Interleaving

Each experiment was designed to measure performance under varying conditions, such as input sizes and hardware capabilities, ensuring robustness across different scenarios. The results aim to identify both generalizable patterns and hardware-specific behaviors in optimization.

3.6 Statistical Analysis

Performance results were analyzed using BenchmarkDotNet, which provides statistical summaries such as mean execution time, standard deviation, and confidence intervals. Each benchmark was run multiple times (N = 10) to ensure consistent results and reduce the impact of environmental variability (e.g., CPU load, thermal throttling).

The statistical tests included:

- **Paired Comparisons:** Each optimization technique was compared against the naïve implementation as a baseline using the percentage improvement in mean execution time.
- **Variance Analysis:** Standard deviations were analyzed to assess the stability of the results across runs and hardware configurations.
- **Hardware Impact Assessment:** Results from AVX-512-capable hardware (System 1) and AVX2-only hardware (System 2) were compared to identify the influence of SIMD capabilities.

All statistical analyses were performed using the built-in tools provided by `BenchmarkDotNet`. In cases where additional granularity was needed, raw data was exported for further inspection using standard spreadsheet software. P-values were calculated where applicable to determine the significance of observed improvements, with P ¡ 0.05 considered statistically significant.

4 Results

5 Discussion

6 Conclusion and Outlook

7 References

8 Appendix

8.1 Compiler and JIT Behavior

C# is a just-in-time (JIT) compiled language, meaning that the source compiler generates intermediate code, known as Common Intermediate Language (CIL), which is then compiled to native machine code at runtime on the target architecture. While complete ahead-of-time (AOT) compilation is supported in newer versions of the .NET framework for certain project types, the experiments in this study found no significant difference between AOT on JIT compilation.

This section provides a deeper look into the source code, CIL, and resulting assembly for different optimization strategies. By dissecting the transformations applied by the compiler and JIT, we gain insights into how each optimization aligns with CPU execution.

The algorithms under study involve checking two chunks of memory for equality, an operation frequently used in comparing strings, matrices, images, or other data structures. The results illustrate the trade-offs and benefits of each strategy.

8.2 Naïve Approach

This approach serves as the baseline for evaluating the impact of subsequent optimizations. This implementation utilizes a simple `for`-loop to compare elements sequentially, focusing on clarity and correctness without applying any additional optimization techniques.

- **Straightforward Execution**: The structure of the code ensures straightforward translation to intermediate and native machine code, resulting in predictable execution behavior across hardware platforms.
- **Minimal Instruction Overhead**: The implementation avoids any unnecessary preprocessing or complex constructs, producing a clean sequence of operations that directly reflect the algorithm's logic.
- **Baseline Comparison**: This approach provides a clear and simple reference point for measuring the effectiveness of further optimizations, offering an unambiguous metric for improvements in execution time, instruction throughput, and memory utilization.

tbd:for-loop

8.3 Bounds-Free

This approach eliminates the overhead introduced by implicit bounds-checks generated by the compiler for each array access. By using unsafe pointers, the implementation bypasses these checks, resulting in improved performance while maintaining the same algorithmic structure.

- **Reduced Instruction Count**: The removal of bounds-checking instructions minimizes the total number of executed instructions, leading to a more streamlined execution path.
- **Improved Execution Speed**: By avoiding the runtime overhead of validating array indices, this approach achieves faster execution, particularly for large datasets.
- **Direct Memory Access**: Utilizing unsafe pointers allows for direct memory access, aligning more closely with hardware operations and reducing the abstraction overhead.
- **Preserved Logical Structure**: Despite these low-level improvements, the implementation preserves the logical simplicity of the original algorithm, ensuring maintainability and clarity.

tbd:unsafe pointers

8.4 Unrolled

This approach reduces the overhead associated with loop control by unrolling the loop, allowing multiple iterations to be processed within a single loop body. By decreasing the frequency of branching and loop counter updates, the performance is significantly enhanced.

- **Reduced Branching Overhead**: By processing multiple elements per iteration, the number of branch instructions is minimized, decreasing the cost associated with loop control.
- **Improved Instruction-Level Parallelism**: Unrolling the loop provides more opportunities for the CPU to execute instructions concurrently, increasing throughput.
- **Streamlined Execution Path**: The simplified control flow reduces pipeline stalls and enhances CPU resource utilization, particularly on hardware with advanced branch prediction mechanisms.
- **Better Utilization of CPU Caches**: Unrolling enables tighter grouping of memory accesses, improving spatial locality and reducing cache misses for large datasets.

tbd:8-times unrolled with lots of if-checks

8.5 Branch Minimized

This approach minimizes the number of conditional branches within the loop by employing bitwise operations and a single conditional check per loop iteration. By reducing reliance on branch prediction, the implementation achieves more predictable and efficient execution.

- **Fewer Conditional Branches**: Consolidating multiple conditional checks into a single branch significantly reduces the likelihood of branch mispredictions.
- **Improved Pipeline Efficiency**: Minimizing branches lowers the risk of pipeline flushes caused by mispredictions, ensuring smoother instruction flow.
- **Predictable Execution Patterns**: The use of bitwise operations introduces a more deterministic execution path, simplifying the CPU's workload.
- **Consistent Performance Gains**: This optimization yields measurable improvements in scenarios with unpredictable input data or irregular patterns that would otherwise hinder branch prediction accuracy.

tbd:unrolled with XOR and sequential OR, employing a single if inside the loop

8.6 Instruction Level Parallelism

This approach exploits the availability of multiple ALUs within modern CPUs by structuring computations to enable parallel execution. By performing multiple operations concurrently and preloading both operands into registers, the implementation maximizes hardware utilization and enhances cache efficiency.

- **Increased ALU Utilization**: Distributing computations across multiple ALUs allows simultaneous execution of independent operations, significantly improving throughput.
- **Reduced Data Dependencies**: Preloading operands minimizes the risk of stalls caused by waiting for data to be fetched, enabling smoother and faster execution.
- **Cache-Friendly Access Patterns**: By preloading both operands, this approach aligns memory access patterns with CPU caching mechanisms, reducing cache misses and improving data locality.
- **Maximized Pipeline Utilization**: Parallelizing computations ensures that the CPU pipeline remains active, reducing idle cycles and enhancing overall efficiency.

tbd:treewise OR

8.7 Chunking with Native General-Purpose Register Width

This approach leverages the full width of general-purpose registers to process multiple data elements simultaneously. By aligning the size of processed chunks with the register width, the implementation reduces the total number of memory operations and improves computational efficiency.

- **Maximized Register Utilization**: Using the full width of general-purpose registers minimizes wasted capacity, allowing more data to be processed in each operation.

- **Fewer Memory Accesses**: Larger chunks reduce the frequency of memory loads and stores, decreasing memory bandwidth usage and improving performance.
- **Improved Cache Efficiency**: Aligning data chunks with register width ensures better spatial locality, enhancing cache performance and reducing misses.
- **Streamlined Execution**: Processing larger chunks at once simplifies the control flow, reducing loop overhead, utilizing more bits of the ALUs' operation results and enabling faster iteration through datasets.

tbd:4 sequential loops using long,int,short,byte

8.8 Extended Chunking

This approach builds upon the benefits of native register-width chunking by combining it with additional optimizations, such as instruction-level parallelism and loop unrolling. By synergizing these techniques, the implementation achieves higher throughput and more efficient utilization of hardware resources.

- **Enhanced Data Throughput**: Combining chunking with instruction-level parallelism allows multiple registers to process data concurrently, further increasing the volume of data processed per cycle.
- **Reduced Loop and Branch Overhead**: Incorporating loop unrolling alongside chunking minimizes the number of loop control instructions and conditional branches, streamlining execution.
- **Improved Cache Utilization**: Larger and parallelized data processing enhances spatial locality and prefetching efficiency, reducing latency associated with cache misses.
- **Higher Resource Utilization**: By coordinating chunking with multiple optimizations, the approach fully exploits available CPU execution units and register file capacity.

tbd:long loop ILP, int, short, byte loops normal

8.9 Branch Tables

This approach replaces conditional branches with precomputed branch tables, enabling direct jumps to predetermined targets. By eliminating the need for branch prediction in loops, this technique improves execution predictability and reduces the penalties associated with mispredictions.

- **Elimination of Conditional Branches**: Using branch tables removes the need for runtime evaluations of conditions, reducing control flow complexity.
- **Improved Predictability**: Direct jumps via branch tables ensure deterministic execution paths, bypassing the CPU's branch prediction mechanisms.

- **Reduced Pipeline Stalls**: The absence of conditional branches minimizes the likelihood of pipeline flushes, maintaining smooth instruction flow.
- **Faster Execution in Dense Scenarios**: For scenarios with numerous discrete cases (e.g., switch statements), branch tables streamline execution by consolidating control flow into a single, efficient construct.

tbd:branch table for 16 cases, the rest as before

8.10 Vectorization

This approach leverages SIMD instructions to process multiple data elements simultaneously using vector registers. By aligning operations with the size of cache lines and utilizing wide vector registers, vectorization improves data throughput and reduces memory access latency.

- **Increased Data Throughput**: Vector instructions allow multiple elements to be processed in parallel, significantly increasing computational efficiency.
- **Improved Cache Utilization**: Aligning vector loads and stores with cache line boundaries optimizes spatial locality, reducing cache misses and leveraging prefetching mechanisms.
- **Reduced Loop Overhead**: Processing multiple elements per instruction minimizes loop iterations and associated control flow overhead.
- **Enhanced Memory Bandwidth Efficiency**: By fetching and processing larger chunks of data in each operation, vectorization maximizes memory bandwidth utilization.

tbd:using vec128, vec256 and vec512

8.11 Interleaving

This approach optimizes instruction scheduling by interleaving load, store, and compute operations. By overlapping these tasks, the implementation takes full advantage of the CPU pipeline's ability to handle multiple types of instructions simultaneously, minimizing idle cycles and reducing reliance on stack operations.

- **Maximized Pipeline Utilization**: Interleaving ensures that load, store, and compute operations are executed in parallel, leveraging multiple execution units and minimizing pipeline stalls.
- **Reduced Stack Dependence**: By focusing on registers for intermediate computations and data storage, the approach avoids the overhead of stack operations, improving execution speed.
- **Improved Instruction Overlap**: Scheduling loads and stores alongside computations reduces latency by hiding memory access delays behind execution tasks.

- **Better Memory Throughput**: Optimized memory access patterns ensure that data prefetching and cache usage are fully utilized, reducing bottlenecks in data movement.

tbd:LOAD C, STORE A, CALCULATE B