Workload-Sensitive Energy Benchmarking of .NET JSON Libraries

cs-26-pt-10-03

Masters Project





AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
http://www.aau.dk

Title:
Workload-Sensitive Energy Benchmarking of .NET JSON Libraries

Theme:
Energy Aware Programming

Project Period:
Spring Semester 2026

Project Group:
CS-IT10, Group 3
cs-26-pt-10-03

Participant(s):
Georgian Voda
Toms Vanders

Supervisor:
Bent Thomsen
Søren Kejser Jensen

Page Numbers:
89

Date of Completion:
June 8, 2026

Abstract:

Software energy consumption is difficult to observe during day-to-day development, and runtime is not always a reliable proxy for energy use. This thesis studies energy consumption at the library level in .NET, using JSON serialisation and deserialisation as the case study. It extends previous work by introducing controlled JSON workload variation, a more controlled RAPL-based measurement environment, and an exploratory Metrion-based comparison under noisy conditions.
Five library configurations are evaluated: `STJRefGen`, `STJSrcGen`, `Newtonsoft.Json`, `SpanJson`, and `Utf8Json`. The results show that energy rankings are workload-dependent. In general, `SpanJson` uses the least energy for deserialisation, `STJSrcGen` uses the least energy for serialisation, and `Newtonsoft.Json` usually uses the most energy. However, rankings change across workload dimensions such as Width, String Length, Numeric Length, and String Composition. Runtime evidence shows that `System.Text.Json` source generation mainly benefits serialisation, where generated write handlers are available. The Metrion results mostly preserve rankings under the tested noisy conditions, but remain exploratory. Overall, the thesis shows that JSON-library energy benchmarking should be workload-sensitive.

The content of this report is freely available, but publication (with reference) may only be pursued due to agreement with the author.

Summary

Software energy consumption is becoming more important in software engineering, but it is still harder to observe than runtime or memory usage. Runtime is often used as a proxy for energy, because faster programs often consume less energy. However, runtime and energy are not the same. Energy also depends on how much power the hardware draws while the program runs. This means that energy should be measured directly when energy efficiency is the question.

This thesis studies energy consumption at the library level in the .NET ecosystem. The case study is JSON serialisation and deserialisation. JSON is widely used in web APIs and data exchange, and .NET developers can choose between several JSON libraries with different implementations. The study compares five library configurations: `STJRefGen`, `STJSrcGen`, `Newtonsoft.Json`, `SpanJson`, and `Utf8Json`.

The thesis extends previous work that measured .NET JSON-library energy consumption on a small set of real-world JSON documents. That work showed that library choice can affect energy use, and that rankings can change with document size. However, the documents used in that study varied several properties at the same time. This made it difficult to say which JSON property caused a ranking change. The previous setup also had limited environmental control, and the results were mainly descriptive.

The present thesis addresses these gaps in three ways. First, it introduces a deterministic synthetic JSON generator. This generator makes it possible to vary one workload dimension at a time while keeping the rest fixed. The main dimensions studied are Size, Depth, Width, String Length, Numeric Length, and String Composition. Second, the thesis strengthens the measurement setup by using a more controlled environment with hardware and operating-system controls. Third, it adds selected runtime explanations and an exploratory noisy-environment comparison using Metrion.

The benchmarks are executed with BenchmarkDotNet. In the clean environment, energy is measured using a RAPL-based Energy Diagnoser. This reports energy in the CPU Package and DRAM domains. In the noisy environment, background load is introduced with `stress-ng`, and energy is measured using Metrion-based process-level attribution. Because the noisy-environment part covers only a subset of the benchmark suite, it is treated as exploratory.

The first research question asks how library rankings and scaling rates differ across JSON workload dimensions. The results show that rankings are workload-dependent. In general, `SpanJson` uses the least energy for deserialisation, `STJSrcGen` uses the least energy for serialisation, and `Newtonsoft.Json` usually uses the most energy. However, these rankings are not fixed. They change across several dimensions, especially Width, String Length, Numeric Length, and String Composition.

For example, SpanJson is the strongest deserialiser at short string lengths, but the System.Text.Json variants become lower-energy as strings grow longer. At the widest object level, Utf8Json becomes the lowest-energy deserialiser. During serialisation, STJSrcGen is usually strongest, but Unicode-heavy and Unicode-escape-heavy strings favour SpanJson and Utf8Json. These results show that one JSON document or one baseline workload is not enough to describe library energy behaviour.

The scaling results explain why rankings change. Some dimensions, such as Size, Depth, and Width, often increase energy per added unit. Other dimensions, such as String Length, Numeric Length, and String Composition, often show decreasing energy per unit as the workload grows. This means that fixed costs are sometimes amortised over larger inputs. It also means that a library can have a low scaling ratio but still use high total energy. Newtonsoft.Json is the clearest example: it sometimes scales less steeply than other libraries, but usually starts from a higher energy level.

The second research question asks what architectural and runtime factors explain the observed differences. The clearest explanation concerns the two System.Text.Json variants. During deserialisation, STJRefGen and STJSrcGen remain close because both use the same steady-state read path after metadata has been prepared. During serialisation, STJSrcGen can use generated write handlers, while STJRefGen uses the general converter path. This explains why source generation mainly benefits serialisation.

The comparison between SpanJson and System.Text.Json shows that read and write paths matter differently. SpanJson often has the lowest deserialisation energy, which is consistent with a more specialised read path. During serialisation, the System.Text.Json variants benefit from their UTF-8 writer path, and STJSrcGen gains further from generated write code. These explanations are supported by documentation, source inspection, generated code, allocation and garbage-collection data, and EventPipe traces inspected in SpeedScope. The diagnostics are used to show which code paths are active, not to assign exact energy costs to individual methods.

The third research question asks how environmental conditions affect rankings and whether Metrion can provide consistent measurements. In the validation case at 0% stress, Metrion largely preserves the rankings measured by the Energy Diagnoser. Deserialisation rankings match exactly, while serialisation differs only by one small adjacent swap between near-tied libraries. Under the tested noisy conditions, rankings also remain mostly stable across the evaluated Depth, Size, and Width subsets. This suggests that Metrion is promising for ranking-oriented comparisons under noise, but more validation is needed.

The main practical result is that library choice can matter for energy, but the best choice depends on the operation and workload. STJSrcGen is the strongest default for serialisation among the evaluated libraries. SpanJson is often strongest for deserialisation, but its advantage depends on workload shape and must be weighed against maintenance and compatibility concerns. Newtonsoft.Json remains mature and feature-rich, but usually uses more energy.

Overall, the thesis shows that energy benchmarking of JSON libraries should be workload-sensitive. Energy consumption is not only a property of the library. It also depends on the JSON document, the operation, and the measurement environment. The benchmark infrastructure and synthetic workload suite developed in this thesis provide a basis for studying these effects more systematically in future work.

Contents

1 Introduction **1**

1.1 Problem Statement . . . 2

1.2 Research Questions . . . 2

1.3 Expected Contributions . . . 3

1.4 Report Structure . . . 3

2 Related Work **4**

2.1 Foundations from Prior Work . . . 4

2.1.1 Energy Measurement . . . 4

2.1.2 Energy Research for C# and .NET . . . 5

2.1.3 JSON Serialisation as a Case Study . . . 5

2.1.4 Benchmarking Methodology . . . 5

2.2 JSON Workload Characterization . . . 6

2.3 Environmental Control in Energy Benchmarking . . . 6

2.3.1 Strict Environmental Control . . . 7

2.3.2 Attribution-Based Measurement . . . 7

3 Experiment Methodology **9**

3.1 Measurement Infrastructure . . . 10

3.1.1 BenchmarkDotNet . . . 10

3.1.2 Energy Measurement with RAPL . . . 10

3.1.2.1 Integration into BenchmarkDotNet . . . 11

3.1.3 Energy Measurement with Metrion . . . 12

3.1.3.1 Metrion . . . 12

3.1.3.2 Integration into BenchmarkDotNet . . . 12
3.1.4 Runtime Diagnostics . . . 14
3.2 Experimental Environment . . . 15
3.2.1 Hardware and Software Platform . . . 15
3.2.2 Clean Environment . . . 15
3.2.2.1 Core Pinning . . . 16
3.2.2.2 Disabling SMT . . . 16
3.2.2.3 Disabling Turbo Boost . . . 16
3.2.2.4 Locking CPU Frequency . . . 16
3.2.2.5 OS and Background Services . . . 17
3.2.2.6 Disabling C-states . . . 17
3.2.2.7 Environment Setup . . . 17
3.2.3 Noisy Environment . . . 18
3.2.3.1 Stress-ng . . . 18
3.3 Benchmark Design . . . 19
3.3.1 Libraries Under Test . . . 19
3.3.2 Operations . . . 20
3.3.3 Workload Dimensions and JSON Generation . . . 20
3.3.3.1 Dimensions . . . 20
3.3.3.2 Workload Generation . . . 21
3.3.4 Factorial and Isolation Benchmark Suites . . . 22
3.3.4.1 Factorial Benchmark Suite . . . 22
3.3.4.2 Isolation Benchmark Suite . . . 23
3.4 Data Analysis . . . 25
3.4.1 Normality Assessment . . . 25
3.4.2 Significance Testing . . . 25
3.4.3 Effect Sizes . . . 26
3.4.4 Ranking and Ratio Summaries . . . 26
3.4.5 Per-Unit Scaling Ratios . . . 26
3.5 Metrion Validation . . . 27

3.5.1 Validation Design . . . 27
3.5.2 Validation Activities . . . 28

4 Implementation **29**

4.1 BenchmarkDotNet Energy Diagnoser Extension . . . 29
4.1.1 Extended domain support . . . 29
4.1.2 Multiple socket systems compatibility . . . 31
4.2 BenchmarkDotNet Metrion Integration . . . 32
4.2.1 Implementation decisions . . . 32
4.2.2 Profiler components . . . 32
4.3 JSON Generator Implementation . . . 37
4.3.1 Configuration Parameters . . . 37
4.3.2 Document Structure . . . 38
4.3.3 Leaf Value Generation . . . 38
4.4 Benchmark Suite Implementation . . . 40
4.4.1 Benchmark Configuration . . . 40
4.4.2 Workload Configuration and Data Generation . . . 41
4.4.3 Data Model and Source Generation . . . 43
4.4.4 Per-Library Operations . . . 44

5 Results **46**

5.1 Factorial Benchmark Overview . . . 47
5.1.1 Rank Composition . . . 47
5.1.2 Library Positioning . . . 49
5.2 RQ1: Library Rankings Across Workload Dimensions . . . 50
5.2.1 Size . . . 50
5.2.2 Depth . . . 53
5.2.3 Width . . . 55
5.2.4 String Length . . . 57
5.2.5 Numeric Length by Type . . . 60
5.2.6 String Composition . . . 63

5.2.7 Cross-Dimension Ranking Overview . . . 66
5.2.8 Cross-Dimension Scaling Overview . . . 67
5.2.9 Key Findings . . . 69
5.3 RQ2: Architectural and Runtime Explanations . . . 71
5.3.1 STJSrcGen and STJRefGen Separate Only on Serialisation . . . 71
5.3.2 SpanJson and System.Text.Json Swap Ranks Across Operations . . . 72
5.4 RQ3: Environmental Effects and Metrion Validation . . . 73
5.4.1 Metrion Validation . . . 73
5.4.1.1 Ranking Agreement . . . 73
5.4.1.2 Magnitude Agreement . . . 74
5.4.2 Environmental effects . . . 75
5.4.2.1 Depth effects . . . 76
5.4.2.2 Size effects . . . 77
5.4.2.3 Width effects . . . 78

6 Discussion **79**

6.1 Synthesis and Implications . . . 79
6.1.1 Workload-dependent Energy Rankings . . . 79
6.1.2 Operation-specific Library Mechanisms . . . 80
6.1.3 Environmental Measurement and Ranking Stability . . . 80
6.1.4 Considerations beyond energy . . . 81
6.2 Reflections . . . 81
6.2.1 Role of the Factorial and Isolation Benchmark Suites . . . 81
6.2.2 Workload Dimensions and Scope Decisions . . . 82
6.2.3 Breadth, Diagnostics, and Explanatory Limits . . . 82
6.2.4 Use of AI Tools . . . 83
6.3 Threats to Validity . . . 83
6.3.1 Construct Validity . . . 83
6.3.2 Internal Validity . . . 84
6.3.3 External Validity . . . 84

6.3.4 Conclusion Validity . . . 85
6.4 Future Work . . . 86
6.4.1 More Realistic and Refined Workload Designs . . . 86
6.4.2 Deeper Architectural Explanations . . . 86
6.4.3 Expanded Noisy-environment Validation . . . 87
6.4.4 Generalisability Across Platforms and Libraries . . . 87

7 Conclusions **88**

A Tables **i**
A.1 BenchmarkDotNet's configuration parameters . . . i
A.2 RAPL Power Domains . . . ii
A.3 Per-Configuration Isolation Benchmark Data . . . ii
A.3.1 Size . . . iii
A.3.2 Depth . . . vi
A.3.3 Width . . . ix
A.3.4 String Length . . . xii
A.3.5 Numeric Length and Type . . . xv
A.3.6 String Composition . . . xix

B Text Listings **xxvi**
B.1 Hardware Component List . . . xxvi
B.2 Jil Deserializer Stack Trace . . . xxvii

C Source code listings **xxviii**
C.1 Repositories details . . . xxviii
C.2 Listings . . . xxix

D Per-Workload Factorial Detail Figures **xxxiii**

E Noisy Environment Effects Detail Figures **xxxvi**
E.1 Depth Environment Effects . . . xxxvi
E.2 Size Environment Effects . . . xxxviii

E.3 Width Environment Effects xxxix

Chapter 1

Introduction

Software energy consumption has become an increasingly relevant concern in software engineering. Software implementation and design choices determine how hardware resources are exercised and can therefore affect energy use [1]. Energy is also less visible to developers than runtime or memory usage, because measuring it requires external power meters, on-chip energy counters such as Intel's Running Average Power Limit (RAPL), or predictive models, each with its own setup requirements and limitations [2].

Runtime is often used as a proxy for energy, since faster programs often consume less energy. However, energy also depends on the average power drawn during execution, so runtime and energy are not identical. Prior programming-language studies show that the two are often correlated, but can produce different rankings [3, 4, 5]. This means energy should be measured directly rather than inferred from execution time alone.

For developers already working inside a fixed ecosystem, language-level comparisons are less actionable than library-level ones. A .NET developer is more likely to choose between serializers, parsers, or frameworks than to change programming language. Studying libraries within one ecosystem keeps the language and runtime fixed and focuses the comparison on implementation differences. This is important because broad language rankings can conflate language effects with runtime details such as JIT compilation, garbage collection, and library choice [6].

JSON serialisation is a suitable case study for such a comparison. JSON is widely used for web APIs and data exchange, accounting for approximately 97% of Cloudflare API requests in 2021 [7]. The .NET ecosystem also provides several JSON libraries with different designs, including the long-established `Newtonsoft.Json`, the default `System.Text.Json`, and performance-oriented alternatives such as `SpanJson` and `Utf8Json`. Prior studies show that .NET serializers differ in runtime and memory behaviour [8, 9], while evidence from Java shows that JSON-library choice can also affect energy consumption [10].

Taken together, this makes .NET JSON serialisation a useful setting for studying library-level software energy consumption. The same high-level task can be implemented by different libraries with different runtime behaviour, and the resulting energy use may depend on both the library implementation and the JSON document being processed.

1.1 Problem Statement

This thesis extends our previous study, which added RAPL-based energy measurement to BenchmarkDotNet through an Energy Diagnoser extension and used it to compare .NET JSON libraries across serialisation and deserialisation [11]. That study showed that library choice can affect energy consumption, and that rankings can change with JSON document size. However, it used only a small set of real-world JSON documents and an initial measurement setup with limited environmental control.

The present thesis addresses three gaps left by the previous study. First, the previous study used real-world JSON documents whose characteristics varied together. This made the measurements realistic, but it also meant that a ranking change could not be attributed to a specific JSON property, nor could it show whether the same ranking would generalise to other document shapes. Second, the results were mainly descriptive. They reported which libraries used less energy, but did not explain the implementation or runtime behaviour behind the differences. Third, the measurement setup itself needed to be strengthened. More environmental controls were needed to reduce measurement variance and establish reliable baseline rankings. At the same time, it remained unclear how such rankings would behave in more production-like environments.

The problem addressed in this thesis is therefore that the energy behaviour of .NET JSON libraries is not yet well characterised across JSON workload variation, implementation behaviour, and measurement conditions. To address this, the thesis develops a more controlled measurement environment, introduces a synthetic JSON benchmark suite for varying workload dimensions, analyses selected runtime explanations, and explores process-level energy attribution with Metrion under noisy conditions.

1.2 Research Questions

Based on the problem statement, this thesis is guided by three research questions:

RQ1 How do libraries' energy rankings and scaling rates differ across JSON workload dimensions?

RQ2 What architectural and runtime factors explain the observed differences between libraries?

RQ3 How do environmental conditions affect library energy rankings, and can Metrion provide consistent measurements across them?

RQ1 addresses the workload-variation gap by measuring how library rankings change when JSON characteristics are varied systematically. RQ2 addresses the explanatory gap by analysing selected implementation and runtime mechanisms behind the observed differences. RQ3 provides an initial investigation of the measurement-condition gap by comparing controlled measurements with a deliberately noisier environment using Metrion-based attribution.

1.3 Expected Contributions

This thesis extends the previous .NET JSON library energy study from a small set of real-world JSON documents to a controlled benchmark study of how energy use changes across JSON workload variation. The main contributions are:

1. A deterministic synthetic JSON workload generator. It produces documents with controlled workload dimensions, allowing Size, Depth, Width, String Length, Numeric Length, and String Composition to be varied independently.
2. An extended BenchmarkDotNet energy measurement infrastructure. This includes additional RAPL-domain support, multi-socket reporting, and Metrion integration for process-level energy attribution.
3. A benchmark methodology for comparing .NET JSON libraries across two complementary workload views:
   (a) a factorial benchmark suite, used as a broad overview of library behaviour across combined Depth, Width, and Content type settings;
   (b) an isolation benchmark suite, used to measure how library rankings, energy ratios, and scaling rates change when one workload dimension is varied at a time.
4. An empirical characterisation of .NET JSON-library energy behaviour with selected runtime explanations. The thesis compares `STJRefGen`, `STJSrcGen`, `Newtonsoft.Json`, `SpanJson`, and `Utf8Json` across serialisation and deserialisation, and explains selected differences using generated code, source inspection, allocation, garbage-collection data, and EventPipe traces.
5. An exploratory comparison of clean-environment RAPL measurements and noisy-environment Metrion attribution, used to investigate whether library-ranking information is preserved when background load is introduced.

1.4 Report Structure

The remainder of this thesis is organised as follows. Chapter 2 reviews related work on software energy measurement, JSON workload characterisation, and environmental control in energy benchmarking. Chapter 3 describes the experimental methodology, including the workload dimensions, benchmark designs, measurement environments, and data analysis procedure. Chapter 4 presents the implemented benchmark infrastructure, including the energy measurement extensions, Metrion integration, JSON generator, and benchmark suite. Chapter 5 reports the empirical results for RQ1–RQ3. Chapter 6 discusses the main findings and implications, reflects on methodological decisions, states threats to validity, and outlines future work. Chapter 7 concludes the thesis.

Chapter 2

Related Work

This chapter summarises the related-work foundations from our previous study [11, Ch. 2] and extends them with literature that motivates the benchmark design used in this thesis. To keep the report self-contained, § 2.1 revisits four foundation areas: software energy measurement, energy-efficiency research for C# and .NET, JSON serialisation as a case study, and benchmarking methodology.

The remaining sections introduce the new related work for this thesis. § 2.2 reviews JSON workload characterisation, focusing on document properties such as size, nesting depth, width, content type, numeric values, and string characteristics. Finally, § 2.3 reviews environmental control in energy benchmarking, including both controlled benchmark environments and attribution-based measurement in noisy or shared systems.

2.1 Foundations from Prior Work

This section summarises the foundations carried forward from our previous study [11, Ch. 2]. These foundations motivate the choice of measurement instrument, runtime platform, case study, and benchmark framework used in the present thesis.

2.1.1 Energy Measurement

Software energy consumption can be measured using external wall-plug meters, on-chip counters such as Intel's Running Average Power Limit (RAPL), or predictive models that estimate energy from system metrics such as CPU utilisation. On Linux, RAPL is exposed through the kernel `powercap` interface and reports accumulated energy for processor power domains such as Package, DRAM, and Core. Khan et al. [12] validate RAPL against a wall-mounted meter and find it sufficiently accurate for software benchmarking, with negligible runtime overhead.

2.1.2 Energy Research for C# and .NET

Cross-language studies [3, 4, 5, 13] generally rank compiled languages as the most energy efficient and interpreted languages as the least energy efficient, with managed languages such as C# in between. They also report cases where energy does not follow execution time, making runtime alone an unreliable proxy for energy consumption. Van Kempen et al. [6] further caution that cross-language rankings often conflate language effects with implementation details such as JIT compilation, garbage collection, and library choice. This motivates a within-ecosystem focus.

Within .NET, prior work has examined energy at the level of language constructs [14] and concurrency primitives [15]. However, no peer-reviewed study compares competing third-party .NET libraries. Evidence from other ecosystems, including framework entropy in .NET [1], method-level measurement [16], Java I/O [17], and Python data analysis [18], shows that library choice can materially affect energy consumption. This leaves a gap for systematic library-level evidence in .NET.

2.1.3 JSON Serialisation as a Case Study

JSON is a widely used data format in web APIs, accounting for approximately 97,% of Cloudflare API requests in 2021 [7]. Comparative studies of .NET serialisers [8, 9] report substantial runtime and memory differences across libraries that implement the same functionality. Noureddine et al. [10] extend this observation to energy in the Java ecosystem, reporting large energy differences across JSON libraries and cases where energy diverges from execution time.

2.1.4 Benchmarking Methodology

Reliable microbenchmarking on managed runtimes requires controlling Just-In-Time Compilation (JIT) effects such as warm-up, dead-code elimination, and inlining, as well as garbage collection and statistical treatment of measurements [19, 20, 21]. BenchmarkDotNet [22] addresses many of these concerns by standardising warm-up, iteration sizing, and process isolation, and is a widely used microbenchmark orchestration framework in .NET [20].

These foundations remain central to the present thesis. We continue to use RAPL for energy measurement and BenchmarkDotNet for benchmark execution, while extending the previous work with a broader workload design, statistical analysis, and a focus on library-level energy behaviour in .NET.

2.2 JSON Workload Characterization

JSON is a tree-structured, text-based data-interchange format standardised in `RFC 8259` [23]. It is composed of primitive values, namely strings, numbers, Booleans, and `null`, combined into two composite types: objects, which are unordered sets of key–value pairs, and arrays, which are ordered sequences of values. In practice, JSON documents vary widely. Viotti and Kinderkhedia [24] analyse 480 documents from the *SchemaStore* repository and show that documents differ by orders of magnitude in size, dominant value type, value length, and structure, including nesting depth and per-level width. A systematic study of JSON library energy use should therefore account for document shape and content, not only total payload size [25, 26, 27]. Our previous work [11] shows a strong time–energy correlation for .NET JSON libraries, which makes workload characteristics that affect runtime relevant to energy measurement as well.

Viotti and Kinderkhedia [24] benchmark 13 binary encodings of JSON for space efficiency using documents from the *SchemaStore* repository [28]. They show that document structure and content type affect serialised size more than the specific values the document contains. They classify JSON documents along four axes: size, dominant content type, redundancy, and structure, where structure combines nesting depth and width. Although their study evaluates encoded size rather than runtime or energy, the same document characteristics are relevant for benchmark design.

Belloni et al. [25] analyse the effect of nesting depth on query runtime in JSON document stores. Using nested documents up to depth 16, they show that for queries requiring a full document scan, runtime grows approximately linearly with the nesting depth at which the queried element is located.

Dhalla [26] analyses the effect of structural width on JSON parsing time in a comparative study of built-in parsers across Java, Python, .NET Core, JavaScript, and PHP. The study shows that increasing the per-level key count from 10 to 100, at a fixed nesting depth of 20, increases parsing time for all five parsers.

Langdale and Lemire [27] analyse parsing time at the level of individual JSON token classes. They show that numeric parsing, especially floating-point parsing, takes more time than most other token types, with number-heavy documents spending roughly a third of total parsing time on numbers alone. UTF-8 validation adds little time for predominantly ASCII input, but becomes more relevant when non-ASCII characters are common. Escape-sequence handling also increases string parsing time.

Together, these studies show that JSON structure and content affect parsing, query runtime, and encoded size. This motivates treating Size, Depth, Width, String Length, Numeric Length by Type, and String Composition as explicit workload dimensions in the benchmark design.

2.3 Environmental Control in Energy Benchmarking

Energy measurements are highly sensitive to the execution environment. Energy consumption can vary substantially with processor frequency behaviour, C-states, thermal conditions, and concurrent system activity [29]. The literature provides two main strate-

gies for software energy measurement under these constraints. The first applies strict experimental controls to reduce environmental variability and make repeated measurements more stable. The second accepts noisy or shared execution environments and estimates per-process or per-container energy through attribution models that combine hardware energy readings with runtime information. § 2.3.1 reviews strict environmental control, while § 2.3.2 reviews attribution-based measurement.

2.3.1 Strict Environmental Control

Van Kempen et al. [6] critique prior comparative studies of programming-language energy efficiency, arguing that several reported rankings are driven by uncontrolled environmental factors and implementation details rather than intrinsic language properties. Through a causal analysis, they identify active core count, operating frequency, and memory activity as major contributors to power draw beyond execution time. They also show that variation in these factors can alter observed energy behaviour and even change benchmark rankings. Their methodology therefore limits each program to a single core, pins the CPU to a fixed frequency with Intel Turbo Boost disabled, and monitors memory activity using hardware performance counters.

Stoico et al. [30] adopt these recommendations in a study of the energy efficiency of code produced by eight Python implementations. They apply the same core controls as Van Kempen et al., including single-core execution and fixed CPU frequency, and additionally disable Intel Hyper-Threading at the BIOS level and deactivate non-essential background processes. They also monitor last-level-cache misses as a memory-activity proxy.

Ournani et al. [29] quantify the impact of several environmental controls on energy variance in systems benchmarking. Using workloads from the NAS Parallel Benchmarks [31] at idle (0%), medium (50%), and high (100%) CPU loads, they report variance reductions of up to 6× from disabling C-states at low load, 5× from disabling Hyper-Threading at medium load, and 30× from pinning execution to a single socket on multi-socket systems.

Khan et al. [12] examine the thermal sensitivity of RAPL energy readings on Haswell processors. They report a correlation coefficient of 0.93 between CPU package temperature and RAPL package power, with package power increasing by 10–12,% from 37,°C to 74,°C. They therefore recommend a warm-up period before measurement to stabilise package temperature.

2.3.2 Attribution-Based Measurement

Where strict environmental control is impractical, attribution-based tools estimate the share of system-level energy used by individual processes, threads, containers, or virtual machines. These tools typically combine RAPL energy readings with runtime information such as CPU time or Hardware Performance Counters (HPCs) [32].

DEEP-mon [33] attributes RAPL energy to individual threads and aggregates it to application containers. It tracks context-switch events in the Linux kernel, uses performance-counter data to account for thread activity, and applies an SMT-aware weighting when two hardware threads share a physical core. In its Docker-container evaluation, the monitoring agent adds less than 5% to benchmark runtime and 0.9–1.7% to total server power

consumption. The paper also discusses Kubernetes as a target for future power-aware scheduling, but the reported validation focuses on Docker workloads.

SmartWatts [34] attributes RAPL package and DRAM energy to processes, containers, and virtual machines through Linux control groups. It learns CPU and DRAM power models from HPCs and recalibrates them at runtime whenever the estimation error exceeds a configured threshold. The authors evaluate the tool using benchmark workloads such as `stress-ng` and the NAS Parallel Benchmarks [31], reporting an average estimation error below 3.5% at a 2,Hz sampling rate. The sensor itself consumes less than 0.2,W on the monitored host.

Scaphandre [35] is an open-source power-monitoring agent designed for production environments, including bare-metal hosts, virtual machines, and Kubernetes clusters. It reads RAPL energy counters through the Linux `powercap` interface and estimates per-process energy consumption from each process's share of CPU time. The collected metrics can be exported through backends such as Prometheus, Riemann, Warp10, and JSON, allowing integration with existing monitoring and observability systems.

EnergAt [36] targets multi-tenant environments where applications share a host with other workloads. It attributes RAPL package and DRAM energy at thread level by estimating each application's share of CPU and memory activity using per-thread CPU time and Non-Uniform Memory Access (NUMA) memory residence information. The authors report that EnergAt remains stable under noisy-neighbour conditions where competing workloads start or stop dynamically, while coarser attribution methods can overestimate energy usage by up to 46% or underestimate it by up to 93%.

Metrion [32] attributes RAPL package and DRAM energy to individual threads in multi-socket systems with Simultaneous Multithreading (SMT). It samples each thread's CPU activity directly in the Linux kernel and uses Hardware Performance Counters to model the combined effects of SMT, frequency scaling, and NUMA. Against ground-truth measurements, the authors report a mean absolute percentage error of 4.2% for CPU energy attribution on CPU-bound workloads and 16.1% for DRAM energy attribution on memory-bound workloads.

The literature provides methodologies for strict environmental control in energy benchmarking and a growing set of attribution tools for measurement in noisy or shared environments. The two strategies, however, have largely been pursued in isolation. To our knowledge, no prior study has applied both to the same workloads, leaving open whether energy-efficiency rankings established under strict control still hold in noisy, shared environments. This thesis addresses that gap by measuring the same JSON libraries under two conditions: direct RAPL readings in a strictly controlled environment, and attribution-based measurement in a noisy, shared environment. We then test whether the resulting library rankings agree.

The literature provides methods for reducing environmental variability in controlled energy benchmarks and a growing set of tools for attributing energy in noisy or shared environments. These two strategies have mostly been studied separately. To our knowledge, no prior study has applied both strategies to the same JSON-library workloads, leaving open whether library rankings measured under strict control also hold under noisy, shared execution. This thesis addresses that gap by measuring the same JSON libraries in two environments: direct RAPL measurement in a clean environment and Metrion-based attribution in a noisy environment.

Chapter 3

Experiment Methodology

This chapter describes the experimental method used to generate JSON workloads, execute benchmarks, collect energy and runtime measurements, and analyse the resulting data. Figure 3.1 summarises the experiment pipeline, with each part described in detail in the sections that follow.

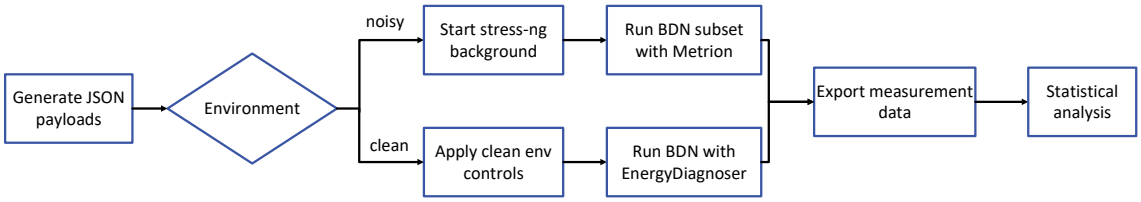


Figure 3.1: End-to-end experiment pipeline.

The pipeline starts with the generation of synthetic JSON workloads across six workload dimensions (§ 3.3.3). The workloads are then executed in two environments. The *clean* environment applies hardware and operating-system controls to reduce measurement variance (§ 3.2.2). The *noisy* environment introduces CPU and memory background stress using `stress-ng` (§ 3.2.3).

Benchmarks are executed with BenchmarkDotNet (BDN) (§ 3.1.1). In the clean environment, energy is measured with the RAPL-based Energy Diagnoser (§ 3.1.2). In the noisy environment, energy is measured with the Metrion Profiler for per-process energy attribution (§ 3.1.3). The Metrion Profiler is first validated against the Energy Diagnoser at 0% stress (§3.5). Due to project time constraints, only a subset of the benchmark suite is executed in the noisy environment.

Finally, the exported measurements are analysed using Shapiro–Wilk normality tests, Holm-adjusted Kruskal–Wallis tests, Cliff's δ, and ranking-based summaries (§ 3.4). These analyses evaluate whether library energy differences are statistically significant, how large those differences are, and how library rankings change across workload configurations.

3.1 Measurement Infrastructure

This section describes the measurement infrastructure used during benchmark execution. BDN [22] acts as the central execution framework and integrates the energy-measurement and runtime-diagnostics tools used throughout the experiment.

The primary energy measurement components are the RAPL-based diagnoser, originally implemented in [11], and the newly integrated Metrion-based profiler, which are used in the clean and noisy environments, respectively. In addition, a set of supporting diagnosers collects allocation, garbage-collection, and runtime metrics alongside the energy measurements.

3.1.1 BenchmarkDotNet

As discussed in our previous work [11, §3.2.1], BDN [22] addresses the main challenges of measuring managed-runtime performance and energy consumption, including process isolation, JIT warmup, garbage collection and dynamic iteration control.

Each benchmark case is executed through *Pilot*, *OverheadWarmup*, *ActualWarmup*, and *ActualWorkload* stages, with the *Result* stage aggregating the collected measurements into the final benchmark summary. The *ActualWorkload* stage is where the final benchmark iterations are executed, and the measurements that contribute to the reported results are recorded. To further isolate the measurements, each benchmark case is executed in a dedicated, high-priority process (the *dedicated process*). This staged execution helps reduce the effect of JIT compilation, startup overhead, and short-term runtime variation before the actual measurements are recorded. The full set of parameters for benchmark configuration is in Appendix A.1.

BDN [22] allows the implementation of custom diagnosers and profilers that can be used to perform additional analysis on measurements. A diagnoser observes the benchmark and contributes additional metrics (columns, warnings) to the summary output, while a profiler is essentially a diagnoser that performs heavier external instrumentation. In this work, we extend the implementation of the Energy Diagnoser [11] (§ 3.1.2), which uses raw RAPL readings, and implement the Metrion Profiler (§ 3.1.3), which integrates Metrion [32] to collect energy measurements.

3.1.2 Energy Measurement with RAPL

RAPL is an interface used to report the accumulated energy consumption of different power domains in a system. In the literature [12], the term `Power Domain` is commonly used, while the official Linux Kernel Power Capping Framework [37] uses the term `Power Zone`. For consistency, this report uses the term `Power Domain`.

RAPL supports different power domains depending on the CPU architecture [12]. Figure 3.2 shows all the energy domains that can be exposed by RAPL:

- **Package**: Energy consumed by the entire CPU socket, including cores, uncore components, LLC, and other integrated parts.

  - **Core**: The energy consumed by all CPU cores.
  - **Uncore**: Energy consumed by components outside the CPU cores, such as the integrated GPU and LLC.
- **DRAM**: Energy consumed by the system memory.
- **Psys**: Energy consumed by the entire SoC platform, including the CPU package, DRAM, and other components on the system-on-chip (e.g., integrated GPU, I/O controllers).

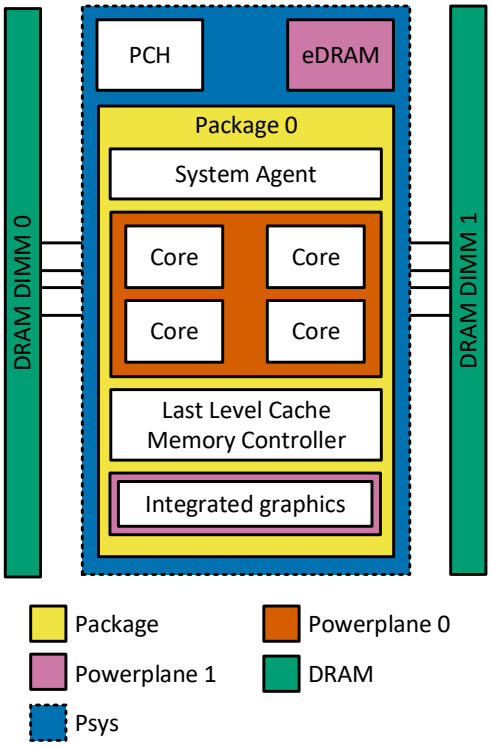


Figure 3.2: Power domains supported by RAPL [12].

The structure of `intel-rapl` on the system used in this experiment, as reported by the Linux Kernel Power Capping Framework [37], is presented in Table A.2 in the Appendix.

3.1.2.1 Integration into BenchmarkDotNet

In our previous work [11], we extended BDN with an Energy Diagnoser. The diagnoser was compatible only with single-socket systems and supported only Package and DRAM measurements. In this work, we extended the Energy Diagnoser to support the remaining domains and subdomains: Core, Uncore, and Psys. We also redesigned the implementation to support multi-socket systems in a generic way. In addition to energy measurements, we also added CPU temperature monitoring.

3.1.3 Energy Measurement with Metrion

This section presents the Metrion tool and its integration into BDN. The implementation serves as the foundation for investigating RQ3 in § 5.4.

3.1.3.1 Metrion

Metrion is described as a framework for accurate software energy measurement [32]. It can approximate the energy consumption of a process with a Mean Absolute Percentage Error of 4.2%. The tool is written in Python, runs through a CLI command, and supports two operating modes:

- **Monitor mode**. In this mode, the tool samples all running system processes and the energy consumed by the CPU package (with RAPL) every second. The measurements are stored in a database format compatible with SQLite[1] or DuckDB[2].
- **Analyse mode**. In this mode, the database generated during `Monitor mode` is queried to calculate the total amount of energy consumed by a specific process or group of processes within a specified time interval. The analysis results are exported in JSON format and, optionally, the raw measurements can also be exported in CSV format.

Metrion has a few limitations:

- During monitoring, it introduces additional system noise because it uses multiple threads to collect measurements.
- Based on our tests, Metrion currently supports only the CPU package Power Domain.

BDN already provides several profilers, such as the `PerfCollectProfiler`, which runs a `Perf`[38] process in the background. Since `Perf` is also a CLI-based tool, Metrion can be similarly integrated into BDN and exposed as a profiler.

3.1.3.2 Integration into BenchmarkDotNet

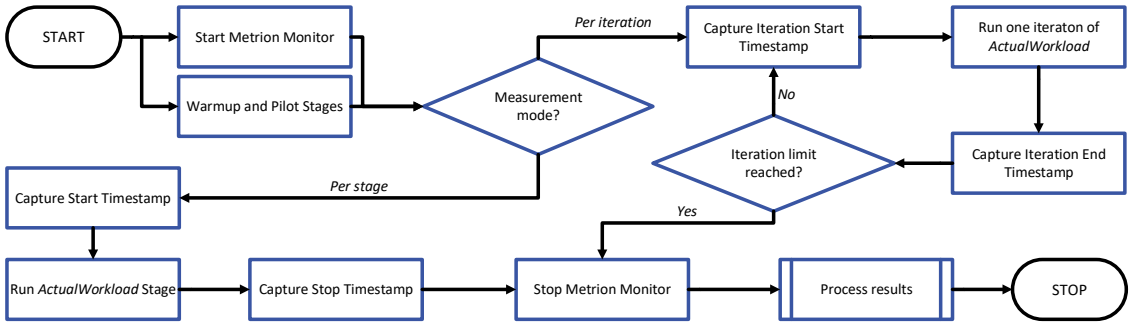


Figure 3.3: Flow of energy measurement coordination between the Metrion monitor and BDN's iteration lifecycle, supporting both Per iteration and Per stage capture modes.

[1] https://sqlite.org/
[2] https://duckdb.org/

The architecture overview of the Metrion integration within BDN is shown in Figure 3.3. Throughout this report, we refer to this integration as the Metrion Profiler. The figure illustrates the workflow executed for each benchmark case. First, the Metrion monitoring process starts to sample, in the background, the energy consumption of all running processes in the system. In parallel, the BDN *Warm-up* and *Pilot* stages are executed. The Metrion Profiler can be configured to extract the energy measurements in two different modes. Depending on mode, it will capture the following timestamps:

- **Per stage**: The timestamps are recorded immediately before *ActualWorkload* stage starts and immediately after it ends. Both timestamps will be used to extract the energy consumed during the entire *ActualWorkload* stage, including all iterations together.
- **Per iteration**: In this case, the timestamps are recorded before and after each iteration. All the timestamps will be used to extract the energy consumed by each iteration of the *ActualWorkload* stage separately.

In both modes, the Metrion process captures the same measurements (because it runs in background); the modes differ only in which timestamps are recorded and how the measurements are later extracted and aggregated during the *Process results* step.

After the benchmark execution finishes, a stop signal is sent to Metrion so all of its internal components can shut down gracefully. Finally, the database generated by Metrion is analysed to determine the amount of energy consumed by the benchmarked process within the recorded timestamps.

The calculation of *energy per iteration* and *energy per operation* depends on the operating mode of the Metrion Profiler. In Per iteration mode, the energy consumption is extracted separately for each iteration using Metrion *Analysis*. In Per stage mode, the energy consumption of the entire *ActualWorkload* stage is extracted in a single step using Metrion *Analysis* and then divided to obtain the *energy per iteration* and *energy per operation*.

As an alternative, Metrion *Analysis* can export the raw measurements, which can then be manually processed and summed to calculate the energy consumption. Figure 3.4 illustrates how the database can be analysed and how the raw measurements are processed and aggregated so that they correspond to the captured timestamps.

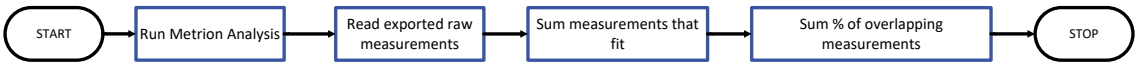


Figure 3.4: Metrion data analysis flow.

When calculating the energy, if a raw measurement only partially overlaps with a time interval, only the corresponding portion of that measurement is included in the final sum. The overlap handling is illustrated in Figure 3.5. For a specific time interval, both the start and end timestamps are known. In the scenario shown, the interval overlaps with three separate raw measurements. For each measurement (*m - 1*, *m* and *m + 1*), the overlap percentage is calculated, and the corresponding portion of the measured energy is attributed to the time interval (e.g. the time interval of an iteration).

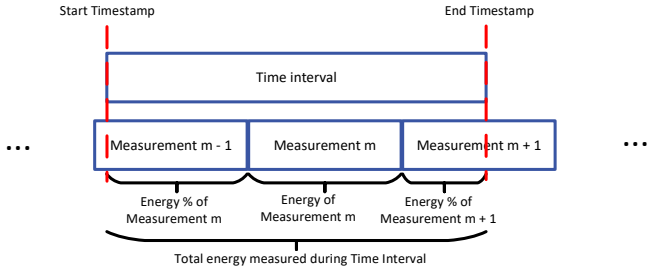


Figure 3.5: Metrion overlapping measurements handling.

3.1.4 Runtime Diagnostics

In addition to the Energy Diagnoser, three BDN diagnosers were considered to help explain the energy differences observed between libraries. The MemoryDiagnoser and the EventPipeProfiler were enabled during benchmark execution, while the DisassemblyDiagnoser was considered but not used in the final study. These tools collect allocation, runtime, and machine-code information alongside the energy measurements.

MemoryDiagnoser

The MemoryDiagnoser reports the number of bytes allocated per operation together with the number of Gen0, Gen1, and Gen2 garbage collections triggered during execution. These metrics help relate energy consumption to allocation pressure and garbage-collection activity, and highlight differences among the evaluated JSON libraries.

EventPipeProfiler

The EventPipeProfiler, configured for CPU sampling, records runtime call-stack samples during benchmark execution. The resulting traces identify which methods dominate execution time and help expose architectural differences between libraries, such as time spent in tokenisation, metadata lookup, or string conversion.

DisassemblyDiagnoser

The DisassemblyDiagnoser outputs the machine code generated by the RyuJIT [39] compiler for the benchmark methods. This is used to inspect low-level implementation details such as SIMD instruction usage, allocation elimination, and method inlining, allowing verification of library optimisations.

3.2 Experimental Environment

Section § 2.3 reviewed two approaches to software energy measurement. The first uses hardware and operating-system controls to reduce measurement variance and improve repeatability. The second accepts noisy or shared execution environments and estimates per-process energy consumption through attribution models.

This study evaluates both approaches on the same JSON workloads. The *clean* environment applies the hardware and operating-system controls identified in prior work and measures energy with the Energy Diagnoser (§ 3.1.2). The *noisy* environment introduces CPU and memory background stress using `stress-ng` and measures per-process energy with the Metrion Profiler (§ 3.1.3).

3.2.1 Hardware and Software Platform

The benchmarks were executed on a single workstation with a fixed software stack. The platform configuration is listed below:

- **OS:** Linux Ubuntu 24.04.4 LTS (Noble Numbat)
- **.NET SDK:** 10.0.107
  - .NET 10.0.7 (10.0.7, 10.0.726.21808), X64 RyuJIT x86-64-v3
- **BenchmarkDotNet**:
  - Forked from version v0.15.5 and extended for energy consumption analysis [40]
- **Metrion**: 0.1.0
- **JSON Libraries:**
  - System.Text.Json: 10.0.6
  - Newtonsoft.Json: 13.0.4
  - SpanJson: 4.2.1
  - Utf8Json: 1.3.7
- **PC specifications**:
  - CPU: Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz
  - RAM: 4x DIMM DDR4 Synchronous 2666 MHz (0.4 ns) 8GB (32GB total)
  - SSD: SAMSUNG MZVLB512HAJQ-00000 512GB (NVME)

The full hardware inventory, as produced by `lshw -short`, is given in Appendix B.1.

3.2.2 Clean Environment

In our previous work [11], environmental control relied mainly on BDN defaults and manual reduction of background activity. This study extends that setup with hardware and operating-system controls motivated by the literature reviewed in § 2.3, while still using BDN as the benchmark framework.

The clean environment applies six controls: core pinning, disabling Simultaneous Multithreading (SMT), disabling Turbo Boost, locking the CPU frequency to remove Dynamic

Voltage and Frequency Scaling (DVFS), reducing background operating-system activity, and disabling C-states.

3.2.2.1 Core Pinning

Benchmarks are pinned to physical core 2. Pinning prevents thread migration between cores, which would otherwise invalidate private cache state and introduce scheduling variability. Cores 0 and 1 are avoided because Linux commonly directs interrupt handling and kernel housekeeping activity to the first cores by default. Avoiding these cores reduces interference from background operating-system activity [41].

This is implemented using BDN's per-job affinity, set in the configuration file (§ A.1):

```
.WithAffinity((IntPtr)(1 << 2)) // pin to CPU 2
```

3.2.2.2 Disabling SMT

SMT is disabled to prevent multiple hardware threads from sharing the execution resources of the same physical core. Prior work reports that concurrent execution on SMT sibling threads can increase total energy consumption relative to single-threaded execution [32]. Disabling SMT also makes each physical core expose one logical CPU, so CPU 2 maps directly to physical core 2.

SMT is disabled by writing `off` to the Linux SMT control interface:

```
echo off > /sys/devices/system/cpu/smt/control
```

3.2.2.3 Disabling Turbo Boost

Turbo Boost allows modern Intel processors to temporarily raise the operating frequency of selected cores when thermal and power conditions permit, improving performance for short, demanding workloads [29]. Disabling Turbo Boost keeps frequency behaviour more stable across benchmark runs.

Turbo Boost is disabled by writing `1` to the Linux `intel_pstate` interface:

```
echo 1 > /sys/devices/system/cpu/intel_pstate/no_turbo
```

3.2.2.4 Locking CPU Frequency

CPU frequency is fixed at 3.6 GHz, which is the base frequency of the Intel Core i9-9900K used in the benchmark machine. This removes DVFS-induced variation from the measurements. With Turbo Boost disabled, both the lower and upper bounds of the `intel_pstate` driver are set to the base frequency, and the governor is set to `performance`.

Frequency is locked through the Linux `cpupower` interface:

```
cpupower frequency-set -g performance
cpupower frequency-set -d 3.6GHz -u 3.6GHz
```

3.2.2.5 OS and Background Services

The benchmark machine runs a fresh Ubuntu Server 24.04 LTS installation using the *Minimized* variant. This image excludes many convenience tools and the GUI, reducing the number of background services. All required runtime dependencies, including the .NET SDK, were installed after the base operating system.

In addition, periodic `systemd` timers are stopped at the start of each benchmark session. Some benchmark sessions run for up to 15 hours and may overlap with default maintenance windows. Stopping these timers prevents background tasks such as package cache updates, unattended upgrades, and filesystem maintenance from running during the measurements.

The stopped timers are:

```
systemctl stop
apt-daily.timer apt-daily-upgrade.timer
motd-news.timer dpkg-db-backup.timer
fstrim.timer e2scrub_all.timer
systemd-tmpfiles-clean.timer
```

3.2.2.6 Disabling C-states

C-states are processor sleep states that place unused cores into progressively deeper low-power modes. When a core enters a deeper C-state, waking it requires additional time and energy, which can introduce measurement variability across benchmark runs [29]. Disabling C-states keeps cores active and removes this source of variation.

C-states are disabled through the Linux `cpupower` interface:

```
cpupower idle-set -D 0
```

3.2.2.7 Environment Setup

All environment controls are applied and verified through automation scripts included in the project repository [42] under `scripts/clean-env/`:

- `clean-env-up.sh` applies the clean-environment configuration: SMT off, Turbo Boost off, CPU frequency locked at 3.6 GHz with the `performance` governor, C-states disabled, and maintenance timers stopped.
- `clean-env-check.sh` verifies that each control is active. It is used as a pre-flight check before benchmarking and as a post-run sanity check before results are accepted.
- `clean-env-down.sh` reverts the controls to the Ubuntu defaults, restoring the system to a state equivalent to a fresh boot.

3.2.3 Noisy Environment

In this context, a noisy environment refers to an operating system where multiple services are continuously running and keeping the CPU usage at a noticeable level. This creates background noise and better reflects how real-world servers behave, where several services are executed in parallel. The CPU settings are kept at their default values, meaning the configuration normally present after a system reboot.

By using a noisy environment, we aim to evaluate whether it is feasible to run energy benchmarks under such conditions and to what extent the measurements are affected. To obtain consistent measurements, we use our Metrion Profiler (§ 3.1.3) and we compare the results with those obtained in the clean environment presented above.

This environment is intended to be representative of a production system. Therefore, we use the default Ubuntu configuration, which can also be restored using the `clean-env-down.sh` script presented earlier in § 3.2.2.7.

The benchmarks are no longer pinned to specific CPU cores. Due to time constraints, only a subset of the Isolated Benchmarks is used: Width, Depth, and Size. The Metrion Profiler is used because it can measure energy consumption on a per-process basis. We perform five runs, each with a different level of CPU noise, while keeping 8 GB of memory allocated. This setup allows us to observe how the tool behaves under varying system loads.

3.2.3.1 Stress-ng

To create a noisy environment, we use `stress-ng` [43], a tool designed to stress different system components. It provides several types of stressors, including CPU, memory, and network stressors. In this experiment we focus only on CPU and memory stress. The following parameters are used:

- `-cpu 0`. Stresses all available CPU cores.
- `-cpu-load` (0%, 25%, 50%, 75%, 100%). Maintains an approximate CPU stress level.
- `-cpu-method all`. Iterates through all available CPU stress methods, such as `ackermann`, `apery`, and others.
- `-vm 1`. Starts a single worker that continuously writes to memory.
- `-vm-bytes 8G`. Limits the worker to 8 GB of memory.
- `-vm-keep -vm-hang 0`. The worker writes to the allocated memory once and keeps it allocated for the remainder of the run.

3.3 Benchmark Design

This section defines the benchmark configuration space. The experiments vary three factors: the JSON library, the operation, and the workload characteristics. The environments in §3.2 define where the benchmarks are executed; this section defines what is executed.

A *workload dimension* is a controllable characteristic of the generated JSON document, such as Depth, Width, or String Length. A *level* is one selected value of a dimension, such as D5, W20, or L500. A *workload configuration* is one concrete combination of workload levels. A *benchmark case* combines a workload configuration with one library and one operation.

The workload configurations are organised into two benchmark suites. The factorial benchmark suite varies several workload dimensions together to provide a broad overview. The isolation benchmark suite varies one workload dimension at a time while holding the others fixed, making each dimension easier to analyse separately.

3.3.1 Libraries Under Test

The library selection follows the rationale established in our previous work [11] and is summarised here for completeness:

- **`System.Text.Json (STJRefGen)`** — Microsoft's built-in JSON library and the default in ASP.NET Core. It discovers type metadata through reflection at first use and caches the result [44].
- **`System.Text.Json (STJSrcGen)`** — the same library configured with its Roslyn source generator. It emits per-type `JsonTypeInfo` metadata at compile time, reducing reliance on reflection at runtime [44].
- **`Newtonsoft.Json`** — the long-standing third-party JSON library that preceded `System.Text.Json` as the de facto .NET standard. It provides a broad feature set and flexible API [45].
- **`SpanJson`** — a performance-oriented JSON library that uses `Span` and stack-allocated buffers for low-allocation processing of UTF-8 and UTF-16 payloads [46].
- **`Utf8Json`** — a performance-oriented JSON library that operates directly on UTF-8 byte sequences while remaining compatible with text JSON [47].

The selected libraries cover the platform default (`System.Text.Json`), the de facto legacy standard (`Newtonsoft.Json`), and two performance-oriented alternatives (`SpanJson` and `Utf8Json`).

Including both `System.Text.Json` variants allows the effect of source generation to be evaluated separately from broader cross-library differences. The reflection-based configuration, `STJRefGen`, discovers type metadata at runtime. The source-generated configuration, `STJSrcGen`, produces metadata at compile time through a `Json-Serializer-Context`. In the source-generated configuration, serialisation uses fast-path generation where available, while deserialisation uses metadata-based generation, as fast-path deserialisation is not currently supported [44].

`Jil` was included in prior work but is excluded here for two reasons. First, the project shows no maintenance activity since `14/3/2019` [48]. Second, it is not compatible with

the .NET 10 runtime used in this study. Its reflection-based deserialiser initialisation fails due to an ambiguous method lookup, resulting in a runtime exception during deserialiser construction. The full stack trace is shown in Appendix B.2.

3.3.2 Operations

Each library is measured on two operations: *serialisation*, which converts an object graph to JSON, and *deserialisation*, which converts JSON to an object graph.

The benchmark data is represented as UTF-8 encoded `byte[]` values. For deserialisation, the benchmark input is a UTF-8 `byte[]` containing the generated JSON document. For serialisation, the library serialises the generated object graph, and the benchmark records the cost of producing UTF-8 JSON output where the library supports it.

The UTF-8 representation is used for two reasons. First, RFC 8259 [23] specifies UTF-8 for interoperable JSON exchange and notes that it is overwhelmingly used in real-world systems. Second, `System.Text.Json`, `SpanJson`, and `Utf8Json` expose APIs that operate directly on UTF-8 data. Using `byte[]` therefore evaluates these libraries on their byte-oriented execution paths instead of forcing an additional UTF-16 conversion.

`Newtonsoft.Json` is the exception. It does not expose a direct UTF-8 byte API. To keep the benchmark representation consistent across libraries, the `Newtonsoft.Json` benchmarks perform UTF-8 transcoding inside the timed region. For deserialisation, the benchmark calls `Encoding.UTF8.GetString` on the input `byte[]` before parsing. For serialisation, it calls `Encoding.UTF8.GetBytes` on the resulting `string`. The reported `Newtonsoft.Json` measurements therefore include this transcoding cost.

A trace-based estimate suggests that this overhead is workload-dependent. In representative EventPipe traces, the transcoding step appears as `UTF8Encoding.GetString` during deserialisation and `Encoding.GetBytes` during serialisation. Its sampled share is small for the W20 workload, approximately 1–2%, but approximately 15% for the larger long-string L10000 workload. These are profiler-derived estimates rather than separate benchmark measurements, but they show that the `Newtonsoft.Json` transcoding cost is negligible in some cases and substantial in string-heavy workloads.

3.3.3 Workload Dimensions and JSON Generation

This subsection defines the workload dimensions used in the benchmark suites and explains how the synthetic JSON documents are generated from those dimensions. The dimensions define what characteristics can vary, while the generator defines how those characteristics are realised as concrete JSON documents.

3.3.3.1 Dimensions

The workload dimensions are selected from the JSON characterisation literature reviewed in § 2.2. In summary, prior work identifies document size, content type, structural complexity, depth, width, and string characteristics as relevant sources of variation in JSON processing. Viotti and Kinderkhedia [24] identify size, dominant content type, structural

complexity, and redundancy in their SchemaStore-derived taxonomy. Belloni et al. [25] study depth as a controlled variable in a JSON document-store benchmark. Dhalla [26] studies width as a dimension distinct from depth in parser evaluation. Langdale and Lemire [27] show that parsing cost also varies with content type and string characteristics, including Unicode usage and escape sequences.

We additionally include string length as an explicit dimension. It is not isolated in the reviewed workload taxonomies, but our pilot measurements showed that library rankings could change between configurations that differed only in string length. We therefore include it as a controlled workload dimension.

The isolation benchmark suite evaluates six workload dimensions:

- **Size** — the number of top-level objects in the document.
- **Depth** — the nesting depth of the generated object structure.
- **Width** — the number of fields per object.
- **String Length** — the number of characters in each generated string value.
- **Numeric Length by Type** — the number of digits in generated integer and floating-point values.
- **String Composition** — the character composition of generated string values, varied through Unicode, Escape, and UnicodeEscape variants.

The factorial benchmark suite also varies **Content type**, defined as the dominant scalar value type in the document: textual strings, numeric values, or Boolean-like values (`true`, `false`, and `null`). Here, a scalar value means a value that is not an object or array.

3.3.3.2 Workload Generation

The workload generator must control each dimension independently. Real-world JSON datasets do not provide this control, because payload size, nesting depth, width, content type, and string composition are usually correlated. Existing JSON generators also tend to be schema-driven or random-data tools rather than benchmark generators that expose the exact dimensions needed for the factorial and isolation benchmark suites. We therefore developed a deterministic synthetic JSON generator for this study.

The generator exposes each workload dimension as a separate parameter. This allows one dimension to vary while the others remain fixed. Given the same seed and configuration, the generator produces the same JSON document and corresponding object graph across runs.

Let D denote the configured Depth and W the configured Width. Each generated document contains a single nested-object path. At every non-final nesting level, the object has W fields. The first $W - 1$ fields contain generated scalar values, and the final field contains the next nested object. At depth D, the final field contains `null`, which terminates the nested structure. Listing 3.1 shows the schematic structure.

```
{
  "Level_1_Key_0":      <value>,
  "Level_1_Key_1":      <value>,
  ...
  "Level_1_Key_{W-2}": <value>,
  "Level_1_Key_{W-1}": {
    "Level_2_Key_0":      <value>,
    ...
    "Level_2_Key_{W-2}": <value>,
    "Level_2_Key_{W-1}": {
      ...
      "Level_D_Key_0":      <value>,
      ...
      "Level_D_Key_{W-2}": <value>,
      "Level_D_Key_{W-1}": null
    }
  }
}
```

Listing 3.1: Schematic structure of generated JSON documents.

Each <*value*> is generated according to the configured content type, String Length, Numeric Length, and String Composition parameters. When Size is greater than one, the baseline object structure is replicated, and the resulting objects are wrapped in a top-level `Items` array. Implementation details are described in §4.3, and the full generator is available in the project repository [42].

3.3.4 Factorial and Isolation Benchmark Suites

The two benchmark suites differ in how workload levels are combined. The *factorial benchmark suite* combines selected levels of Depth, Width, and Content type. It is used to provide a broad overview of library behaviour across different document shapes. The *isolation benchmark suite* varies one workload dimension at a time while all other dimensions remain fixed at a baseline configuration. It is used for the detailed per-dimension analysis, where ranking and scaling changes can be related to a single workload characteristic.

3.3.4.1 Factorial Benchmark Suite

The factorial benchmark suite varies three workload dimensions: Depth, Width, and Content type. Combining the selected levels gives $4 \times 4 \times 3 = 48$ workload configurations, shown in Table 3.1. Each configuration is executed for both operations and all libraries.

Depth and Width are included because they describe the structural shape of the generated JSON document. Dhalla [26] describes nesting depth and per-level key count as the two main factors affecting JSON parser complexity. Content type is included following Viotti and Kinderkhedia [24], who group documents by the dominant value type carried by the document.

The remaining workload dimensions are not included in the factorial benchmark suite. Including all six dimensions in one full factorial grid would multiply the number of configurations and make the experiment too large for the available measurement time. Instead,

the factorial suite focuses on document shape and dominant content type, while String Length, Numeric Length by Type, and String Composition are evaluated in the isolation benchmark suite.

Table 3.1: Levels per dimension in the factorial benchmark suite. The suite contains $4 \times 4 \times 3 = 48$ workload configurations.

Dimension	Levels
Depth	D2, D5, D10, D20
Width	W5, W20, W50, W100
Content type	Textual (T), Numeric integer (N), Boolean (B)

The levels are chosen to cover small, baseline, and high-complexity documents without making the factorial suite too large. Depth and Width follow a sparse, approximately logarithmic progression. The upper bounds align with the ranges used by Dhalla [26], where Width varies up to 100 keys per object and deeply nested documents use a Depth of 20. The intermediate levels expose scaling changes between the lower and upper bounds.

To reduce the influence of raw payload size, individual scalar values are size-normalised across Content types. Short textual values, small integers, and Boolean literals are selected so that each occupies roughly five bytes in the final JSON representation. Without this normalisation, differences between factorial configurations would mainly reflect payload size rather than the combined effect of Depth, Width, and Content type.

3.3.4.2 Isolation Benchmark Suite

The isolation benchmark suite provides detailed measurements for each workload dimension. Each group varies one dimension across a set of levels while keeping all other dimensions fixed at the baseline configuration. This makes it possible to analyse how rankings, energy ratios, and per-unit energy change when a single workload characteristic changes.

Baseline configuration

Table 3.2 shows the baseline configuration used in the isolation benchmark suite. The baseline represents a small-to-medium textual JSON document with moderate nesting and width. It is a practical reference point rather than a statistical claim about the average JSON document.

Table 3.2: Baseline configuration used as the fixed reference for the isolation benchmark suite.

Dimension	Baseline value
Content type	Textual
String Composition	ASCII-only
String Length	L20 (20 characters)
Depth	D5
Width	W20
Size	C1 (one generated object)

Textual content was selected because Viotti and Kinderkhedia report that most SchemaStore documents are primarily textual [24]. The remaining values were chosen heuristically to provide a stable reference configuration. Existing workload studies describe distributions of JSON characteristics across real datasets, but do not define a single representative JSON structure [24], [49].

Workload levels

Table 3.3 lists the evaluated levels for each isolation group. The selected ranges are sparse by design: they cover small, baseline, and high-stress configurations without making the benchmark suite too large to execute repeatedly.

Table 3.3: Levels per workload dimension in the isolation benchmark suite.

Dimension	Levels
Size	C10, C100, C1K, C10K, C100K
Depth	D1, D2, D4, D8, D15, D25, D40
Width	W2, W5, W10, W20, W50, W100, W200
String Length	L5, L20, L50, L200, L500, L2000, L10000
Numeric Length by Type	I2, I3, I5, I7, I10; F2, F3, F5, F7, F10
String Composition	A0; U5, U10, U25, U50, U100; E5, E10, E25, E50, E100; UE5, UE10, UE25, UE50, UE100

Size. Size is varied by replicating the baseline object structure rather than by changing the structure itself. Each generated document contains a JSON array of N copies of the baseline object under a top-level `Items` field. The object count ranges from C10 to C100K, producing payloads from small API-like arrays to large batch-style documents. The levels follow a logarithmic progression to cover several orders of magnitude.

Depth. Depth ranges from D1 to D40. The lower levels represent flat or lightly nested documents, while the higher levels stress nested traversal and stack handling. The intermediate levels are chosen to expose non-linear changes between shallow and deeply nested structures.

Width. Width ranges from W2 to W200. Low Width levels represent narrow objects with few fields, while high Width levels represent wide objects that may stress property lookup, metadata handling, and field iteration.

String Length. String Length ranges from L5 to L10000. The lower levels represent short identifiers or labels, while the higher levels represent large text values. This dimension was included because pilot measurements showed that library rankings could change between otherwise identical workloads that differed only in string length.

Numeric Length by Type. Numeric Length is evaluated separately for Integer and Float values. Integer levels vary the number of digits in generated integer values, while Float levels vary the number of digits in generated floating-point values. This separates the effect of numeric length from the effect of numeric representation. Floating-point parsing is included because Langdale and Lemire [27] report that floating-point values are substantially more expensive to parse than integers.

String Composition. String Composition varies the character composition of generated string values. A0 is the shared ASCII-only baseline. The Unicode variant increases the density of non-ASCII UTF-8 characters, the Escape variant increases the density of simple JSON escape sequences, and the UnicodeEscape variant increases the density of `\uXXXX` sequences. Langdale and Lemire [27] show that ASCII input can benefit from fast parsing paths, while Unicode and escape sequences add validation and normalisation work. The density levels cover low, medium, and fully special-character strings.

3.4 Data Analysis

This section describes how benchmark measurements are processed and compared. The same analysis procedure is applied to the factorial and isolation benchmark suites (§3.3.4) so that library rankings and energy differences are interpreted consistently.

The analysis has two purposes. First, statistical tests check whether the observed energy differences between libraries are reliable across benchmark iterations. Second, ranking and ratio summaries describe the size and direction of those differences across workload configurations.

Methodologically, the analysis follows the statistical workflow used in software energy-efficiency experiments: normality assessment, significance testing, and effect-size analysis [50]. Because the energy samples rarely follow a normal distribution, the analysis uses non-parametric tests, following the decision framework in Wohlin et al. [51, Ch. 11] and the test composition used by Stoico et al. [30].

3.4.1 Normality Assessment

For each library and workload configuration, the per-iteration energy samples are tested for normality using the Shapiro–Wilk test at $\alpha = 0.05$. Here, normality means that the samples follow a normal distribution. Q–Q plots are used as a visual complement.

The test is applied separately to each library within each operation and workload configuration. In the factorial benchmark suite, this gives 480 library-configuration sample sets: 48 workload configurations, two operations, and five libraries. In the isolation benchmark suite, this gives 540 sample sets across 54 workload configurations, two operations, and five libraries. The shared ASCII baseline (A0) is counted once within each String Composition variant (Unicode, Escape, and UnicodeEscape), since each variant is analysed relative to its own baseline. These results determine whether parametric or non-parametric tests should be used for the subsequent comparisons.

3.4.2 Significance Testing

Within each operation and workload configuration, the five libraries are compared using the Kruskal–Wallis test at $\alpha = 0.05$. Kruskal–Wallis is the non-parametric counterpart to one-way ANOVA for comparing more than two groups. A significant result indicates that at least one library has a different energy distribution from the others within that

configuration.

Because many workload configurations are tested, raw p-values are adjusted using the Holm–Bonferroni procedure [52]. The correction is applied separately within each benchmark suite, so that the factorial and isolation results are controlled within their own analysis families.

3.4.3 Effect Sizes

Statistical significance does not show how large a difference is. Pairwise library differences are therefore also measured using Cliff's δ [53], a non-parametric effect-size measure for comparing two distributions.

Effect sizes are interpreted using the thresholds proposed by Romano et al. [54]: negligible ($|\delta| < 0.147$), small ($0.147 \leq |\delta| < 0.33$), medium ($0.33 \leq |\delta| < 0.474$), and large ($|\delta| \geq 0.474$).

Cliff's δ is used to assess whether one library's energy measurements are consistently lower than another's. The practical size of the energy gap is read separately from the ratio summaries described below. This distinction is important because two libraries can have consistently ordered measurements while still having a small ratio difference.

3.4.4 Ranking and Ratio Summaries

The Results chapter reports library rankings using mean energy per operation. For each operation and workload configuration, libraries are ranked from Rank 1 to Rank 5. Rank 1 denotes the lowest mean energy, and Rank 5 denotes the highest mean energy.

To show the size of the gap between libraries, each library's mean energy is divided by the lowest mean energy in the same operation and workload configuration. This produces the energy ratio used in the rank-and-ratio heatmaps. A value of $1.00\times$ means that the library has the lowest mean energy for that configuration. A value of $2.00\times$ means that the library uses twice as much energy as the lowest-energy library in that configuration.

For cross-dimension summaries, the per-level ranks are averaged within each workload group. These mean ranks summarise how often each library appears near the low-energy or high-energy end of the ranking. Mean rank does not show the size of the energy gap, so it is interpreted together with the energy-ratio figures.

3.4.5 Per-Unit Scaling Ratios

For the isolation benchmark suite, the analysis also reports how energy scales as a workload dimension increases. For each library and operation, mean energy is normalised by the relevant unit of the dimension: objects for Size, nesting levels for Depth, fields for Width, characters for String Length, digits for Numeric Length, and special characters for String Composition.

Adjacent per-unit ratios compare consecutive levels. For two adjacent levels a and b, the

ratio is calculated as:

$$\frac{E_b / U_b}{E_a / U_a}$$

where E is mean energy and U is the dimension unit at that level. A value of 1.00× means that energy per unit is unchanged between the two levels. A value above 1.00× means that energy per unit increases, while a value below 1.00× means that energy per unit decreases, so total energy increases less than proportionally to the dimension size. For String Composition, the A0 baseline is excluded from the per-special-character calculation because it contains no special characters.

The cross-dimension scaling overview uses the same idea, but compares the highest and lowest levels of each workload group. These endpoint ratios summarise the overall change in energy per unit across the dimension, while the adjacent-level heatmaps show where the changes occur.

3.5 Metrion Validation

The clean and noisy environments (§ 3.2) use different measurement instruments: the RAPL-based Energy Diagnoser in the clean environment, Metrion Profiler in the noisy environment. Any observed difference in library rankings between the two environments therefore combines two effects — the change in environment and the change in instrument — and cannot be attributed to either cleanly.

Hence we need to validate Metrion Profiler against Energy Diagnoser in the noisy environment with 0% stress. If Metrion Profiler reproduces Energy Diagnoser's results there, its measurements in the noisy environment (25% to 100% stress) are credible for the RQ3 ranking comparison.

3.5.1 Validation Design

The benchmarks used for validation are the Smoke Benchmarks set, which contains a total of 10 benchmarks: one serialisation and one deserialization operation for each library. This setup represents the minimum benchmark configuration required for the validation.

The validation compares two runs, both executed in the noisy environment with 0% stress:

- **Main run.** Smoke Benchmarks executed with the Energy Diagnoser.
- **Metrion validation run.** Smoke Benchmarks executed with the Metrion Profiler.

The Metrion Profiler validation run is executed five times to account for between-run variance. The main Energy Diagnoser run is also executed five times to ensure a fair comparison.

Energy is reported and compared at *per-operation* (μJ/op) granularity for both tools. BDN's *ActualWorkload*-stage iteration count is adaptive ($\approx$ 15 with the Energy Diagnoser, up to $\approx$ 100 with Metrion Profiler) and the invocations-per-iteration count is also adaptive

(chosen to hit a target iteration time), so per-stage and per-iteration totals are not directly comparable across libraries or tools.

3.5.2 Validation Activities

Within the Smoke Benchmarks the following are computed:

- **Ranking agreement (Spearman ρ).** Spearman rank correlation between the five-library orderings produced by the two tools. A one-number summary of whether the tools order the libraries the same way.
- **Magnitude agreement (ratios).** Per library, normalize the library's energy to that of the lowest-energy library in the configuration. Compare these ratios across the two tools to check whether the relative gaps between libraries are preserved.

Chapter 4

Implementation

This chapter describes the implementation of the experimental infrastructure used in this study. It covers four components. First, the Energy Diagnoser was extended to support additional RAPL domains and multi-socket reporting. Second, Metrion was integrated into BDN as an optional profiler for attribution-based measurements. Third, a deterministic synthetic JSON generator was implemented to produce the workload configurations defined in Chapter 4.3. Finally, the benchmark suite was implemented to execute the selected libraries on the generated documents under a shared BDN configuration.

4.1 BenchmarkDotNet Energy Diagnoser Extension

This section presents the modifications made to the Energy Diagnoser to support multiple-socket systems. In addition, support for all available RAPL domains and subdomains was added.

4.1.1 Extended domain support

To read RAPL domains, the previous work integrated a C# library called `CSharpRapl` [55]. We extended this library to support all available domains and subdomains. Listing 4.1 shows the method responsible for loading all supported domains. Each domain has a corresponding `Sensor` implementation. Lines 3 to 8 load all sensors into a dictionary.

```
public void LoadAllSensors()
{
    sensors.Add(SensorType.PackageSensor, new Sensor("package", new
        PackageAPI(), CollectionApproach.Difference));
    sensors.Add(SensorType.DramSensor, new Sensor("dram", new DramAPI(),
        CollectionApproach.Difference));
    sensors.Add(SensorType.UncoreSensor, new Sensor("uncore", new UncoreAPI(),
        CollectionApproach.Difference));
    sensors.Add(SensorType.CoreSensor, new Sensor("core", new CoreAPI(),
        CollectionApproach.Difference));
    sensors.Add(SensorType.PsysSensor, new Sensor("psys", new PsysAPI(),
        CollectionApproach.Difference));
    sensors.Add(SensorType.TemperatureSensor, new Sensor("temperature", new
        TempAPI(), CollectionApproach.Average));
}
```

Listing 4.1: Energy Diagnoser: loading the supported energy domains.

Listing 4.2 shows the implementation of the Psys sensor responsible for scanning the available domains. The function returns a list of tuples. The first value in the tuple represents the path where the energy values can be read, while the second value represents the maximum energy value before the counter overflows. When this limit is reached, the energy counter resets back to 0.

```
public override List<(string, double)> openRAPLFiles()
{
    string directoryName = this.GetRaplDir();
    List<(string, double)> raplFiles = new List<(string, double)>();

    int raplSocketId = 0;
    while (Directory.Exists(directoryName + "/intel-rapl:" + raplSocketId))
    {
        var dirName = directoryName + "/intel-rapl:" + raplSocketId;
        var content = File.ReadAllText(dirName + "/name").Trim();
        if (content.Equals("psys"))
        {
            double maxEnergyRange = double.Parse(File.ReadAllText(dirName +
                "/max_energy_range_uj"));
            raplFiles.Add((dirName + "/energy_uj", maxEnergyRange));
        }

        raplSocketId += 1;
    }

    return raplFiles;
}
```

Listing 4.2: Energy Diagnoser: PSys sensor implementation.

Line 3 initializes the RAPL directory. Line 7 starts scanning all directories corresponding to the sockets available on the system. Line 10 checks whether the current domain is a psys domain, and if so, its details are stored. The other sensor implementations follow the

same approach.

Listing C.1 from Appendix presents legacy code that was originally implemented in the `CSharpRapl` project [55], but was later removed in the previous work [11]. We reintroduced this functionality to observe how CPU temperature changes before and after each benchmark execution.

4.1.2 Multiple socket systems compatibility

All energy measurements are stored in a model that groups them by `SocketId`. Listing 4.3 shows the fields stored in an `EnergyMeasurement` object.

```
1  public class EnergyMeasurement
2  {
3      public int SocketId { get; }
4      public double PackageEnergy { get; }
5      public double DramEnergy { get; }
6      public double CoreEnergy { get; }
7      public double UncoreEnergy { get; }
8      public double PsysEnergy { get; }
9      public double AverageCpuTemperature { get; }
10
11     public EnergyMeasurement( ... ) { ... }
12     ...
13 }
```

Listing 4.3: Energy Diagnoser: energy measurement data model.

To display the measurements in the final BDN result table, each column name must be explicitly defined, as shown in Listing 4.4. The `{0}` placeholder in Line 4 is replaced with the socket index, allowing measurements from different sockets to be identified correctly. The same approach is used for all domains and subdomains.

```
1  public static class Column
2  {
3      ...
4      public const string PackageEnergyPerOp = "PkgE{0} (uJ/op)";
5      public const string PackageEnergyPerOpStdDev = "StdDev{0} (uJ/op)";
6      public const string PackageEnergyPerOpCvPct = "CV{0} (%)";
7      ...
8      public const string AverageTemperaturePerIter = "Avg Temp{0} (degC)";
9  }
```

Listing 4.4: Energy Diagnoser: report column definitions.

Listing 4.5 shows how a measurement column is exported. Since the implementation must follow the internal BDN export format, an adaptation is required. Line 2 defines the column name. Lines 3 to 7 concatenate the `PackageEnergy` values from all sockets into a single string, separated by `/`.

```
columns.Add(new MeasurementColumn(
    "Measurement_PackageEnergy",
    (summary, report, m) => string.Join(
        "/",
        m.EnergyMeasurements.Select(em =>
            em.PackageEnergy.ToString("0.##", summary.GetCultureInfo())
        )
    )
));
```

Listing 4.5: Energy Diagnoser: exporting the PackageEnergy column to CSV.

4.2 BenchmarkDotNet Metrion Integration

This section presents how we integrated Metrion within BDN. During the development and testing phases, we had to modify some aspects of the original design (§ 4.2.1). § 4.2.2 describes the functionality of the main implemented components, along with relevant source code examples.

4.2.1 Implementation decisions

Initially, as presented in the Design chapter (§ 3.1.3.2), the calculation of *energy per iteration* was based on the aggregated results provided by the Metrion *Analysis* mode. During testing, however, we observed that the *Analysis* mode cannot aggregate measurements with millisecond precision; its finest granularity is one second.

In Per iteration mode, each iteration lasts approximately one second, but the exact time interval varies and always includes additional milliseconds. As a result, the values computed by the *Analysis* mode did not accurately align with the iteration boundaries and therefore could not provide reliable energy measurements. To address this issue, we switched to an alternative approach for measuring energy by using the raw measurements. Specifically, we enabled the exportation of the raw data provided by Metrion *Analysis* mode and matched measurements using their timestamps.

A second limitation is that the *Analysis* mode appears to exclude partially overlapping measurements at the start and end of the selected time interval. To address this issue, we implemented a custom aggregation method that calculates the overlap percentage and includes the corresponding fraction of the edge raw measurements in the total energy value.

As an overview, we implemented custom methods to aggregate the raw measurements for both running modes (Per stage and Per iteration).

4.2.2 Profiler components

As mentioned before, Metrion is integrated as a profiler and implements the `IProfiler` interface. Listing 4.6 presents the model used to store the information required to calcu-

late energy consumption using Metrion. One instance of this model is created for each benchmark case and stored in a dictionary (Line 17).

The `ProcessId` field stores the id of the dedicated process. This identifier is later used by the Metrion Analyzer to filter the energy measurements related to the specific process.

The `TraceFile` field represents the database file where Metrion stores all collected measurements while monitoring the system. This file is saved in the BDN artifacts directory only when explicitly enabled in the configuration.

```
1  internal class EnergyInterval
2  {
3      public int ProcessId { get; set; }
4      public DateTime StartTimestamp { get; set; }
5      public DateTime EndTimestamp { get; set; }
6      public double TotalEnergyFromSummary { get; set; }
7      public double TotalEnergyFromRawMeasurements { get; set; }
8      public FileInfo TraceFile { get; set; }
9      public List<DateTime> IterationTimestamps { get; } = new();
10     public List<double> EnergyPerIteration { get; } = new();
11     public List<long> OperationsPerIteration { get; } = new();
12 }
13
14 public class MetrionEnergyProfiler : IProfiler
15 {
16     ...
17     private readonly Dictionary<BenchmarkCase, EnergyInterval> energyIntervals
           = new();
18     ...
19 }
```

Listing 4.6: Metrion Profiler: energy interval data model.

When Metrion runs in Per stage mode, the following fields are used:

- `StartTimestamp` and `EndTimestamp` represent the start and end timestamps of the *ActualWorkload* stage, which is the stage where BDN performs the final benchmark iterations.
- `TotalEnergyFromSummary` stores the total energy measured within the *ActualWorkload* stage interval, as calculated by the Metrion analyzer script.
- `TotalEnergyFromRawMeasurements` stores the total energy calculated directly from the raw measurements exported by Metrion. This calculation also includes the percentage of overlapping edge measurements.

When Metrion runs in Per iteration mode, the following fields are used:

- `IterationTimestamps` stores the timestamps marking the start and end of each iteration in the *ActualWorkload* stage.
- `EnergyPerIteration` stores the energy consumed per iteration, calculated from the raw measurements exported by Metrion after the trace database is analyzed.
- `OperationsPerIteration` stores the number of operations executed by BDN during

each iteration, representing how many times the benchmarked method was invoked. This information is required to calculate the energy consumed per operation.

An important part of the Metrion Profiler logic is shown in Listing 4.7. The listing handles the different signals received by the Metrion Profiler while BDN executes the benchmarks.

At Line 7, a background Metrion monitoring process is started before the benchmark execution begins. If the Metrion Profiler is configured to run in Per iteration mode (Line 8), an *Event Profiler* is also started (Line 9). The purpose of the *Event Profiler* is to capture the timestamps of each benchmark iteration (this functionality is explained further in Listing 4.9). After the dedicated process is started (Line 11), its identifier is captured and stored.

Before the *ActualWorkload* stage begins (Line 14), the start timestamp is recorded. Similarly, after the *ActualWorkload* stage finishes (Line 17), the end timestamp is captured.

Finally, after all benchmark stages are completed, the Metrion monitoring process is stopped (Line 21). If the *Event Profiler* was enabled, it is also stopped. The generated Metrion database is then analyzed to calculate the final energy measurements (Line 23).

```
1  energyIntervals.TryGetValue(parameters.BenchmarkCase, out var energyInterval);
2  ...
3  switch (signal)
4  {
5      case HostSignal.BeforeAnythingElse:
6          KillOrphanedMetrionProcesses(parameters);
7          StartMetrion(parameters);
8          if (config.MeasurePerIteration)
9              StartEventProfiling(parameters);
10         break;
11     case HostSignal.AfterProcessStart:
12         energyInterval.ProcessId = parameters.Process.Id;
13         break;
14     case HostSignal.BeforeActualRun:
15         energyInterval.StartTimestamp = DateTime.Now;
16         break;
17     case HostSignal.AfterActualRun:
18         energyInterval.EndTimestamp = DateTime.Now;
19         break;
20     case HostSignal.AfterAll:
21         StopMetrion(parameters);
22         if (config.MeasurePerIteration)
23             StopEventProfiling();
24         AnalyzeMetrionDatabase(parameters);
25         break;
26 }
```

Listing 4.7: Metrion Profiler: benchmark signal handling.

Listing 4.8 shows how the Metrion monitoring process is started. A `ProcessStartInfo` object is created to provide better control over the process configuration and execution (Lines 2–10).

```
1  var processStartInfo = new ProcessStartInfo
2  {
3      FileName = config.MetrionBinaryPath.FullName,
4      Arguments = $"monitor",
5      UseShellExecute = false,
6      RedirectStandardOutput = false,
7      CreateNoWindow = true,
8      WorkingDirectory = config.MetrionBinaryPath.Directory.FullName,
9  };
10 if (config.AffinityMask != null)
11 {
12     processStartInfo.FileName = "taskset";
13     processStartInfo.Arguments = $"{(long)config.AffinityMask:X}
           {config.MetrionBinaryPath.FullName} monitor";
14 }
15 ...
16 metrionProcess = Process.Start(processStartInfo);
```

Listing 4.8: Metrion Profiler: starting the Metrion monitoring process.

If a CPU affinity mask is explicitly configured (Line 11), the Metrion process is restricted to the CPU cores specified by the mask. On Linux, this is achieved using the `taskset` command (Line 13).

The extraction of iteration timestamps is shown in Listing 4.9. This method is called as a callback whenever BDN's Engine emits an EventSource signal. Lines 5 and 6 check whether the received signal corresponds to the start or end of an iteration in the *Actual-Workload* stage. If this condition is met, the timestamp is stored in the model introduced earlier (Line 8).

```
1  private void AddIterationTimestamp(BenchmarkCase benchmarkCase, TraceEvent
       traceEvent)
2  {
3      switch ((int)traceEvent.ID)
4      {
5          case EngineEventSource.WorkloadActualStartEventId:
6          case EngineEventSource.WorkloadActualStopEventId:
7              if (energyIntervals.TryGetValue(benchmarkCase, out var
                   energyInterval))
8                  energyInterval.IterationTimestamps.Add(traceEvent.TimeStamp);
9              break;
10     }
11 }
```

Listing 4.9: Metrion Profiler: extracting iteration timestamps from engine events.

During implementation testing on the remote machine used for the experiments, we observed that Metrion orphaned processes occasionally remained running after the benchmarks had completed. To prevent this issue, we implemented a method that scans the system for active Metrion processes before starting each benchmark case and terminates any that are found.

The method is shown in Listing 4.10. Line 4 retrieves all running Metrion processes from the system. For each detected process, a graceful stop signal is first sent (Line 9). If the process does not terminate within 2 seconds (Line 10), it is forcefully killed (Line 12).

```
1  private void KillOrphanedMetrionProcesses(DiagnoserActionParameters
       parameters)
2  {
3      ...
4      var orphans = Process.GetProcessesByName(metrionFileName);
5      foreach (var orphan in orphans)
6      {
7          try
8          {
9              libc.kill(orphan.Id, libc.Signals.SIGINT);
10             if (!orphan.WaitForExit(2000))
11             {
12                 orphan.Kill();
13                 orphan.WaitForExit(1000);
14             }
15         }
16         catch (Exception ex) {...}
17         finally {...}
18     }
19 }
```

Listing 4.10: Metrion Profiler: handling orphaned Metrion processes.

Listing 4.11 presents the method that calculates the overlapping energy value between an iteration and one measurement. Line 3 and 4 decide the overlapping interval edges. If the timestamps do not intersect, the method returns 0 (Line 6). Otherwise, the energy percent is calculated (Line 9) and the corresponding energy is returned (Line 10).

```
1  private static double CalculateOverlappingEnergy(DateTime mStart, DateTime
       mEnd, double energy, DateTime iterStart, DateTime iterEnd)
2  {
3      var overlapStart = mStart > iterStart ? mStart : iterStart;
4      var overlapEnd = mEnd < iterEnd ? mEnd : iterEnd;
5
6      if (overlapEnd <= overlapStart)
7          return 0;
8
9      var overlapFraction = (overlapEnd - overlapStart).TotalSeconds / (mEnd -
           mStart).TotalSeconds;
10     return energy * overlapFraction;
11 }
```

Listing 4.11: Metrion Profiler: handling overlapping measurements.

The remaining source code listings are exposed in Appendix C. Listing C.3 presents the Metrion Profiler's `Config` class, which contains all available configuration parameters. The method that starts analysing the database generated by Metrion and extracts the raw mea-

surements is presented in Listing C.2. The raw measurements parsing is presented in Listing C.6. The source code that analyse the raw measurements for Per iteration running mode is presented in Listing C.4, and for Per stage running mode is presented in Listing C.5.

4.3 JSON Generator Implementation

This section describes the synthetic JSON generator used to create the benchmark documents. The generator consists of three main components. `JsonGenConfig` stores the workload configuration, `JsonTreeBuilder` writes the object structure, and `ValueGenerator` writes the generated terminal values, referred to here as *leaf values*. For a fixed seed and configuration, the generator is deterministic. Each document is written once to disk and reused unchanged by every library in the benchmark suite

4.3.1 Configuration Parameters

Listing 4.12 shows the configuration record used by the generator. The parameters in Lines 3,4,9–14 expose the main workload dimensions: `NestingDepth` and `Width` control the object structure, `Count` controls the number of top-level objects used by the Size benchmark, `StringLength` and `IntegerDigits` control value length, and `RedundancyRatio` controls the fraction of duplicated leaf values. Lines 5–8 group the probabilistic mixes used for content type, string composition, numeric and bool representation. The fixed `Seed` in Line 15 makes the generator reproducible for a given configuration.

```
1  public record JsonGenConfig
2  {
3      public int NestingDepth { get; init; } = 4;
4      public int Width { get; init; } = 10;
5      public ContentMix ContentMix { get; init; } = new();// textual / numeric /
           boolean
6      public StringMix StringMix { get; init; } = new();// ascii / unicode /
           escape
7      public NumericMix NumericMix { get; init; } = new();// integer / float
8      public BoolMix BoolMix { get; init; } = new(); true / false / null
9      public double RedundancyRatio { get; init; } = 0.0; // fraction of
           duplicates
10     public int StringLength { get; init; } = 20;
11     public int IntegerDigits { get; init; } = 6;
12     public int FloatIntegerDigits { get; init; } = 5;
13     public int FloatDecimalPlaces { get; init; } = 2;
14     public int Count { get; init; } = 1; // top-level objects (size)
15     public int Seed { get; init; } = 42;
16 }
```

Listing 4.12: JSON generator workload configuration record.

4.3.2 Document Structure

`JsonTreeBuilder` produces the object-only chain structure used by the benchmarks. Listing 4.13 shows the recursive method that writes each object. At every level, the loop in Lines 9–15 writes `Width - 1` leaf-value properties. The final property is handled in Lines 18–23: it either recurses into the next object or, at the deepest level, writes a JSON `null` sentinel. This implements the Depth and Width structure defined in the experimental design.

The document is written directly through `Utf8JsonWriter` to a stream, avoiding construction of the full JSON document in memory before writing it to disk. When `Count > 1`, the same object chain is generated multiple times and wrapped in a top-level `Items` array. This is how the Size dimension increases the number of top-level objects while preserving the same baseline object structure.

```
1  private void WriteObject(Utf8JsonWriter writer, int currentDepth)
2  {
3      writer.WriteStartObject();
4
5      var isLeafLevel = currentDepth + 1 >= _config.NestingDepth;
6      var keyIndex = 0;
7
8      // (Width - 1) value fields
9      for (var i = 0; i < _config.Width - 1; i++)
10     {
11         writer.WritePropertyName($"key_{keyIndex++}");
12         _valueGenerator.WriteValue(writer);
13         _leafCount++;
14         _totalKeyCount++;
15     }
16
17     // Last field: nested object, or a null sentinel at the leaf level
18     writer.WritePropertyName($"key_{keyIndex}");
19     _totalKeyCount++;
20     if (isLeafLevel)
21         writer.WriteNullValue();
22     else
23         WriteObject(writer, currentDepth + 1);
24
25     writer.WriteEndObject();
26 }
```

Listing 4.13: JSON generator: recursive object-chain construction.

4.3.3 Leaf Value Generation

`ValueGenerator` produces the leaf values in the generated documents. Listing 4.14 shows the main dispatch method. Lines 4–8 implement redundancy by replaying a previously generated value with probability `RedundancyRatio`. If no value is replayed, Lines 10–16 select the next value type from the configured `ContentMix`. The selected writer is executed in Line 17 and, when redundancy is enabled, stored for possible reuse in Lines 19–20.

```
1  public void WriteValue(Utf8JsonWriter writer)
2  {
3      // Redundancy: replay a previous value with probability RedundancyRatio
4      if (_valuePool.Count > 0 && _random.NextDouble() <
           _config.RedundancyRatio)
5      {
6          _valuePool[_random.Next(_valuePool.Count)](writer);
7          return;
8      }
9
10     Action<Utf8JsonWriter> writeAction = PickContentType() switch
11     {
12         LeafType.String  => CreateStringWriter(),
13         LeafType.Number  => CreateNumberWriter(),
14         LeafType.Boolean => CreateBoolWriter(),
15         _ => throw new InvalidOperationException()
16     };
17     writeAction(writer);
18
19     if (_config.RedundancyRatio > 0)   // retain for replay only when
           redundancy is on
20         _valuePool.Add(writeAction);
21 }
```

Listing 4.14: JSON generator: leaf-value dispatch and redundancy pool.

String composition is applied at the character level. Listing 4.15 shows the string generation logic. The pure-ASCII case uses the direct path in Lines 5–6. Otherwise, the loop in Lines 10–17 assigns each character position according to `StringMix`, allowing controlled proportions of ASCII characters, non-ASCII Unicode characters, JSON escape sequences, and Unicode escape sequences. Numeric values are generated to the configured digit lengths. For floating-point values, the final decimal digit is forced to be non-zero, because the libraries emit the shortest round-trippable representation and may otherwise remove trailing zeros.

```
1  private string GenerateString()
2  {
3      var length = _config.StringLength;
4
5      if (_stringAsciiThreshold >= 1.0)         // fast path: pure-ASCII
           baseline
6          return GenerateAsciiString(length);
7
8      // Each character position is independently assigned a type from StringMix
9      var sb = new StringBuilder(length);
10     for (var i = 0; i < length; i++)
11     {
12         var roll = _random.NextDouble();
13         if      (roll < _stringAsciiThreshold)   AppendAsciiChar(sb);
14         else if (roll < _stringUnicodeThreshold) AppendUnicodeChar(sb);
15         else if (roll < _stringEscapeThreshold)  AppendEscapeSequence(sb);
16         else
               AppendUnicodeEscapeSequence(sb);
17     }
18     return sb.ToString();
19 }
```

Listing 4.15: JSON generator: per-character string composition.

4.4 Benchmark Suite Implementation

The benchmark suite executes serialisation and deserialisation for all selected libraries across the factorial and isolation workload configurations. Generated JSON documents are loaded as UTF-8 `byte[]` values before measurement. For serialisation benchmarks, the corresponding object graph is also prepared during setup so that the measured method contains only the library operation itself.

4.4.1 Benchmark Configuration

All benchmarks use the shared BDN configuration shown in Listing 4.16. The job fixes a one-second iteration target in Line 7, allowing BDN to choose the invocation count needed to fill a stable measurement window. Line 8 disables outlier removal so the measurement structure remains unchanged between configurations, and Line 9 pins execution to logical CPU 2, which corresponds to physical core 2 in the clean environment because SMT is disabled.

The configuration enables the extended Energy Diagnoser in Line 14, the `MemoryDiagnoser` in Line 16, and the `EventPipeProfiler` in Lines 17–18. The Metrion Profiler shown in Line 15 is enabled instead for the noisy-environment configuration. Benchmarks are grouped by category in Line 12 so rankings and ratios can be computed within each operation and workload level. Raw per-iteration measurements are exported to CSV through Line 22.

```
1  public class BenchConfig : ManualConfig
2  {
3      public BenchConfig()
4      {
5          AddJob(Job.Default
6              .WithId("Energy")
7              .WithIterationTime(TimeInterval.Second)  // 1 s measurement window
8              .WithOutlierMode(OutlierMode.DontRemove)
9              .WithAffinity(1 << 2));                  // pin to logical CPU 2
10
11         WithArtifactsPath(SerializationHelper.BenchmarkArtifactPath());
12         AddLogicalGroupRules(BenchmarkLogicalGroupRule.ByCategory);
13
14         AddDiagnoser(EnergyDiagnoser.Default);
15         //AddDiagnoser(MetrionEnergyProfiler.Default);  // enabled in the
               noisy-env config
16         AddDiagnoser(MemoryDiagnoser.Default);
17         AddDiagnoser(new EventPipeProfiler(EventPipeProfile.CpuSampling,
18                                            performExtraBenchmarksRun: true));
19
20         AddColumn(StatisticColumn.Iterations);
21         AddColumn(new InvocationCountColumn());
22         AddExporter(CsvMeasurementsExporter.Default);
23     }
24 }
```

Listing 4.16: Shared BDN configuration.

4.4.2 Workload Configuration and Data Generation

Each isolation benchmark is defined by a small configuration class. The shared base class in Listing 4.17 stores the baseline configuration used across the isolation benchmarks. Lines 3–10 define the baseline values: depth 5, width 20, textual ASCII content, no redundancy, and a fixed seed. Lines 13–18 expose these values as a `BaseConfig` record. Lines 23–32 generate one JSON file per level in the benchmark's `TestData` directory.

```
1  public abstract class IsolationBaseConfig
2  {
3      protected const int    BaseDepth = 5;
4      protected const int    BaseWidth = 20;
5      protected const double BaseRedundancy = 0.0;
6      protected const int    BaseSeed  = 42;
7      protected static readonly ContentMix BaseContent =
8          new() { Textual = 1.0, Numeric = 0.0, Boolean = 0.0 };
9      protected static readonly StringMix  BaseStringMix =
10         new() { Ascii = 1.0, Unicode = 0.0, Escape = 0.0 };
11
12     // All-baseline config; sweeps override one property with a 'with'
           expression
13     protected static JsonGenConfig BaseConfig => new()
14     {
15         NestingDepth = BaseDepth, Width = BaseWidth,
16         ContentMix = BaseContent, StringMix = BaseStringMix,
17         RedundancyRatio = BaseRedundancy, Seed = BaseSeed,
18     };
19
20     public abstract IEnumerable<(string Id, JsonGenConfig Config)> GetAll();
21     protected abstract string SubDir { get; }
22
23     public void GenerateAll(string? outputDir = null)
24     {
25         outputDir ??= SerializationHelper.TestDataPath(SubDir);
26         Directory.CreateDirectory(outputDir);
27         foreach (var (id, config) in GetAll())
28         {
29             using var fs = File.Create(Path.Combine(outputDir, $"{id}.json"));
30             new JsonTreeBuilder(config).Generate(fs);
31         }
32     }
33 }
```

Listing 4.17: Isolation benchmark baseline and file generation.

Concrete benchmarks inherit the baseline configuration and override their specific configuration property while leaving the rest at baseline. Listing 4.18 shows this pattern for Depth. Lines 3–5 define the evaluated Depth levels, and Lines 11–12 produce the corresponding workload configurations by replacing only `NestingDepth`. This directly implements the isolation design described in §3.3.4.2.

```
public class DepthIsolationConfig : IsolationBaseConfig
{
    private static readonly (string Label, int Value)[] Depths =
        [("D1", 1), ("D2", 2), ("D4", 4), ("D8", 8),
         ("D15", 15), ("D25", 25), ("D40", 40)];

    protected override string SubDir => "IsoDepth";

    public override IEnumerable<(string Id, JsonGenConfig Config)> GetAll()
    {
        foreach (var (label, depth) in Depths)
            yield return (label, BaseConfig with { NestingDepth = depth });
    }
}
```

Listing 4.18: Depth benchmark configuration, varying only `NestingDepth`.

4.4.3 Data Model and Source Generation

The generated documents are deserialised into a family of generic `Node` classes that mirror the generated chain structure. Listing 4.19 shows `Node20` as an example. Lines 3–5 represent the leaf-value properties, while Line 6 stores the nested child node. Parameterising the model over `T` allows the same structure to represent string, integer, float and bool workloads.

```
public class Node20<T>
{
    public T key_0 { get; set; }
    public T key_1 { get; set; }
    // key_2 ... key_18 : all of type T
    public Node20<T>? key_19 { get; set; }   // last key holds the nested node
}
```

Listing 4.19: Generic chain model and the source-generation context.

The `STJSrcGen` variant requires compile-time metadata. Listing 4.20 shows the `IsolationJsonContext`, which registers the concrete benchmark model types, annotated in Lines 2–12, with `[JsonSerializable]`. The generated `JsonTypeInfo` objects are passed explicitly to `System.Text.Json` in the source-generated benchmarks. The reflection-based variant, `STJRefGen`, uses the same model types but relies on runtime metadata discovery.

```
1
2  [JsonSourceGenerationOptions(GenerationMode =
       JsonSourceGenerationMode.Default)]
3  [JsonSerializable(typeof(Node2<string>))]
4  [JsonSerializable(typeof(Node5<string>))]
5  [JsonSerializable(typeof(Node10<string>))]
6  [JsonSerializable(typeof(Node20<string>))]
7  [JsonSerializable(typeof(Node20<double>))]
8  [JsonSerializable(typeof(Node20<int>))]
9  [JsonSerializable(typeof(Node50<string>))]
10 [JsonSerializable(typeof(Node100<string>))]
11 [JsonSerializable(typeof(Node200<string>))]
12 [JsonSerializable(typeof(ItemsWrapper<Node20<string>>))]
13 public partial class IsolationJsonContext : JsonSerializerContext { }
14
```

Listing 4.20: Source-generation context.

4.4.4 Per-Library Operations

Listing 4.21 shows the benchmark methods for one Depth level. The same structure is repeated for the remaining levels and dimensions. Lines 7–12 show the setup phase, where each generated document is loaded as a `byte[]` and deserialised once into the corresponding object graph. This ensures that the measured deserialisation methods start from an already-loaded byte buffer, while the measured serialisation methods start from an already-materialised object.

Lines 16–30 show the deserialisation methods for the five libraries, and Lines 32–43 show the corresponding serialisation methods. Each library is invoked through its idiomatic UTF-8 path where available. `STJRefGen` on Lines 17–18 and 33 uses `JsonSerializer` with reflection-based metadata, while `STJSrcGen` on Lines 20–21 and 35–36 passes the generated `JsonTypeInfo`. `SpanJson` and `Utf8Json` use their native UTF-8 generic APIs. `Newtonsoft.Json`, on Lines 23–24 and 38–39, does not expose an equivalent UTF-8 byte API, so its deserialisation path converts bytes to a string before parsing, and its serialisation path converts the output string back to bytes. Its reported measurements therefore include this transcoding step.

```
[Config(typeof(BenchConfig))]
public class DepthIsolationByteBench
{
    private byte[] _d1_b = null!; private Node20<string> _d1 = null!;
    // ... _d2 ... _d40 ...

    [GlobalSetup]
    public void Setup()
    {
        _d1_b = Load("D1"); _d1 =
            JsonSerializer.Deserialize<Node20<string>>(_d1_b)!;
        // ... D2 ... D40 loaded and pre-deserialised the same way ...
    }
    private static byte[] Load(string id)
        => File.ReadAllBytes(SerializationHelper.TestDataFile("IsoDepth",
            $"{id}.json"));

    [Benchmark, BenchmarkCategory("Deserialize-D1")]
    public Node20<string> STJRefGen_Deser_D1()
        => JsonSerializer.Deserialize<Node20<string>>(_d1_b)!;
    [Benchmark, BenchmarkCategory("Deserialize-D1")]
    public Node20<string> STJSrcGen_Deser_D1()
        => JsonSerializer.Deserialize(_d1_b,
            IsolationJsonContext.Default.Node20String)!;
    [Benchmark, BenchmarkCategory("Deserialize-D1")]
    public Node20<string> Newtonsoft_Deser_D1()
        =>
            JsonConvert.DeserializeObject<Node20<string>>(Encoding.UTF8.GetString(_d1_b))!;
    [Benchmark, BenchmarkCategory("Deserialize-D1")]
    public Node20<string> SpanJson_Deser_D1()
        =>
            SpanJson.JsonSerializer.Generic.Utf8.Deserialize<Node20<string>>(_d1_b)!;
    [Benchmark, BenchmarkCategory("Deserialize-D1")]
    public Node20<string> Utf8Json_Deser_D1()
        => Utf8Json.JsonSerializer.Deserialize<Node20<string>>(_d1_b)!;

    [Benchmark, BenchmarkCategory("Serialize-D1")]
    public byte[] STJRefGen_Ser_D1() =>
        JsonSerializer.SerializeToUtf8Bytes(_d1);
    [Benchmark, BenchmarkCategory("Serialize-D1")]
    public byte[] STJSrcGen_Ser_D1()
        => JsonSerializer.SerializeToUtf8Bytes(_d1,
            IsolationJsonContext.Default.Node20String);
    [Benchmark, BenchmarkCategory("Serialize-D1")]
    public byte[] Newtonsoft_Ser_D1()
        => Encoding.UTF8.GetBytes(JsonConvert.SerializeObject(_d1));
    [Benchmark, BenchmarkCategory("Serialize-D1")]
    public byte[] SpanJson_Ser_D1() =>
        SpanJson.JsonSerializer.Generic.Utf8.Serialize(_d1);
    [Benchmark, BenchmarkCategory("Serialize-D1")]
    public byte[] Utf8Json_Ser_D1() => Utf8Json.JsonSerializer.Serialize(_d1);

    // D2 ... D40 repeat the same ten methods.
}
```

Listing 4.21: Per-library serialisation and deserialisation operations for Depth level D1.

Chapter 5

Results

This chapter reports the empirical findings of the study. §5.1 establishes an overview of the factorial benchmarks §3.3.4.1, which vary Depth, Width, and Content together across all 48 combinations. §5.2 presents per-dimension rankings from the six isolation benchmarks §3.3.4.2, each varying a single dimension while holding the others at a fixed baseline (RQ1). §5.3 explains the observed rankings through architectural and runtime evidence (RQ2). §5.4 reports the Metrion Profiler validation and environmental-effects analysis (RQ3).

All reported energy values combine the Package and DRAM RAPL domains and are expressed in microjoules per operation (μJ/op). Because most library energy distributions fail the Shapiro–Wilk normality test, comparisons use the non-parametric tests described in §3.4. The Holm-adjusted Kruskal–Wallis test determines whether libraries differ within each workload configuration. Pairwise Cliff's δ then measures how consistently one library's energy distribution falls below another's, using the Romano et al. [54] tiers: negligible, small, medium, and large. This separates dependable rankings from near-ties. The practical size of each difference is read separately from the ratio-to-best figures.

The full set of figures, tables, and statistical outputs exceeds the appendix scope and is available in the project repository [42]. Plots and CSV tables are under `analysis/figures/`, split into `isolation/` and `factorial/` subfolders, each with its own `stats/` and `tables/` layout. Within `stats/`, the per-cell Shapiro–Wilk, Kruskal–Wallis, and Cliff's δ tables sit under `normality/`, `kruskal_wallis/`, and `cliffs_delta/`; for the isolation benchmarks, the pooled cross-dimension summaries are under `isolation/_aggregate/stats/`. Raw per-iteration BDN measurements are stored under `BenchmarkArtifacts/results/`. The generated `System.Text.Json` source-generation files are available under `JsonBench/Generated/`. Representative EventPipe/Speedscope traces are included under `BenchmarkArtifacts/` for the minimum and maximum Size isolation workloads for each library; the complete trace set was not included because of storage size. Inline figures and tables use filenames that match the corresponding repository paths.

5.1 Factorial Benchmark Overview

This section provides a descriptive overview of the factorial benchmarks: the $4 \times 4 \times 3 = 48$ workloads per operation, formed by 4 Depth levels, 4 Width levels, and 3 Content types (§3.3.4.1). It establishes the libraries' overall standing before the detailed per-dimension analysis in §5.2, focusing on broad patterns and visible ranking shifts rather than isolated workload effects. The analysis proceeds in two parts: rank composition (§5.1.1) reports how often each library is placed at each rank, and library positioning (§5.1.2) reports each library's energy relative to the lowest-energy library in each workload.

As a statistical check, we first apply the tests from §3.4. Shapiro–Wilk rejects normality in 455/480 per-library measurement groups (94.8%), so non-parametric tests are used. The Holm-adjusted Kruskal–Wallis test finds significant differences between libraries in all 96 workload configurations. Pairwise Cliff's δ is *large* in 958/960 comparisons (99.8%). The 2 below-large cases are near-ties, both in deserialisation comparisons involving `STJSrcGen`.

Table 5.1 summarises per-library energy across the grid using the median, minimum, and maximum over the 48 workloads per operation. Serialisation consistently uses less energy than deserialisation. Overall, `SpanJson` uses the least energy in both operations, while `Newtonsoft` uses the most energy. The two `System.Text.Json` variants are almost identical at the median during deserialisation (620.8 vs 627.2 μJ/op), but diverge during serialisation, where `STJSrcGen` uses less energy than `STJRefGen` (143.5 vs 240.4 μJ/op). This split is also visible in the rank and ratio analyses below.

Table 5.1: Per-library energy summary across the factorial benchmark. Median, minimum, and maximum are computed over the 48 workloads for each library; values are in μJ/op. Median cells are shaded separately within each operation block, from lower to higher median energy.

Library	Operation	Median	Min	Max
SpanJson	Deserialise	252.4	8.6	6,174.4
Utf8Json	Deserialise	276.6	12.8	4,615.6
STJSrcGen	Deserialise	620.8	27.8	12,166.8
STJRefGen	Deserialise	627.2	29.0	11,481.4
Newtonsoft	Deserialise	1,118.9	55.2	12,447.8
SpanJson	Serialise	98.7	3.3	2,193.8
STJSrcGen	Serialise	143.5	9.6	2,219.1
Utf8Json	Serialise	147.9	6.2	2,649.2
STJRefGen	Serialise	240.4	14.6	3,791.5
Newtonsoft	Serialise	595.4	33.6	7,026.6

5.1.1 Rank Composition

Figure 5.1 summarises how often each library appears at each rank across the 48 workloads per operation. For each workload, libraries are ranked from 1 (lowest energy consumption) to 5 (highest energy consumption), and the stacked bars show the distribution of ranks for each library.

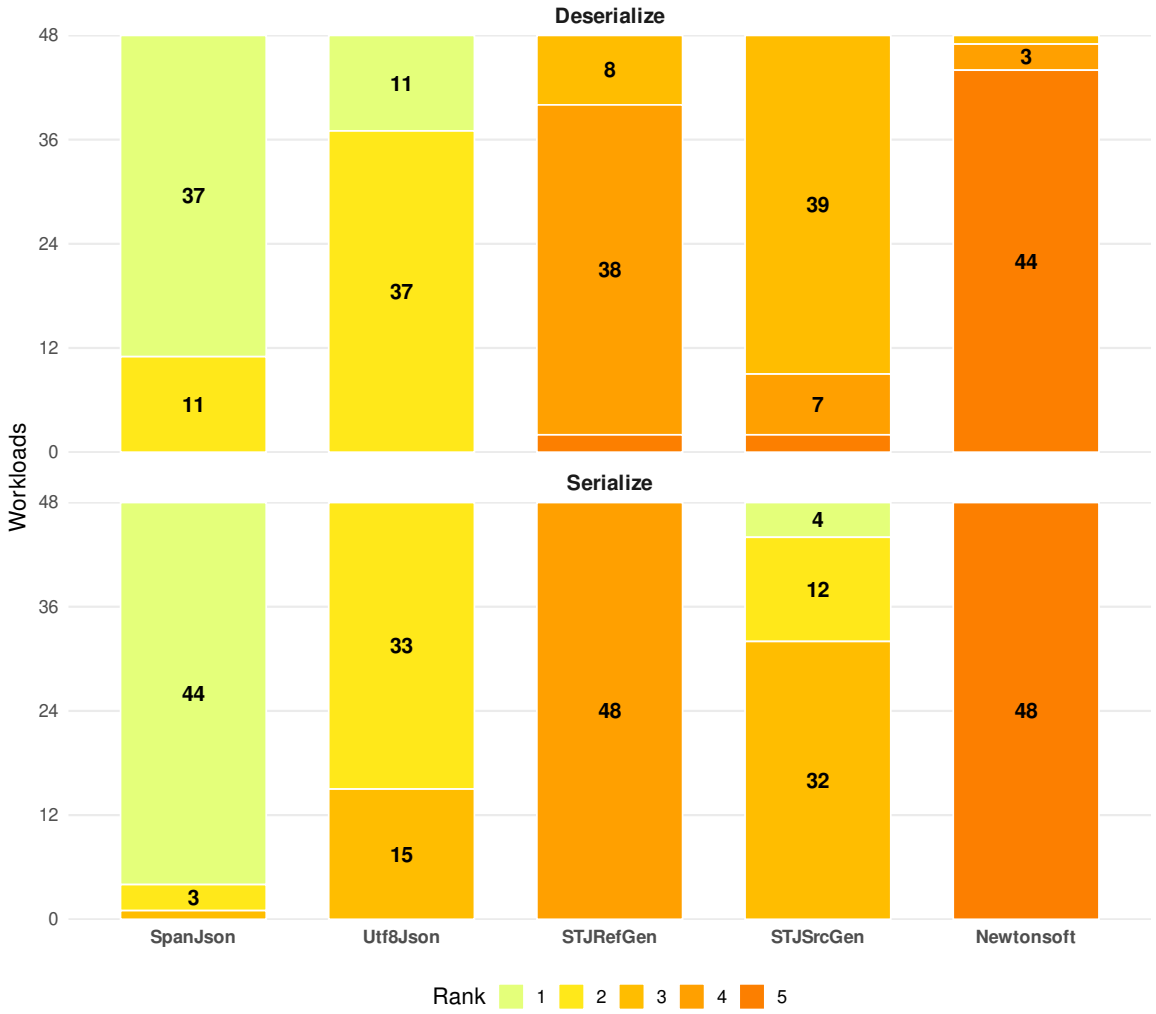


Figure 5.1: Library rank composition across the 48 workloads, shown per operation. Each bar shows how many of the 48 workloads the library placed at Rank 1 (lowest energy) through Rank 5 (highest energy).

Two broad patterns are visible. First, `SpanJson` appears most often at Rank 1. It has the lowest energy consumption on 37 of 48 deserialisation workloads and 44 of 48 serialisation workloads, and is never below Rank 2 during deserialisation. `Utf8Json` is the next lowest-energy library in deserialisation: it appears at Rank 1 on the remaining 11 deserialisation workloads and otherwise usually ranks directly after `SpanJson`.

Second, the difference between the source-generated and reflection-based `System.Text.Json` variants appears mainly during serialisation. `STJRefGen` is Rank 4 on all 48 serialisation workloads, while `STJSrcGen` usually is Rank 3 and reaches Rank 1 in 4 workloads. During deserialisation, the two variants behave almost identically, occupying the lower-middle ranks across most workloads. `Newtonsoft` is consistently the highest-energy library, at Rank 5 on nearly all workloads in both operations.

The deserialisation workloads where `Utf8Json` appears at Rank 1 are also highly clustered. They account for 11 of the 12 W100 configurations, spanning all evaluated depths and content types. The only exception is the D20/W100/Boolean workload, where `SpanJson` remains at Rank 1.

5.1.2 Library Positioning

Figure 5.2 reports each library's energy consumption relative to the library with the lowest energy consumption in the same workload. Each point represents one of the 48 workloads. A value of 1.00× means that the library has the lowest energy consumption for that workload. The boxplots summarise the median and interquartile range for each library.

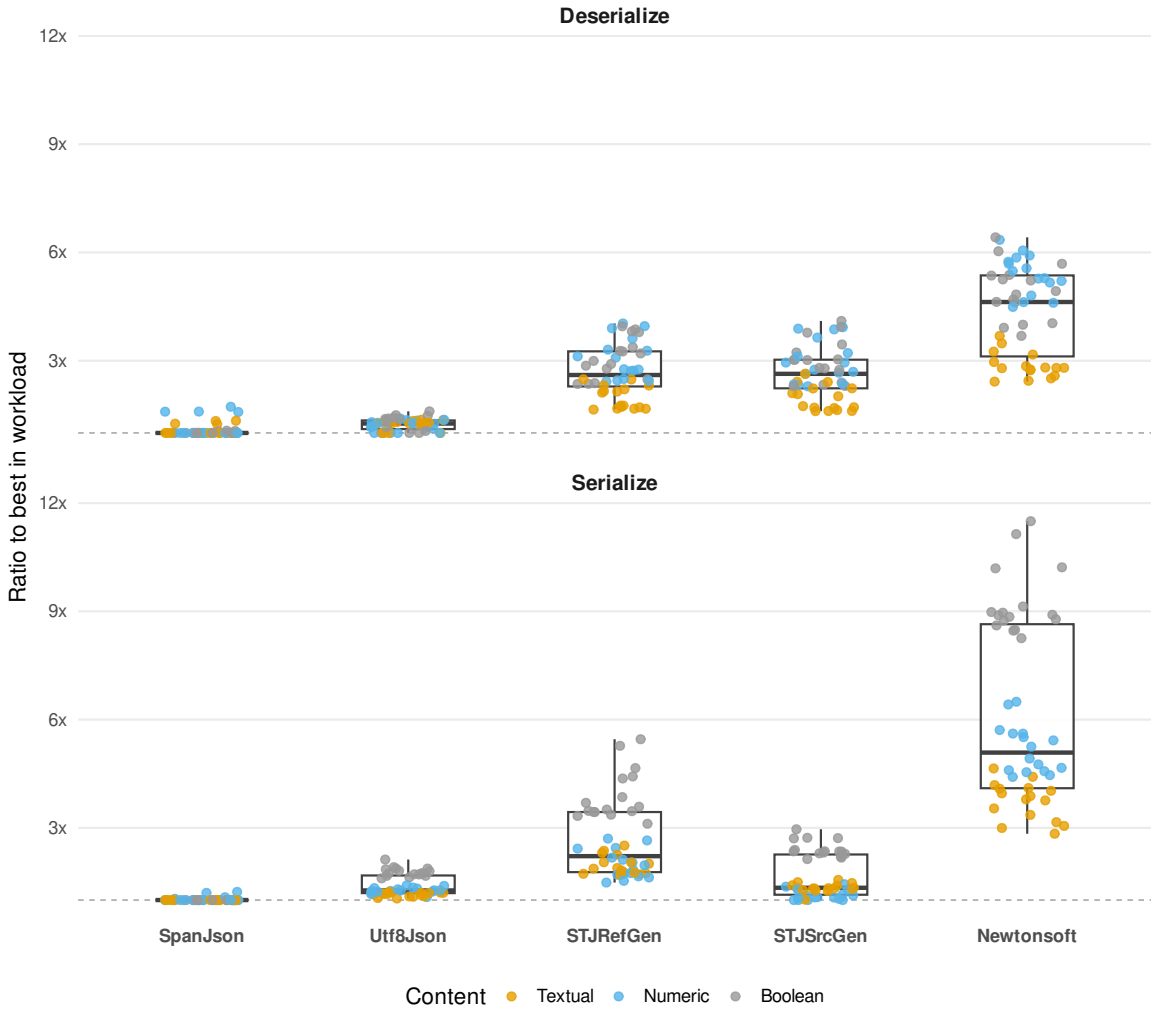


Figure 5.2: Per-library energy across the 48 workloads per operation, expressed as a ratio to the lowest-energy library on the same workload. Each point is one workload; dot colour is content type (Textual, Numeric, Boolean).

The ratios form three broad groups. `SpanJson` is closest to 1.00× across almost all workloads in both serialisation and deserialisation, matching the high Rank 1 frequency. `Utf8Json` is the next lowest-energy library, with a median of 1.26× the lowest-energy library in deserialisation and 1.27× in serialisation.

The two `System.Text.Json` variants form a middle group. During deserialisation, their distributions are almost indistinguishable, with medians of 2.61× and 2.64× compared to the lowest-energy library. During serialisation, `STJSrcGen` has a much lower median (1.34× the lowest-energy library) than `STJRefGen` (2.22× the lowest-energy library), matching the rank separation observed earlier (§5.1.1).

`Newtonsoft` uses the most energy in both operations, with a median of 4.63× the lowest-energy library and a maximum of 6.42× in deserialisation, and a median of 5.09× and a maximum of 11.50× in serialisation.

The point colours show an additional content-type effect. Among the highest-energy libraries, ratios are highest on Boolean workloads, lowest on Textual workloads, and intermediate on Numeric workloads. The spread between the lowest- and highest-energy libraries reaches 11.50× on Boolean serialisation and narrows to 2.42× on Textual deserialisation. This content-type pattern is clearest for the highest-energy libraries because `SpanJson` and `Utf8Json` remain close to 1.00× on most workloads. The architectural explanation is examined in §5.3. The full per-workload Depth × Width × Content heatmaps is provided in Appendix D.

5.2 RQ1: Library Rankings Across Workload Dimensions

This section reports how library rankings change across the six isolation benchmarks, each of which varies one workload dimension across a set of discrete *levels* while holding the others at a fixed baseline. Each subsection presents the same three views. First, a table reports execution time and energy at the lowest and highest levels of the dimension. This anchors the relative analyses in absolute values and shows the overall scale of change. Second, a rank-and-ratio heatmap shows the energy-consumption ranking at every level. The rank gives each library's position (1 = lowest energy consumption, 5 = highest energy consumption), while the ratio shows its energy consumption relative to the lowest-energy library (1.00×). Third, a per-unit heatmap shows how energy scales between adjacent levels, helping explain why relative energy gaps widen, narrow, or change the ranking.

As a statistical check, we first apply the tests from §3.4. Shapiro–Wilk rejects normality in 485/540 per-library measurement groups (89.8%), so non-parametric tests are used. The Holm-adjusted Kruskal–Wallis test finds significant differences between libraries in all 108 workload configurations. Pairwise Cliff's δ is *large* in 1,065/1,080 comparisons (98.6%). The 15 below-large cases are near-ties, occurring between `STJRefGen` and `STJSrcGen` in deserialisation and between `SpanJson` and `Utf8Json` in serialisation.

§5.2.7 presents the per-level rankings as a mean rank for each library and dimension. §5.2.8 presents the corresponding per-unit ratio: the energy per added unit at the highest level divided by the energy per added unit at the lowest level. Values above 1.00× indicate increasing per-unit cost, while values below 1.00× indicate amortisation as the dimension grows. §5.2.9 brings the main findings together. Full per-benchmark results, including timing, energy, garbage collection, and allocation, are reported in Appendix A.3.

5.2.1 Size

Table 5.2 shows that energy use increases sharply with Size. At C10, values range from 1,311 μJ/op to 7,257 μJ/op, while at C100K, they range from 19,500,000 μJ/op to 104,200,000 μJ/op. `SpanJson` uses the least energy for deserialisation at both ends, while `STJSrcGen` uses the least energy for serialisation. At C100K, `Newtonsoft` uses the most energy in both opera-

tions: 104,200,000 μJ/op for deserialisation and 63,600,000 μJ/op for serialisation. At both ends of the Size dimension, execution time and energy consumption produce the same library ranking in both operations. The observed energy differences therefore follow the execution time differences at these levels.

Table 5.2: Execution time (t, μs/op) and energy (E, μJ/op) at the lowest and highest Size levels. Within each energy column, cell colour is scaled from the column's lowest energy (light) to its highest (dark); colours are not comparable across columns.

	Deserialisation				Serialisation			
	C10		C100K		C10		C100K	
Library	t	E	t	E	t	E	t	E
SpanJson	75.24	2,728	1,405,684.65	50,971,301	141.51	5,019	1,573,516.28	56,050,524
Utf8Json	112.82	4,105	1,765,264.00	64,262,171	130.59	4,630	1,699,722.13	60,696,553
STJRefGen	103.18	3,755	1,669,657.50	60,947,582	63.65	2,321	640,462.16	23,937,451
STJSrcGen	100.04	3,640	1,650,024.23	60,240,476	35.14	1,311	521,863.37	19,490,059
Newtonsoft	199.66	7,257	2,873,212.59	104,204,765	115.10	4,191	1,744,143.29	63,579,339

Figure 5.3 shows that the same library uses the least energy at every Size level in both operations. During deserialisation, `SpanJson` is Rank 1 throughout. `STJSrcGen` and `STJRefGen` occupy Ranks 2 and 3, with ratios between 1.18× and 1.38×, while `Utf8Json` is Rank 4 and `Newtonsoft` is Rank 5. `STJRefGen` and `STJSrcGen` exchange Ranks 2 and 3 at C1K, but their ratios differ by only 0.02×.

The other libraries' ratios to `SpanJson` decrease as Size increases. For example, `Newtonsoft`'s ratio decreases from 2.66× at C10 to 2.04× at C100K. Figure 5.4 shows the scaling context. Size is implemented as the number of objects, so from C100 onward, `Newtonsoft` generally has lower adjacent energy per object ratios than the other libraries, reaching 1.07× in the final interval. The ratio to `SpanJson` therefore narrows because the other libraries' energy per object increases more, not because `Newtonsoft`'s energy per object decreases.

During serialisation, `STJSrcGen` and `STJRefGen` remain Ranks 1 and 2 throughout. `SpanJson`, `Utf8Json`, and `Newtonsoft` occupy Ranks 3–5, with ratios between 2.88× and 3.83× the per-level minimum. Among these three libraries, `SpanJson` moves from Rank 5 at C10 to Rank 3 at C100K. Figure 5.4 shows that the final narrowing relative to `STJSrcGen` occurs because `STJSrcGen`'s energy per object increases more than `SpanJson`'s from C10K to C100K (1.22× vs 1.13×). Nevertheless, `SpanJson` still uses 2.88× as much energy as `STJSrcGen` at C100K.

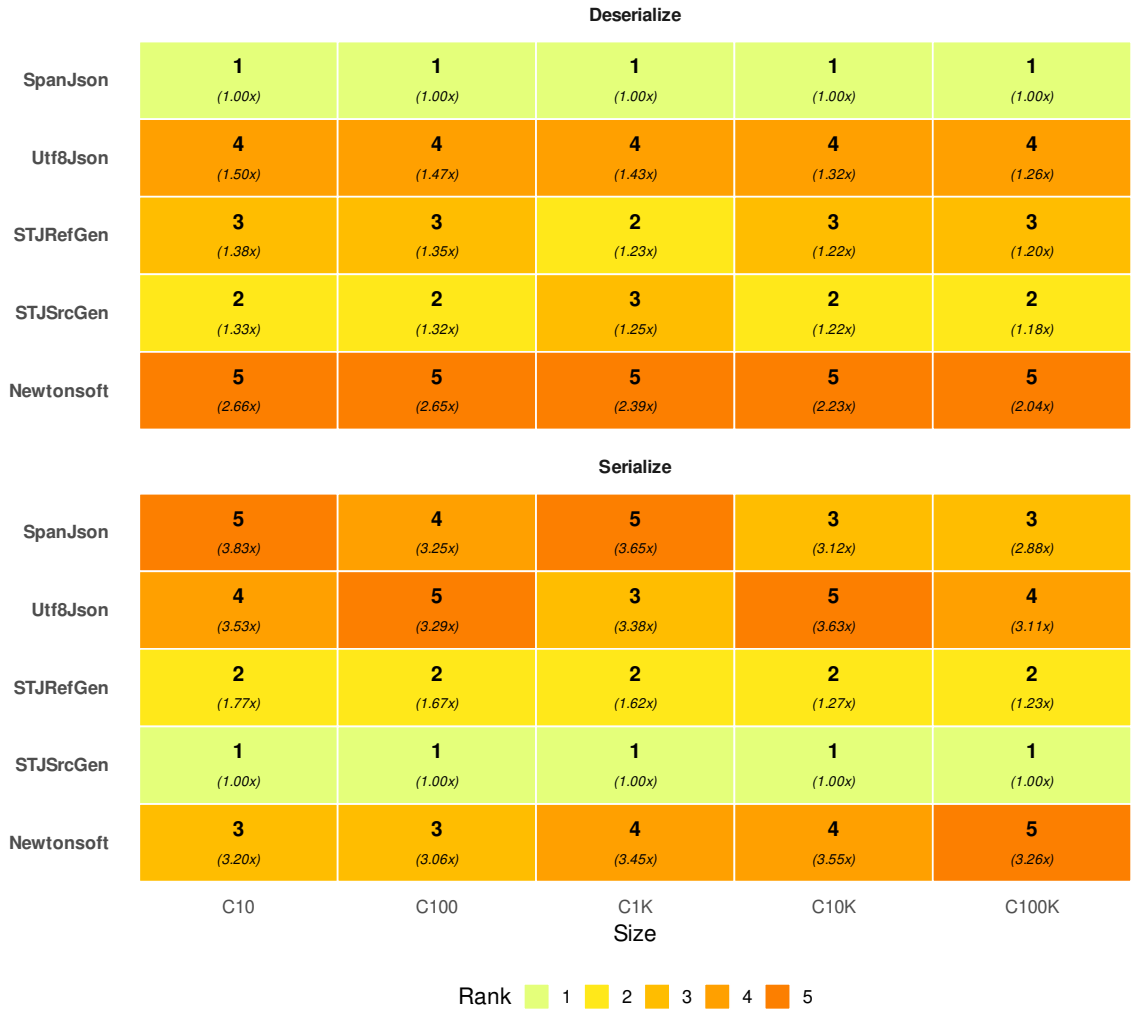


Figure 5.3: Library rank and energy ratio at each Size level, per operation. Rank 1 corresponds to the lowest energy consumption and Rank 5 to the highest energy consumption at that level. Cell text shows the rank (bold) and energy relative to the lowest energy consumption (italic).

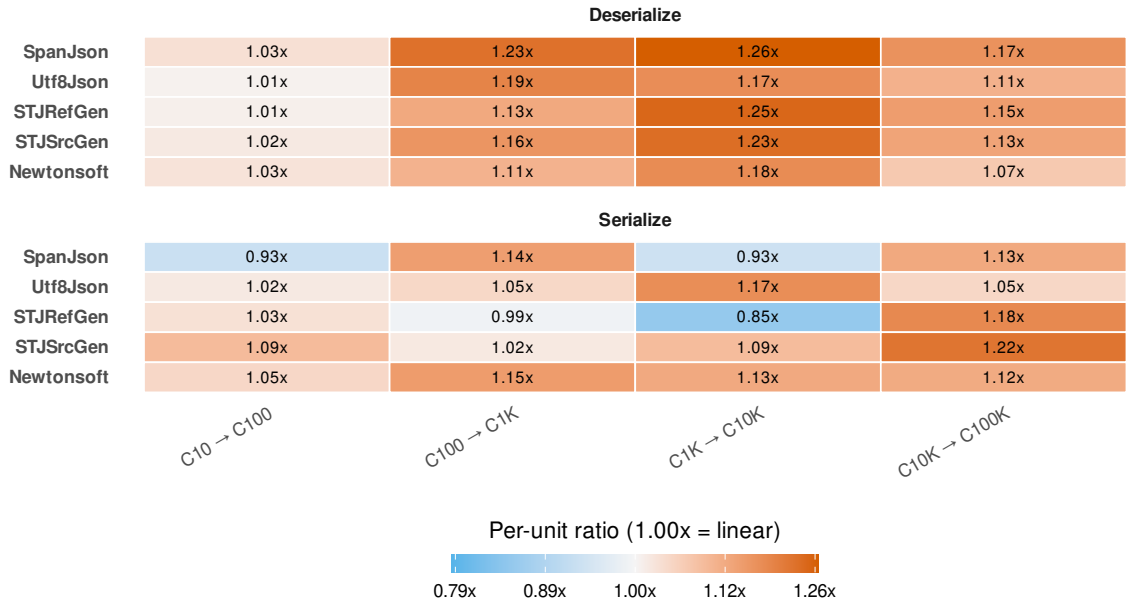


Figure 5.4: Adjacent ratio of energy per object between consecutive Size levels. A value of 1.00× indicates unchanged energy per object; values above 1.00× indicate an increase, and values below 1.00× indicate a decrease.

5.2.2 Depth

Table 5.3 shows energy use at the lowest and highest Depth levels. At D1, energy ranges from 29.5 μJ/op to 137 μJ/op across the libraries, while at D40, it ranges from 1,037 μJ/op to 5,020 μJ/op. `SpanJson` uses the least energy for deserialisation at both endpoints, increasing from 52.4 μJ/op at D1 to 2,124 μJ/op at D40. `STJSrcGen` uses the least energy for serialisation, increasing from 29.5 μJ/op to 1,037 μJ/op. `Newtonsoft` uses the most energy for deserialisation at both ends, increasing from 137 μJ/op to 5,020 μJ/op. During serialisation, `SpanJson` increases from 37.5 μJ/op at D1 to 3,802 μJ/op at D40, where it uses more energy than the other libraries. At both ends, execution time and energy consumption produce the same library ranking in both operations.

Table 5.3: Execution time (t, μs/op) and energy (E, μJ/op) at the lowest and highest Depth levels. Within each energy column, cell colour is scaled from the column's lowest energy (light) to its highest (dark); colours are not comparable across columns.

	Deserialisation				Serialisation			
	D1		D40		D1		D40	
Library	t	E	t	E	t	E	t	E
SpanJson	1.48	52.4	59.90	2,124	1.06	37.5	110.06	3,802
Utf8Json	2.28	80.7	86.20	3,057	1.33	47.2	99.83	3,444
STJRefGen	1.98	70.3	82.03	2,915	0.94	34.4	50.25	1,792
STJSrcGen	1.99	70.9	80.70	2,874	0.81	29.5	28.89	1,037
Newtonsoft	3.85	137	141.60	5,020	2.49	88.5	91.67	3,255

Figure 5.5 shows that deserialisation rankings remain stable across Depth. `SpanJson` is Rank 1 and `Newtonsoft` is Rank 5 at every level. The two `System.Text.Json` variants

occupy Ranks 2 and 3, with ratios between 1.34× and 1.39×, while `Utf8Json` remains at Rank 4.

The serialisation ranking changes more as Depth increases. `STJSrcGen` and `STJRefGen` remain at Ranks 1 and 2 throughout. `SpanJson` moves from Rank 3 at D1, with a ratio of 1.27×, to Rank 5 by D25, reaching 3.67× at D40. `Newtonsoft` moves from Rank 5 to Rank 3 by D40, while its ratio remains close to 3.0–3.3× across the levels. Its higher rank at D40 therefore reflects the increasing energy ratios of `SpanJson` and `Utf8Json` rather than a substantial change in `Newtonsoft`'s own ratio.

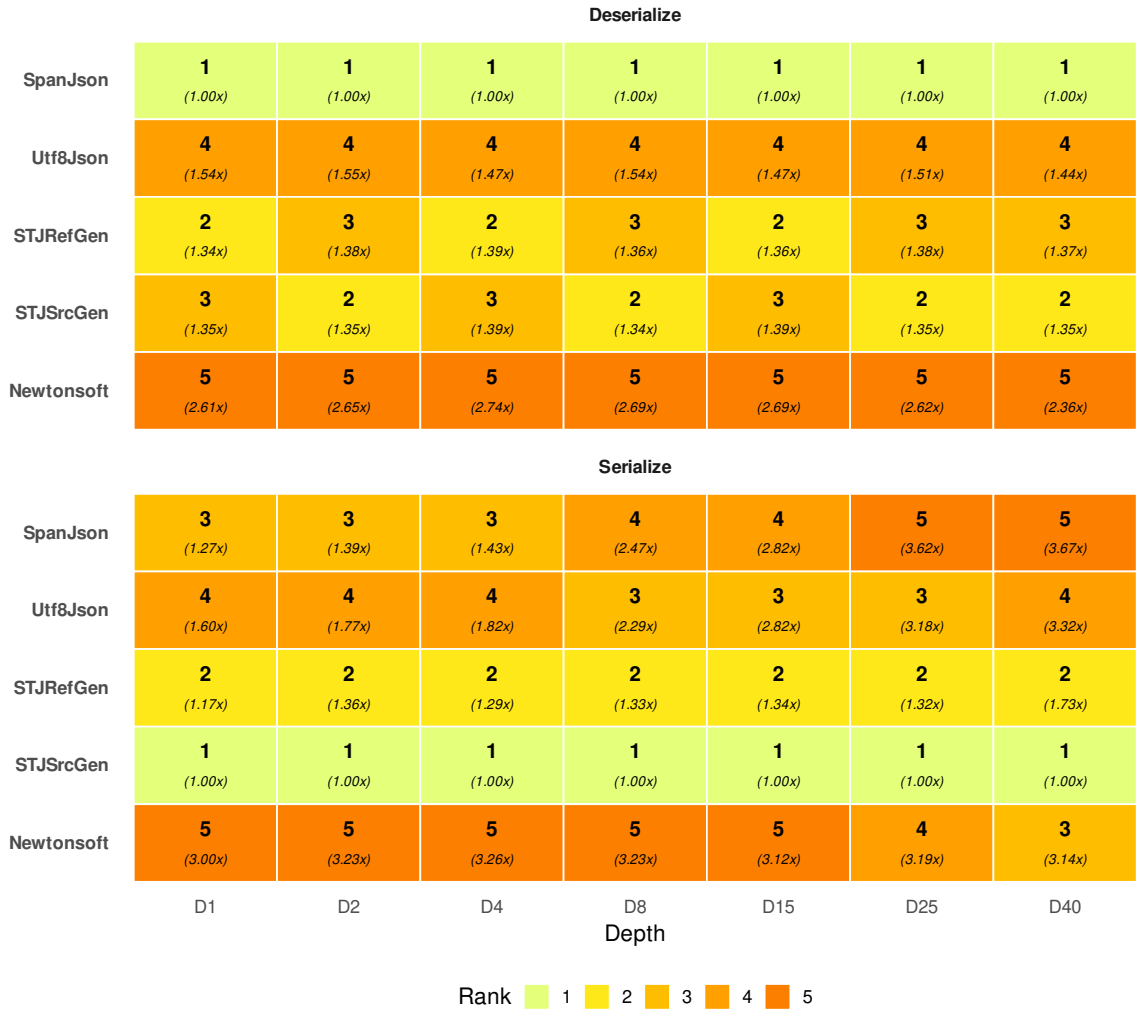


Figure 5.5: Library rank and energy ratio at each Depth level, per operation. Rank 1 corresponds to the lowest energy consumption and Rank 5 to the highest energy consumption at that level. Cell text shows the rank (bold) and energy relative to the lowest energy consumption (italic).

Figure 5.6 shows the scaling pattern behind rankings. During deserialisation, the adjacent per-nesting-level energy ratios remain close to 1.00× for all libraries, indicating that energy increases approximately in proportion to Depth. This supports the stable ranking.

During serialisation, `SpanJson` and `Utf8Json` show higher per-nesting-level energy ratios through the middle of the dimension. `SpanJson`'s energy per nesting level rises to 1.66× at the D4–D8 step, while `Utf8Json` reaches 1.21× over the same interval and 1.26× over D8–D15. `STJSrcGen` remains close to 1.00× throughout. The serialisation rank changes therefore follow the stronger increase in energy per nesting level for `SpanJson` and `Utf8Json`.

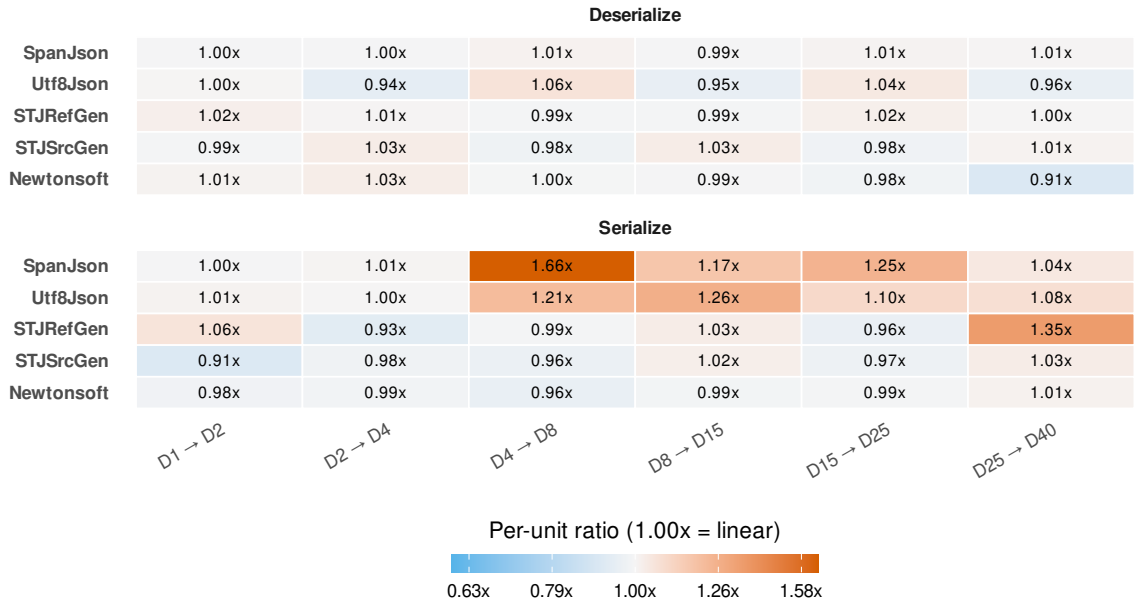


Figure 5.6: Adjacent ratio of energy per nesting level between consecutive Depth levels. A value of 1.00× indicates unchanged energy per nesting level; values above 1.00× indicate an increase, and values below 1.00× indicate a decrease.

5.2.3 Width

Table 5.4 shows energy use at the lowest and highest Width levels. At W2, `SpanJson` uses the least energy in both operations, with 18.1 μJ/op in deserialisation and 12.9 μJ/op in serialisation. `Newtonsoft` uses the most energy at W2, with 73.1 μJ/op and 46.4 μJ/op. At W200, the ranking changes. During deserialisation, `Utf8Json` uses the least energy at 4,074 μJ/op, while `STJRefGen` and `STJSrcGen` use the most at 9,156 μJ/op and 8,855 μJ/op. During serialisation, `STJSrcGen` uses the least at 1,370 μJ/op, while `SpanJson`, `Utf8Json`, and `Newtonsoft` are grouped near 4,800 μJ/op. At both ends, execution time and energy consumption produce the same library ranking in both operations.

Table 5.4: Execution time (t, μs/op) and energy (E, μJ/op) at the lowest and highest Width levels. Within each energy column, cell colour is scaled from the column's lowest energy (light) to its highest (dark); colours are not comparable across columns.

	Deserialisation				Serialisation			
	W2		W200		W2		W200	
Library	t	E	t	E	t	E	t	E
SpanJson	0.51	18.1	127.17	4,498	0.36	12.9	139.55	4,810
Utf8Json	0.77	27.5	114.87	4,074	0.46	16.4	138.79	4,798
STJRefGen	1.18	42.1	258.07	9,156	0.67	24.2	81.05	2,875
STJSrcGen	1.15	40.9	249.75	8,855	0.36	13.2	37.71	1,370
Newtonsoft	2.06	73.1	214.70	7,608	1.30	46.4	134.03	4,749

Figure 5.7 shows that the deserialisation ranking changes mainly at the widest level. `SpanJson` is Rank 1 from W2 through W100. At W200, `Utf8Json` becomes Rank 1, while `SpanJson` has a ratio of 1.10×. `STJRefGen` and `STJSrcGen` start in the lower ranks at W2

and W5, then move closer to `Utf8Json` over W10–W50. Their ratios increase sharply at the high Width levels, reaching 2.25× and 2.17× at W200. `Newtonsoft` shows the opposite relative pattern. Its ratio decreases from 4.03× at W2 to 1.87× at W200, moving from Rank 5 to Rank 3.

During serialisation, `STJSrcGen` and `STJRefGen` are Ranks 1 and 2 from W5 onward. `SpanJson` is Rank 1 at W2, but reaches Rank 5 at W100 and W200. At W200, `SpanJson`, `Utf8Json`, and `Newtonsoft` have very similar ratios: 3.51×, 3.50×, and 3.47×. The Rank 3–5 ranking at W200 therefore reflects a 0.04× spread within the higher-energy group rather than a large separation between those three libraries.

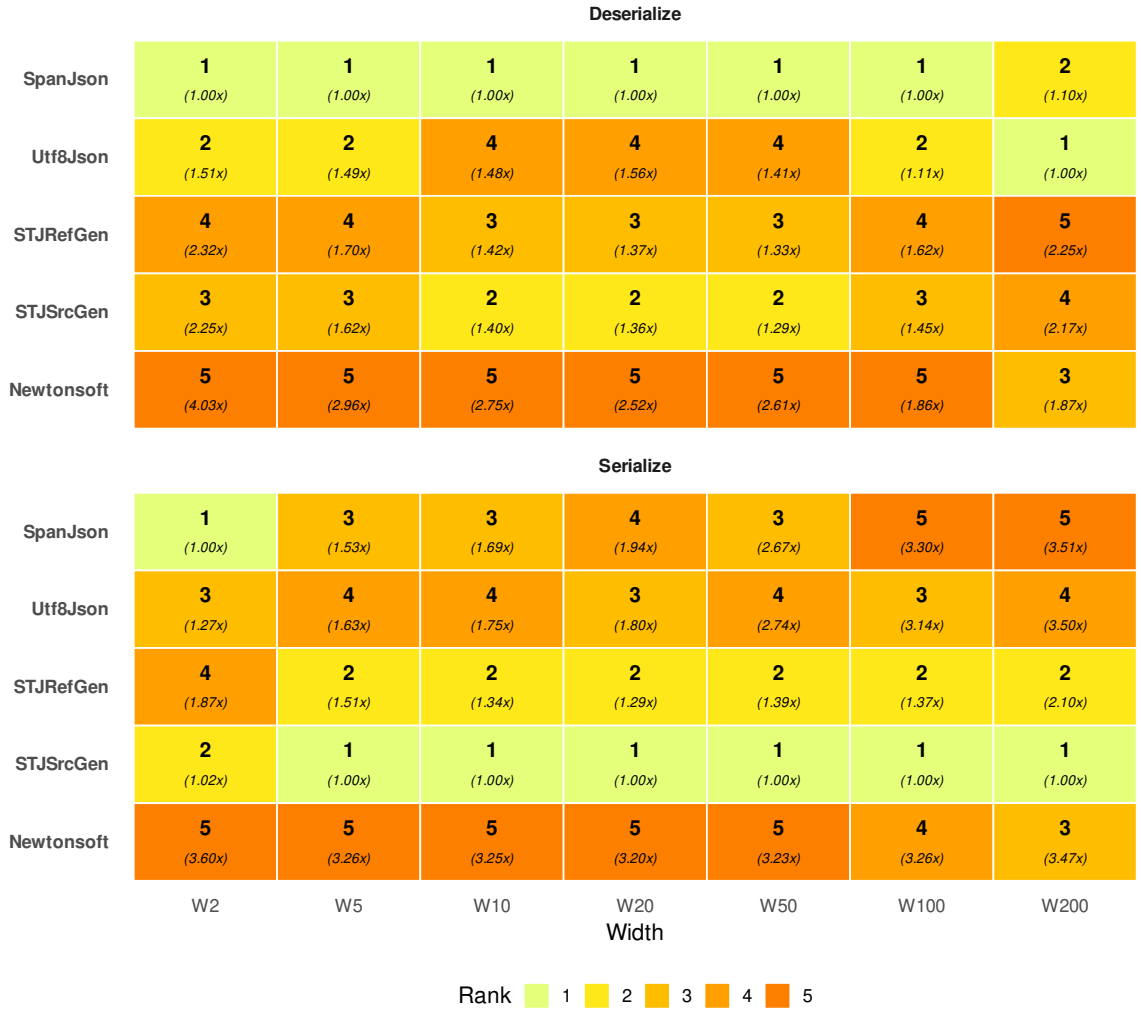


Figure 5.7: Library rank and energy ratio at each Width level, per operation. Rank 1 corresponds to the lowest energy consumption and Rank 5 to the highest energy consumption at that level. Cell text shows the rank (bold) and energy relative to the lowest energy consumption (italic).

Figure 5.8 shows the scaling pattern behind the rank changes. During deserialisation, `STJRefGen` and `STJSrcGen` remain close to or below 1.00× through the middle of the dimension, but their energy per field increases sharply at the high Width intervals. `STJRefGen` reaches 1.78× at W50–W100 and 1.41× at W100–W200, while `STJSrcGen` reaches 1.64× and 1.53×. In contrast, `Newtonsoft` remains close to 1.00× over the same intervals. Its lower ratio at W200 therefore comes from the stronger increase in the `System.Text.Json` variants' energy per field, not from a decrease in `Newtonsoft`'s energy per field.

During serialisation, `STJSrcGen` stays close to 1.00× at every interval, which is consistent with its Rank 1 position from W5 onward. `SpanJson` and `Utf8Json` have higher energy-per-field ratios across several intervals, especially W2–W5 and W20–W100. This explains why their ratios to `STJSrcGen` increase as Width grows. `Newtonsoft` has ratios close to 1.00× across most intervals, which is why it becomes close to `SpanJson` and `Utf8Json` by W200. `STJRefGen` is also close to or below 1.00× for most intervals, but increases to 1.58× at W100–W200.

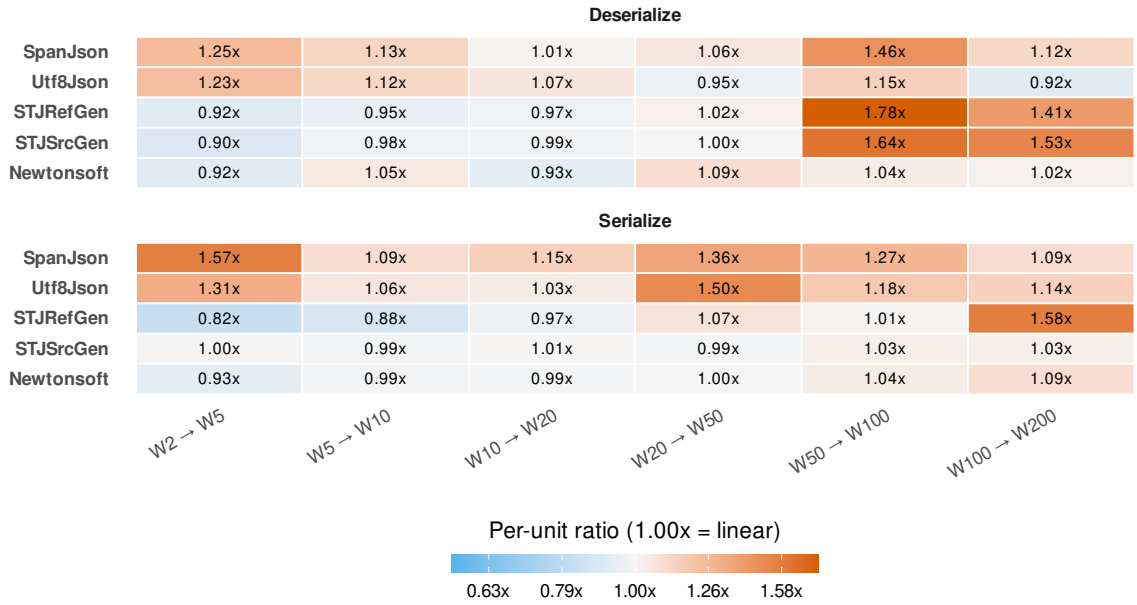


Figure 5.8: Adjacent ratio of energy per field between consecutive Width levels. A value of 1.00× indicates unchanged energy per field; values above 1.00× indicate an increase, and values below 1.00× indicate a decrease.

5.2.4 String Length

Table 5.5 shows energy use at the lowest and highest String Length levels. At L5, `SpanJson` uses the least energy in both operations, with 207 μJ/op in deserialisation and 98.6 μJ/op in serialisation. `Newtonsoft` uses the most energy at L5, with 580 μJ/op and 282 μJ/op. At L10000, `STJRefGen` and `STJSrcGen` use the least energy for deserialisation, with 7,972 μJ/op and 7,988 μJ/op. `STJSrcGen` and `STJRefGen` use the least energy for serialisation, with 15,438 μJ/op and 16,049 μJ/op. `Newtonsoft` uses the most energy for deserialisation at 113,595 μJ/op, while `Utf8Json` uses the most energy for serialisation at 206,958 μJ/op. At both ends, execution time and energy consumption produce the same library ranking in both operations.

Table 5.5: Execution time (t, μs/op) and energy (E, μJ/op) at the lowest and highest String Length levels. Within each energy column, cell colour is scaled from the column's lowest energy (light) to its highest (dark); colours are not comparable across columns.

	Deserialisation				Serialisation			
	L5		L10000		L5		L10000	
Library	t	E	t	E	t	E	t	E
SpanJson	5.69	207	1,156.50	41,156	2.71	98.6	5,426.69	190,047
Utf8Json	7.33	267	2,038.29	74,331	3.05	111	5,884.04	206,958
STJRefGen	9.63	351	215.77	7,972	4.42	161	447.37	16,049
STJSrcGen	9.12	333	215.98	7,988	3.32	121	427.83	15,438
Newtonsoft	15.93	580	3,230.05	113,595	7.74	282	2,670.78	93,342

Figure 5.9 shows that the energy ranking changes as the string becomes longer. During deserialisation, `SpanJson` is Rank 1 at L5 and L20. From L50 onward, `STJRefGen` and `STJSrcGen` occupy Ranks 1 and 2, with nearly identical energy values. Over the same levels, `SpanJson`'s ratio increases from 1.01× at L50 to 5.16× at L10000. `Utf8Json` and `Newtonsoft` reach 9.32× and 14.25× at L10000.

During serialisation, `SpanJson` is Rank 1 at L5. `STJSrcGen` is Rank 1 from L20 onward, while `STJRefGen` remains Rank 2 over the same levels. The ratios for `SpanJson`, `Utf8Json`, and `Newtonsoft` increase the most through L500, where they reach 25.70×, 20.90×, and 14.70×. Their ratios decrease again by L10000, but remain at 12.30×, 13.40×, and 6.05×.

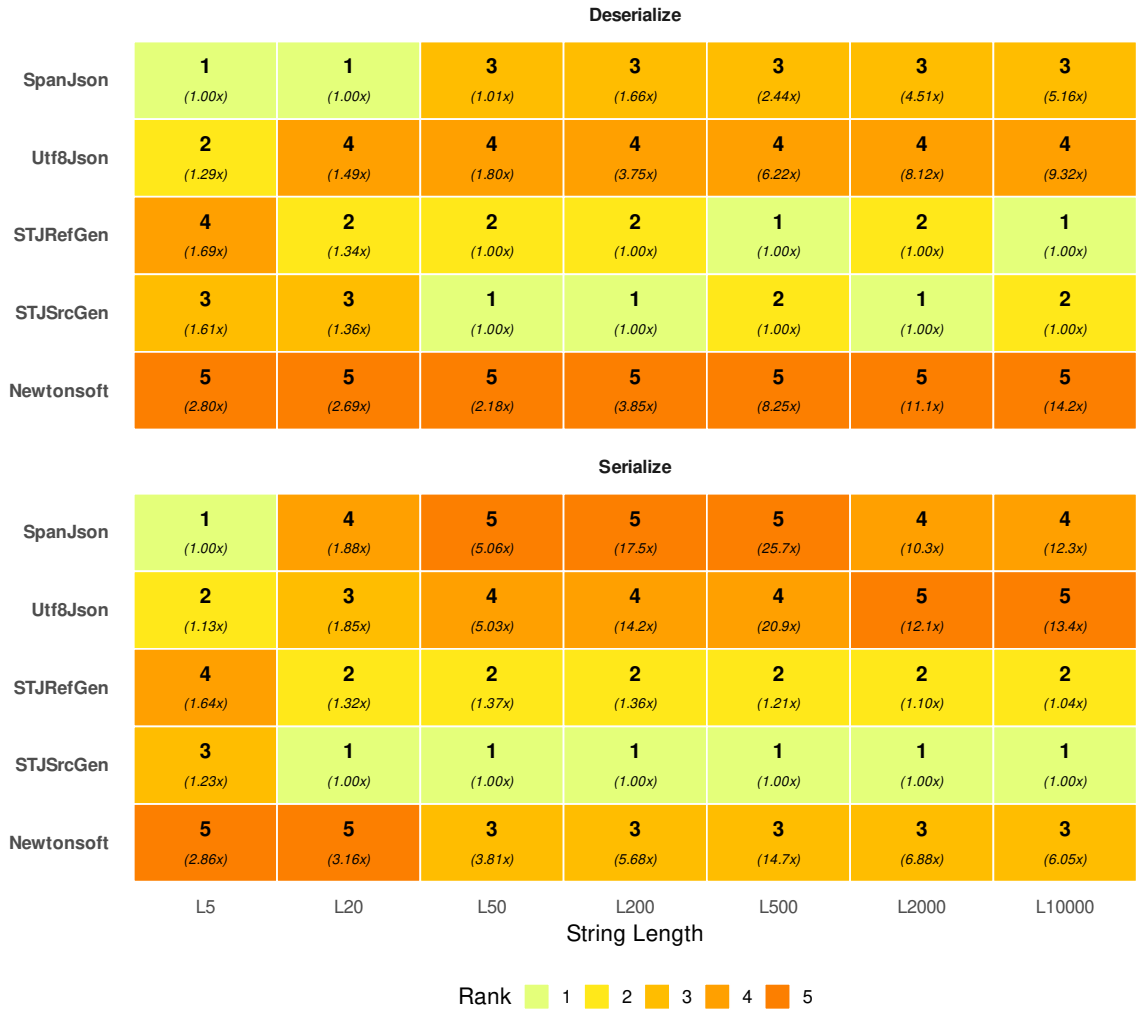


Figure 5.9: Library rank and energy ratio at each String Length level, per operation. Rank 1 corresponds to the lowest energy consumption and Rank 5 to the highest energy consumption at that level. Cell text shows the rank (bold) and energy relative to the lowest energy consumption (italic).

Figure 5.10 shows the scaling pattern behind rank changes. During deserialisation, `STJRefGen` and `STJSrcGen` have energy per character ratios below 1.00× at every interval, and their ratios are also lower than those of the other libraries. Their energy per character therefore improves relative to the other libraries as String Length increases. This is consistent with their Rank 1 and Rank 2 positions from L50 onward and with the widening ratios of the remaining libraries. `SpanJson` and `Newtonsoft` exceed 1.00× in some later intervals, while `Utf8Json` approaches 1.00×, so their energy per character decreases less or begins to increase.

During serialisation, `STJRefGen` and `STJSrcGen` also have low energy-per-character ratios through L500. Over L5–L500, their ratios remain between 0.27× and 0.69×. `SpanJson` and `Utf8Json` exceed 1.00× over L20–L200, so their energy per character increases while the `STJRefGen` and `STJSrcGen` continue to decrease. At L500–L2000, `STJRefGen` and `STJSrcGen` increase to 1.87× and 2.04×, which corresponds to the lower ratios of the other libraries at L2000. Despite this increase, `STJRefGen` and `STJSrcGen` retain the lowest total energy because of their lower energy-per-character ratios over the earlier intervals. `Newtonsoft` moves to Rank 3 in serialisation from L50 onward. Its energy-per-character ratios are lower than those of `SpanJson` and `Utf8Json` over L5–L200, which helps explain this change in

ranking.

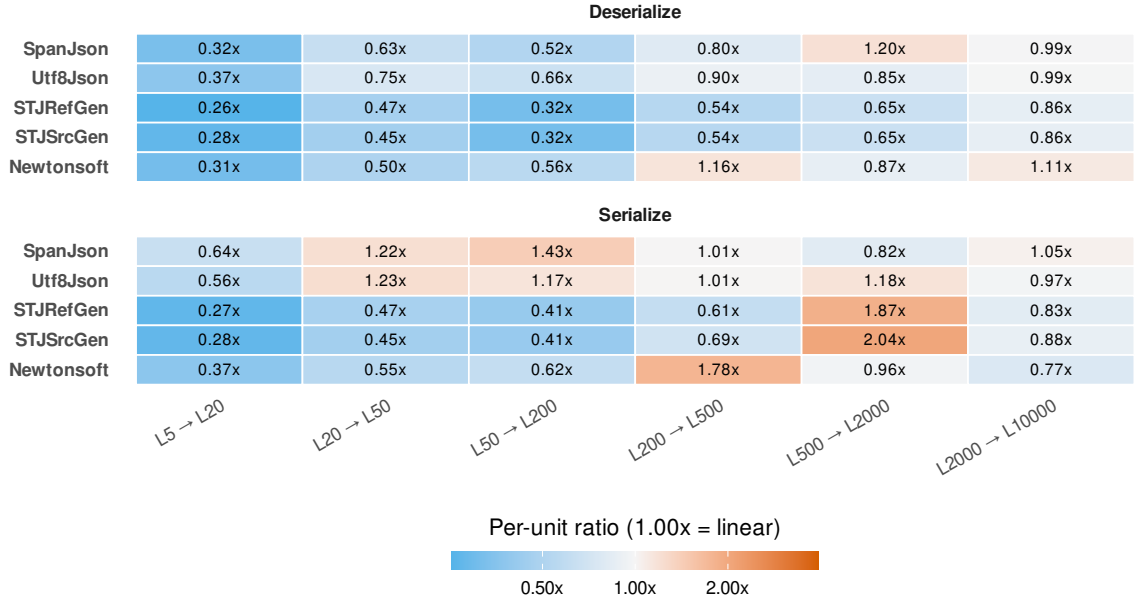


Figure 5.10: Adjacent ratio of energy per character between consecutive String Length levels. A value of 1.00× indicates unchanged energy per character; values above 1.00× indicate an increase, and values below 1.00× indicate a decrease.

5.2.5 Numeric Length by Type

Tables 5.6 and 5.7 show energy use at the lowest and highest Numeric Length levels for the Integer and Float variants. At matched digit lengths, Float values use more energy than Integer values for every library and operation.

`SpanJson` uses the least energy for deserialisation at both endpoints of both variants. For Integer values, its energy increases from 89.3 μJ/op at I2 to 132 μJ/op at I10. For Float values, it increases from 220 μJ/op at F2 to 324 μJ/op at F10. During serialisation, `SpanJson` uses the least energy at I2 (44.3 μJ/op), while `STJSrcGen` uses the least energy at I10 (83.6 μJ/op). For Float serialisation, `Utf8Json` uses the least energy at F2 (270 μJ/op), while `STJSrcGen` uses the least energy at F10 (462 μJ/op). `Newtonsoft` uses the most energy in all 4 workloads. At both ends of both variants, execution time and energy consumption produce the same library ranking.

Table 5.6: Execution time (t, μs/op) and energy (E, μJ/op) for the Integer variant at the lowest (I2) and highest (I10) Numeric Length levels. Within each energy column, cell colour is scaled from the column's lowest energy (light) to its highest (dark); colours are not comparable across columns.

	Deserialisation				Serialisation			
	I2		I10		I2		I10	
Library	t	E	t	E	t	E	t	E
SpanJson	2.52	89.3	3.73	132	1.25	44.3	2.91	103
Utf8Json	3.36	119	4.41	156	1.70	60.2	3.59	124
STJRefGen	7.31	259	9.10	323	3.09	110	3.59	128
STJSrcGen	7.45	264	9.03	320	1.77	63.1	2.35	83.6
Newtonsoft	15.09	535	18.98	673	7.18	255	10.43	370

Table 5.7: Execution time (t, μs/op) and energy (E, μJ/op) for the Float variant at the lowest (F2) and highest (F10) Numeric Length levels. Within each energy column, cell colour is scaled from the column's lowest energy (light) to its highest (dark); colours are not comparable across columns.

	Deserialisation				Serialisation			
	F2		F10		F2		F10	
Library	t	E	t	E	t	E	t	E
SpanJson	6.36	220	9.40	324	8.03	279	14.07	487
Utf8Json	8.39	295	13.04	453	7.65	270	13.36	471
STJRefGen	11.52	407	14.19	498	10.69	379	15.74	551
STJSrcGen	11.19	394	14.11	496	8.29	291	13.31	462
Newtonsoft	22.19	784	28.85	1,017	17.09	606	22.45	796

Figure 5.11 shows that the deserialisation ranking remains stable across both variants. `SpanJson` is Rank 1, `Utf8Json` is Rank 2, and `Newtonsoft` is Rank 5 at every level. `STJRefGen` and `STJSrcGen` exchange Ranks 3 and 4, while their energy ratios differ by at most 0.07×. For Integer deserialisation, the other libraries' energy ratios relative to `SpanJson` generally decrease as Numeric Length increases. For example, `Newtonsoft` decreases from 6.00× at I2 to 5.09× at I10. Float deserialisation shows a similar narrowing for `STJRefGen`, `STJSrcGen`, and `Newtonsoft`. `Utf8Json` differs slightly. Its ratio increases from 1.34× at F2 to 1.48× at F7, then decreases to 1.39× at F10.

The serialisation ranking changes across the Numeric Length levels. For Integer values, `SpanJson` is Rank 1 at I2–I5, while `STJSrcGen` is Rank 1 at I7 and I10. For Float values, `Utf8Json` is Rank 1 at F2–F7, while `STJSrcGen` is Rank 1 at F10. At F10, `STJSrcGen`, `Utf8Json`, and `SpanJson` cluster close together and have energy ratios of 1.00×, 1.02×, and 1.05×, respectively. `Newtonsoft` remains Rank 5 in both variants, although its energy ratios are lower for Float values than for Integer values.

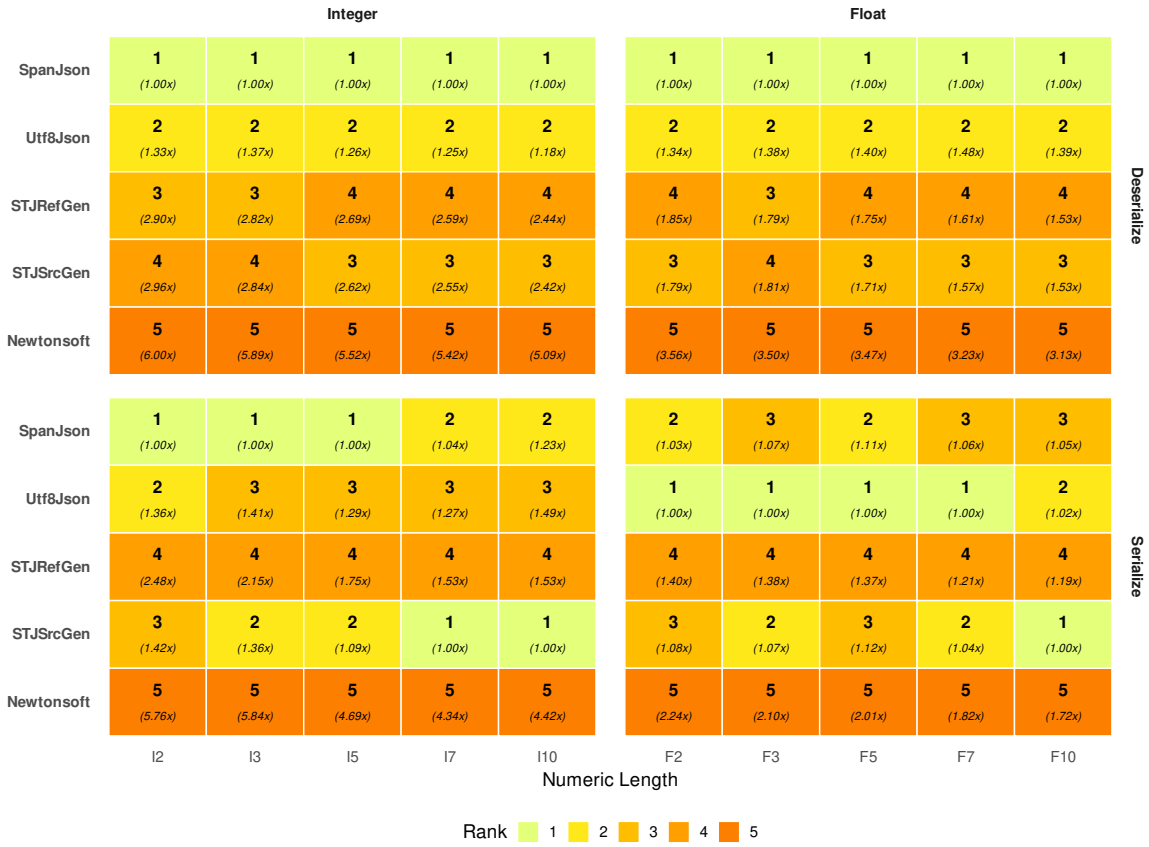


Figure 5.11: Library rank and energy ratio at each Numeric Length level, per operation and variant (Integer / Float). Rank 1 corresponds to the lowest energy consumption and Rank 5 to the highest energy consumption at that level. Cell text shows the rank (bold) and energy relative to the lowest energy consumption (italic).

Figure 5.12 shows that for deserialisation, adjacent per-digit ratios stay below 1.00× for every library across both Integer and Float variants. Energy, therefore, increases less than proportionally with numeric length. Because this pattern is similar across libraries, the ranking remains stable. For Integer values, `STJRefGen`, `STJSrcGen`, and `Newtonsoft` generally have slightly lower energy-per-digit ratios than `SpanJson`, so their total-energy ratios relative to `SpanJson` narrow as Numeric Length increases. For Float values, the pattern is similar, except that `Utf8Json`'s ratio relative to `SpanJson` increases through F7 before decreasing at F10.

Serialisation shows the same overall decrease in per-digit cost, but the differences between libraries matter more for the leading ranks. For Integer values, `STJSrcGen` has lower per-digit ratios than `SpanJson` on most adjacent steps, especially toward the longer values. This explains why it uses less energy than `SpanJson` from I7.

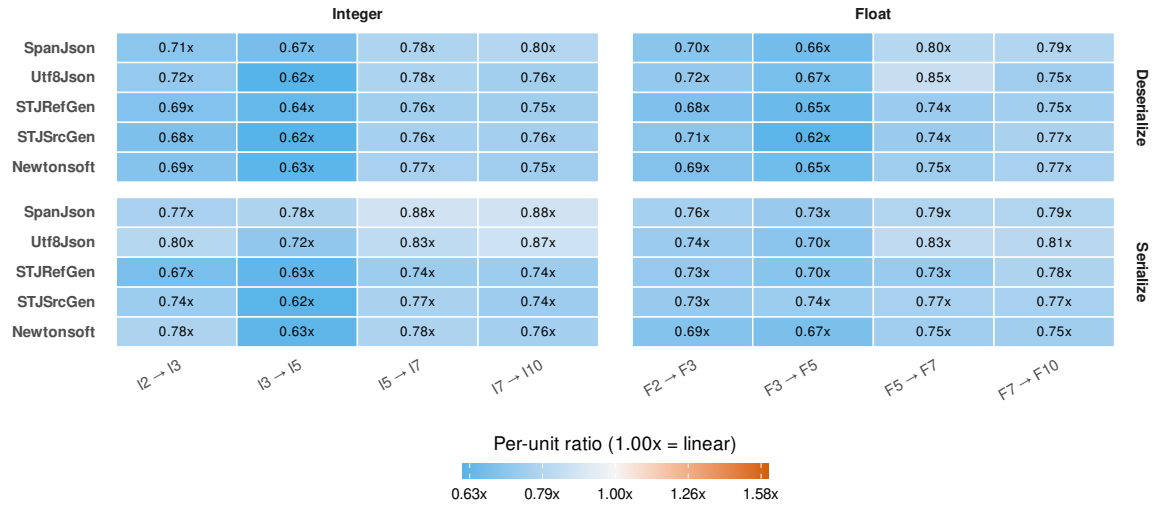


Figure 5.12: Adjacent ratio of energy per digit between consecutive Numeric Length levels. A value of 1.00× indicates unchanged energy per digit; values above 1.00× indicate an increase, and values below 1.00× indicate a decrease.

5.2.6 String Composition

Tables 5.8, 5.9, and 5.10 show energy use at the shared ASCII baseline and the 100% density level for each String Composition variant. At A0, `SpanJson` uses the least energy for deserialisation at 261 μJ/op, while `STJSrcGen` uses the least energy for serialisation at 129 μJ/op. `Newtonsoft` uses the most energy in both operations, with 701 μJ/op in deserialisation and 426 μJ/op in serialisation.

At U100 and UE100, `SpanJson` uses the least energy for deserialisation, with 1,280 μJ/op and 1,253 μJ/op, while `Utf8Json` uses the most energy, with 2,394 μJ/op and 2,372 μJ/op. Escape produces a different ranking. At E100, `Newtonsoft` uses the least energy for deserialisation at 1,087 μJ/op, while `Utf8Json` uses the most at 1,781 μJ/op.

During serialisation, `SpanJson` uses the least energy at U100 and UE100, with 280 μJ/op and 296 μJ/op. `STJRefGen` uses the most energy at those levels, with 1,007 μJ/op and 1,080 μJ/op. At E100, `STJSrcGen` uses the least energy at 904 μJ/op, while `Utf8Json` uses the most at 1,606 μJ/op. At both ends of each variant, execution time and energy consumption produce the same library ranking.

Table 5.8: Execution time (t, μs/op) and energy (E, μJ/op) for the Unicode variant at the lowest (A0) and highest (U100) Special-character density (%) levels. Within each energy column, cell colour is scaled from the column's lowest energy (light) to its highest (dark); colours are not comparable across columns.

	Deserialisation				Serialisation			
	A0		U100		A0		U100	
Library	t	E	t	E	t	E	t	E
SpanJson	7.37	261	36.39	1,280	6.15	218	7.93	280
Utf8Json	10.82	384	69.32	2,394	6.82	241	9.24	320
STJRefGen	9.74	346	62.54	2,201	4.95	179	28.35	1,007
STJSrcGen	9.84	350	64.26	2,271	3.59	129	24.88	891
Newtonsoft	19.78	701	54.63	1,935	12.01	426	16.88	598

Table 5.9: Execution time (t, μs/op) and energy (E, μJ/op) for the Escape variant at the lowest (A0) and highest (E100) Special-character density (%) levels. Within each energy column, cell colour is scaled from the column's lowest energy (light) to its highest (dark); colours are not comparable across columns.

	Deserialisation				Serialisation			
	A0		E100		A0		E100	
Library	t	E	t	E	t	E	t	E
SpanJson	7.37	261	36.92	1,283	6.15	218	29.00	1,028
Utf8Json	10.82	384	50.35	1,781	6.82	241	45.45	1,606
STJRefGen	9.74	346	48.09	1,695	4.95	179	28.79	1,020
STJSrcGen	9.84	350	47.62	1,671	3.59	129	25.50	904
Newtonsoft	19.78	701	30.66	1,087	12.01	426	34.82	1,234

Table 5.10: Execution time (t, μs/op) and energy (E, μJ/op) for the UnicodeEscape variant at the lowest (A0) and highest (UE100) Special-character density (%) levels. Within each energy column, cell colour is scaled from the column's lowest energy (light) to its highest (dark); colours are not comparable across columns.

	Deserialisation				Serialisation			
	A0		UE100		A0		UE100	
Library	t	E	t	E	t	E	t	E
SpanJson	7.37	261	35.97	1,253	6.15	218	8.36	296
Utf8Json	10.82	384	68.76	2,372	6.82	241	8.77	308
STJRefGen	9.74	346	63.88	2,249	4.95	179	30.42	1,080
STJSrcGen	9.84	350	65.56	2,309	3.59	129	24.91	885
Newtonsoft	19.78	701	52.39	1,848	12.01	426	16.74	593

Figure 5.13 shows that `SpanJson` is Rank 1 for deserialisation at every level except E100. At E100, `Newtonsoft` is Rank 1, while `SpanJson` has an energy ratio of 1.18×. `Newtonsoft`'s ratio decreases from 2.68× at A0 to 1.00× at E100. It also improves on Unicode and UnicodeEscape variants, moving to Rank 2 from U25 and UE25 onward, with its ratio decreasing to 1.51× at U100 and 1.47× at UE100. `Utf8Json`, `STJRefGen`, and `STJSrcGen` exchange ranks across the three variants, but their ratios are usually close. At U100, their energy ratios are 1.87×, 1.72×, and 1.77×. At E100, the corresponding ratios are 1.64×, 1.56×, and 1.54×. At UE100, they are 1.89×, 1.79×, and 1.84×.

During serialisation, at A0, `STJSrcGen` is Rank 1 and `STJRefGen` is Rank 2. For Unicode and UnicodeEscape, `SpanJson` and `Utf8Json` occupy Ranks 1 and 2 at the highest density, while `STJSrcGen` and `STJRefGen` move to Ranks 4 and 5. At U100, `STJSrcGen` and `STJRefGen` have ratios of 3.18× and 3.59×. At UE100, their ratios are 2.99× and 3.65×. Escape produces a different serialisation ranking. `STJSrcGen` is Rank 1 at E50 and E100. At E100, `STJRefGen`, `SpanJson`, `Newtonsoft`, and `Utf8Json` have ratios of 1.13×, 1.14×, 1.37×, and 1.78×, respectively. `Newtonsoft`'s ratio decreases from 3.30× at A0 to 1.37× at E100.

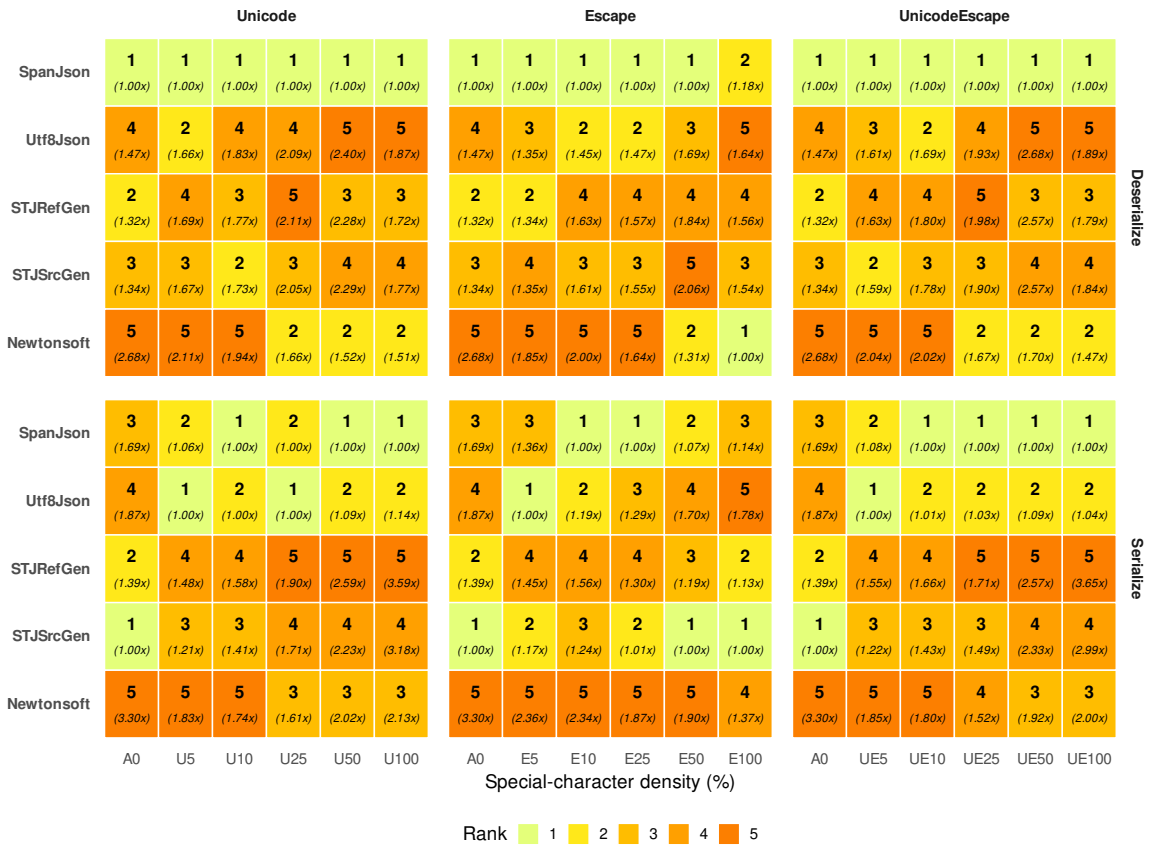


Figure 5.13: Library rank and energy ratio at each String Composition level, per operation and variant (Unicode / Escape / UnicodeEscape). Rank 1 corresponds to the lowest energy consumption and Rank 5 to the highest energy consumption at that level. Cell text shows the rank (bold) and energy relative to the lowest energy consumption (italic). The `A0` column (shared ASCII baseline) appears in every variant facet as the 0% point, computed off the same measurement.

Figure 5.14 shows the change in energy per special character between adjacent non-zero density levels. Every ratio is below 1.00×, so total energy increases with special-character density, but less than proportionally.

During deserialisation, `Newtonsoft` has some of the lowest energy-per-special-character ratios, especially for Escape, where its ratios range from 0.45× to 0.56× across E5–E100. This is consistent with its movement from Rank 5 at A0 to Rank 1 at E100. For Unicode and UnicodeEscape, its ratios are also among the lowest across most intervals, which is consistent with its movement to Rank 2 at U100 and UE100. `SpanJson` remains at Rank 1 at every deserialisation level except E100. Its low energy at A0 and energy-per-special-character ratios below 1.00× across every interval help preserve this position.

A common pattern also appears at the highest densities. In 12 of the 15 library–variant combinations, the energy-per-special-character ratio is lower over the 50%–100% interval than over the preceding 25%–50% interval. The only increases occur for `SpanJson` on Escape and UnicodeEscape, while `Newtonsoft` remains unchanged on Escape. Energy per special character, therefore, usually decreases more strongly at the highest density.

During serialisation, `SpanJson` and `Utf8Json` have lower energy-per-special-character ratios than `STJRefGen` and `STJSrcGen` across most Unicode and UnicodeEscape intervals. This is consistent with their Rank 1 and Rank 2 positions at the highest densities. Escape again differs. `STJSrcGen` begins with the lowest energy at A0 and has ratios of 0.61×,

0.50×, 0.71×, and 0.83× across the 4 Escape intervals. `Utf8Json` has higher ratios over the same intervals: 0.69×, 0.67×, 0.93×, and 0.87×. This is consistent with `STJSrcGen` being Rank 1 and `Utf8Json` being Rank 5 at E100.

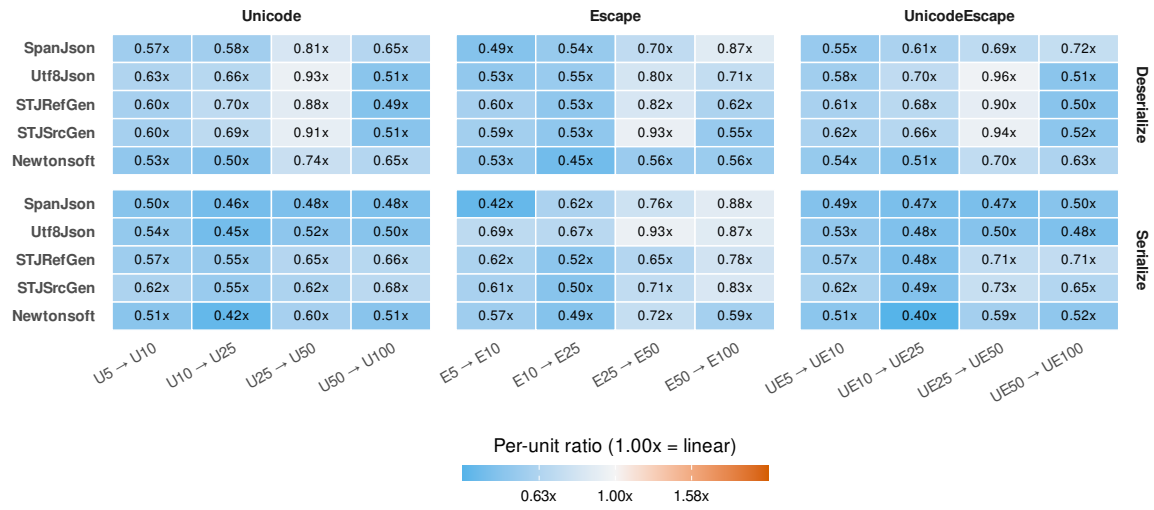


Figure 5.14: Adjacent ratio of energy per special character between consecutive String Composition levels. A value of 1.00× indicates unchanged energy per special character; values above 1.00× indicate an increase, and values below 1.00× indicate a decrease.

5.2.7 Cross-Dimension Ranking Overview

Figure 5.15 shows the per-level rankings as a mean rank for each library and workload group (a single dimension, with Numeric Length and String Composition split by variant). Numeric Length is shown separately for Integer and Float values, while String Composition is shown separately for Unicode, Escape, and UnicodeEscape. Lower mean ranks indicate that a library more often records lower energy within that workload group.

There is a clear distinction between operations. `SpanJson` records the lowest or near-lowest deserialisation energy in 8 of the 9 workload groups, while `STJSrcGen` most often records the lowest serialisation energy. The serialisation results also divide into two broad groupings: Size, Depth, Width, and String Length favour the `System.Text.Json` variants, whereas Numeric Length and the Unicode and UnicodeEscape String Composition variants favour `SpanJson` or `Utf8Json`.

`SpanJson` has the lowest mean deserialisation rank for Size, Depth, both Numeric Length variants, Unicode, and UnicodeEscape, with a mean rank of 1.00 in each group. It also remains close to Rank 1 for Width and Escape, with mean ranks of 1.14 and 1.17, while String Length is the main exception at 2.43. During serialisation, its profile changes. `SpanJson` records its lowest mean ranks for Numeric Length with Integer values, Unicode, and UnicodeEscape, with values of 1.40, 1.67, and 1.50. Its mean ranks are higher for Size, Depth, Width, and String Length, at 4.00, 3.86, 3.43, and 4.00.

`Utf8Json` has its lowest mean deserialisation ranks for both Numeric Length variants, with a value of 2.00 in each group, followed by Width at 2.71. Its mean ranks are higher for Size and Depth, both at 4.00, String Length at 3.71, and the String Composition variants, which range from 3.17 to 4.00. During serialisation, `Utf8Json` records its lowest mean rank for Numeric Length with Float values at 1.20, followed by Unicode at 2.00 and UnicodeEscape

at 2.17. Its mean ranks for Size, Depth, Width, and String Length range from 3.57 to 4.20.

`STJRefGen` and `STJSrcGen` occupy more similar positions during deserialisation than during serialisation. `STJRefGen` records its lowest deserialisation mean ranks for String Length, Depth, and Size, with values of 2.00, 2.57, and 2.80. `STJSrcGen` records its lowest deserialisation mean ranks for String Length, Size, and Depth, with values of 1.86, 2.20, and 2.43. During serialisation, `STJSrcGen` has a lower mean rank than `STJRefGen` in every workload group. The difference is largest for Size, Depth, Width, and String Length, where `STJSrcGen` has mean ranks between 1.00 and 1.29, while `STJRefGen` ranges from 2.00 to 2.29. Both variants have higher mean ranks for Unicode and UnicodeEscape serialisation: `STJSrcGen` records 3.17 and 3.00, while `STJRefGen` records 4.17 in both groups.

`Newtonsoft` usually uses the most energy. It has the highest mean deserialisation rank for Size, Depth, String Length, and both Numeric Length variants, with a value of 5.00 in each group. Its relative position improves for String Composition deserialisation, where its mean ranks range from 3.50 to 3.83. During serialisation, its lowest mean ranks occur for String Length and Size, at 3.57 and 3.80, while its highest mean ranks occur for both Numeric Length variants, at 5.00, and Escape at 4.83.

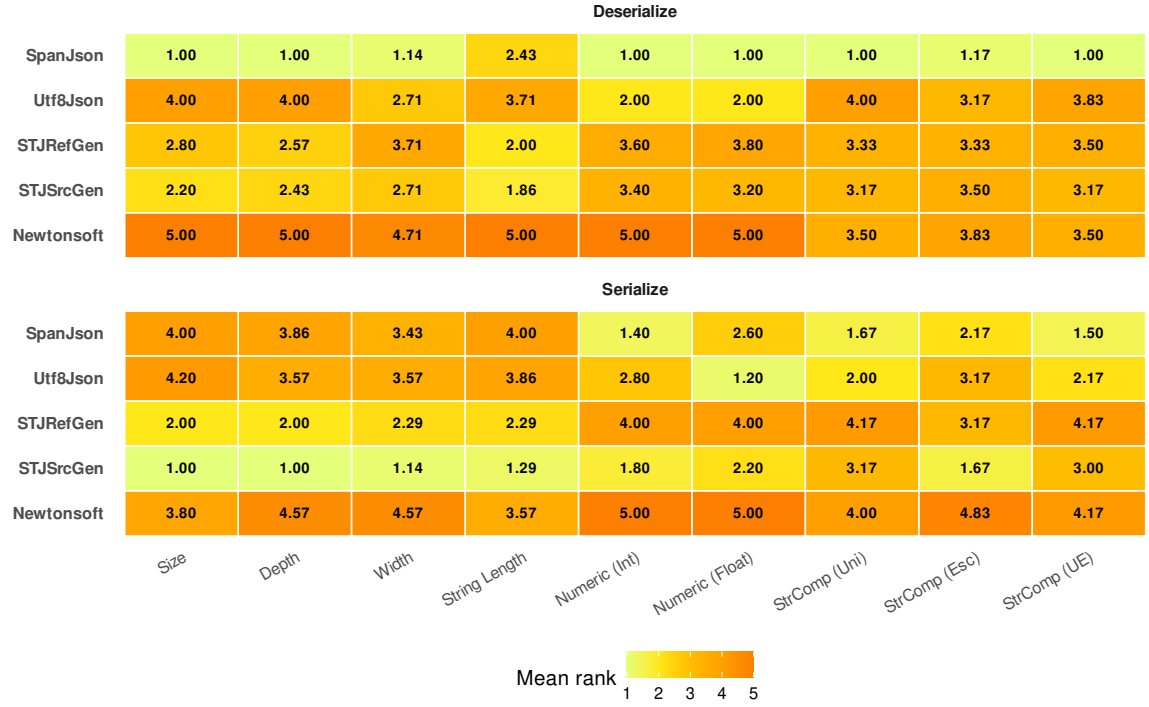


Figure 5.15: Cross-dimension mean rank by library and operation. Each cell is the mean rank across the levels of one workload group. Rank 1 corresponds to the lowest energy and Rank 5 to the highest energy at each level. Numeric Length and String Composition are shown separately by variant.

5.2.8 Cross-Dimension Scaling Overview

Figure 5.16 summarises how per-unit energy changes from the lowest to the highest level of each dimension. A value above 1.00× means that the library uses more energy per added unit as the dimension grows, a value below 1.00× means that the added cost is amortised. The relevant unit depends on the dimension: objects for Size, nesting levels for Depth, fields for Width, characters for String Length, digits for Numeric Length, and

special characters for String Composition.

The broadest pattern is that Size, Depth, and Width contain most of the ratios near or above 1.00×, while String Length, Numeric Length, and String Composition mostly have ratios below 1.00×. The extent of this difference varies by library and operation.

`SpanJson` shows its largest deserialisation increases for Width and Size, where its energy per unit ratios are 2.48× and 1.87×. Depth remains close to unchanged at 1.01×, while String Length, Numeric Length, and String Composition remain below 0.30×. During serialisation, `SpanJson` records the largest increases in the figure for Width and Depth, with ratios of 3.73× and 2.53×. Its String Length ratio is 0.96×, while Numeric Length and String Composition range from 0.05× to 0.47×. These higher Width and Depth ratios are consistent with its higher serialisation mean ranks for those dimensions.

`Utf8Json` follows a similar pattern, but with smaller increases than `SpanJson` for Width and Depth. During deserialisation, its Size and Width ratios are 1.57× and 1.48×, while Depth remains close to unchanged at 0.95×. During serialisation, its Width and Depth ratios increase to 2.92× and 1.82×, and its Size ratio is 1.31×. String Length remains close to proportional at 0.93×, while Numeric Length and String Composition range from 0.06× to 0.41×.

`STJRefGen`, during deserialisation, has its largest ratios for Size and Width, at 1.62× and 2.18×, while Depth remains close to unchanged at 1.04×. Its String Length ratio is 0.01×, and its Numeric Length and String Composition ratios range from 0.16× to 0.25×. During serialisation, `STJRefGen` remains much closer to 1.00× for Size, Depth, and Width, with ratios of 1.03×, 1.30×, and 1.19×. Its String Length ratio is 0.05×, while the remaining workload groups range from 0.13× to 0.29×.

`STJSrcGen` has a deserialisation profile close to `STJRefGen`. Its Size and Width ratios are 1.65× and 2.17×, Depth remains close to unchanged at 1.01×, and String Length falls to 0.01×. During serialisation, `STJSrcGen` remains close to unchanged for Depth and Width, with ratios of 0.88× and 1.04×, while its String Length ratio is 0.06×. Size is the main exception at 1.49×. Despite this increase in energy per object, `STJSrcGen` retains the lowest serialisation energy across the Size levels because it begins at a lower absolute energy.

`Newtonsoft` often has lower per-unit ratios than the other libraries, despite usually having higher energy consumption. During deserialisation, its Width ratio is 1.04×, compared with ratios between 1.48× and 2.48× for the other libraries. Its Depth ratio is 0.92×, while String Length, Numeric Length, and String Composition range from 0.07× to 0.26×. During serialisation, Width and Depth remain close to unchanged at 1.02× and 0.92×, while String Length, Numeric Length, and String Composition range from 0.06× to 0.29×. These lower scaling ratios help explain why `Newtonsoft`'s relative position improves in some Width and String Composition results, even though its total energy remains high.

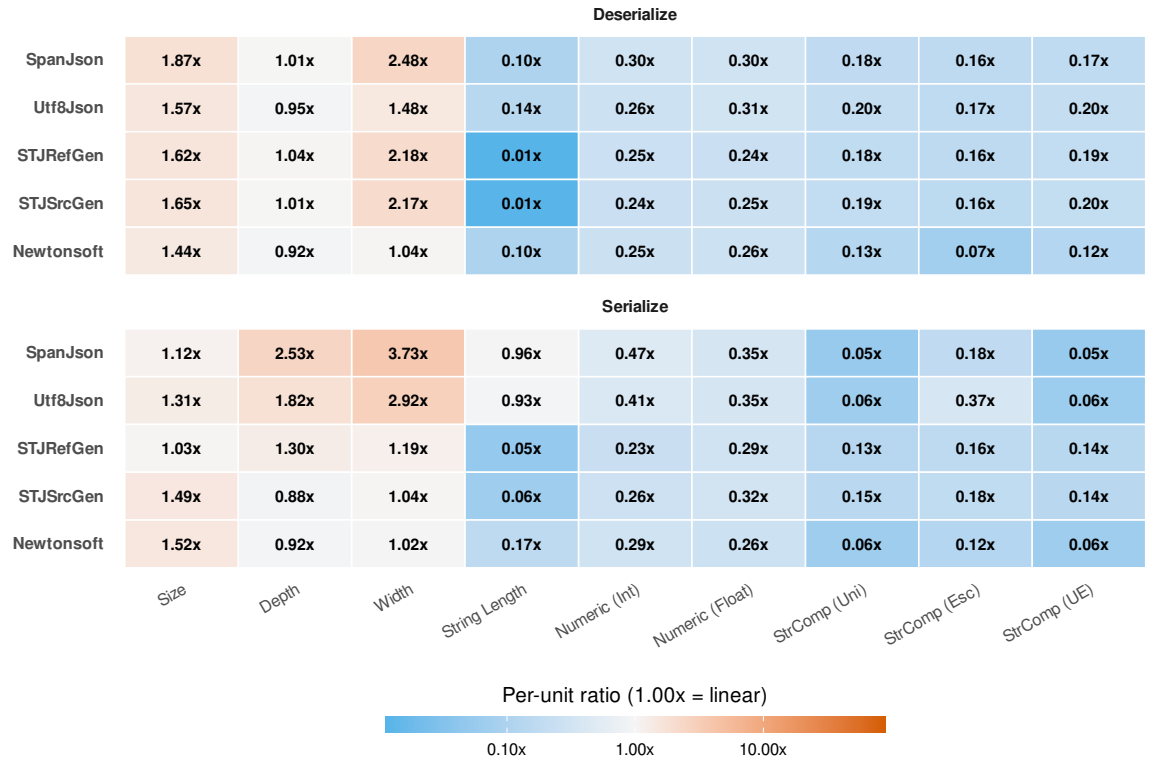


Figure 5.16: Per-unit energy ratio by library and operation. Each cell compares energy per unit at the highest level with energy per unit at the lowest level for one workload group. A value of 1.00× indicates unchanged energy per unit; values above 1.00× indicate an increase, and values below 1.00× indicate a decrease. Numeric Length and String Composition are shown separately by variant.

5.2.9 Key Findings

The isolation benchmarks show that library rankings depend on workload dimension and operation. Overall, SpanJson uses the least energy for deserialisation, STJSrcGen uses the least energy for serialisation, and Newtonsoft uses the most energy in both operations. The library ranking persists in some cases, especially Size and Depth deserialisation, but changes across Width, String Length, Numeric Length, and String Composition. The main findings are summarised below.

- **Size preserves the leading libraries while several energy gaps narrow.** SpanJson remains Rank 1 for deserialisation, and STJSrcGen remains Rank 1 for serialisation across all Size levels. The deserialisation ranking is stable, while the lower serialisation ranks change without affecting the lowest-energy library. As Size increases, several ratios to the per-level minimum decrease. For example, Newtonsoft's deserialisation ratio decreases from 2.66× at C10 to 2.04× at C100K.
- **Depth mainly affects serialisation.** Deserialisation rankings remain stable across depth, with SpanJson first and Newtonsoft last throughout. Serialisation changes more: SpanJson falls from rank 3 at D1 to rank 5 by D25, reaching 3.67× the cheapest library at D40. The per-element view shows why. SpanJson and Utf8Json become more expensive per depth level, while STJSrcGen stays close to proportional scaling.
- **Depth preserves the leading libraries but widens SpanJson's serialisation gap.** SpanJson remains Rank 1 for deserialisation and STJSrcGen remains Rank 1 for serialisation across all Depth levels. The deserialisation ranking is stable, but the lower

serialisation ranks change as Depth increases. SpanJson moves from Rank 3 at D1 to Rank 5 at D25 and D40, while its energy ratio increases from 1.27× to 3.67×. Its energy per nesting level increases by 1.66× over D4–D8, whereas STJSrcGen remains close to 1.00× across the adjacent intervals.

- **Width produces different ranking changes in the two operations.** During deserialisation, SpanJson remains Rank 1 through W100, while Utf8Json becomes Rank 1 at W200 and SpanJson has an energy ratio of 1.10×. STJRefGen and STJSrcGen reach ratios of 2.25× and 2.17× at W200 after late increases in energy per field. During serialisation, STJSrcGen and STJRefGen occupy Ranks 1 and 2 from W5 onward. At W200, SpanJson, Utf8Json, and Newtonsoft have ratios of 3.51×, 3.50×, and 3.47×, respectively.
- **String Length changes Rank 1 in both operations.** SpanJson is Rank 1 for both operations at L5. During deserialisation, STJRefGen and STJSrcGen occupy Ranks 1 and 2 from L50 onward. During serialisation, STJSrcGen is Rank 1 and STJRefGen is Rank 2 from L20 onward. Over L5–L500, the serialisation energy-per-character ratios of STJRefGen and STJSrcGen remain between 0.27× and 0.69×, while SpanJson and Utf8Json exceed 1.00× over L20–L200.
- **Numeric Length preserves the deserialisation ranking but changes the serialisation ranking.** Float values use more energy than Integer values at matched digit lengths, but the deserialisation ranking remains unchanged for both variants. During serialisation, STJSrcGen becomes Rank 1 at I7 for Integer values and at F10 for Float values. At F10, STJSrcGen, Utf8Json, and SpanJson have energy ratios of 1.00×, 1.02×, and 1.05×, respectively.
- **String Composition is variant-dependent.** In deserialisation, SpanJson is usually cheapest, but Newtonsoft improves as special-character density increases and becomes cheapest for Escape at E100. In serialisation, Unicode and UnicodeEscape favour SpanJson and Utf8Json, while the STJ pair loses its baseline advantage. Escape behaves differently: STJSrcGen largely preserves its serialisation lead, while Utf8Json falls to the bottom at E100.
- **String Composition produces variant-specific rankings.** During deserialisation, SpanJson is Rank 1 at every level except E100. Newtonsoft decreases from an energy ratio of 2.68× at A0 to Rank 1 at E100. During serialisation, SpanJson and Utf8Json occupy Ranks 1 and 2 at U100 and UE100, while STJSrcGen and STJRefGen have ratios of 3.18× and 3.59× at U100, and 2.99× and 3.65× at UE100. Escape produces a different result: STJSrcGen is Rank 1 at E100, while Utf8Json is Rank 5 with a ratio of 1.78×.
- **STJSrcGen and STJRefGen separate more during serialisation than deserialisation.** During deserialisation, the two variants generally occupy more similar positions. During serialisation, STJSrcGen has a lower mean rank than STJRefGen in every workload group. For Size, Depth, Width, and String Length, STJSrcGen has mean ranks between 1.00 and 1.29, while STJRefGen ranges from 2.00 to 2.29. Unicode and UnicodeEscape are the main cases where both variants use substantially more energy relative to SpanJson and Utf8Json.
- **Lower scaling ratios do not necessarily imply low total energy.** Newtonsoft uses the most energy in many workload groups, but its relative position improves in some cases because its energy per unit increases less than that of competing libraries. During Width deserialisation, its energy ratio decreases from 4.03× at W2 to 1.87× at W200. During Escape deserialisation, it moves from Rank 5 at A0 to Rank 1 at

E100. These cases show why rankings should be interpreted together with energy ratios and scaling results.

These findings provide the basis for the architectural analysis in §5.3, which focuses on two selected operation-level patterns: the separation between the two `System.Text.Json` variants during serialisation, and the read/write reversal between `SpanJson` and `System.Text.Json`.

5.3 RQ2: Architectural and Runtime Explanations

RQ1 showed that library rankings change across operations and workload dimensions. RQ2 examines selected cases where the results can be connected to implementation behaviour. The aim is not to explain every ranking change, but to explain the clearest patterns using library documentation, source code, generated code, allocation and garbage-collection data, and EventPipe traces inspected in SpeedScope.

The section focuses on two operation-level patterns: why `STJSrcGen` separates from `STJRefGen` mainly during serialisation, and why `SpanJson` is usually the lowest-energy deserialiser while the `System.Text.Json` variants use less energy during serialisation. The diagnostics support the interpretation by showing which code paths are active, but they are not used for exact per-method energy attribution.

5.3.1 STJSrcGen and STJRefGen Separate Only on Serialisation

RQ1 showed a clear operation split between the two `System.Text.Json` variants. During deserialisation, `STJRefGen` and `STJSrcGen` remain close and frequently exchange neighbouring ranks. During serialisation, however, `STJSrcGen` consistently uses less energy than `STJRefGen`. Because the two variants use the same library and differ only in how metadata and generated code are provided, this comparison isolates the effect of source generation.

To serialise or deserialise a type, `System.Text.Json` needs *metadata* that describes how the type is accessed: getters and fields for writing, and constructors, setters, and fields for reading. `STJRefGen` builds this metadata at runtime through reflection the first time it encounters a type and then caches the result. This discovery costs both time and memory [44].

`STJSrcGen` addresses this cost through source generation, which has two relevant modes. First, metadata mode provides the type metadata at compile time through a generated `JsonSerializerContext`, avoiding runtime reflection-based discovery [56]. Second, serialisation-optimisation mode, or the *fast path*, emits type-specific code for writing objects directly through the JSON writer instead of routing each value through the general-purpose converter used at runtime [44, 56]. The measured difference between the variants therefore depends on whether an operation only benefits from generated metadata or can also use generated write code.

For deserialisation, only metadata mode applies. `STJSrcGen` avoids runtime metadata discovery but generates no deserialisation fast path [44, 56]. Under BenchmarkDotNet, the reflection-based metadata discovery in `STJRefGen` is paid during warm-up and cached

before measurement begins. Both variants therefore use the same steady-state read path, which explains why their deserialisation energy remains close.

Serialisation differs because both source-generation modes apply: `STJSrcGen` has generated metadata *and* generated write code. Listing 5.1 shows this generated code for a benchmark node type. It writes each property with a direct `Utf8JsonWriter` call and recurses into the nested child. `STJRefGen` has no such code and writes the same values through its reflection-backed general converter at runtime, whereas source generation removes this per-property converter step.

```
// Emitted by the System.Text.Json source generator for Node20<string>
// (fully-qualified names and a redundant cast removed for readability)
private void Node20StringSerializeHandler(Utf8JsonWriter writer,
    Node20<string>? value)
{
    if (value == null) { writer.WriteNullValue(); return; }
    writer.WriteStartObject();
    writer.WriteString(PropName_key_0, value.key_0);
    writer.WriteString(PropName_key_1, value.key_1);
    // ... key_2 through key_18 ...
    writer.WritePropertyName(PropName_key_19);
    Node20StringSerializeHandler(writer, value.key_19); // nested node
    writer.WriteEndObject();
}
```

Listing 5.1: Simplified source-generated serialisation handler for the benchmark node type. The generated handler writes properties directly through `Utf8JsonWriter` and recurses for the nested child node.

EventPipe traces confirm that the code path in Listing 5.1 is used at runtime. During `STJRefGen` serialisation, the sampled stacks pass through the general converter path, including `ObjectDefaultConverter.OnTryWrite` and `JsonPropertyInfo.GetMemberAndWriteJson`. During `STJSrcGen` serialisation, those methods are absent, and the generated `SerializeHandler` appears instead. Deserialisation shows the corresponding shared path: both variants execute `ObjectDefaultConverter.OnTryRead`, with no generated read method.

5.3.2 SpanJson and System.Text.Json Swap Ranks Across Operations

The operation ranking reversal between `SpanJson` and `System.Text.Json` cannot be explained by source generation alone, since `STJRefGen` also uses less serialisation energy than `SpanJson`. Part of the explanation must lie in the shared `System.Text.Json` write path.

Both libraries are span-based and allocate comparably, so neither the use of `Span` nor garbage collection differentiates them. So the difference is in CPU work. The baseline configuration in our benchmarks is textual, so the values written are strings (§3.3), and serialisation repeatedly encodes string values as UTF-8. SpanJson performs much of this work inside `JsonWriter.WriteUtf8String`, which scans the span for characters that require escaping, transcodes to UTF-8, and copies into the output buffer, growing it as needed [57]. In `System.Text.Json` the same work is split across `Utf8JsonWriter` and several helpers, such

as `WriteStringMinimized`, `OptimizedInboxTextEncoder`, `IndexOfAnyExcept`, `IndexOfAnyAsciiSearcher`, and `Utf8Utility.TranscodeToUtf8`.

The .NET runtime sources for `IndexOfAnyAsciiSearcher` and `Utf8Utility.TranscodeToUtf8` show that these helper paths include vectorised implementations where supported [58, 59]. Vectorisation, or SIMD, allows one instruction to operate on multiple bytes or characters at once, which can reduce the cost of repeated scanning and transcoding in string-heavy output paths [60]. So a plausible explanation could be that during serialisation, the `System.Text.Json` writer does less work on this string-heavy output than `SpanJson`. Source generation then gives `STJSrcGen` an additional advantage over `STJRefGen`.

During deserialisation, the trace evidence is less conclusive. `SpanJson` reads each type with a reader specialised to that type, whereas the `System.Text.Json` variants still proceed through the general reader, converter, and property-setting pipeline (§5.3.1). The trace comparison therefore suggests that `SpanJson` may do less general per-property dispatch during reading, which is consistent with its lower deserialisation energy.

5.4 RQ3: Environmental Effects and Metrion Validation

5.4.1 Metrion Validation

For validating that Metrion Profiler provide similar results when compared to the Energy Diagnoser, we have used the Smoke Benchmarks suite. We run each experiment 5 times with the Energy Diagnoser and 5 times with the Metrion Profiler, in the noisy environment with 0% stress. As mentioned before, we stick to only 5 times due to time limits. Over the 5 results, we computed the mean energy and compared the rankings using Sperman's Rank Correlation and Magnitude Agreement (§ 3.5.1, § 3.5.2).

5.4.1.1 Ranking Agreement

To verify that Metrion Profiler orders libraries consistently with Energy Diagnoser, we rank the five libraries by mean energy per operation under each diagnoser (rank 1 = lowest energy) and correlate the two orderings with Spearman's ρ. Deserialization rankings are identical across tools ($\rho = 1.0$; Figure 5.17, left). Serialization differs by a single adjacent swap ($\rho = 0.9$): Utf8Json swaps places with SpanJson. This swap is not surprising, as the next section shows a very small difference between the two libraries. Figure 5.18 shows that, for Energy Diagnoser, the normalized ratio difference between Utf8Json and SpanJson is only 0.02. This suggests that the serialization ranking obtained with Energy Diagnoser may be affected by between-run variance, since the values for the two libraries are very close to each other.

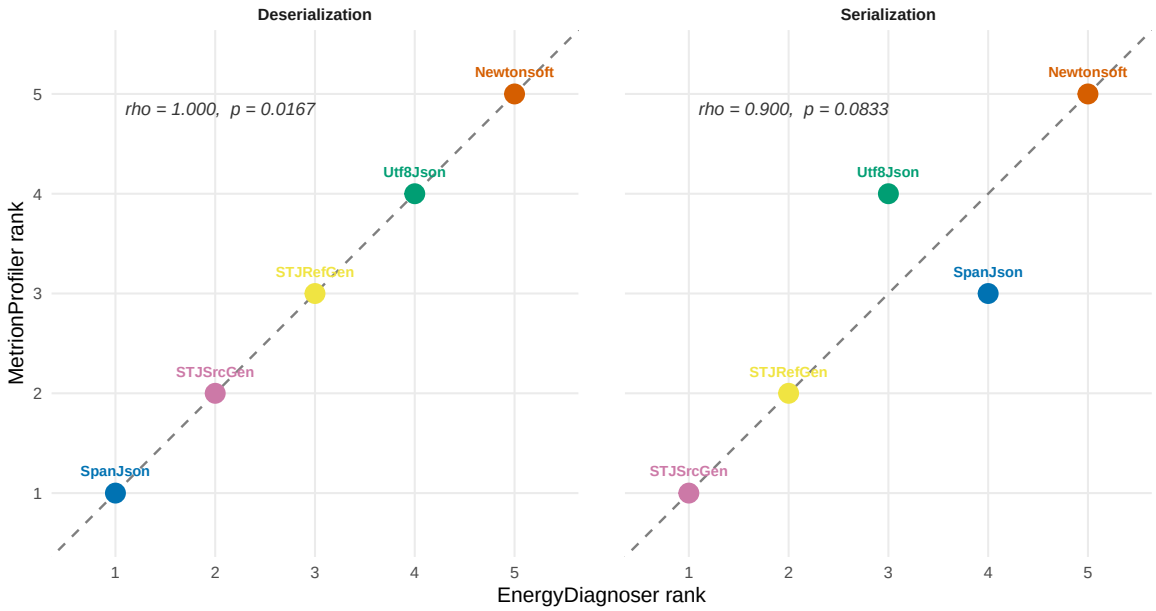


Figure 5.17: Rank agreement between Energy Diagnoser and Metrion at noisy environment with 0 % stress. Each point is one library; points on the dashed diagonal indicate identical rank under both tools.

5.4.1.2 Magnitude Agreement

Figure 5.18 shows the normalized energy for every library under both diagnosers. The two instruments reconstruct essentially the same relative-energy structure — both place Newtonsoft far above the rest (2.61× vs. 2.69× for deserialization; 3.18× vs. 3.24× for serialization) and group the System.Text.Json variants tightly together.

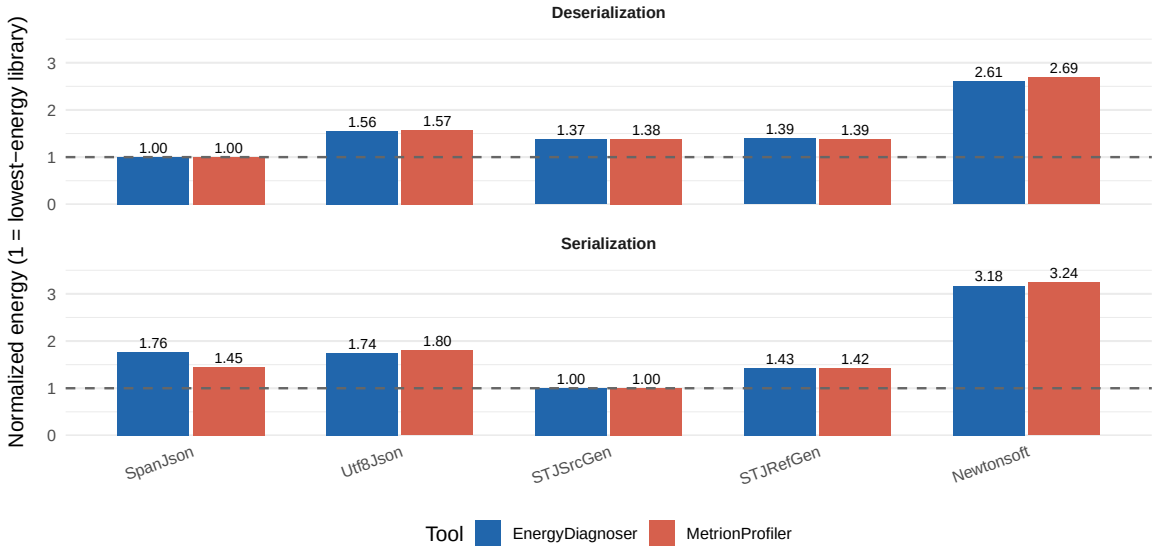


Figure 5.18: Normalized energy per library at noisy environment with 0 % stress. Each library's mean energy is normalised to the lowest-energy library within the same tool and operation. Matching bar heights indicate that the two tools agree on the relative energy gap.

The lone exception is SpanJson during serialization, visible in Figure 5.19 ($\Delta = +0.31$). Here Energy Diagnoser rates SpanJson as roughly equivalent to Utf8Json (1.76× vs. 1.74×

the best library), whereas Metrion rates it more efficient (1.45× vs. 1.80×). This is the same SpanJson–Utf8Json pair responsible for the single rank inversion observed in the Spearman analysis ($\rho = 0.90$). The convergence is informative: under Energy Diagnoser these two libraries are nearly tied, so even a small measurement discrepancy is sufficient to reorder them. The disagreement is therefore both localised and explicable.

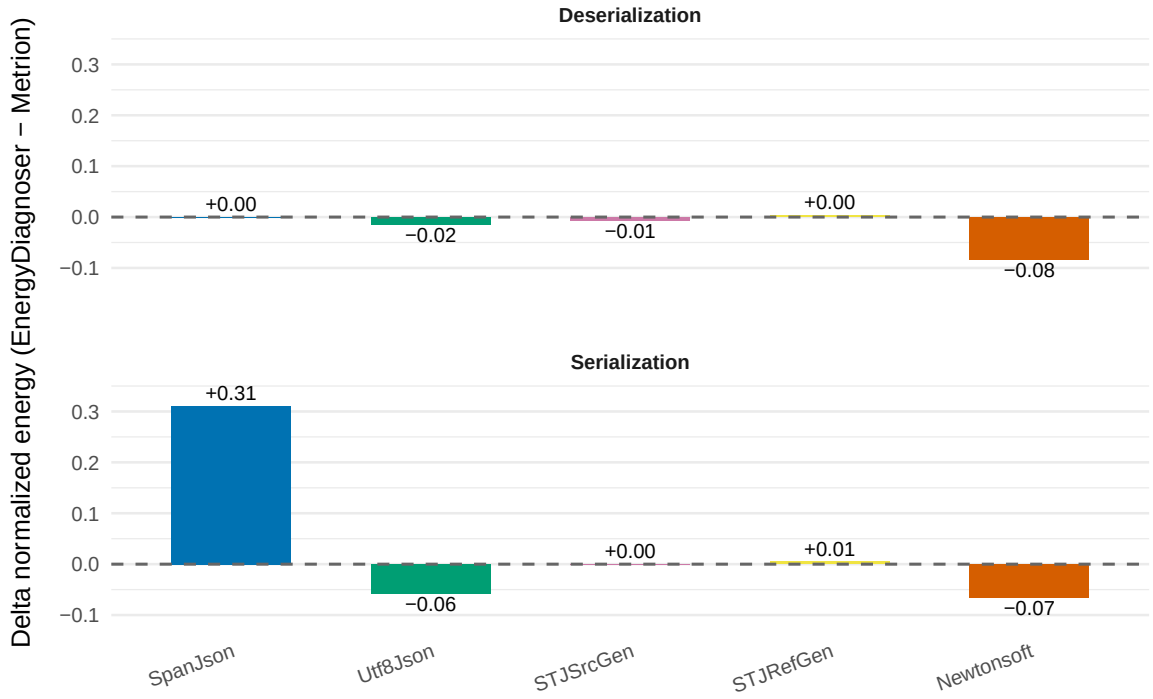


Figure 5.19: Difference in normalized energy, $\Delta = r_{\text{Energy Diagnoser}} - r_{\text{Metrion Profiler}}$, per library and operation. Values near zero denote agreement.

Taken together with the ranking results, the magnitude comparison supports the conclusion that Metrion Profiler reproduces Energy Diagnoser's view of the workload closely: the two tools agree on both the order and the relative spacing of the libraries, with a single exception arising where two libraries are already statistically close. This strengthens the case for treating Metrion Profiler as a viable substitute for Energy Diagnoser on the noisy environment.

5.4.2 Environmental effects

This section presents an analysis of how different noisy environment conditions affect the ranking of the Isolation Benchmark suite used in the context of RQ1 (§ 5.3). A subset of benchmarks is used: dimensions Depth, Width, and Size. We compare the rankings across five CPU stress levels: 0%, 25%, 50%, 75%, and 100%. For each stress level, five runs are executed, and the mean energy across runs is used. To improve readability, the results are condensed into a single graph for each benchmark dimension. More detailed figures are provided in Appendix C.

5.4.2.1 Depth effects

Figure 5.20 shows the mean ranks for depth level across different environment stress conditions. The central finding is that library rankings are highly stable under environmental stress for both operations.

For deserialization, `SpanJson`, `Utf8Json`, and `Newtonsoft` maintain stable rankings across all stress levels. `SpanJson` consistently remains in first place, making it the most energy-efficient library regardless of nesting depth or environmental conditions. `Newtonsoft` consistently ranks last, while `Utf8Json` consistently occupies fourth place. Both `System.Text.Json` variants tend to alternate between second and third place depending on the system stress. `STJRefGen` more frequently ranks third across the different stress levels, whereas `STJSrcGen` more often ranks second. At the 25% stress level, all libraries maintained the same ranking.

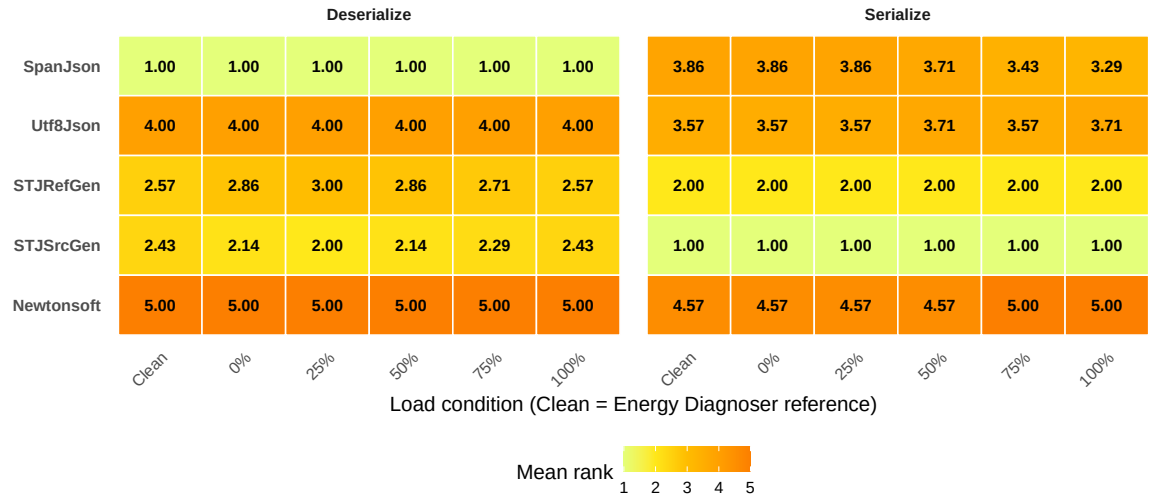


Figure 5.20: Mean energy rank of each library across background stress conditions, averaged over all nesting depth levels. Rank 1 indicates the most energy-efficient library on average. Consistency across stress levels indicates that nesting depth does not alter the relative library ordering under environmental stress.

For serialization, both `System.Text.Json` variants exhibit stable rankings across all stress levels. `STJSrcGen` consistently remains in first place, while `STJRefGen` consistently ranks second. `Newtonsoft` tends to perform worse at higher stress levels (75% and 100%), suggesting that it is more sensitive to environmental noise during serialization. In contrast, `SpanJson` appears to improve as the stress increases from 0% to 100%, starting closer to fourth place and ending closer to third place.

As an overview, the rankings obtained using the clean environment with Energy Diagnoser closely align with those produced by Metrion Profiler at 0% noise across all operations and libraries. This supports the use of Metrion Profiler as a valid ranking tool and is consistent with the validation results presented in the previous section.

Full per-level rank trajectories (Figure E.2), rank heatmaps (Figure E.3), and stress sensitivity curves (Figure E.1) for the depth dimension are provided in the Appendix.

5.4.2.2 Size effects

Figure 5.21 shows the mean ranks for size level across different environment stress conditions. The overall picture is one of ranking stability.

For deserialization, `SpanJson` maintains the first rank across all stress levels, except at 75% stress, where its mean rank increases to 1.60. In contrast, `Utf8Json` improves from rank 4 to a mean rank of 3.40 at the same stress level, suggesting a partial rank swap between the two libraries under 75% stress. Figure E.5 in the Appendix shows that `Utf8Json` overtakes `SpanJson` only for size C10K, indicating that this change is not persistent. At 100% stress, both libraries return to their baseline rankings. `STJSrcGen` and `STJRefGen` consistently occupy the second and third positions, with mean ranks ranging from 2.00–2.40 and 2.60–3.00, respectively. `Newtonsoft` remains in last place across all conditions, with mean ranks between 4.80 and 5.00.

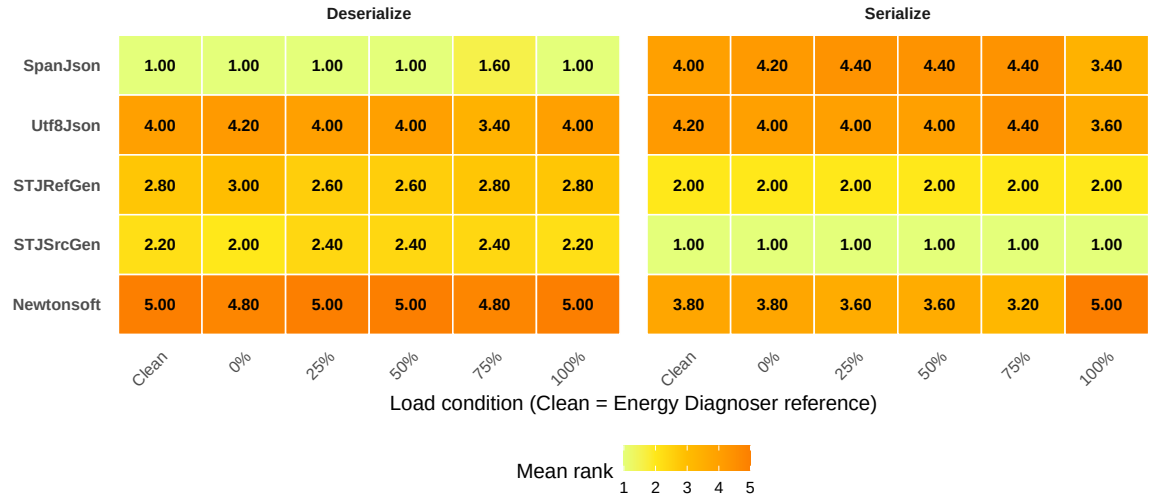


Figure 5.21: Mean energy rank of each library across background stress conditions, averaged over all Size levels. Rank 1 indicates the most energy-efficient library on average. Stable rankings across stress levels suggest that Size does not interact with environmental noise in a way that disrupts library ordering.

For serialization, the top two positions are consistently occupied by the two `System.Text.Json` variants. `STJSrcGen` maintains first place, while `STJRefGen` consistently ranks second across all stress levels. The most notable observation is that `Newtonsoft` tends to improve its position, with its mean rank decreasing from 3.8 to 3.2 between 0% and 75% stress. However, under maximum stress, its ranking deteriorates, making it the most energy-consuming library at the 100% stress level. In contrast, `SpanJson` and `Utf8Json` exhibit the opposite trend: their mean ranks improve from 4.40 at 75% stress to 3.40 and 3.60, respectively, at 100% stress.

Full per-level rank trajectories (Figure E.5), rank heatmaps (Figure E.6), and stress sensitivity curves (Figure E.4) for the size dimension are provided in the Appendix.

5.4.2.3 Width effects

Figure 5.22 shows the mean ranks for width level across different environment stress conditions. The width dimension produces the most dynamic behaviour of the three dimensions studied.

For deserialization, `SpanJson` leads across all conditions, although its position is not as stable as it is for depth. Its mean rank fluctuates between 1.00 and 1.57 depending on the stress level. The most notable behavior is observed for `Utf8Json` and `STJSrcGen`, which share similar rankings across all stress levels, indicating frequent rank ties across width levels rather than a clear separation. Figure E.8 in the Appendix shows that `Utf8Json` ranks behind `STJSrcGen` for W10 to W50 across all stress levels, and the opposite is observed for the remaining width levels, which may explain this behavior. `STJRefGen` consistently occupies a mean rank between 3.29 and 3.71, while `Newtonsoft` remains the most energy-consuming library, reaching a mean rank of 5.00 at 25% stress and returning to 4.71 at the other stress levels.

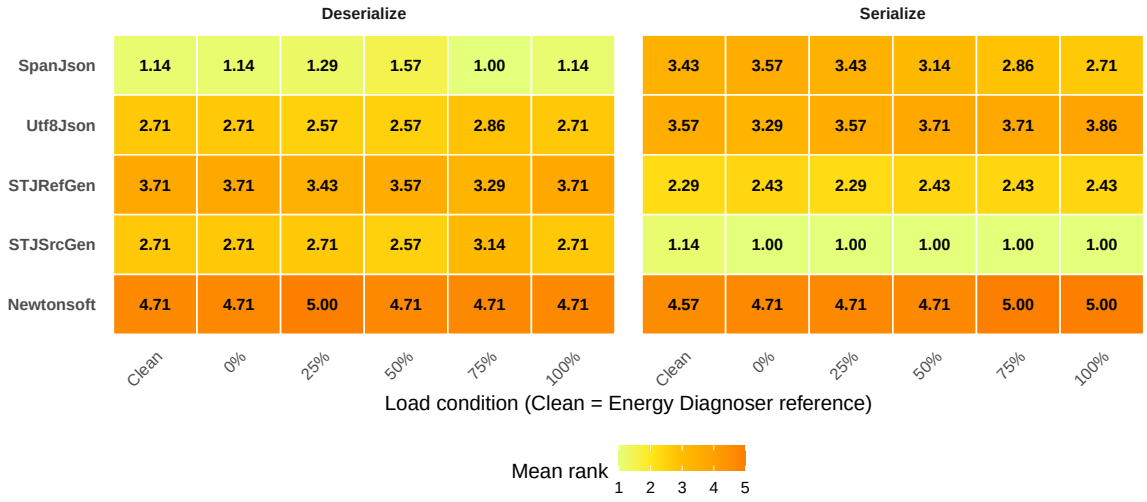


Figure 5.22: Mean energy rank of each library across background stress conditions, averaged over all width levels. Rank 1 indicates the most energy-efficient library on average. Stable values across stress conditions suggest rankings are robust to environmental noise for this dimension.

For serialization, `STJSrcGen` consistently maintains the first position across all stress levels. The most notable effect is observed for `SpanJson`, whose ranking improves as the stress level increases, with its mean rank decreasing from 3.57 at 0% stress to 2.71 at 100% stress, representing a shift of nearly one full rank position. `Utf8Json` exhibits the opposite trend, with its mean rank increasing from 3.29 to 3.86. `Newtonsoft` again consumes the most energy, reaching a mean rank of 5.00 at both 75% and 100% stress levels.

Full per-level rank trajectories (Figure E.8), rank heatmaps (Figure E.9), and stress sensitivity curves (Figure E.7) for the Width dimension are provided in the Appendix.

Chapter 6

Discussion

This chapter steps back from the empirical findings of Chapter 5 and reflects on the study as a whole. §6.2 groups general reflections on design choices that worked, design choices that did not produce the expected signal, and aspects of the experimental setup that constrained the conclusions that can be drawn. §6.4 sketches concrete follow-on work, including a redesigned probe for the Redundancy dimension that the present design was unable to characterise.

6.1 Synthesis and Implications

This section brings together the answers to the three research questions and discusses what they imply for JSON-library benchmarking and library selection.

6.1.1 Workload-dependent Energy Rankings

RQ1 asked how library energy rankings and scaling rates differ across workload dimensions. The main result is that the rankings are workload-dependent, but not arbitrary. Across most configurations, `SpanJson` records the lowest deserialisation energy, `STJSrcGen` records the lowest serialisation energy, and `Newtonsoft` usually uses the most energy. These are useful defaults, but they do not hold uniformly. Rankings can change within a single dimension as the level increases. For example, `SpanJson` leads String Length deserialisation at short strings but is overtaken by the `System.Text.Json` variants at longer strings (§5.2.4) while `Utf8Json` becomes the lowest-energy deserialiser only at the widest Width level (§5.2.3). These cases show that a library's position at one level does not necessarily predict its position at another, and scaling behaviour matters.

The practical implication is that library choice should start from the operation mix and then be refined by workload shape. Serialisation and deserialisation have different lowest-energy libraries, so knowing whether an application is read-heavy or write-heavy already points to different candidates. Workload shape matters because long strings, wide objects, and deep nesting are reasons the rankings can change. String Length is especially important because the other isolation dimensions use a short string baseline. `SpanJson`'s deserialisation lead is clear at the baseline length, but it disappears by L50 and reverses

at longer levels (§5.2.4). Therefore, a claim that one library is the lowest-energy option is only meaningful if the operation and relevant workload characteristics are fixed, which in practise can be difficult to determine.

The importance of library choice also depends on the size of the energy gap. Some rank changes occur between near-ties and are unlikely to matter in practice. Others coincide with large ratio differences or steep scaling changes, where library choice can have a substantial energy effect. For benchmarking, this means that a representative JSON-library comparison should not rely on a single baseline payload. It should vary the relevant workload dimensions and report both rankings and energy ratios, because both the ordering and the size of the differences can change across the dimension range.

6.1.2 Operation-specific Library Mechanisms

RQ2 asked which architectural and runtime factors explain the observed differences. A clear pattern is the difference between the read and write paths. `System.Text.Json` source generation changes serialisation more than deserialisation, because generated code is available for the serialisation path but not for the deserialisation path. This explains why `STJSrcGen` and `STJRefGen` remain close during deserialisation but separate during serialisation. The comparison with `SpanJson` further suggests that low-energy deserialisation and low-energy serialisation depend on different implementation choices.

The practical implication is that source generation should be used when serialising known model types with `System.Text.Json`, especially in write-heavy workloads, while the reflection-based resolver remains the option that needs no compile-time setup. However, source generation should not be assumed to provide the same benefit for deserialisation. For read-heavy workloads, the results suggest that alternatives such as `SpanJson` can remain among the lowest-energy options, depending on the workload shape. Though overall, when the workload mix is unknown, `System.Text.Json` is the most defensible default, as it is rarely the highest-energy option.

6.1.3 Environmental Measurement and Ranking Stability

RQ3 investigated how environmental conditions affect library energy rankings and whether Metrion can provide consistent measurements across them. The results show that library energy rankings are robust to environmental stress across the depth, size, and width benchmark dimensions. In a few isolated cases, partial rank swaps were observed, most notably a rank swap between `SpanJson` and `Utf8Json` at 75% load for a single size level, and occasional alternations between `STJSrcGen` and `STJSrcRef`.

These findings suggest that rank-based comparisons between libraries are tolerant to environmental noise under the tested stress conditions. Although slight variations in the measurements occur, the relative ordering of the libraries generally remains unchanged. In the few cases where rankings shift, the affected libraries are those with the smallest differences in energy consumption.

A profiler that preserves library rankings across the full range of background load levels tested in this study can be used outside perfectly isolated conditions. Combined with the agreement between Metrion Profiler and Energy Diagnoser at 0% stress, shown in

the previous section, these results support treating Metrion Profiler as a valid ranking instrument in noisy environments.

6.1.4 Considerations beyond energy

Energy is only one part of library selection. Production adoption also depends on maintenance, runtime compatibility, feature coverage, and migration cost. These factors are uneven across the libraries studied. For serialisation, the energy and maintenance results align. `STJSrcGen` most often records the lowest serialisation energy and is also the maintained in-box option for modern .NET. The main trade-off appears on deserialisation. `SpanJson` most often records the lowest deserialisation energy, and `Utf8Json` records the lowest energy at the widest objects, but both projects are archived and read-only: `SpanJson` since 23 July 2025 [46] and `Utf8Json` since 16 May 2022 [47]. Choosing them for energy reasons therefore means accepting the risk of an unmaintained dependency, while `System.Text.Json` remains the maintained fallback. This risk is visible in the present study, as `Jil` was considered as a candidate library but was excluded because it no longer worked on the .NET version used here (§6.3).

`Newtonsoft` represents the opposite trade-off. It is mature, broadly used, and feature-rich, but it also uses the most energy in most of the measurements. Moving from `Newtonsoft` to `System.Text.Json` may therefore improve energy use as well as maintenance alignment with the .NET platform. However, such a migration is not purely mechanical. The two libraries also differ in defaults, supported features, and compatibility behaviour.

6.2 Reflections

This section reflects on the methodological choices that shaped the final study. It focuses on how the benchmark suites, workload dimensions, runtime diagnostics, and scope decisions changed during the project, and how those choices affect the interpretation of the results.

6.2.1 Role of the Factorial and Isolation Benchmark Suites

The factorial benchmark suite was originally intended as a second analytical lens alongside the isolation benchmark suite. In practice, its role became more descriptive. The $4 \times 4 \times 3$ grid is useful because it gives a compact overview of library behaviour across combinations of Depth, Width, and Content type. The rank-composition and library-positioning figures show the broad library ordering and the spread between libraries before the more detailed per-dimension analysis begins.

The isolation benchmark suite became the main analytical basis for RQ1. Varying one workload dimension at a time made it possible to relate ranking changes and scaling behaviour to specific characteristics such as Width, String Length, Numeric Length, and String Composition. This was harder to do with the factorial suite because several characteristics changed at once. The factorial suite therefore works best as a landscape view, while the isolation suite carries the main explanatory weight.

6.2.2 Workload Dimensions and Scope Decisions

Some workload dimensions changed role during the study. Content type in the factorial benchmark suite (Textual / Numeric / Boolean) was originally treated as a workload dimension on the same footing as Depth and Width. In the final analysis, it is used more descriptively because it compares all-textual, all-numeric, and all-Boolean workloads across the Depth $\times$ Width grid. This is a coarse comparison. The variants differ not only in value type but also in the number of bytes written per value, so the result combines content effects with workload-size effects. Fully textual, numeric, or Boolean documents are also less representative of many real JSON payloads, which often mix strings, numbers, Booleans, and nulls within the same document.

Some of the questions originally associated with Content type are covered in isolation benchmarks. First, Numeric Length separates integer and floating-point values at matched digit lengths, while String Composition isolates variation inside string values, including ASCII, Unicode, and escape-heavy inputs. What remains is the broader cross-type question: whether, at comparable byte size, a number-heavy document uses less energy than a text-heavy or Boolean-heavy one. Answering that cleanly would require a different generator design, either by fixing the byte budget across content types or by generating controlled mixtures of strings, numbers, Booleans, and nulls. The current generator does not support this kind of graded content mix, so Content type is retained as part of the descriptive factorial overview but is not included in the per-dimension RQ1 analysis.

Redundancy dimension was also removed from the final results. It was included in the original design, but it did not produce a clear energy effect in the generated workloads. In hindsight, the benchmark structure was not well suited, and the workloads were possibly too small to show any value-reuse path the libraries might have. Keeping the dimension would have added length without adding a strong finding, so it was dropped from the main analysis.

6.2.3 Breadth, Diagnostics, and Explanatory Limits

The broad coverage of workload dimensions was necessary to answer RQ1, but it limited how far RQ2 could go. The RQ1 results raise many explanatory questions involving string handling, object traversal, allocation and runtime-specific optimisations. Explaining all of them would require a much deeper profiling study. RQ2 therefore focuses on a small number of mechanisms where the evidence was strongest, especially `System.Text.Json` source generation and selected read/write path differences.

Runtime diagnostics were most useful when treated qualitatively. EventPipe traces helped identify which implementation paths were active, such as the generated serialisation handlers in `STJSrcGen`. Allocation and garbage-collection data helped rule out memory behaviour as the main explanation in some cases. However, the traces were not used to assign precise energy costs to individual methods. Sampling, inlining, and aggregation make that level of attribution unreliable in this study.

Some tooling was considered but not used in the final analysis. BenchmarkDotNet's DisassemblyDiagnoser was integrated for possible low-level inspection, but the output was verbose, architecture-specific, and difficult to connect directly to energy results. Source-level review, allocation data, and EventPipe sampling provided enough evidence for the

selected RQ2 explanations. Instruction-level analysis remains a useful direction for future work.

The dual-environment setup also required a scope trade-off. The clean environment provides the main basis for the library-ranking analysis. The noisy Metrion environment addresses RQ3, but only for a subset of the benchmark suite because full coverage would have required substantially more measurement time. The noisy-environment results should therefore be read as an exploratory comparison of ranking stability and attribution-based measurement, not as a complete repetition of RQ1 under noise.

6.2.4 Use of AI Tools

AI tools were used in a supporting role during the project. They assisted with bash and R scripting, figure generation, grammar and phrasing, integration-test scaffolding, and exploration of implementation ideas. The experimental design, benchmark execution, data analysis, interpretation, and conclusions are the authors' own.

6.3 Threats to Validity

This section discusses threats to validity that affect how the results should be interpreted. Where possible, threats were mitigated through the experimental design, benchmark configuration, and statistical analysis.

6.3.1 Construct Validity

Construct validity concerns whether the measurements capture the intended concept. In this study, energy is measured as the sum of the Package and DRAM RAPL domains. These domains capture CPU- and memory-related energy available through RAPL, but they do not represent whole-system energy. Components such as storage, networking, motherboard power, cooling, and power-supply losses are outside the measured scope.

The benchmark suite also measures JSON serialisation and deserialisation in isolation. This makes library-level comparison possible, but it does not capture all costs present in a production application, such as request handling, database access, logging, network I/O, or application-level processing. The results should therefore be interpreted as controlled library-level measurements rather than total application energy.

The workloads are synthetic. This was necessary to vary Size, Depth, Width, String Length, Numeric Length, and String Composition independently, but it also means that the generated documents do not fully represent production JSON. In particular, the generated nesting structure is object-based and does not cover array-only or mixed object–array nesting. Real payloads may also contain more irregular schemas, optional fields, mixed content types, and less predictable property distributions.

Another construct threat relates to input representation. The benchmarks use UTF-8 `byte[]` inputs and outputs to match the native byte-oriented APIs of `System.Text.Json`, `SpanJson`, and `Utf8Json`. `Newtonsoft.Json` does not expose an equivalent UTF-8 byte API,

so its measurements include explicit UTF-8 transcoding inside the timed region. This keeps the benchmark representation consistent across libraries, but it also means that part of the measured `Newtonsoft.Json` cost comes from API mismatch rather than JSON processing alone. Trace inspection indicates that this overhead is workload-dependent. It is small in some workloads, but substantial for long-string serialisation. The `Newtonsoft.Json` results should therefore be interpreted as the cost of using the library in this byte-oriented benchmark harness, not as an isolated measure of its parser or writer alone.

6.3.2 Internal Validity

Internal validity concerns whether the observed differences are caused by the factors under study rather than by measurement artefacts. Energy measurements are sensitive to background activity, CPU frequency changes, scheduling, thermal state, and operating-system noise. The clean environment mitigates these risks through core pinning, disabled SMT and Turbo Boost, fixed CPU frequency, reduced background services, and disabled C-states. BDN further provides warm-up, repeated measurement, and process isolation. These controls reduce variance, but they do not remove all noise.

Thermal variation was monitored in the clean environment and remained small, varying by roughly 1,° C across runs (§A). Thermal drift is therefore unlikely to explain the clean-environment rankings, although it cannot be ruled out completely.

The noisy-environment experiment has additional limitations. The memory stressor is not used at its full write intensity. Continuously writing to memory would increase CPU utilisation and interfere with the configured CPU stress level, so the memory stressor allocates memory and keeps it resident rather than continuously modifying it. This means that the noisy environment increases memory pressure, but does not fully represent continuous memory activity. Regarding the repeatability across runs, due to time constraints, we did not manage to repeat the experiments for a satisfactory amount of time. All Metrion Profiler runs performed for validation and for evaluating environmental effects in the noisy environment were repeated five times. Additionally, we used a constant stress level throughout each run rather than a varying one. While this approach provides controlled experimental conditions, a fluctuating stress level may better reflect a production environment, where system load naturally changes.

6.3.3 External Validity

External validity concerns how far the findings generalise beyond the experimental setup. The study uses one hardware platform, one operating-system configuration, one .NET runtime version, and one set of library versions. Results may differ on other CPUs, memory hierarchies, operating systems, .NET versions, or future library releases. This is especially relevant for source generation and string handling, where runtime and JIT improvements can change library behaviour.

The results are also limited to the selected library set. The five libraries are sufficient to show that energy rankings change with workload dimensions, but they do not represent every .NET serialisation option. Some high-performing third-party libraries also have reduced practical relevance because they are no longer actively maintained. For example,

`SpanJson` and `Utf8Json` are archived, while `Jil` was excluded because it was not compatible with the .NET version used in this study.

The isolation benchmark suite varies one dimension at a time while holding the others at fixed baseline values. This improves interpretability, but it means that each dimension is studied at a particular operating point. String Length is the clearest example. Most dimensions use a short string baseline, while the String Length benchmark shows that rankings can change substantially at longer values. Therefore, the frequent deserialisation advantage of `SpanJson` across other dimensions is conditioned on the short-string baseline and should not be assumed to hold for long-string workloads.

Metrion's Python components introduce overhead, but since energy is attributed per-process via CPU metrics, that overhead is assigned to Metrion's own process rather than the benchmark. The authors describe the current implementation as initial [32], with Linux and Intel-specific extensions, leaving room for broader platform support and attribution refinements in future versions.

6.3.4 Conclusion Validity

Conclusion validity concerns whether the analysis supports the conclusions drawn from it. The study performs many comparisons across libraries, operations, and workload configurations. This is mitigated by using non-parametric tests, Holm-adjusted Kruskal–Wallis tests, and Cliff's δ effect sizes. However, statistical significance does not by itself imply practical importance, so the Results chapter interprets test outcomes together with energy ratios and scaling behaviour.

Rank-based summaries also have limits. A rank change can represent either a large energy difference or a near-tie, and a stable rank can hide changes in the size of the gap between libraries. This is why the analysis reports rankings together with ratio-to-minimum and per-unit scaling figures.

The per-unit scaling comparisons also depend on the selected level ranges. Levels were chosen from available prior-work bounds, extended for the isolation benchmarks, and spaced roughly logarithmically. They were not selected through a separate calibration study. As a result, transition points may fall between sampled levels, and comparisons between dimensions only apply to the level ranges tested here, not to all possible JSON workloads.

Finally, the correctness of the generated output is also a threat in serializer benchmarking. To reduce this risk, the benchmark suite includes integration tests that validate both serialisation and deserialisation for each generated JSON file and each library. The tests check that a payload can pass through the library-specific serialise–deserialise path and still represent the same document. This guards against a library appearing faster or lower-energy because it silently omits part of the input or output. The check was motivated by earlier experience with `Jil`, where a large document was not deserialised completely, but no exception was thrown.

6.4 Future Work

This thesis leaves several directions for future work. The most immediate is to extend the workload design with cases that were reduced or removed from the final analysis, and to combine the synthetic benchmark suite with more realistic workloads

6.4.1 More Realistic and Refined Workload Designs

The synthetic workloads were useful because they isolated individual workload dimensions, but future work should combine them with real-world JSON documents. A curated corpus of API responses, configuration files, and data-exchange documents would show whether the ranking changes observed in the synthetic benchmarks also appear when several dimensions vary together in less regular documents.

Content type should also be revisited with a stronger design. In this thesis, Content type is limited to all-textual, all-numeric, and all-Boolean variants in the factorial benchmark suite. A more useful design would either hold total byte size fixed across content types or generate controlled mixtures of strings, numbers, Booleans, and nulls. This would make it possible to separate value-type effects from payload-size effects and to study more realistic mixed-content documents.

Redundancy also deserves a more targeted probe. The removed Redundancy dimension did not show a clear energy effect in the generated workloads, but the design may not have exercised the mechanisms that repeated values could affect. Future work could vary the size of the unique-value pool while holding total document size fixed, or run larger documents where cache and working-set effects are more likely to appear. This would test whether repeated values influence energy through locality, allocation behaviour, or library-specific reuse mechanisms.

Finally, future workload designs could extend the nesting structure beyond the object-only pattern used here. Array-only and mixed object-array nesting may exercise different library code paths, especially where libraries handle arrays and objects through separate optimised routines.

6.4.2 Deeper Architectural Explanations

RQ2 focused on a small number of mechanisms where the evidence was strongest. Several ranking changes remain open, including the high-Width behaviour of the `System.Text.Json` variants, the String Length crossover, and the sharp Depth serialisation increase for `SpanJson`. These questions would require deeper profiling than was possible here.

A follow-up study could combine EventPipe traces with hardware performance counters, disassembly, allocation analysis, and targeted microbenchmarks. This would make it possible to separate costs from string escaping, UTF-8 transcoding, property lookup, object construction, buffer growth, garbage collection, and SIMD use. BDN's DisassemblyDiagnoser, which was considered but not used in the final analysis, would be useful for this kind of instruction-level follow-up.

6.4.3 Expanded Noisy-environment Validation

In this thesis, the Metrion-based measurements cover only a subset of the benchmark suite, so they cannot show whether ranking stability holds across all workload dimensions. Future work should repeat a larger part of the isolation benchmark suite under noisy conditions and vary both the type and intensity of background stress. It should also compare Metrion with other attribution-based tools discussed in the related work, such as Scaphandre, SmartWatts, DEEP-mon, or EnergAt. This would make it possible to separate three questions that are only partially addressed here: whether library rankings remain stable under noise, whether Metrion preserves rankings relative to direct RAPL measurement, and whether different attribution models produce consistent conclusions.

6.4.4 Generalisability Across Platforms and Libraries

Finally, the results should also be tested across more platforms. This study uses one machine, one operating-system configuration, one .NET runtime version, and one set of library versions. Repeating the benchmark suite across different CPUs, memory configurations, operating systems and .NET versions would show which findings are stable and which depend on the specific hardware or runtime environment.

Chapter 7

Conclusions

This thesis investigated the energy consumption of .NET JSON libraries in a microbenchmark setting. It extended previous work by introducing controlled JSON workload variation, a more controlled measurement environment, runtime diagnostics, and an exploratory noisy-environment comparison using Metrion.

RQ1 asked how library rankings and scaling rates differ across JSON workload dimensions. The results show that energy rankings are workload-dependent. In general, `SpanJson` uses the least energy for deserialisation, `STJSrcGen` uses the least energy for serialisation, and `Newtonsoft.Json` usually uses the most energy. However, these rankings are not fixed. They change with Width, String Length, Numeric Length, and String Composition. This means that a single JSON document or baseline workload is not enough to characterise library energy behaviour. Energy rankings should be interpreted together with the operation, workload shape, energy ratios, and scaling behaviour.

RQ2 asked what architectural and runtime factors explain the observed differences. The clearest result is the operation-specific effect of `System.Text.Json` source generation. During deserialisation, `STJRefGen` and `STJSrcGen` remain close because both use the same steady-state read path after warm-up. During serialisation, `STJSrcGen` can use generated write handlers, which explains its consistent advantage over `STJRefGen`. The comparison with `SpanJson` also shows that low-energy deserialisation and low-energy serialisation depend on different implementation paths. The diagnostics support these explanations, but they are used qualitatively rather than as exact per-method energy attribution.

RQ3 asked how environmental conditions affect rankings and whether Metrion can provide consistent measurements. In the validation run at 0% stress, Metrion largely preserved the rankings produced by the RAPL-based Energy Diagnoser. Under the tested noisy conditions, rankings also remained mostly stable across the evaluated Depth, Size, and Width subsets. The few rank changes mainly involved near-ties. This suggests that Metrion is promising for ranking-oriented energy comparison under noise, but the evidence is exploratory because only part of the benchmark suite and one noisy setup were tested.

The main practical implication is that library choice can matter for energy, but the best choice depends on the operation and workload. `STJSrcGen` is the strongest default for serialisation among the evaluated libraries. `SpanJson` is often strongest for deserialisation, but its advantage depends on the workload and must be weighed against maintenance concerns. `Newtonsoft.Json` remains mature and feature-rich, but usually uses more en-

ergy.

Overall, the thesis shows that energy benchmarking of JSON libraries should be workload-sensitive. Energy consumption is not only a property of the library; it also depends on the JSON document, the operation, and the measurement environment. The benchmark infrastructure and synthetic workload suite developed here provide a basis for studying these effects more systematically in future work.

Special Terms

BDN BenchmarkDotNet. i, 2, 3, 5, 9–16, 27, 29, 31–35, 40, 41, 46, 84, 86, 90

dedicated process A separate, high-priority operating system process in which a single BDN benchmark case is executed, used to isolate the measured code from the main BDN runner and reduce interference from other system activity. 10, 33, 34

Energy Diagnoser BDN diagnoser that uses RAPL to collect energy measurements from the supported power domains. 2, 9–11, 14, 15, 27, 29–32, 40, 73–76, 80, 88

JIT Just-In-Time Compilation. 1, 5, 10, 84

Metrion A CLI tool that captures per-process energy.. xxxi, xxxii, 2, 3, 8, 10, 12–15, 27, 29, 32–36, 74, 75, 80, 83, 87, 88, 90

Metrion Profiler The BDN profiler implemented in this work that integrates Metrion to measure the energy consumed by benchmarks in noisy environments. Supports both Per stage and Per iteration measurement modes.. xxix–xxxii, 9, 10, 13, 15, 18, 27, 33–36, 40, 46, 73, 75, 76, 80, 81, 84, 90

Per iteration A configurable mode of the Metrion Profiler that extracts the energy consumed by each iteration of the *ActualWorkload* stage individually.. xxxi, 12, 13, 32–34, 37, 90

Per stage A configurable mode of the Metrion Profiler that extracts the total energy consumed during the entire *ActualWorkload* stage.. xxxi, 12, 13, 32, 33, 37, 90

RAPL Intel's Running Average Power Limit. ii, 1–5, 7–12, 27, 29, 30, 46, 87, 88, 90

SIMD Single Instruction, Multiple Data a CPU parallelism technique that applies one operation to multiple data elements simultaneously. 14, 73, 86

Smoke Benchmarks A minimal ten-method benchmark (one serialization + one deserialization per library) run against a single fixed workload configuration (D5, W20, textual, object-only, ASCII, no redundancy). 27, 28, 73

SMT Simultaneous Multithreading. 7, 8, 15, 16, 40, 84

Bibliography

[1] E. Capra, C. Francalanci, and S. A. Slaughter, "Is software "green"? application development environments and energy efficiency in open source applications", *Information and Software Technology*, vol. 54, no. 1, pp. 60–71, 2012, ISSN: 0950-5849. DOI: 10.1016/j.infsof.2011.07.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584911001777.

[2] M. Fahad, A. Shahid, R. R. Manumachu, and A. Lastovetsky, "A comparative study of methods for measurement of energy of computing", *Energies*, vol. 12, no. 11, 2019, ISSN: 1996-1073. DOI: 10.3390/en12112204. [Online]. Available: https://www.mdpi.com/1996-1073/12/11/2204.

[3] M. Couto, R. Pereira, F. Ribeiro, R. Rua, and J. Saraiva, "Towards a green ranking for programming languages", in *Proceedings of the 21st Brazilian Symposium on Programming Languages*, ser. SBLP '17, Fortaleza, CE, Brazil: Association for Computing Machinery, 2017. DOI: 10.1145/3125374.3125382. [Online]. Available: https://doi.org/10.1145/3125374.3125382.

[4] R. Pereira et al., "Ranking programming languages by energy efficiency", *Science of Computer Programming*, vol. 205, p. 102 609, 2021, ISSN: 0167-6423. DOI: 10.1016/j.scico.2021.102609. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642321000022.

[5] L. Koedijk and A. Oprescu, "Finding significant differences in the energy consumption when comparing programming languages and programs", in *2022 International Conference on ICT for Sustainability (ICT4S)*, IEEE, 2022, pp. 1–12. DOI: 10.1109/ICT4S55073.2022.00012.

[6] N. van Kempen, H.-J. Kwon, D. T. Nguyen, and E. D. Berger. "It's not easy being green: On the energy efficiency of programming languages", Accessed: May 6, 2026. [Online]. Available: https://arxiv.org/abs/2410.05460.

[7] D. Molteni. "Landscape of api traffic", Cloudflare, Accessed: Dec. 9, 2025. [Online]. Available: https://blog.cloudflare.com/landscape-of-api-traffic/.

[8] A. Nagy and B. Kovari, "Analyzing .net serialization components", in *2016 IEEE 11th International Symposium on Applied Computational Intelligence and Informatics (SACI)*, 2016, pp. 425–430. DOI: 10.1109/SACI.2016.7507414.

[9] E. Balalayeva, I. Marchenko, and A. Serhiienko, "Investigation of the performance efficiency of c# programming language data serializers using the developed software product for testing", in *2024 IEEE 19th International Conference on Computer Science and Information Technologies (CSIT)*, 2024, pp. 1–4. DOI: 10.1109/CSIT65290.2024.10982567.

[10] A. Noureddine, M. Martinez, H. Kanso, and N. Bru, "Is well-tested code more energy efficient?", in *11th Workshop on the Reliability of Intelligent Environments*, Biarritz, France, Jun. 2022. DOI: 10.3233/AISE220044. [Online]. Available: https://hal.science/hal-03635797.

[11] T. Vanders and G. Voda, "Microbenchmarking energy consumption of .net libraries", M.S. thesis, Aalborg University, 2026. [Online]. Available: https://kbdk-aub.primo.exlibrisgroup.com/permalink/45KBDK_AUB/3b147k/alma9922293346705762.

[12] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, and Z. Ou, "Rapl in action: Experiences in using rapl for power measurements", *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 3, no. 2, Mar. 2018, ISSN: 2376-3639. DOI: 10.1145/3177754. [Online]. Available: https://doi.org/10.1145/3177754.

[13] A. Gordillo et al., "Programming languages ranking based on energy measurements", *Software Quality Journal*, vol. 32, no. 4, pp. 1539–1580, 2024, ISSN: 1573-1367. DOI: 10.1007/s11219-024-09690-4. [Online]. Available: https://doi.org/10.1007/s11219-024-09690-4.

[14] L. S. E. Rasmussen, M. K. Lindof, S. B. Christensen, R. S. Lindholt, K. Jepsen, and A. O. Nielsen, "Benchmarking c# for energy consumption: Energy implications of different language constructs and collections", English, Student project, Aalborg University, 2021, 167 pp.

[15] R. S. Lindholt, K. Jepsen, and A. O. Nielsen, "Analyzing c# energy efficiency of concurrency and language construct combinations", English, Student project, Aalborg University, 2022, 243 pp.

[16] A. Noureddine, R. Rouvoy, and L. Seinturier, "Unit testing of energy consumption of software libraries", in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, ser. SAC '14, New York, NY, USA: Association for Computing Machinery, 2014, pp. 1200–1205. DOI: 10.1145/2554850.2554932. [Online]. Available: https://doi.org/10.1145/2554850.2554932.

[17] Z. Ournani, R. Rouvoy, P. Rust, and J. Penhoat, "Evaluating the energy consumption of java i/o apis", in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2021, pp. 1–11. DOI: 10.1109/ICSME52107.2021.00007.

[18] F. Nahrstedt, M. Karmouche, K. Bargie, P. Banijamali, A. Nalini Pradeep Kumar, and I. Malavolta, "An empirical study on the energy usage and performance of pandas and polars data analysis python libraries", in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '24, New York, NY, USA: Association for Computing Machinery, 2024, pp. 58–68. DOI: 10.1145/3661167.3661203. [Online]. Available: https://doi.org/10.1145/3661167.3661203.

[19] T. Kalibera and R. E. Jones, "Rigorous benchmarking in reasonable time", in *Proceedings of the 2013 ACM SIGPLAN International Symposium on Memory Management (ISMM '13)*, ACM, 2013, pp. 63–74. DOI: 10.1145/2464157.2464160.

[20] A. Akinshin, *Pro .NET Benchmarking, The Art of Performance Measurement*, 1st ed. Berkeley, CA: Apress, 2019, ISBN: 978-1-4842-4940-6. DOI: 10.1007/978-1-4842-4941-3. Accessed: Dec. 10, 2025. [Online]. Available: https://link.springer.com/book/10.1007/978-1-4842-4941-3.

[21] P. Sestoft, *Microbenchmarks in java and c#*, English, Lecture notes (online), Sep. 2013. Accessed: Dec. 10, 2025. [Online]. Available: https://www.itu.dk/~sestoft/papers/benchmarking.pdf.

[22] .NET Foundation, *Benchmarkdotnet documentation: Overview*. Accessed: Jan. 4, 2025. [Online]. Available: https://benchmarkdotnet.org/articles/overview.html.

[23] T. Bray, *The JavaScript Object Notation (JSON) Data Interchange Format*, RFC 8259, Internet Engineering Task Force, Dec. 2017. DOI: 10.17487/RFC8259. Accessed: Apr. 20, 2026. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8259.

[24] J. C. Viotti and M. Kinderkhedia, "A benchmark of JSON-compatible binary serialization specifications", *CoRR*, vol. abs/2201.03051, 2022. [Online]. Available: https://arxiv.org/abs/2201.03051.

[25] S. Belloni, D. Ritter, M. Schröder, and N. Rörup, "Deepbench: Benchmarking JSON document stores", in *DBTest@SIGMOD '22: Proceedings of the 9th International Workshop of Testing Database Systems, Philadelphia, PA, USA, 17 June 2022*, M. Rigger and P. Tözün, Eds., ACM, 2022, pp. 1–9. DOI: 10.1145/3531348.3532176. [Online]. Available: https://doi.org/10.1145/3531348.3532176.

[26] H. K. Dhalla, "A performance analysis of native json parsers in java, python, ms.net core, javascript, and php", in *2020 16th International Conference on Network and Service Management (CNSM)*, 2020, pp. 1–5. DOI: 10.23919/CNSM50824.2020.9269101.

[27] G. Langdale and D. Lemire, "Parsing gigabytes of JSON per second", *CoRR*, vol. abs/1902.08318, 2019. [Online]. Available: http://arxiv.org/abs/1902.08318.

[28] SchemaStore Contributors. "SchemaStore: A community-maintained collection of JSON schemas", Accessed: May 19, 2026. [Online]. Available: https://github.com/schemastore/schemastore.

[29] Z. Ournani, M. C. Belgaid, R. Rouvoy, P. Rust, J. Penhoat, and L. Seinturier, "Taming energy consumption variations in systems benchmarking", in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '20, Edmonton AB, Canada: Association for Computing Machinery, 2020, pp. 36–47, ISBN: 9781450369916. DOI: 10.1145/3358960.3379142. [Online]. Available: https://doi.org/10.1145/3358960.3379142.

[30] V. Stoico, A. C. Dragomir, and P. Lago, "An empirical study on the performance and energy usage of compiled python code", in *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '25, New York, NY, USA: Association for Computing Machinery, 2025, pp. 46–56, ISBN: 9798400713859. DOI: 10.1145/3756681.3756972. [Online]. Available: https://doi.org/10.1145/3756681.3756972.

[31] NASA Advanced Supercomputing (NAS) Division. "Nas parallel benchmarks", Accessed: May 19, 2026. [Online]. Available: https://www.nas.nasa.gov/software/npb.html.

[32] B. Weigell, S. Hornung, and B. Bauer, "Metrion: A framework for accurate software energy measurement", *TBD*, 2025. Accessed: May 6, 2026. [Online]. Available: https://arxiv.org/abs/2512.06806.

[33] R. Brondolin, T. Sardelli, and M. D. Santambrogio, "Deep-mon: Dynamic and energy efficient power monitoring for container-based infrastructures", in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, pp. 676–684. DOI: 10.1109/IPDPSW.2018.00110.

[34] G. Fieni, R. Rouvoy, and L. Seinturier, "Smartwatts: Self-calibrating software-defined power meter for containers", in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, 2020, pp. 479–488. DOI: 10.1109/CCGrid49817.2020.00-45.

[35] Hubblo. "Scaphandre: Energy consumption metrology agent.", Accessed: May 8, 2026. [Online]. Available: https://github.com/hubblo-org/scaphandre.

[36] H. Hè, M. Friedman, and T. Rekatsinas, "Energat: Fine-grained energy attribution for multi-tenancy", *SIGENERGY Energy Inform. Rev.*, vol. 4, no. 3, pp. 18–25, Sep. 2024. DOI: 10.1145/3698365.3698369. [Online]. Available: https://doi.org/10.1145/3698365.3698369.

[37] T. kernel development community, *Power capping framework*, https://www.kernel.org/doc/html/latest/power/powercap/powercap.html, May 2025. Accessed: Dec. 8, 2025.

[38] I. Molnár. "Perf: Linux profiling with performance counters", Accessed: Dec. 18, 2025. [Online]. Available: https://perfwiki.github.io/main/.

[39] dotnet, *Jit compiler structure*. Accessed: Apr. 4, 2026. [Online]. Available: https://github.com/dotnet/runtime/blob/main/docs/design/coreclr/jit/ryujit-overview.md.

[40] T. Vanders and G. Voda, *Benchmarkdotnet*. Accessed: Jun. 8, 2026. [Online]. Available: https://github.com/geovoda/BenchmarkDotNet.Energy.

[41] M. Bernhardt. "On pinning and isolating cpu cores", Accessed: May 6, 2026. [Online]. Available: https://manuel.bernhardt.io/posts/2023-11-16-core-pinning/#deciding-which-cores-to-use.

[42] T. Vanders and G. Voda, *.net library energy benchmarking*. Accessed: Jun. 22, 2026. [Online]. Available: https://github.com/toms-vanders/json-energy-bench.

[43] C. I. King, "Stress-ng", *URL: http://kernel. ubuntu. com/git/cking/stressng. git/(visited on 28/03/2018)*, vol. 39, 2017.

[44] Microsoft Docs. "Reflection versus source generation in system.text.json", Accessed: Apr. 20, 2026. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/serialization/system-text-json/reflection-vs-source-generation.

[45] Newtonsoft. "Json.net documentation, Introduction", Accessed: Apr. 20, 2026. [Online]. Available: https://www.newtonsoft.com/json/help/html/Introduction.htm.

[46] A. Köplinger. "Spanjson", Accessed: Apr. 20, 2026. [Online]. Available: https://github.com/Tornhoof/SpanJson.

[47] Y. Kawai. "Utf8json - fast json serializer for c#", Accessed: Apr. 20, 2026. [Online]. Available: https://github.com/neuecc/Utf8Json.

[48] K. Montrose. "Jil", Accessed: Apr. 20, 2026. [Online]. Available: https://github.com/kevin-montrose/Jil.

[49] B. Maiwald, B. Riedle, and S. Scherzinger, "What are real json schemas like? an empirical analysis of structural properties", in *Advances in Conceptual Modeling: ER 2019 Workshops FAIR, MREBA, EmpER, MoBiD, OntoCom, and ER Doctoral Symposium Papers, Salvador, Brazil, November 47, 2019, Proceedings*, Salvador, Brazil: Springer-Verlag, 2019, pp. 95–105, ISBN: 978-3-030-34145-9. DOI: 10.1007/978-3-030-34146-6_9. [Online]. Available: https://doi.org/10.1007/978-3-030-34146-6_9.

[50] L. Cruz, *Green software engineering done right: A scientific guide to set up energy efficiency experiments*, http://luiscruz.github.io/2021/10/10/scientific-guide.html, Blog post., 2021. DOI: 10.6084/m9.figshare.22067846.v1.

[51] C. Wohlin, P. Runeson, M. Höst,
bibinitperiod Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering (2024 edition)*, English. 2024, Claes Wohlin, Blekinge Institute of Technology Per Runeson, Lund University Martin Höst, Malmö University Magnus C. Ohlsson, System Verification Sweden AB Björn Regnell, Lund University Anders Wesslén, Ericsson AB. DOI: 10.1007/978-3-662-69306-3.

[52] S. Holm, "A simple sequentially rejective multiple test procedure", *Scandinavian Journal of Statistics*, vol. 6, no. 2, pp. 65–70, 1979.

[53] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions", *Psychological Bulletin*, vol. 114, no. 3, pp. 494–509, 1993. DOI: 10.1037/0033-2909.114.3.494.

[54] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, "Appropriate statistics for ordinal level data: Should we really be using t-test and cohensd for evaluating group differences on the nsse and other surveys", in *annual meeting of the Florida Association of Institutional Research*, vol. 177, 2006.

[55] G. Nitram1995, *Paradigmcomparison/csharprapl*. Accessed: Dec. 27, 2025. [Online]. Available: https://github.com/lrecht/ParadigmComparison/tree/master/CsharpRAPL.

[56] Microsoft Docs. "Source-generation modes in system.text.json", Accessed: Apr. 20, 2026. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/serialization/system-text-json/source-generation-modes.

[57] A. Köplinger. "Jsonwriter.utf8.cs", Accessed: Jun. 6, 2026. [Online]. Available: https://github.com/Tornhoof/SpanJson/blob/master/SpanJson/JsonWriter.Utf8.cs#L290.

[58] dotnet, *Indexofanyasciisearcher.cs*. Accessed: Jun. 6, 2026. [Online]. Available: https://github.com/dotnet/runtime/blob/2d241825745f8be946892fd25492fa4664590241/src/libraries/System.Private.CoreLib/src/System/SearchValues/IndexOfAnyAsciiSearcher.cs#L34.

[59] dotnet, *Utf8utility.transcoding.cs*. Accessed: Jun. 6, 2026. [Online]. Available: https://github.com/dotnet/runtime/blob/2d241825745f8be946892fd25492fa4664590241/src/libraries/System.Private.CoreLib/src/System/Text/Unicode/Utf8Utility.Transcoding.cs#L851.

[60] Microsoft. "SIMD-accelerated types in .NET", Accessed: Jun. 6, 2026. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/simd.

Appendix A

Tables

A.1 BenchmarkDotNet's configuration parameters

Table A.1: BDN configuration parameters with their default values and experimental overrides, based on [11] and adjusted as necessary.

Parameter	Default	Our
Minimum time of a single iteration	500ms	1s
Forces full garbage collection after each benchmark invocation.	True	
Run garbage collection concurrently.	True	
The unroll factor - How many times the benchmark method will be invoked per one iteration of a generated loop	16	
Maximum acceptable error for a benchmark	0.02	
Minimum count of benchmark invocations per iteration	4	
Minimum count of warmup iterations	6	
Maximum count of warmup iterations	50	
Minimum count of target iterations	15	
Maximum count of target iterations	100	
Specifies if the overhead should be evaluated (Idle runs) and it's average value subtracted from every result	True	
ProcessorAffinity *	0	`(1 « 2)`

*Applicable to the *clean* environment configuration only (See § 3.2.2.1).

A.2 RAPL Power Domains

Table A.2: Power domains and constraints exposed by the Linux Power Capping Framework [37] on our system.

Domain	Property	Value
Package	Monitored power domain	package-0
Package	Maximum counter before wraparound	262 143 328 850 uJ
Package	Long-term sustained power limit (PL1)	95 W
Package	Averaging window for PL1	27 983 872 us
Package	Short-term burst power limit (PL2)	210 W
Package	Averaging window for PL2	2.44 ms
Core*	Maximum counter before wraparound	262 143 328 850 uJ
Uncore*	Maximum counter before wraparound	262 143 328 850 uJ

* The controls of the domains are disabled.

A.3 Per-Configuration Isolation Benchmark Data

This appendix section collects the complete per-configuration BenchmarkDotNet output for every isolation sweep referenced from §5.2 and §5.3. Each table lists one row per (library, operation, sweep level), reporting execution time (`Mean/Error/StdDev`), per-operation Package and DRAM energy (with its standard deviation), socket temperature, garbage-collection counts (Gen0/Gen1), and bytes allocated. Energy is reported per operation in μJ; values are from the main clean RAPL run. Tables are rendered in landscape and may span multiple pages.

A.3.1 Size

Table A.3: Full per-configuration BenchmarkDotNet report for the Size isolation sweep: timing, Package and DRAM energy per operation, GC counts, and allocations. Rows are grouped by operation and then by Size level, with the five libraries listed within each level. Within each level the Package-energy cell is shaded from the lowest-energy library (light) to the highest (dark); the shading is not comparable across levels.

Library	Iters	Ops	Mean	Error	StdDev	Pkg (µJ/op)	StdDev (µJ/op)	DRAM (µJ/op)	Temp (°C)	Gen0	Gen1	Allocated
Deserialisation												
C10												
SpanJson	15	13291	75.24 s	0.353 s	0.330 s	2643.57 uj	12.47 uj	84.52 uj	48.8	8.3515	1.4295	68.31 KB
Utf8Json	15	8868	112.82 s	0.544 s	0.509 s	3978.59 uj	16.52 uj	126.82 uj	48.97	8.3446	1.4659	68.31 KB
STJRefGen	15	9696	103.18 s	0.521 s	0.488 s	3638.94 uj	16.81 uj	115.90 uj	48.6	8.4571	1.4439	69.67 KB
STJSrcGen	15	10000	100.04 s	0.603 s	0.564 s	3527.98 uj	19.57 uj	112.42 uj	48.6	8.5	1.5	69.67 KB
Newtonsoft	15	5008	199.66 s	0.889 s	0.832 s	7032.96 uj	31.00 uj	224.20 uj	48.63	15.9744	3.7939	130.81 KB
C100												
SpanJson	15	1287	777.49 s	4.051 s	3.789 s	27275.00 uj	140.49 uj	886.81 uj	48.77	83.1391	47.397	681.85 KB
Utf8Json	15	881	1,135.78 s	5.459 s	5.106 s	40034.66 uj	169.46 uj	1291.12 uj	48.97	82.8604	47.6731	681.85 KB
STJRefGen	15	960	1,041.26 s	2.496 s	2.335 s	36719.73 uj	87.10 uj	1185.94 uj	48.6	83.3333	44.7917	683.21 KB
STJSrcGen	15	944	1,018.75 s	1.631 s	1.526 s	35936.20 uj	56.43 uj	1161.76 uj	49.03	82.6271	44.4915	683.21 KB
Newtonsoft	15	488	2,043.55 s	8.835 s	8.264 s	72067.76 uj	284.78 uj	2485.78 uj	48.53	81.9672	40.9836	1280.48 KB
C1K												
SpanJson	16	42	9,524.57 s	180.229 s	177.009 s	333451.83 uj	6015.49 uj	12226.43 uj	48.78	833.3333	809.5238	6813.11 KB
Utf8Json	15	18	13,522.25 s	230.669 s	215.768 s	475705.20 uj	7511.45 uj	16893.80 uj	48.7	833.3333	777.7778	6813.11 KB
STJRefGen	15	96	11,719.15 s	21.251 s	19.878 s	411782.33 uj	671.96 uj	14818.79 uj	49.17	833.3333	822.9167	6814.47 KB
STJSrcGen	15	64	11,823.60 s	81.655 s	76.380 s	415519.48 uj	2641.81 uj	14960.38 uj	49.5	828.125	812.5	6814.47 KB
Newtonsoft	15	10	22,659.87 s	393.062 s	367.671 s	797135.01 uj	13274.63 uj	29400.55 uj	49.5	800	700	12773.07 KB
C10K												
SpanJson	15	4	119,507.63 s	1,664.214 s	1,556.707 s	4175006.63 uj	56878.10 uj	193214.45 uj	49.33	8250	8000	68225.08 KB
Utf8Json	15	7	158,284.34 s	1,087.833 s	1,017.560 s	5544906.50 uj	36548.52 uj	241313.79 uj	49.7	8285.7143	8142.8571	68225.08 KB
STJRefGen	15	6	145,145.94 s	1,531.609 s	1,432.668 s	5091237.24 uj	52673.49 uj	225989.54 uj	49.17	8333.3333	8166.6667	68226.44 KB
STJSrcGen	15	4	145,172.32 s	1,301.903 s	1,217.801 s	5086608.10 uj	44416.94 uj	226295.42 uj	49.53	8250	8000	68226.44 KB
Newtonsoft	27	1	267,329.42 s	5,191.836 s	7,278.230 s	9369231.81 uj	253393.03 uj	368090.78 uj	49.09	8000	7000	127799.3 KB
C100K												

continued on next page

Library	Iters	Ops	Mean	Error	StdDev	Pkg (µJ/op)	StdDev (µJ/op)	DRAM (µJ/op)	Temp (°C)	Gen0	Gen1	Allocated
SpanJson	100	1	1,405,684.65 s	58,278.950 s	171,836.810 s	48596976.39 uj	5799253.43 uj	2374324.42 uj	49.15	83000	82000	681735.9 KB
Utf8Json	100	1	1,765,264.00 s	57,641.518 s	169,957.328 s	61522147.77 uj	5744480.97 uj	2740022.89 uj	49.59	83000	82000	681735.9 KB
STJRefGen	100	1	1,669,657.50 s	57,612.391 s	169,871.446 s	58261837.58 uj	5755616.87 uj	2685744.60 uj	48.71	84000	83000	681737.96 KB
STJSrcGen	100	1	1,650,024.23 s	58,400.556 s	172,195.370 s	57577967.86 uj	5844393.50 uj	2662507.96 uj	48.94	84000	83000	681737.96 KB
Newtonsoft	30	1	2,873,212.59 s	56,963.060 s	85,259.633 s	99975387.00 uj	2983884.56 uj	4229377.53 uj	49.25	83000	82000	1277441.95 KB
Serialisation												
C10												
SpanJson	15	7074	141.51 s	0.991 s	0.927 s	4859.90 uj	31.81 uj	158.78 uj	48.3	3.5341	0	29.69 KB
Utf8Json	15	7661	130.59 s	0.457 s	0.428 s	4483.71 uj	15.46 uj	146.52 uj	48.5	3.5243	0	29.82 KB
STJRefGen	15	15728	63.65 s	0.418 s	0.391 s	2249.45 uj	14.72 uj	71.45 uj	49.67	3.6877	0	30.71 KB
STJSrcGen	15	28432	35.14 s	0.079 s	0.074 s	1271.54 uj	3.06 uj	39.45 uj	49.87	3.6227	0	29.82 KB
Newtonsoft	15	8688	115.10 s	0.168 s	0.157 s	4061.46 uj	5.59 uj	129.54 uj	49.17	18.7615	0	154.13 KB
C100												
SpanJson	15	761	1,313.66 s	7.903 s	7.392 s	45125.04 uj	253.43 uj	1510.60 uj	48.5	0	0	296.53 KB
Utf8Json	15	756	1,322.48 s	7.547 s	7.060 s	45550.39 uj	262.15 uj	1634.69 uj	48.3	0	0	1193.96 KB
STJRefGen	15	1376	655.85 s	0.585 s	0.547 s	23176.80 uj	62.89 uj	772.99 uj	49.17	0	0	298.78 KB
STJSrcGen	15	2224	389.62 s	1.491 s	1.394 s	13861.47 uj	58.92 uj	472.08 uj	49.27	0	0	297.89 KB
Newtonsoft	15	830	1,197.07 s	4.438 s	4.151 s	42162.94 uj	160.16 uj	1692.04 uj	48.83	74.6988	33.7349	1507.66 KB
C1K												
SpanJson	15	34	14,981.84 s	202.234 s	189.170 s	514702.89 uj	6606.15 uj	18833.72 uj	48.37	0	0	2964.88 KB
Utf8Json	15	24	13,801.14 s	163.403 s	152.848 s	474399.46 uj	5416.22 uj	19287.91 uj	48.23	0	0	11042.7 KB
STJRefGen	15	64	6,366.15 s	40.988 s	38.340 s	227131.01 uj	1425.44 uj	9892.50 uj	49.17	0	0	2979.45 KB
STJSrcGen	15	256	3,857.54 s	16.405 s	15.345 s	139459.33 uj	530.20 uj	6644.06 uj	49.13	0	0	2978.56 KB
Newtonsoft	70	17	13,691.59 s	272.033 s	662.166 s	481532.61 uj	23518.53 uj	22029.79 uj	48.42	705.8824	588.2353	14933.15 KB
C10K												
SpanJson	15	4	139,384.35 s	2,034.001 s	1,902.606 s	4782519.47 uj	70294.78 uj	196320.10 uj	46.87	0	0	29648.48 KB
Utf8Json	15	4	162,249.08 s	1,561.748 s	1,460.860 s	5572370.37 uj	49624.69 uj	224338.28 uj	48.17	0	0	95193.41 KB
STJRefGen	41	6	53,463.68 s	1,065.423 s	1,921.178 s	1923420.02 uj	67111.38 uj	101598.47 uj	48	0	0	29786.09 KB
STJSrcGen	29	8	42,027.71 s	823.921 s	1,207.694 s	1511052.18 uj	40722.30 uj	85648.41 uj	47.55	0	0	29785.2 KB
Newtonsoft	16	5	154,255.95 s	3,056.870 s	3,002.255 s	5404753.79 uj	105491.40 uj	260932.99 uj	47.59	7200	7000	149203.69 KB
C100K												
SpanJson	15	1	1,573,516.28 s	2,228.703 s	2,084.730 s	53865788.27 uj	64499.16 uj	2184735.60 uj	47.3	0	0	296484.41 KB

continued on next page

Library	Iters	Ops	Mean	Error	StdDev	Pkg (µJ/op)	StdDev (µJ/op)	DRAM (µJ/op)	Temp (°C)	Gen0	Gen1	Allocated
`Utf8Json`	15	1	1,699,722.13 s	4,087.827 s	3,823.756 s	58200025.73 uj	141388.76 uj	2496526.93 uj	48.6	0	0	1346299.91 KB
`STJRefGen`	15	1	640,462.16 s	2,970.435 s	2,778.547 s	22766657.40 uj	116169.01 uj	1170793.53 uj	47.33	0	0	297852.49 KB
`STJSrcGen`	15	1	521,863.37 s	8,917.998 s	8,341.901 s	18478737.00 uj	276500.25 uj	1011321.53 uj	47.5	0	0	297851.6 KB
`Newtonsoft`	84	1	1,744,143.29 s	34,847.142 s	93,614.590 s	60697948.73 uj	3159853.33 uj	2881390.39 uj	48.89	74000	73000	1491949.41 KB

A.3.2 Depth

Table A.4: Full per-configuration BenchmarkDotNet report for the Depth isolation sweep: timing, Package and DRAM energy per operation, GC counts, and allocations. Rows are grouped by operation and then by Depth level, with the five libraries listed within each level. Within each level the Package-energy cell is shaded from the lowest-energy library (light) to the highest (dark); the shading is not comparable across levels.

Library	Iters	Ops	Mean	Error	StdDev	Pkg (µJ/op)	StdDev (µJ/op)	DRAM (µJ/op)	Temp (°C)	Gen0	Gen1	Allocated
Deserialisation												
D1												
`SpanJson`	15	676291	1,479.6 ns	7.59 ns	7.10 ns	50.75 uj	0.24 uj	1.68 uj	49.17	0.1656	0	1392 B
`Utf8Json`	15	441151	2,275.1 ns	14.73 ns	13.78 ns	78.10 uj	0.49 uj	2.58 uj	49.7	0.1655	0	1392 B
`STJRefGen`	15	490496	1,977.8 ns	7.40 ns	6.92 ns	68.03 uj	0.24 uj	2.22 uj	49.4	0.1651	0	1392 B
`STJSrcGen`	15	501264	1,994.6 ns	7.28 ns	6.81 ns	68.67 uj	0.31 uj	2.24 uj	49.7	0.1656	0	1392 B
`Newtonsoft`	15	260960	3,853.7 ns	23.70 ns	22.17 ns	132.37 uj	0.76 uj	4.33 uj	49.43	0.6208	0.0115	5224 B
D2												
`SpanJson`	15	338499	2,947.0 ns	19.59 ns	18.32 ns	101.10 uj	0.62 uj	3.30 uj	49.13	0.3309	0.003	2784 B
`Utf8Json`	15	219726	4,567.1 ns	23.52 ns	22.00 ns	156.77 uj	0.76 uj	5.16 uj	49.53	0.3322	0	2784 B
`STJRefGen`	15	247712	4,044.2 ns	19.92 ns	18.64 ns	139.02 uj	0.65 uj	4.63 uj	49.3	0.3875	0	3256 B
`STJSrcGen`	15	252128	3,960.0 ns	24.61 ns	23.02 ns	136.11 uj	0.82 uj	4.49 uj	49.77	0.3887	0.004	3256 B
`Newtonsoft`	15	128832	7,789.9 ns	43.44 ns	40.64 ns	267.57 uj	1.41 uj	8.87 uj	49.43	0.9547	0.0155	7992 B
D4												
`SpanJson`	15	171373	5,861.7 ns	40.16 ns	37.57 ns	201.14 uj	1.28 uj	6.66 uj	49	0.6652	0.0117	5568 B
`Utf8Json`	15	109054	8,588.5 ns	70.73 ns	66.16 ns	295.34 uj	2.22 uj	9.69 uj	49.37	0.6602	0	5568 B
`STJRefGen`	15	48768	8,137.8 ns	124.49 ns	116.45 ns	279.51 uj	3.90 uj	9.24 uj	49.57	0.7177	0	6040 B
`STJSrcGen`	15	122496	8,154.5 ns	40.22 ns	37.62 ns	280.21 uj	1.30 uj	9.25 uj	49.27	0.7184	0.0082	6040 B
`Newtonsoft`	15	62544	16,033.3 ns	104.39 ns	97.65 ns	550.12 uj	3.32 uj	18.24 uj	48.97	1.5669	0.064	13208 B
D8												
`SpanJson`	15	84433	11,877.4 ns	71.12 ns	66.52 ns	407.52 uj	2.17 uj	13.31 uj	48.8	1.3265	0.0474	11136 B
`Utf8Json`	15	54851	18,236.3 ns	161.38 ns	150.95 ns	626.35 uj	5.04 uj	20.52 uj	49.33	1.3309	0.0182	11136 B
`STJRefGen`	15	61952	16,128.3 ns	108.19 ns	101.20 ns	555.50 uj	3.70 uj	18.35 uj	49.47	1.485	0.0484	12528 B
`STJSrcGen`	15	63296	15,841.3 ns	107.53 ns	100.58 ns	546.90 uj	3.74 uj	17.81 uj	49.4	1.4851	0.0474	12528 B
`Newtonsoft`	15	31520	31,936.9 ns	189.11 ns	176.89 ns	1096.14 uj	6.15 uj	36.35 uj	48.83	2.8236	0.1586	23864 B
D15												

continued on next page

Library	Iters	Ops	Mean	Error	StdDev	Pkg (µJ/op)	StdDev (µJ/op)	DRAM (µJ/op)	Temp (°C)	Gen0	Gen1	Allocated
`SpanJson`	15	45326	22,143.7 ns	109.77 ns	102.68 ns	759.96 uj	3.58 uj	25.07 uj	49.33	2.4931	0.1544	20880 B
`Utf8Json`	15	30938	32,437.0 ns	285.33 ns	266.89 ns	1117.20 uj	9.10 uj	36.67 uj	49.67	2.4888	0.0646	20880 B
`STJRefGen`	15	33264	30,075.5 ns	175.54 ns	164.20 ns	1034.35 uj	5.33 uj	34.24 uj	49.67	2.8559	0.2104	24088 B
`STJSrcGen`	15	32432	30,711.8 ns	181.35 ns	169.64 ns	1055.05 uj	6.34 uj	34.57 uj	49.6	2.8675	0.2158	24088 B
`Newtonsoft`	100	3696	59,564.2 ns	1,490.76 ns	4,395.53 ns	2044.52 uj	149.73 uj	67.28 uj	49.49	4.8701	0.2706	42544 B
D25												
`SpanJson`	15	26993	37,266.3 ns	427.41 ns	399.80 ns	1278.21 uj	13.31 uj	42.21 uj	49.03	4.1492	0.4446	34800 B
`Utf8Json`	15	17782	56,413.4 ns	346.43 ns	324.05 ns	1936.38 uj	11.03 uj	63.51 uj	49.3	4.1053	0.2249	34800 B
`STJRefGen`	58	7344	51,223.6 ns	1,012.54 ns	2,222.55 ns	1760.42 uj	75.70 uj	57.88 uj	49.67	4.902	0.5447	41616 B
`STJSrcGen`	15	8400	50,021.0 ns	639.24 ns	597.95 ns	1719.78 uj	23.60 uj	56.34 uj	49.57	4.881	0.5952	41616 B
`Newtonsoft`	15	9824	97,482.1 ns	551.42 ns	515.80 ns	3345.83 uj	18.21 uj	110.65 uj	49.23	8.2451	0.8143	69432 B
D40												
`SpanJson`	15	16559	59,902.4 ns	485.86 ns	454.47 ns	2055.95 uj	15.81 uj	67.70 uj	49.23	6.6429	0.9662	55680 B
`Utf8Json`	15	10815	86,201.0 ns	930.88 ns	870.75 ns	2959.97 uj	29.85 uj	97.39 uj	49.53	6.565	0.5548	55680 B
`STJRefGen`	15	12256	82,029.0 ns	494.87 ns	462.90 ns	2821.68 uj	16.16 uj	92.87 uj	49.53	8.3225	1.4687	69688 B
`STJSrcGen`	15	12288	80,703.0 ns	377.09 ns	352.73 ns	2783.69 uj	17.24 uj	90.45 uj	49.4	8.3008	1.4648	69688 B
`Newtonsoft`	15	7088	141,600.6 ns	826.86 ns	773.45 ns	4861.65 uj	26.74 uj	158.67 uj	48.93	13.1208	2.2573	110144 B
Serialisation												
D1												
`SpanJson`	15	910668	1,059.1 ns	15.78 ns	14.76 ns	36.35 uj	0.51 uj	1.19 uj	49.6	0.0736	0	624 B
`Utf8Json`	15	754524	1,330.9 ns	11.67 ns	10.92 ns	45.68 uj	0.38 uj	1.50 uj	49.13	0.0755	0	640 B
`STJRefGen`	15	1060176	942.2 ns	6.11 ns	5.71 ns	33.38 uj	0.20 uj	1.06 uj	50.3	0.0764	0	640 B
`STJSrcGen`	15	1232704	810.0 ns	4.67 ns	4.36 ns	28.60 uj	0.14 uj	0.92 uj	50.1	0.0763	0	640 B
`Newtonsoft`	15	401312	2,487.6 ns	19.28 ns	18.04 ns	85.58 uj	0.61 uj	2.88 uj	49.4	0.5806	0.005	4864 B
D2												
`SpanJson`	15	474512	2,111.2 ns	18.31 ns	17.13 ns	72.38 uj	0.60 uj	2.39 uj	49.27	0.1454	0	1232 B
`Utf8Json`	15	371473	2,707.5 ns	20.19 ns	18.89 ns	92.26 uj	0.70 uj	3.05 uj	48.93	0.1481	0	1248 B
`STJRefGen`	15	490384	2,021.9 ns	11.70 ns	10.95 ns	70.65 uj	0.53 uj	2.31 uj	50.07	0.1856	0	1560 B
`STJSrcGen`	15	665776	1,501.7 ns	8.21 ns	7.68 ns	52.02 uj	0.35 uj	1.71 uj	49.77	0.1487	0	1248 B
`Newtonsoft`	15	205008	4,883.6 ns	32.89 ns	30.77 ns	167.82 uj	1.04 uj	5.62 uj	49.3	1.0683	0.0098	8968 B
D4												
`SpanJson`	15	237608	4,245.8 ns	31.12 ns	29.11 ns	145.67 uj	0.99 uj	4.80 uj	49.2	0.2904	0	2456 B

continued on next page

Library	Iters	Ops	Mean	Error	StdDev	Pkg (µJ/op)	StdDev (µJ/op)	DRAM (µJ/op)	Temp (°C)	Gen0	Gen1	Allocated
`Utf8Json`	15	185666	5,396.0 ns	57.32 ns	53.62 ns	184.96 uj	1.79 uj	6.09 uj	49.17	0.2908	0	2464 B
`STJRefGen`	15	264208	3,764.8 ns	28.59 ns	26.75 ns	131.49 uj	1.03 uj	4.26 uj	50.23	0.3293	0	2776 B
`STJSrcGen`	15	344384	2,903.7 ns	19.54 ns	18.27 ns	101.59 uj	0.68 uj	3.27 uj	50.03	0.2933	0	2464 B
`Newtonsoft`	15	103616	9,633.4 ns	52.06 ns	48.70 ns	330.90 uj	1.92 uj	10.82 uj	49.47	1.9978	0.0579	16784 B
D8												
`SpanJson`	15	67201	14,437.7 ns	146.98 ns	137.48 ns	483.16 uj	4.58 uj	16.22 uj	48.57	0.5803	0	4888 B
`Utf8Json`	15	77358	13,088.9 ns	176.34 ns	164.95 ns	448.03 uj	5.96 uj	14.75 uj	48.53	0.5817	0	4904 B
`STJRefGen`	15	132752	7,424.4 ns	50.55 ns	47.28 ns	260.74 uj	1.57 uj	8.42 uj	50.23	0.693	0	5816 B
`STJSrcGen`	15	177168	5,648.1 ns	33.84 ns	31.66 ns	195.93 uj	1.28 uj	6.39 uj	50.07	0.5814	0	4904 B
`Newtonsoft`	15	54224	18,409.4 ns	118.04 ns	110.41 ns	631.99 uj	3.76 uj	20.80 uj	49.7	3.8913	0.2397	32664 B
D15												
`SpanJson`	15	30396	31,796.5 ns	301.47 ns	281.99 ns	1061.29 uj	9.48 uj	35.76 uj	48.37	1.0857	0	9152 B
`Utf8Json`	15	31542	31,771.1 ns	200.53 ns	187.58 ns	1060.43 uj	6.43 uj	35.63 uj	48.17	1.0779	0	9168 B
`STJRefGen`	15	69728	14,288.4 ns	100.00 ns	93.54 ns	503.05 uj	3.51 uj	16.21 uj	50.33	1.3338	0	11256 B
`STJSrcGen`	15	93520	10,704.7 ns	62.97 ns	58.90 ns	376.41 uj	1.93 uj	12.07 uj	50.07	1.0907	0	9168 B
`Newtonsoft`	15	29472	34,128.0 ns	207.57 ns	194.16 ns	1172.08 uj	7.17 uj	38.83 uj	49.57	7.3969	0.7125	62088 B
D25												
`SpanJson`	15	15278	65,871.2 ns	570.20 ns	533.36 ns	2203.39 uj	17.52 uj	73.89 uj	48.07	1.7672	0	15240 B
`Utf8Json`	15	17300	58,002.1 ns	407.05 ns	380.75 ns	1936.82 uj	12.82 uj	65.29 uj	47.97	1.7919	0	15256 B
`STJRefGen`	15	43456	23,007.2 ns	124.34 ns	116.31 ns	803.21 uj	3.89 uj	26.05 uj	50.13	2.3472	0	19672 B
`STJSrcGen`	15	57968	17,564.5 ns	194.77 ns	182.19 ns	609.34 uj	5.58 uj	19.81 uj	50.17	1.8113	0	15256 B
`Newtonsoft`	15	17776	56,597.4 ns	349.56 ns	326.98 ns	1942.51 uj	11.15 uj	63.62 uj	49.47	9.676	0.9563	81424 B
D40												
`SpanJson`	15	9119	110,058.3 ns	1,143.37 ns	1,069.51 ns	3678.72 uj	36.53 uj	123.40 uj	48.2	2.8512	0	24376 B
`Utf8Json`	15	10053	99,826.6 ns	566.32 ns	529.73 ns	3331.42 uj	17.52 uj	112.22 uj	48.07	2.8847	0	24392 B
`STJRefGen`	15	19872	50,249.9 ns	298.24 ns	278.98 ns	1735.22 uj	9.85 uj	56.82 uj	49.8	3.9754	0	33440 B
`STJSrcGen`	15	34528	28,893.8 ns	173.48 ns	162.27 ns	1004.36 uj	7.75 uj	32.50 uj	50.33	2.8962	0	24392 B
`Newtonsoft`	15	10976	91,671.2 ns	628.17 ns	587.59 ns	3149.76 uj	23.65 uj	105.11 uj	49.63	17.0372	2.0955	143072 B

A.3.3 Width

Table A.5: Full per-configuration BenchmarkDotNet report for the Width isolation sweep: timing, Package and DRAM energy per operation, GC counts, and allocations. Rows are grouped by operation and then by Width level, with the five libraries listed within each level. Within each level the Package-energy cell is shaded from the lowest-energy library (light) to the highest (dark); the shading is not comparable across levels.

Library	Iters	Ops	Mean	Error	StdDev	Pkg (μJ/op)	StdDev (μJ/op)	DRAM (μJ/op)	Temp (°C)	Gen0	Gen1	Allocated
Deserialisation												
W2												
`SpanJson`	15	1961696	507.4 ns	2.95 ns	2.76 ns	17.56 uj	0.09 uj	0.58 uj	49.27	0.0571	0	480 B
`Utf8Json`	15	1297443	774.2 ns	7.28 ns	6.81 ns	26.60 uj	0.23 uj	0.88 uj	49.53	0.057	0	480 B
`STJRefGen`	15	849056	1,178.0 ns	6.84 ns	6.39 ns	40.73 uj	0.22 uj	1.34 uj	49.47	0.1131	0	952 B
`STJSrcGen`	15	872064	1,146.4 ns	6.63 ns	6.21 ns	39.59 uj	0.22 uj	1.31 uj	49.63	0.1135	0	952 B
`Newtonsoft`	15	481792	2,056.9 ns	13.42 ns	12.56 ns	70.74 uj	0.44 uj	2.38 uj	49.23	0.4359	0.0021	3656 B
W5												
`SpanJson`	15	626272	1,596.1 ns	8.58 ns	8.02 ns	54.92 uj	0.29 uj	1.80 uj	49.23	0.1852	0	1560 B
`Utf8Json`	15	412016	2,390.7 ns	18.55 ns	17.35 ns	82.09 uj	0.60 uj	2.70 uj	49.77	0.1845	0	1560 B
`STJRefGen`	15	369296	2,712.5 ns	15.06 ns	14.09 ns	93.31 uj	0.48 uj	3.08 uj	49.23	0.241	0	2032 B
`STJSrcGen`	15	386992	2,582.0 ns	14.69 ns	13.74 ns	88.82 uj	0.47 uj	2.90 uj	49.57	0.2429	0	2032 B
`Newtonsoft`	15	199200	4,728.4 ns	30.18 ns	28.23 ns	162.43 uj	0.96 uj	5.41 uj	49.43	0.6777	0.005	5672 B
W10												
`SpanJson`	15	276832	3,622.7 ns	22.58 ns	21.12 ns	124.42 uj	0.75 uj	4.11 uj	49.13	0.401	0.0036	3360 B
`Utf8Json`	15	183895	5,358.2 ns	53.29 ns	49.84 ns	183.96 uj	1.65 uj	6.01 uj	49.33	0.397	0	3360 B
`STJRefGen`	15	193840	5,157.7 ns	27.75 ns	25.96 ns	177.16 uj	0.90 uj	5.86 uj	49.43	0.454	0.0052	3832 B
`STJSrcGen`	15	196880	5,069.2 ns	28.96 ns	27.09 ns	174.18 uj	0.94 uj	5.71 uj	49.53	0.4571	0.0051	3832 B
`Newtonsoft`	15	99984	9,956.7 ns	54.03 ns	50.54 ns	341.73 uj	1.74 uj	11.30 uj	49.1	1.0702	0.02	9016 B
W20												
`SpanJson`	15	136512	7,338.1 ns	43.65 ns	40.83 ns	251.71 uj	1.38 uj	8.23 uj	49.13	0.8278	0.0147	6960 B
`Utf8Json`	15	87639	11,430.9 ns	76.87 ns	71.90 ns	392.62 uj	2.53 uj	12.82 uj	49.6	0.8216	0	6960 B
`STJRefGen`	57	40608	10,043.9 ns	198.55 ns	431.63 ns	345.29 uj	14.74 uj	11.36 uj	49.59	0.8865	0	7432 B
`STJSrcGen`	15	99408	9,976.4 ns	73.07 ns	68.35 ns	343.04 uj	2.43 uj	11.35 uj	49.5	0.8852	0.0201	7432 B
`Newtonsoft`	15	54320	18,500.0 ns	109.21 ns	102.16 ns	635.19 uj	3.63 uj	21.14 uj	49.3	1.8778	0.0368	15816 B
W50												

continued on next page

Library	Iters	Ops	Mean	Error	StdDev	Pkg (µJ/op)	StdDev (µJ/op)	DRAM (µJ/op)	Temp (°C)	Gen0	Gen1	Allocated
`SpanJson`	15	51817	19,375.4 ns	191.59 ns	179.22 ns	664.52 uj	6.35 uj	21.81 uj	49.03	2.1229	0.1158	17760 B
`Utf8Json`	15	36779	27,274.7 ns	147.93 ns	138.37 ns	936.95 uj	4.38 uj	30.77 uj	49.63	2.1208	0.0544	17760 B
`STJRefGen`	15	38896	25,686.0 ns	121.23 ns	113.40 ns	882.93 uj	3.95 uj	29.24 uj	49.5	2.1596	0.1285	18232 B
`STJSrcGen`	15	40096	24,922.1 ns	156.17 ns	146.08 ns	856.15 uj	5.08 uj	28.39 uj	49.53	2.1698	0.1247	18232 B
`Newtonsoft`	15	19840	50,571.6 ns	252.11 ns	235.83 ns	1735.31 uj	7.54 uj	57.16 uj	49.53	4.2843	0.252	36216 B
W100												
`SpanJson`	15	17353	57,383.8 ns	571.99 ns	535.04 ns	1940.95 uj	22.17 uj	64.65 uj	48.53	4.2644	0.4034	35760 B
`Utf8Json`	15	15996	62,744.4 ns	364.60 ns	341.05 ns	2153.59 uj	11.21 uj	70.78 uj	49.43	4.2511	0.1875	35760 B
`STJRefGen`	15	10960	91,364.6 ns	475.36 ns	444.65 ns	3138.10 uj	15.97 uj	103.21 uj	49.1	4.2883	0.4562	36232 B
`STJSrcGen`	15	12240	81,764.9 ns	444.55 ns	415.84 ns	2807.71 uj	14.48 uj	91.71 uj	49.2	4.3301	0.4085	36232 B
`Newtonsoft`	15	9027	105,214.9 ns	453.80 ns	424.48 ns	3609.03 uj	14.34 uj	119.37 uj	48.63	8.3084	1.1078	70216 B
W200												
`SpanJson`	15	7822	127,168.8 ns	837.13 ns	783.05 ns	4354.85 uj	29.48 uj	143.23 uj	48.7	8.5656	1.5341	71760 B
`Utf8Json`	15	8737	114,873.3 ns	1,065.89 ns	997.04 ns	3943.95 uj	34.21 uj	129.69 uj	49.43	8.4697	0.9156	71760 B
`STJRefGen`	15	3888	258,074.7 ns	1,443.75 ns	1,350.48 ns	8864.40 uj	46.63 uj	291.67 uj	49.4	8.4877	1.5432	72232 B
`STJSrcGen`	15	4016	249,749.1 ns	1,420.53 ns	1,328.76 ns	8574.53 uj	44.44 uj	280.36 uj	49.37	8.4661	1.494	72232 B
`Newtonsoft`	15	4442	214,702.5 ns	917.86 ns	858.57 ns	7366.91 uj	28.59 uj	241.17 uj	49.1	16.434	4.0522	139216 B
Serialisation												
W2												
`SpanJson`	15	2763264	359.1 ns	2.03 ns	1.89 ns	12.48 uj	0.06 uj	0.41 uj	49.57	0.0268	0	224 B
`Utf8Json`	15	2165999	463.1 ns	3.48 ns	3.26 ns	15.89 uj	0.11 uj	0.52 uj	49.37	0.0286	0	240 B
`STJRefGen`	15	1481776	672.2 ns	4.25 ns	3.97 ns	23.41 uj	0.15 uj	0.77 uj	49.4	0.0655	0	552 B
`STJSrcGen`	15	2730576	363.2 ns	2.50 ns	2.34 ns	12.77 uj	0.11 uj	0.42 uj	49.73	0.0286	0	240 B
`Newtonsoft`	15	763888	1,299.3 ns	7.32 ns	6.85 ns	44.90 uj	0.24 uj	1.46 uj	49.53	0.2644	0	2216 B
W5												
`SpanJson`	15	705424	1,425.3 ns	10.97 ns	10.26 ns	49.00 uj	0.34 uj	1.60 uj	49.27	0.0822	0	688 B
`Utf8Json`	15	657876	1,524.8 ns	13.43 ns	12.56 ns	52.18 uj	0.47 uj	1.71 uj	49.4	0.0836	0	704 B
`STJRefGen`	15	719776	1,384.3 ns	8.40 ns	7.86 ns	48.20 uj	0.36 uj	1.58 uj	49.87	0.1209	0	1016 B
`STJSrcGen`	15	1098752	908.3 ns	5.29 ns	4.95 ns	31.98 uj	0.21 uj	1.03 uj	50	0.0837	0	704 B
`Newtonsoft`	15	329888	3,029.3 ns	17.12 ns	16.01 ns	104.23 uj	0.60 uj	3.45 uj	49.37	0.6305	0.003	5296 B
W10												
`SpanJson`	15	320480	3,143.5 ns	20.73 ns	19.40 ns	107.09 uj	0.72 uj	3.56 uj	49.13	0.1747	0	1464 B

continued on next page

Library	Iters	Ops	Mean	Error	StdDev	Pkg (µJ/op)	StdDev (µJ/op)	DRAM (µJ/op)	Temp (°C)	Gen0	Gen1	Allocated
`Utf8Json`	15	301377	3,228.7 ns	30.70 ns	28.72 ns	110.60 uj	1.03 uj	3.64 uj	49.13	0.1759	0	1480 B
`STJRefGen`	15	406080	2,423.1 ns	16.75 ns	15.67 ns	84.76 uj	0.60 uj	2.72 uj	49.93	0.2142	0	1792 B
`STJSrcGen`	15	546256	1,829.9 ns	10.96 ns	10.25 ns	63.23 uj	0.43 uj	2.07 uj	50	0.1757	0	1480 B
`Newtonsoft`	15	166080	5,980.2 ns	38.97 ns	36.45 ns	205.37 uj	1.20 uj	6.92 uj	49.57	1.1621	0.012	9736 B
W20												
`SpanJson`	15	139420	7,190.1 ns	113.04 ns	105.74 ns	246.48 uj	3.60 uj	8.09 uj	48.77	0.3658	0	3064 B
`Utf8Json`	15	146838	6,676.6 ns	45.90 ns	42.93 ns	228.91 uj	1.57 uj	7.55 uj	49.17	0.3678	0	3080 B
`STJRefGen`	15	214304	4,665.7 ns	27.34 ns	25.57 ns	163.64 uj	0.96 uj	5.30 uj	50.03	0.4013	0	3392 B
`STJSrcGen`	15	277472	3,612.0 ns	22.79 ns	21.32 ns	127.20 uj	0.71 uj	4.09 uj	49.93	0.3676	0	3080 B
`Newtonsoft`	15	84560	11,809.1 ns	61.67 ns	57.69 ns	406.04 uj	1.88 uj	13.54 uj	49.47	2.2351	0.071	18704 B
W50												
`SpanJson`	15	40047	25,086.0 ns	229.17 ns	214.36 ns	837.61 uj	7.13 uj	28.24 uj	48.4	0.9239	0	7864 B
`Utf8Json`	15	39241	25,444.9 ns	112.57 ns	105.30 ns	859.88 uj	3.93 uj	28.53 uj	48.53	0.9174	0	7880 B
`STJRefGen`	15	80560	12,429.8 ns	68.23 ns	63.82 ns	438.47 uj	2.16 uj	13.96 uj	50.07	0.9682	0	8192 B
`STJSrcGen`	15	107680	9,013.6 ns	55.86 ns	52.25 ns	314.50 uj	2.39 uj	10.16 uj	50.1	0.938	0	7880 B
`Newtonsoft`	15	34096	29,562.5 ns	173.80 ns	162.58 ns	1014.91 uj	5.45 uj	33.95 uj	49.47	4.9273	0.176	41368 B
W100												
`SpanJson`	15	15283	63,628.8 ns	594.68 ns	556.26 ns	2128.52 uj	19.00 uj	71.51 uj	48.07	1.8321	0	15864 B
`Utf8Json`	15	16529	60,555.0 ns	466.54 ns	436.40 ns	2027.66 uj	15.03 uj	68.09 uj	48.17	1.8755	0	15880 B
`STJRefGen`	15	40224	25,033.7 ns	138.59 ns	129.64 ns	882.91 uj	4.45 uj	28.07 uj	49.9	1.9143	0	16192 B
`STJSrcGen`	15	53744	18,344.6 ns	136.15 ns	127.36 ns	646.35 uj	4.43 uj	20.81 uj	49.9	1.8793	0	15880 B
`Newtonsoft`	15	16434	61,276.5 ns	545.94 ns	510.67 ns	2102.38 uj	17.33 uj	70.99 uj	49.13	9.6751	0.9127	81440 B
W200												
`SpanJson`	15	7184	139,554.9 ns	990.97 ns	926.95 ns	4653.57 uj	30.48 uj	156.51 uj	47.8	3.7584	0	32360 B
`Utf8Json`	15	7215	138,790.8 ns	834.44 ns	780.53 ns	4642.37 uj	24.56 uj	155.66 uj	47.77	3.7422	0	32376 B
`STJRefGen`	15	12384	81,047.5 ns	522.17 ns	488.44 ns	2782.98 uj	16.54 uj	91.86 uj	49.4	3.876	0	32688 B
`STJSrcGen`	15	26608	37,712.7 ns	221.99 ns	207.65 ns	1326.61 uj	8.25 uj	42.93 uj	49.87	3.8334	0	32376 B
`Newtonsoft`	15	7097	134,029.2 ns	959.65 ns	897.65 ns	4598.51 uj	30.32 uj	150.76 uj	49.2	21.2766	3.5226	179152 B

A.3.4 String Length

Table A.6: Full per-configuration BenchmarkDotNet report for the String Length isolation sweep: timing, Package and DRAM energy per operation, GC counts, and allocations. Rows are grouped by operation and then by String Length level, with the five libraries listed within each level. Within each level the Package-energy cell is shaded from the lowest-energy library (light) to the highest (dark); the shading is not comparable across levels.

Library	Iters	Ops	Mean	Error	StdDev	Pkg (µJ/op)	StdDev (µJ/op)	DRAM (µJ/op)	Temp (°C)	Gen0	Gen1	Allocated
Deserialisation												
L5												
`SpanJson`	15	176525	5.689 s	0.0424 s	0.0397 s	200.62 uj	1.39 uj	6.44 uj	50.7	0.4645	0.0057	3.83 KB
`Utf8Json`	15	133906	7.329 s	0.0680 s	0.0636 s	258.62 uj	2.21 uj	8.24 uj	51.03	0.463	0	3.83 KB
`STJRefGen`	15	103120	9.629 s	0.0519 s	0.0485 s	339.91 uj	1.73 uj	10.88 uj	50.73	0.5237	0	4.29 KB
`STJSrcGen`	15	109344	9.123 s	0.0591 s	0.0553 s	322.46 uj	2.00 uj	10.34 uj	50.87	0.5213	0	4.29 KB
`Newtonsoft`	15	62736	15.933 s	0.0970 s	0.0907 s	562.23 uj	3.19 uj	18.00 uj	50.57	1.1795	0.0159	9.7 KB
L20												
`SpanJson`	15	135990	7.384 s	0.0478 s	0.0447 s	260.31 uj	1.52 uj	8.37 uj	50.7	0.8309	0.0147	6.8 KB
`Utf8Json`	15	91417	10.997 s	0.0488 s	0.0457 s	387.87 uj	1.62 uj	12.34 uj	51.4	0.8314	0.0109	6.8 KB
`STJRefGen`	15	100688	9.849 s	0.0621 s	0.0581 s	348.12 uj	1.92 uj	11.23 uj	51.2	0.8839	0.0199	7.26 KB
`STJSrcGen`	66	41056	10.060 s	0.1986 s	0.4682 s	355.08 uj	16.41 uj	11.36 uj	51.39	0.8769	0	7.26 KB
`Newtonsoft`	15	50496	19.887 s	0.1586 s	0.1484 s	701.27 uj	5.41 uj	22.63 uj	50.87	1.8813	0.0396	15.45 KB
L50												
`SpanJson`	15	85532	11.678 s	0.1233 s	0.1153 s	408.15 uj	4.44 uj	13.29 uj	50.53	1.555	0.0585	12.73 KB
`Utf8Json`	15	48864	20.542 s	0.1009 s	0.0944 s	724.45 uj	3.30 uj	23.25 uj	50.7	1.5553	0.0205	12.73 KB
`STJRefGen`	15	86208	11.467 s	0.0874 s	0.0818 s	404.92 uj	2.76 uj	12.98 uj	51.03	1.6124	0.0696	13.2 KB
`STJSrcGen`	15	87488	11.415 s	0.0739 s	0.0691 s	403.07 uj	2.53 uj	13.11 uj	50.8	1.6116	0.0686	13.2 KB
`Newtonsoft`	15	38560	24.861 s	0.1462 s	0.1368 s	877.16 uj	4.88 uj	28.63 uj	50.67	3.2936	0.2593	26.95 KB
L200												
`SpanJson`	15	41047	24.499 s	0.1606 s	0.1503 s	843.75 uj	5.18 uj	27.48 uj	50.4	4.8968	0.5603	40.2 KB
`Utf8Json`	15	18559	54.053 s	0.3295 s	0.3082 s	1906.73 uj	10.36 uj	60.77 uj	50.87	4.9033	0.3233	40.2 KB
`STJRefGen`	15	69696	14.418 s	0.0861 s	0.0806 s	509.98 uj	2.94 uj	17.00 uj	50.87	4.9644	0.5883	40.66 KB
`STJSrcGen`	15	70000	14.379 s	0.0866 s	0.0810 s	507.66 uj	3.15 uj	16.84 uj	50.83	4.9714	0.6	40.66 KB
`Newtonsoft`	15	18064	55.560 s	0.2390 s	0.2236 s	1953.23 uj	5.90 uj	64.41 uj	50.43	10.5182	1.3286	86.27 KB
L500												

continued on next page

Library	Iters	Ops	Mean	Error	StdDev	Pkg (µJ/op)	StdDev (µJ/op)	DRAM (µJ/op)	Temp (°C)	Gen0	Gen1	Allocated
`SpanJson`	15	20112	48.913 s	0.4915 s	0.4597 s	1680.04 uj	16.10 uj	55.00 uj	50.3	11.7343	2.4364	95.86 KB
`Utf8Json`	15	7934	121.468 s	0.9876 s	0.9238 s	4284.61 uj	32.31 uj	137.48 uj	51.27	11.7217	1.7646	95.86 KB
`STJRefGen`	15	52080	19.467 s	0.1262 s	0.1181 s	687.46 uj	4.10 uj	23.75 uj	50.63	11.7896	2.3425	96.32 KB
`STJSrcGen`	15	52272	19.386 s	0.1282 s	0.1199 s	687.60 uj	4.61 uj	23.69 uj	50.67	11.7845	2.3531	96.32 KB
`Newtonsoft`	15	5088	167.191 s	0.9974 s	0.9330 s	5675.50 uj	36.69 uj	191.34 uj	49.03	38.7186	25.5503	221.69 KB
L2000												
`SpanJson`	15	4218	234.740 s	2.0756 s	1.9416 s	8075.05 uj	65.32 uj	269.29 uj	49.93	45.7563	16.5955	374.18 KB
`Utf8Json`	15	2417	411.298 s	3.2627 s	3.0519 s	14545.59 uj	99.85 uj	464.73 uj	51.33	45.511	16.1357	374.18 KB
`STJRefGen`	66	7168	50.542 s	0.9964 s	2.3485 s	1792.82 uj	81.88 uj	59.45 uj	50.24	45.7589	17.8571	374.64 KB
`STJSrcGen`	15	20416	49.835 s	0.3894 s	0.3642 s	1785.28 uj	14.03 uj	64.08 uj	50.17	45.8464	17.8781	374.64 KB
`Newtonsoft`	15	1714	583.412 s	2.4924 s	2.3314 s	19760.63 uj	85.23 uj	678.98 uj	49.2	129.5216	98.5998	754.42 KB
L10000												
`SpanJson`	37	486	1,156.502 s	22.6938 s	38.5358 s	39798.18 uj	1363.63 uj	1357.59 uj	50.19	226.3374	135.8025	1858.55 KB
`Utf8Json`	34	295	2,038.287 s	39.8426 s	64.3384 s	71956.91 uj	2232.56 uj	2373.61 uj	51.38	227.1186	179.661	1858.55 KB
`STJRefGen`	15	4752	215.770 s	3.0763 s	2.8776 s	7682.61 uj	108.28 uj	289.18 uj	49.97	227.0623	136.1532	1859.02 KB
`STJSrcGen`	72	1680	215.984 s	4.2847 s	10.5907 s	7696.81 uj	371.59 uj	291.40 uj	49.97	226.7857	135.7143	1859.02 KB
`Newtonsoft`	15	310	3,230.050 s	27.7344 s	25.9427 s	109470.66 uj	872.16 uj	4124.18 uj	48.93	490.3226	441.9355	3843.84 KB
Serialisation												
L5												
`SpanJson`	15	358610	2.706 s	0.0251 s	0.0234 s	95.49 uj	0.79 uj	3.06 uj	51.4	0.1952	0	1.6 KB
`Utf8Json`	15	319092	3.055 s	0.0346 s	0.0324 s	107.61 uj	1.15 uj	3.43 uj	50.6	0.1943	0	1.61 KB
`STJRefGen`	15	225856	4.423 s	0.0250 s	0.0234 s	156.29 uj	0.84 uj	4.98 uj	51.3	0.2302	0	1.91 KB
`STJSrcGen`	15	301120	3.316 s	0.0217 s	0.0203 s	117.12 uj	0.72 uj	3.75 uj	51.33	0.1959	0	1.61 KB
`Newtonsoft`	15	128608	7.743 s	0.0428 s	0.0400 s	273.41 uj	1.43 uj	8.88 uj	50.93	1.2208	0.0156	10.02 KB
L20												
`SpanJson`	15	144379	6.944 s	0.0567 s	0.0530 s	243.81 uj	2.08 uj	7.79 uj	50.83	0.3602	0	2.99 KB
`Utf8Json`	15	142604	6.837 s	0.0618 s	0.0578 s	239.87 uj	1.62 uj	7.69 uj	50.7	0.3646	0	3.01 KB
`STJRefGen`	15	210528	4.730 s	0.0317 s	0.0297 s	170.46 uj	1.00 uj	5.39 uj	51.43	0.4037	0	3.31 KB
`STJSrcGen`	15	273808	3.642 s	0.0240 s	0.0225 s	129.39 uj	0.82 uj	4.10 uj	51.53	0.3652	0	3.01 KB
`Newtonsoft`	15	85856	11.573 s	0.0678 s	0.0634 s	408.37 uj	2.25 uj	13.36 uj	50.73	2.2247	0.0699	18.27 KB
L50												
`SpanJson`	15	44730	21.662 s	0.1843 s	0.1724 s	743.17 uj	6.01 uj	24.32 uj	49.67	0.693	0	5.77 KB

continued on next page

Library	Iters	Ops	Mean	Error	StdDev	Pkg (µJ/op)	StdDev (µJ/op)	DRAM (µJ/op)	Temp (°C)	Gen0	Gen1	Allocated
`Utf8Json`	15	46545	21.514 s	0.2251 s	0.2105 s	738.55 uj	7.07 uj	24.12 uj	49.33	0.6875	0	5.79 KB
`STJRefGen`	15	175568	5.634 s	0.0326 s	0.0305 s	201.29 uj	1.34 uj	6.33 uj	51.33	0.7405	0	6.09 KB
`STJSrcGen`	15	242864	4.132 s	0.0249 s	0.0233 s	147.01 uj	0.71 uj	4.67 uj	51.5	0.7082	0	5.79 KB
`Newtonsoft`	15	59952	15.875 s	0.1439 s	0.1346 s	560.54 uj	5.04 uj	18.09 uj	51.07	4.2367	0.1334	34.69 KB
L200												
`SpanJson`	15	8108	123.918 s	0.9358 s	0.8754 s	4250.98 uj	30.56 uj	138.89 uj	49.1	2.3434	0	19.69 KB
`Utf8Json`	15	9703	100.186 s	0.8647 s	0.8089 s	3451.89 uj	28.86 uj	112.57 uj	49.83	2.3704	0	19.7 KB
`STJRefGen`	15	109968	9.170 s	0.0525 s	0.0491 s	329.25 uj	2.02 uj	10.68 uj	51.13	2.4371	0	20.01 KB
`STJSrcGen`	15	150128	6.715 s	0.0491 s	0.0459 s	242.98 uj	1.56 uj	7.74 uj	51.13	2.398	0	19.7 KB
`Newtonsoft`	15	25792	38.935 s	0.2275 s	0.2128 s	1379.13 uj	7.10 uj	45.24 uj	50.87	13.1436	1.6284	107.82 KB
L500												
`SpanJson`	15	3098	312.254 s	2.7705 s	2.5915 s	10709.90 uj	88.98 uj	350.57 uj	48.93	5.4874	0	47.52 KB
`Utf8Json`	15	3861	254.788 s	1.9277 s	1.8031 s	8734.67 uj	62.48 uj	285.50 uj	49.23	5.698	0	47.54 KB
`STJRefGen`	15	71968	13.974 s	0.0849 s	0.0794 s	504.03 uj	2.88 uj	16.04 uj	50.37	5.8359	0	47.84 KB
`STJSrcGen`	15	86064	11.626 s	0.1010 s	0.0945 s	417.56 uj	3.28 uj	13.53 uj	50.27	5.7748	0	47.54 KB
`Newtonsoft`	15	5360	179.656 s	1.9875 s	1.8591 s	6121.72 uj	63.83 uj	211.54 uj	49.07	43.2836	27.6119	254.13 KB
L2000												
`SpanJson`	15	992	1,031.391 s	19.2505 s	18.0070 s	35089.90 uj	579.13 uj	1155.38 uj	49.3	43.3468	43.3468	186.73 KB
`Utf8Json`	19	833	1,215.914 s	23.6683 s	26.3072 s	41113.53 uj	867.62 uj	1375.04 uj	49.08	96.0384	96.0384	570.83 KB
`STJRefGen`	45	8416	108.380 s	2.1362 s	4.0644 s	3745.11 uj	134.75 uj	144.92 uj	48.64	12.8327	12.8327	187.06 KB
`STJSrcGen`	41	10928	98.108 s	1.9576 s	3.5299 s	3386.69 uj	117.41 uj	134.52 uj	48.33	10.7064	10.7064	186.7 KB
`Newtonsoft`	15	1392	691.293 s	3.4823 s	3.2573 s	23425.70 uj	119.05 uj	793.61 uj	48.73	161.6379	135.0575	938.82 KB
L10000												
`SpanJson`	15	178	5,426.689 s	77.6622 s	72.6452 s	183923.74 uj	2238.46 uj	6122.91 uj	49.13	168.5393	168.5393	929.08 KB
`Utf8Json`	23	166	5,884.040 s	114.8055 s	145.1922 s	200160.41 uj	4564.03 uj	6797.35 uj	49.46	433.7349	433.7349	2849.37 KB
`STJRefGen`	66	944	447.367 s	8.8871 s	20.9479 s	15398.52 uj	705.99 uj	650.96 uj	48.73	56.1441	56.1441	929.58 KB
`STJSrcGen`	23	2272	427.831 s	8.4063 s	10.6312 s	14808.51 uj	351.54 uj	629.11 uj	48.7	56.7782	56.7782	928.93 KB
`Newtonsoft`	15	304	2,670.778 s	14.6863 s	13.7376 s	89739.61 uj	573.77 uj	3602.82 uj	48.43	730.2632	667.7632	4665.65 KB

A.3.5 Numeric Length and Type

Table A.7: Full per-configuration BenchmarkDotNet report for the Numeric Length isolation sweep: timing, Package and DRAM energy per operation, GC counts, and allocations. Rows are grouped by operation and then by Numeric Length level, with the five libraries listed within each level. Within each level the Package-energy cell is shaded from the lowest-energy library (light) to the highest (dark); the shading is not comparable across levels.

Library	Iters	Ops	Mean	Error	StdDev	Pkg (μJ/op)	StdDev (μJ/op)	DRAM (μJ/op)	Temp (°C)	Gen0	Gen1	Allocated
Deserialisation												
F2												
`SpanJson`	15	157872	6.360 s	0.0484 s	0.0453 s	212.78 uj	1.63 uj	7.16 uj	49.47	0.1013	0	880 B
`Utf8Json`	15	119530	8.385 s	0.0426 s	0.0399 s	285.26 uj	1.78 uj	9.40 uj	49.77	0.1004	0	880 B
`STJRefGen`	15	87104	11.515 s	0.0890 s	0.0832 s	394.06 uj	3.25 uj	12.95 uj	49.27	0.1607	0	1352 B
`STJSrcGen`	15	89648	11.187 s	0.0897 s	0.0839 s	381.65 uj	3.15 uj	12.54 uj	49.37	0.1562	0	1352 B
`Newtonsoft`	15	45280	22.186 s	0.1465 s	0.1370 s	758.55 uj	4.69 uj	25.08 uj	49.27	1.3472	0.0221	11448 B
I2												
`SpanJson`	15	398494	2.518 s	0.0152 s	0.0142 s	86.43 uj	0.48 uj	2.82 uj	50.43	0.0602	0	520 B
`Utf8Json`	15	290567	3.361 s	0.0240 s	0.0224 s	115.31 uj	0.75 uj	3.78 uj	50.43	0.0619	0	520 B
`STJRefGen`	15	137280	7.308 s	0.0466 s	0.0436 s	251.03 uj	1.48 uj	8.22 uj	50.43	0.1166	0	992 B
`STJSrcGen`	15	134880	7.451 s	0.0507 s	0.0474 s	255.97 uj	1.59 uj	8.36 uj	50.87	0.1112	0	992 B
`Newtonsoft`	15	66496	15.094 s	0.1072 s	0.1002 s	518.26 uj	3.36 uj	17.09 uj	50.6	0.9324	0	7856 B
F3												
`SpanJson`	15	152672	6.565 s	0.0409 s	0.0382 s	223.89 uj	1.37 uj	7.38 uj	50.07	0.1048	0	880 B
`Utf8Json`	15	110626	9.042 s	0.0641 s	0.0599 s	308.87 uj	1.80 uj	10.13 uj	49.93	0.0994	0	880 B
`STJRefGen`	15	85680	11.711 s	0.0783 s	0.0732 s	401.98 uj	2.41 uj	13.13 uj	49.8	0.1517	0	1352 B
`STJSrcGen`	15	84864	11.814 s	0.0788 s	0.0737 s	405.46 uj	2.48 uj	13.29 uj	49.83	0.1532	0	1352 B
`Newtonsoft`	15	43872	22.850 s	0.1599 s	0.1496 s	784.17 uj	5.06 uj	25.82 uj	49.93	1.3904	0.0228	11640 B
I3												
`SpanJson`	15	376287	2.668 s	0.0208 s	0.0194 s	91.55 uj	0.67 uj	2.99 uj	50.5	0.0611	0	520 B
`Utf8Json`	15	266290	3.672 s	0.0314 s	0.0294 s	125.05 uj	1.12 uj	4.13 uj	50.33	0.0601	0	520 B
`STJRefGen`	15	133584	7.506 s	0.0469 s	0.0439 s	257.99 uj	1.51 uj	8.44 uj	50.53	0.1123	0	992 B
`STJSrcGen`	15	132752	7.558 s	0.0574 s	0.0537 s	259.69 uj	1.82 uj	8.51 uj	50.47	0.113	0	992 B
`Newtonsoft`	15	63872	15.691 s	0.1135 s	0.1062 s	538.74 uj	3.59 uj	17.73 uj	50.43	0.955	0.0157	8048 B
F5												

continued on next page

Library	Iters	Ops	Mean	Error	StdDev	Pkg (µJ/op)	StdDev (µJ/op)	DRAM (µJ/op)	Temp (°C)	Gen0	Gen1	Allocated
`SpanJson`	15	134736	7.226 s	0.0488 s	0.0457 s	246.97 uj	1.47 uj	8.12 uj	49.9	0.1039	0	880 B
`Utf8Json`	15	99677	10.064 s	0.0552 s	0.0516 s	345.09 uj	1.76 uj	11.28 uj	50.37	0.1003	0	880 B
`STJRefGen`	15	79600	12.604 s	0.0864 s	0.0808 s	432.76 uj	2.79 uj	14.14 uj	50.2	0.1508	0	1352 B
`STJSrcGen`	15	81488	12.305 s	0.0748 s	0.0700 s	422.32 uj	2.46 uj	13.82 uj	50.43	0.1595	0	1352 B
`Newtonsoft`	15	40288	24.940 s	0.1842 s	0.1723 s	855.95 uj	5.66 uj	28.07 uj	50.3	1.5141	0.0248	12776 B
I5												
`SpanJson`	15	337954	2.974 s	0.0144 s	0.0134 s	102.04 uj	0.44 uj	3.33 uj	50.3	0.0621	0	520 B
`Utf8Json`	15	258879	3.775 s	0.0292 s	0.0273 s	128.78 uj	0.98 uj	4.24 uj	50.4	0.0618	0	520 B
`STJRefGen`	15	125600	7.982 s	0.0479 s	0.0448 s	274.21 uj	1.70 uj	8.96 uj	50.7	0.1115	0	992 B
`STJSrcGen`	15	128672	7.791 s	0.0511 s	0.0478 s	267.49 uj	1.62 uj	8.76 uj	51.07	0.1166	0	992 B
`Newtonsoft`	15	61280	16.401 s	0.1211 s	0.1133 s	562.60 uj	3.82 uj	18.55 uj	50.53	0.9954	0.0163	8432 B
F7												
`SpanJson`	15	121072	8.242 s	0.0707 s	0.0662 s	276.94 uj	2.20 uj	9.25 uj	49.8	0.0991	0	880 B
`Utf8Json`	15	80989	12.003 s	0.0676 s	0.0632 s	410.08 uj	2.11 uj	13.48 uj	50.27	0.0988	0	880 B
`STJRefGen`	15	77040	13.017 s	0.0900 s	0.0842 s	446.79 uj	2.84 uj	14.59 uj	50.4	0.1558	0	1352 B
`STJSrcGen`	15	79120	12.671 s	0.0814 s	0.0762 s	435.00 uj	2.65 uj	14.21 uj	50.33	0.1517	0	1352 B
`Newtonsoft`	15	38464	26.089 s	0.1976 s	0.1849 s	894.98 uj	6.40 uj	29.32 uj	50.3	1.5599	0.026	13160 B
I7												
`SpanJson`	15	308010	3.260 s	0.0301 s	0.0282 s	111.90 uj	0.93 uj	3.67 uj	50.47	0.0617	0	520 B
`Utf8Json`	15	244274	4.095 s	0.0309 s	0.0289 s	140.23 uj	1.05 uj	4.60 uj	50.2	0.0614	0	520 B
`STJRefGen`	15	118752	8.449 s	0.0565 s	0.0529 s	290.19 uj	1.87 uj	9.50 uj	50.7	0.1179	0	992 B
`STJSrcGen`	15	120640	8.320 s	0.0567 s	0.0530 s	285.78 uj	1.78 uj	9.36 uj	50.5	0.116	0	992 B
`Newtonsoft`	15	56848	17.659 s	0.1104 s	0.1032 s	606.17 uj	3.61 uj	19.99 uj	50.27	1.0379	0.0176	8808 B
F10												
`SpanJson`	15	106048	9.398 s	0.0648 s	0.0606 s	313.90 uj	2.01 uj	10.56 uj	48.67	0.1037	0	880 B
`Utf8Json`	15	74798	13.039 s	0.0912 s	0.0853 s	437.99 uj	2.72 uj	14.62 uj	48.83	0.0936	0	880 B
`STJRefGen`	15	70560	14.188 s	0.0992 s	0.0928 s	481.68 uj	3.96 uj	15.91 uj	48.2	0.1559	0	1352 B
`STJSrcGen`	15	71104	14.109 s	0.0848 s	0.0793 s	480.36 uj	2.76 uj	15.82 uj	48.4	0.1547	0	1352 B
`Newtonsoft`	15	34816	28.849 s	0.2283 s	0.2135 s	984.34 uj	8.85 uj	32.41 uj	48.47	1.7233	0.0287	14488 B
I10												
`SpanJson`	15	269063	3.728 s	0.0154 s	0.0144 s	127.91 uj	0.47 uj	4.18 uj	50.43	0.0595	0	520 B
`Utf8Json`	15	227238	4.413 s	0.0312 s	0.0292 s	151.30 uj	0.99 uj	4.96 uj	50.33	0.0616	0	520 B
`STJRefGen`	15	110368	9.097 s	0.0603 s	0.0564 s	312.30 uj	1.89 uj	10.22 uj	50.63	0.1178	0	992 B

continued on next page

Library	Iters	Ops	Mean	Error	StdDev	Pkg (µJ/op)	StdDev (µJ/op)	DRAM (µJ/op)	Temp (°C)	Gen0	Gen1	Allocated
STJSrcGen	15	111152	9.028 s	0.0564 s	0.0527 s	310.16 uj	1.82 uj	10.16 uj	50.7	0.117	0	992 B
Newtonsoft	15	52912	18.980 s	0.1211 s	0.1133 s	651.33 uj	3.88 uj	21.48 uj	50.2	1.1151	0.0189	9376 B
Serialisation												
F2												
SpanJson	15	125264	8.035 s	0.1171 s	0.1095 s	270.30 uj	4.06 uj	9.05 uj	49.97	0.1437	0	1256 B
Utf8Json	15	131561	7.647 s	0.0501 s	0.0468 s	261.60 uj	1.80 uj	8.57 uj	50.07	0.152	0	1272 B
STJRefGen	15	39456	10.692 s	0.1605 s	0.1501 s	366.81 uj	5.06 uj	12.06 uj	50.2	0.1774	0	1584 B
STJSrcGen	15	121072	8.294 s	0.1298 s	0.1214 s	281.22 uj	3.58 uj	9.34 uj	50.2	0.1487	0	1272 B
Newtonsoft	15	58752	17.091 s	0.1228 s	0.1149 s	586.72 uj	3.78 uj	19.24 uj	50.1	1.7021	0.017	14344 B
I2												
SpanJson	15	804671	1.247 s	0.0077 s	0.0072 s	42.89 uj	0.30 uj	1.40 uj	50.77	0.1379	0	1160 B
Utf8Json	15	590032	1.697 s	0.0114 s	0.0106 s	58.25 uj	0.35 uj	1.91 uj	50.4	0.139	0	1176 B
STJRefGen	15	318336	3.085 s	0.0232 s	0.0217 s	106.27 uj	0.74 uj	3.49 uj	50.77	0.1759	0	1488 B
STJSrcGen	15	565568	1.774 s	0.0122 s	0.0114 s	61.09 uj	0.39 uj	2.02 uj	51.23	0.1397	0	1176 B
Newtonsoft	15	138944	7.185 s	0.0549 s	0.0513 s	246.99 uj	1.77 uj	8.28 uj	50.53	1.3243	0.0144	11112 B
F3												
SpanJson	15	109328	9.226 s	0.0690 s	0.0646 s	309.28 uj	2.26 uj	10.38 uj	49.63	0.1555	0	1352 B
Utf8Json	15	114462	8.509 s	0.0739 s	0.0691 s	289.10 uj	2.46 uj	9.56 uj	50.17	0.1573	0	1368 B
STJRefGen	15	29936	11.643 s	0.2006 s	0.1876 s	399.75 uj	6.48 uj	13.09 uj	50.23	0.2004	0	1680 B
STJSrcGen	15	109376	9.166 s	0.1005 s	0.0940 s	308.06 uj	3.49 uj	10.31 uj	50	0.1554	0	1368 B
Newtonsoft	15	56560	17.692 s	0.1240 s	0.1160 s	607.29 uj	3.92 uj	19.85 uj	50.4	1.7327	0.0177	14632 B
I3												
SpanJson	15	696337	1.443 s	0.0180 s	0.0168 s	49.62 uj	0.57 uj	1.63 uj	50.97	0.1494	0	1256 B
Utf8Json	15	502201	2.041 s	0.0380 s	0.0356 s	70.03 uj	1.21 uj	2.30 uj	50.4	0.1513	0	1272 B
STJRefGen	15	319600	3.071 s	0.0224 s	0.0209 s	106.95 uj	0.76 uj	3.46 uj	51.2	0.1877	0	1584 B
STJSrcGen	15	509616	1.968 s	0.0149 s	0.0139 s	67.72 uj	0.47 uj	2.23 uj	51.23	0.1511	0	1272 B
Newtonsoft	15	117888	8.428 s	0.0662 s	0.0619 s	289.46 uj	2.09 uj	9.65 uj	50.43	1.3572	0.017	11400 B
F5												
SpanJson	15	88928	11.301 s	0.1445 s	0.1352 s	377.34 uj	4.68 uj	12.71 uj	49.37	0.1799	0	1544 B
Utf8Json	15	97405	10.008 s	0.0509 s	0.0477 s	339.10 uj	2.17 uj	11.24 uj	50.2	0.1848	0	1560 B
STJRefGen	15	35664	13.874 s	0.1999 s	0.1870 s	466.05 uj	6.11 uj	15.57 uj	49.67	0.1963	0	1872 B
STJSrcGen	15	88144	11.378 s	0.1303 s	0.1219 s	380.15 uj	4.22 uj	12.79 uj	49.6	0.1815	0	1560 B

continued on next page

Library	Iters	Ops	Mean	Error	StdDev	Pkg (µJ/op)	StdDev (µJ/op)	DRAM (µJ/op)	Temp (°C)	Gen0	Gen1	Allocated
`Newtonsoft`	15	50320	19.985 s	0.1449 s	0.1355 s	680.36 uj	5.60 uj	22.73 uj	49.8	1.9078	0.0199	15960 B
I5												
`SpanJson`	15	533219	1.881 s	0.0137 s	0.0128 s	64.61 uj	0.44 uj	2.11 uj	50.77	0.1725	0	1448 B
`Utf8Json`	15	407579	2.453 s	0.0207 s	0.0194 s	83.50 uj	0.73 uj	2.77 uj	50.33	0.1742	0	1464 B
`STJRefGen`	15	298752	3.266 s	0.0252 s	0.0235 s	112.74 uj	0.85 uj	3.70 uj	50.93	0.2109	0	1776 B
`STJSrcGen`	15	489392	2.046 s	0.0133 s	0.0125 s	70.38 uj	0.43 uj	2.30 uj	50.8	0.1737	0	1464 B
`Newtonsoft`	15	113296	8.818 s	0.0638 s	0.0597 s	302.91 uj	2.00 uj	10.12 uj	50.5	1.4299	0.0177	11976 B
F7												
`SpanJson`	15	77776	12.453 s	0.0956 s	0.0894 s	417.16 uj	3.17 uj	14.00 uj	49.97	0.2057	0	1736 B
`Utf8Json`	15	87521	11.485 s	0.1232 s	0.1153 s	394.05 uj	3.94 uj	12.90 uj	50.03	0.2057	0	1744 B
`STJRefGen`	15	36000	14.227 s	0.1432 s	0.1339 s	477.87 uj	4.62 uj	15.97 uj	49.97	0.2222	0	2056 B
`STJSrcGen`	15	83104	12.115 s	0.0873 s	0.0816 s	407.99 uj	2.68 uj	13.63 uj	49.9	0.2046	0	1744 B
`Newtonsoft`	15	47920	20.885 s	0.1329 s	0.1243 s	716.29 uj	4.34 uj	23.76 uj	50.23	1.9616	0.0209	16528 B
I7												
`SpanJson`	15	434323	2.313 s	0.0151 s	0.0142 s	79.46 uj	0.46 uj	2.62 uj	50.73	0.1957	0	1640 B
`Utf8Json`	15	348599	2.866 s	0.0221 s	0.0207 s	96.84 uj	0.77 uj	3.23 uj	50.13	0.1951	0	1648 B
`STJRefGen`	15	292256	3.379 s	0.0245 s	0.0229 s	116.78 uj	0.89 uj	3.84 uj	51.43	0.2327	0	1960 B
`STJSrcGen`	15	451088	2.217 s	0.0163 s	0.0152 s	76.22 uj	0.53 uj	2.51 uj	51.07	0.1951	0	1648 B
`Newtonsoft`	15	101760	9.629 s	0.0765 s	0.0716 s	330.77 uj	2.50 uj	11.03 uj	50.67	1.4937	0.0295	12536 B
F10												
`SpanJson`	15	68064	14.069 s	0.1139 s	0.1065 s	470.94 uj	3.95 uj	15.82 uj	49.77	0.2351	0	2016 B
`Utf8Json`	15	73293	13.360 s	0.0618 s	0.0578 s	456.18 uj	2.44 uj	15.01 uj	50.27	0.2456	0	2104 B
`STJRefGen`	16	24368	15.739 s	0.3084 s	0.3029 s	533.45 uj	9.75 uj	17.66 uj	49.91	0.2462	0	2344 B
`STJSrcGen`	15	75472	13.311 s	0.0926 s	0.0866 s	446.80 uj	2.79 uj	14.96 uj	49.9	0.2385	0	2032 B
`Newtonsoft`	15	44752	22.451 s	0.1551 s	0.1451 s	770.16 uj	5.08 uj	25.61 uj	50.1	2.1675	0.0223	18144 B
I10												
`SpanJson`	15	345322	2.910 s	0.0298 s	0.0279 s	99.88 uj	0.93 uj	3.29 uj	50.53	0.2288	0	1920 B
`Utf8Json`	15	279408	3.586 s	0.0227 s	0.0212 s	120.15 uj	0.79 uj	4.03 uj	50.1	0.2291	0	1936 B
`STJRefGen`	15	272976	3.589 s	0.0271 s	0.0254 s	123.56 uj	0.90 uj	4.07 uj	50.77	0.2674	0	2248 B
`STJSrcGen`	15	424944	2.353 s	0.0160 s	0.0150 s	80.92 uj	0.51 uj	2.67 uj	50.63	0.2306	0	1936 B
`Newtonsoft`	15	94560	10.432 s	0.0817 s	0.0764 s	358.07 uj	2.69 uj	11.75 uj	50.4	1.5969	0.0212	13392 B

A.3.6 String Composition

Table A.8: Full per-configuration BenchmarkDotNet report for the Special-character density (%) isolation sweep: timing, Package and DRAM energy per operation, GC counts, and allocations. Rows are grouped by operation and then by Special-character density (%) level, with the five libraries listed within each level. Within each level the Package-energy cell is shaded from the lowest-energy library (light) to the highest (dark); the shading is not comparable across levels.

Library	Iters	Ops	Mean	Error	StdDev	Pkg (μJ/op)	StdDev (μJ/op)	DRAM (μJ/op)	Temp (°C)	Gen0	Gen1	Allocated
Deserialisation												
A0												
SpanJson	15	136323	7.373 s	0.0471 s	0.0441 s	252.95 uj	1.51 uj	8.33 uj	50.27	0.8289	0.0147	6.8 KB
Utf8Json	15	92889	10.825 s	0.1156 s	0.1081 s	371.53 uj	3.66 uj	12.20 uj	50.83	0.8289	0.0108	6.8 KB
STJRefGen	15	101584	9.738 s	0.0708 s	0.0662 s	334.96 uj	2.34 uj	10.98 uj	50.57	0.886	0.0197	7.26 KB
STJSrcGen	16	35648	9.841 s	0.1863 s	0.1830 s	338.50 uj	6.18 uj	11.15 uj	50.53	0.8696	0	7.26 KB
Newtonsoft	15	50832	19.775 s	0.1467 s	0.1372 s	678.92 uj	4.87 uj	22.52 uj	50.17	1.8886	0.0393	15.45 KB
E5												
SpanJson	15	89762	11.191 s	0.0659 s	0.0617 s	381.22 uj	2.77 uj	12.62 uj	50.1	0.8244	0.0111	6.8 KB
Utf8Json	15	64609	14.952 s	0.1286 s	0.1203 s	513.15 uj	4.12 uj	16.88 uj	50.4	0.8203	0	6.8 KB
STJRefGen	15	66656	14.902 s	0.1219 s	0.1140 s	512.05 uj	3.97 uj	16.89 uj	50.6	0.8851	0.015	7.26 KB
STJSrcGen	15	66640	14.982 s	0.1172 s	0.1096 s	514.69 uj	3.72 uj	16.97 uj	50.53	0.8854	0.015	7.26 KB
Newtonsoft	15	49056	20.481 s	0.1940 s	0.1814 s	703.42 uj	6.25 uj	23.43 uj	50.43	2.1608	0.0815	17.82 KB
U5												
SpanJson	15	98363	10.211 s	0.0557 s	0.0521 s	350.33 uj	1.80 uj	11.46 uj	50.6	0.8235	0.0203	6.8 KB
Utf8Json	15	56680	16.979 s	0.1372 s	0.1283 s	582.74 uj	4.42 uj	19.12 uj	51.1	0.8292	0	6.8 KB
STJRefGen	15	57440	17.266 s	0.1312 s	0.1228 s	592.87 uj	4.24 uj	19.53 uj	50.67	0.8879	0.0174	7.26 KB
STJSrcGen	15	58128	17.051 s	0.1118 s	0.1046 s	585.68 uj	3.91 uj	19.17 uj	50.7	0.8774	0.0172	7.26 KB
Newtonsoft	15	46528	21.551 s	0.1687 s	0.1578 s	739.77 uj	5.35 uj	24.55 uj	50.87	2.2567	0.086	18.46 KB
UE5												
SpanJson	15	96186	10.473 s	0.0695 s	0.0650 s	359.18 uj	2.29 uj	11.86 uj	50.7	0.8317	0.0208	6.8 KB
Utf8Json	15	57498	16.867 s	0.1401 s	0.1310 s	578.96 uj	4.36 uj	18.99 uj	51.07	0.8174	0	6.8 KB
STJRefGen	15	58432	17.024 s	0.1252 s	0.1171 s	584.44 uj	4.02 uj	19.24 uj	50.57	0.8728	0.0171	7.26 KB
STJSrcGen	15	59264	16.643 s	0.1365 s	0.1277 s	571.31 uj	4.26 uj	18.67 uj	50.77	0.8774	0.0169	7.26 KB
Newtonsoft	15	46816	21.396 s	0.1431 s	0.1338 s	734.41 uj	4.59 uj	24.09 uj	50.7	2.2428	0.0854	18.46 KB
E10												

continued on next page

Library	Iters	Ops	Mean	Error	StdDev	Pkg (µJ/op)	StdDev (µJ/op)	DRAM (µJ/op)	Temp (°C)	Gen0	Gen1	Allocated
SpanJson	15	87868	10.980 s	0.0932 s	0.0872 s	376.14 uj	3.09 uj	12.37 uj	50.5	0.8308	0.0114	6.8 KB
Utf8Json	15	60617	15.891 s	0.1660 s	0.1553 s	545.48 uj	5.26 uj	17.87 uj	50.47	0.8249	0	6.8 KB
STJRefGen	15	55504	17.857 s	0.1112 s	0.1040 s	613.00 uj	3.48 uj	20.12 uj	50.63	0.8828	0.018	7.26 KB
STJSrcGen	15	56608	17.593 s	0.1290 s	0.1207 s	603.94 uj	4.13 uj	19.87 uj	50.37	0.8833	0.0177	7.26 KB
Newtonsoft	15	45904	21.885 s	0.1621 s	0.1517 s	751.08 uj	5.06 uj	24.57 uj	50.4	2.2002	0.1307	18.1 KB
U10												
SpanJson	15	82820	11.717 s	0.1276 s	0.1193 s	401.94 uj	4.10 uj	13.18 uj	50.4	0.8211	0.0121	6.8 KB
Utf8Json	15	45336	21.462 s	0.2509 s	0.2347 s	736.47 uj	7.85 uj	24.15 uj	50.4	0.8161	0	6.8 KB
STJRefGen	15	48240	20.681 s	0.1563 s	0.1462 s	709.85 uj	5.00 uj	23.22 uj	50.47	0.8706	0.0207	7.26 KB
STJSrcGen	15	49088	20.310 s	0.1571 s	0.1470 s	697.18 uj	5.03 uj	22.79 uj	50.73	0.876	0.0204	7.26 KB
Newtonsoft	15	44240	22.733 s	0.1863 s	0.1743 s	780.17 uj	5.97 uj	25.61 uj	50.8	2.3508	0.0678	19.32 KB
UE10												
SpanJson	15	84131	11.494 s	0.1308 s	0.1224 s	394.52 uj	4.16 uj	12.96 uj	50.93	0.832	0.0119	6.8 KB
Utf8Json	15	49503	19.471 s	0.1739 s	0.1627 s	668.10 uj	5.48 uj	21.92 uj	50.73	0.8282	0	6.8 KB
STJRefGen	15	47776	20.692 s	0.1629 s	0.1524 s	710.32 uj	5.28 uj	23.26 uj	50.7	0.8791	0.0209	7.26 KB
STJSrcGen	15	48352	20.475 s	0.1793 s	0.1677 s	702.83 uj	5.62 uj	23.02 uj	50.5	0.8686	0.0207	7.26 KB
Newtonsoft	15	43120	23.268 s	0.1940 s	0.1814 s	798.83 uj	6.15 uj	26.15 uj	50.5	2.3423	0.0464	19.32 KB
E25												
SpanJson	15	64032	14.922 s	0.1519 s	0.1421 s	512.08 uj	4.96 uj	16.79 uj	50.43	0.8277	0.0156	6.8 KB
Utf8Json	15	44143	21.979 s	0.2013 s	0.1883 s	754.67 uj	6.44 uj	24.76 uj	50.47	0.8155	0	6.8 KB
STJRefGen	15	42048	23.493 s	0.1778 s	0.1663 s	806.15 uj	5.89 uj	26.52 uj	50.37	0.8799	0	7.26 KB
STJSrcGen	15	42672	23.145 s	0.1826 s	0.1708 s	794.34 uj	5.88 uj	26.15 uj	50.23	0.8671	0	7.26 KB
Newtonsoft	15	40976	24.530 s	0.1622 s	0.1518 s	842.02 uj	5.22 uj	27.88 uj	50.53	2.3184	0.122	19.01 KB
U25												
SpanJson	15	57048	17.026 s	0.1627 s	0.1522 s	584.21 uj	5.23 uj	19.17 uj	50.37	0.8239	0.0175	6.8 KB
Utf8Json	15	27183	35.591 s	0.2754 s	0.2576 s	1221.31 uj	8.46 uj	39.94 uj	50.33	0.8093	0	6.8 KB
STJRefGen	15	27552	36.049 s	0.3173 s	0.2968 s	1235.62 uj	10.54 uj	40.47 uj	50.13	0.8711	0	7.26 KB
STJSrcGen	15	28512	34.899 s	0.2628 s	0.2458 s	1197.12 uj	8.32 uj	39.32 uj	50.23	0.8768	0	7.26 KB
Newtonsoft	15	35408	28.353 s	0.2226 s	0.2082 s	972.73 uj	7.06 uj	31.89 uj	50.33	2.683	0.1695	22.1 KB
UE25												
SpanJson	15	54642	17.742 s	0.1537 s	0.1438 s	606.29 uj	5.56 uj	19.98 uj	50.5	0.8235	0.0183	6.8 KB
Utf8Json	15	28275	34.166 s	0.3708 s	0.3469 s	1171.49 uj	11.75 uj	38.31 uj	50.5	0.8134	0	6.8 KB
STJRefGen	15	28352	35.091 s	0.2870 s	0.2685 s	1198.67 uj	8.72 uj	39.55 uj	50.77	0.8818	0	7.26 KB

continued on next page

Library	Iters	Ops	Mean	Error	StdDev	Pkg (µJ/op)	StdDev (µJ/op)	DRAM (µJ/op)	Temp (°C)	Gen0	Gen1	Allocated
`STJSrcGen`	15	29328	33.897 s	0.2569 s	0.2403 s	1154.34 uj	8.43 uj	38.18 uj	50.63	0.8865	0	7.26 KB
`Newtonsoft`	100	5472	29.601 s	0.8901 s	2.6245 s	1014.54 uj	88.62 uj	33.45 uj	50.58	2.5585	0	22.1 KB
E50												
`SpanJson`	15	45717	20.893 s	0.2018 s	0.1888 s	716.51 uj	6.22 uj	23.44 uj	50.37	0.8312	0	6.8 KB
`Utf8Json`	15	27530	35.271 s	0.3285 s	0.3073 s	1209.72 uj	10.30 uj	39.57 uj	50.03	0.7991	0	6.8 KB
`STJRefGen`	15	25520	38.653 s	0.3724 s	0.3483 s	1315.73 uj	11.82 uj	43.37 uj	49.83	0.8621	0	7.26 KB
`STJSrcGen`	15	22944	43.470 s	0.2831 s	0.2649 s	1471.82 uj	12.96 uj	48.92 uj	49.77	0.8717	0	7.26 KB
`Newtonsoft`	15	36736	27.355 s	0.2295 s	0.2147 s	938.69 uj	7.43 uj	31.17 uj	50.5	2.4771	0.0817	20.4 KB
U50												
`SpanJson`	15	34687	28.006 s	0.2598 s	0.2430 s	949.92 uj	8.78 uj	31.46 uj	50.37	0.8072	0	6.8 KB
`Utf8Json`	15	14435	68.044 s	0.5490 s	0.5135 s	2277.87 uj	16.25 uj	76.29 uj	49.97	0.8313	0	6.8 KB
`STJRefGen`	15	15424	63.982 s	0.4514 s	0.4223 s	2163.61 uj	14.75 uj	71.93 uj	49.77	0.8428	0	7.26 KB
`STJSrcGen`	15	15488	64.389 s	0.5111 s	0.4780 s	2172.40 uj	19.65 uj	72.20 uj	49.73	0.8394	0	7.26 KB
`Newtonsoft`	15	22720	42.093 s	0.2948 s	0.2757 s	1443.11 uj	9.39 uj	47.23 uj	50.3	3.213	0.1761	26.59 KB
UE50												
`SpanJson`	15	39448	24.660 s	0.2473 s	0.2313 s	840.69 uj	8.02 uj	27.66 uj	50.6	0.8112	0	6.8 KB
`Utf8Json`	15	14716	67.184 s	0.4906 s	0.4589 s	2253.04 uj	19.00 uj	75.33 uj	50.17	0.8154	0	6.8 KB
`STJRefGen`	15	15648	63.719 s	0.4591 s	0.4295 s	2157.23 uj	17.53 uj	71.46 uj	50.03	0.8308	0	7.26 KB
`STJSrcGen`	15	15456	64.080 s	0.4096 s	0.3831 s	2159.31 uj	13.81 uj	71.84 uj	49.87	0.8411	0	7.26 KB
`Newtonsoft`	15	24048	41.729 s	0.3048 s	0.2851 s	1429.13 uj	9.73 uj	47.37 uj	50.53	3.2435	0.1663	26.59 KB
E100												
`SpanJson`	15	26424	36.925 s	0.3197 s	0.2990 s	1241.82 uj	10.80 uj	41.41 uj	49.9	0.7947	0	6.8 KB
`Utf8Json`	29	6577	50.347 s	0.9925 s	1.4548 s	1724.26 uj	49.13 uj	56.67 uj	50.16	0.7602	0	6.8 KB
`STJRefGen`	15	20400	48.092 s	0.3721 s	0.3481 s	1641.17 uj	14.22 uj	54.10 uj	50.03	0.8824	0	7.26 KB
`STJSrcGen`	15	20592	47.622 s	0.3586 s	0.3355 s	1617.45 uj	12.95 uj	53.50 uj	49.7	0.8741	0	7.26 KB
`Newtonsoft`	15	32752	30.662 s	0.2699 s	0.2524 s	1052.09 uj	8.45 uj	34.64 uj	50.47	2.8395	0.1832	23.36 KB
U100												
`SpanJson`	15	27173	36.393 s	0.2799 s	0.2618 s	1239.68 uj	11.91 uj	40.79 uj	50.23	0.8096	0	6.8 KB
`Utf8Json`	15	14353	69.324 s	0.5600 s	0.5238 s	2316.08 uj	17.55 uj	77.82 uj	49.53	0.7664	0	6.8 KB
`STJRefGen`	15	15984	62.537 s	0.5087 s	0.4759 s	2130.67 uj	18.18 uj	70.30 uj	50.3	0.8759	0	7.26 KB
`STJSrcGen`	15	15568	64.264 s	0.5292 s	0.4950 s	2198.56 uj	16.53 uj	72.05 uj	50.23	0.835	0	7.26 KB
`Newtonsoft`	15	17504	54.630 s	0.3719 s	0.3479 s	1872.64 uj	12.44 uj	62.19 uj	50.27	4.856	0.5142	40.05 KB
UE100												

continued on next page

Library	Iters	Ops	Mean	Error	StdDev	Pkg (µJ/op)	StdDev (µJ/op)	DRAM (µJ/op)	Temp (°C)	Gen0	Gen1	Allocated
SpanJson	15	27460	35.970 s	0.2862 s	0.2677 s	1213.14 uj	10.60 uj	40.34 uj	50.6	0.8012	0	6.8 KB
Utf8Json	15	14424	68.759 s	0.5429 s	0.5078 s	2294.58 uj	17.04 uj	77.10 uj	49.97	0.8319	0	6.8 KB
STJRefGen	15	15600	63.884 s	0.5221 s	0.4884 s	2177.35 uj	16.73 uj	71.62 uj	50.4	0.8333	0	7.26 KB
STJSrcGen	15	15168	65.558 s	0.4338 s	0.4057 s	2235.23 uj	16.87 uj	73.45 uj	50.4	0.8571	0	7.26 KB
Newtonsoft	15	19200	52.388 s	0.4476 s	0.4187 s	1788.16 uj	13.39 uj	59.51 uj	50.5	4.8438	0.5208	40.05 KB
Serialisation												
A0												
SpanJson	15	156923	6.152 s	0.1100 s	0.1029 s	211.07 uj	3.46 uj	6.94 uj	51.1	0.3632	0	2.99 KB
Utf8Json	15	146371	6.816 s	0.0999 s	0.0934 s	233.61 uj	3.28 uj	7.67 uj	50.67	0.3621	0	3.01 KB
STJRefGen	15	199632	4.947 s	0.0355 s	0.0332 s	173.41 uj	1.37 uj	5.61 uj	51.77	0.4007	0	3.31 KB
STJSrcGen	15	277968	3.590 s	0.0262 s	0.0245 s	125.14 uj	0.92 uj	4.04 uj	51.67	0.3669	0	3.01 KB
Newtonsoft	15	83808	12.007 s	0.0847 s	0.0793 s	412.51 uj	2.71 uj	13.81 uj	50.7	2.2313	0.0716	18.27 KB
E5												
SpanJson	15	121736	8.293 s	0.1223 s	0.1144 s	283.88 uj	3.71 uj	9.31 uj	50.03	0.3697	0	3.09 KB
Utf8Json	15	160767	6.072 s	0.0764 s	0.0714 s	208.57 uj	2.45 uj	6.88 uj	51.23	0.3794	0	3.1 KB
STJRefGen	15	112560	8.775 s	0.0756 s	0.0707 s	303.35 uj	2.49 uj	9.84 uj	51.37	0.4264	0	3.48 KB
STJSrcGen	15	140448	7.101 s	0.0493 s	0.0462 s	244.52 uj	1.57 uj	7.99 uj	51.27	0.3845	0	3.18 KB
Newtonsoft	15	70256	14.304 s	0.1099 s	0.1028 s	491.22 uj	3.49 uj	16.46 uj	50.83	2.2632	0.0569	18.55 KB
U5												
SpanJson	15	130379	7.706 s	0.0786 s	0.0735 s	259.50 uj	2.71 uj	8.66 uj	50.4	0.3758	0	3.12 KB
Utf8Json	15	139437	7.240 s	0.0528 s	0.0494 s	244.23 uj	1.89 uj	8.15 uj	50.47	0.3801	0	3.13 KB
STJRefGen	15	91344	10.558 s	0.0874 s	0.0817 s	362.75 uj	2.75 uj	11.84 uj	51.53	0.4598	0	3.8 KB
STJSrcGen	15	116064	8.603 s	0.0608 s	0.0569 s	295.42 uj	1.94 uj	9.69 uj	51.57	0.4222	0	3.5 KB
Newtonsoft	15	74848	13.048 s	0.1056 s	0.0988 s	447.65 uj	3.35 uj	14.91 uj	50.63	2.2445	0.0668	18.39 KB
UE5												
SpanJson	15	128799	7.770 s	0.1195 s	0.1118 s	263.38 uj	3.62 uj	8.75 uj	50.4	0.3804	0	3.12 KB
Utf8Json	15	135188	7.164 s	0.0783 s	0.0733 s	244.30 uj	2.74 uj	8.04 uj	50.4	0.3773	0	3.13 KB
STJRefGen	15	85712	11.027 s	0.0889 s	0.0831 s	379.30 uj	3.01 uj	12.37 uj	50.9	0.455	0	3.8 KB
STJSrcGen	15	115328	8.674 s	0.0583 s	0.0545 s	299.27 uj	2.06 uj	9.82 uj	51.47	0.4249	0	3.5 KB
Newtonsoft	15	71616	13.175 s	0.1109 s	0.1038 s	452.25 uj	3.50 uj	15.09 uj	50.63	2.2481	0.0698	18.39 KB
E10												
SpanJson	15	143293	7.023 s	0.0953 s	0.0892 s	240.96 uj	3.05 uj	7.87 uj	51.13	0.3838	0	3.19 KB

continued on next page

Library	Iters	Ops	Mean	Error	StdDev	Pkg (µJ/op)	StdDev (µJ/op)	DRAM (µJ/op)	Temp (°C)	Gen0	Gen1	Allocated
`Utf8Json`	15	120740	8.350 s	0.0376 s	0.0352 s	286.90 uj	1.27 uj	9.39 uj	50.83	0.3893	0	3.2 KB
`STJRefGen`	15	90656	10.885 s	0.0889 s	0.0831 s	375.34 uj	2.88 uj	12.21 uj	51.03	0.4412	0	3.63 KB
`STJSrcGen`	15	115344	8.642 s	0.0856 s	0.0800 s	297.80 uj	2.66 uj	9.70 uj	51.6	0.3988	0	3.32 KB
`Newtonsoft`	15	61296	16.399 s	0.1339 s	0.1252 s	562.96 uj	4.34 uj	18.82 uj	50.83	2.3003	0.0653	18.86 KB
U10												
`SpanJson`	15	125409	7.607 s	0.1073 s	0.1003 s	260.88 uj	3.43 uj	8.54 uj	50.7	0.3907	0	3.21 KB
`Utf8Json`	15	125859	7.645 s	0.1037 s	0.0970 s	261.87 uj	3.17 uj	8.61 uj	50.73	0.3893	0	3.23 KB
`STJRefGen`	15	77744	11.960 s	0.1012 s	0.0947 s	411.69 uj	3.27 uj	13.49 uj	51.43	0.5145	0	4.23 KB
`STJSrcGen`	15	93424	10.702 s	0.0681 s	0.0637 s	368.24 uj	2.22 uj	12.09 uj	51.57	0.471	0	3.93 KB
`Newtonsoft`	15	70128	13.251 s	0.1193 s	0.1116 s	454.73 uj	3.77 uj	15.11 uj	50.67	2.253	0.0713	18.48 KB
UE10												
`SpanJson`	15	133561	7.598 s	0.0872 s	0.0816 s	258.89 uj	2.71 uj	8.52 uj	50.63	0.3893	0	3.21 KB
`Utf8Json`	15	127463	7.617 s	0.0952 s	0.0890 s	260.48 uj	3.11 uj	8.56 uj	50.5	0.3923	0	3.23 KB
`STJRefGen`	15	79952	12.540 s	0.1049 s	0.0981 s	430.77 uj	3.30 uj	14.07 uj	51.63	0.5128	0	4.23 KB
`STJSrcGen`	15	92576	10.775 s	0.0750 s	0.0702 s	370.06 uj	2.39 uj	12.16 uj	51.6	0.4753	0	3.93 KB
`Newtonsoft`	15	69984	13.554 s	0.1287 s	0.1204 s	465.20 uj	4.15 uj	15.46 uj	50.73	2.2577	0.0714	18.48 KB
E25												
`SpanJson`	15	87877	10.828 s	0.0741 s	0.0693 s	371.71 uj	2.34 uj	12.20 uj	50.5	0.421	0	3.46 KB
`Utf8Json`	15	71795	14.015 s	0.1110 s	0.1038 s	481.08 uj	3.44 uj	15.72 uj	50.63	0.4179	0	3.48 KB
`STJRefGen`	15	69744	14.110 s	0.1190 s	0.1113 s	485.07 uj	3.86 uj	15.83 uj	50.97	0.4875	0	4.08 KB
`STJSrcGen`	15	91936	10.860 s	0.0943 s	0.0882 s	374.74 uj	3.00 uj	12.18 uj	51.4	0.4568	0	3.77 KB
`Newtonsoft`	15	49696	20.182 s	0.1536 s	0.1436 s	694.02 uj	4.76 uj	22.94 uj	50.73	2.3946	0.0604	19.67 KB
U25												
`SpanJson`	15	113058	8.893 s	0.1181 s	0.1105 s	298.02 uj	3.62 uj	10.02 uj	50.5	0.4334	0	3.55 KB
`Utf8Json`	15	110931	8.834 s	0.1037 s	0.0970 s	296.78 uj	3.57 uj	9.95 uj	50.53	0.4327	0	3.57 KB
`STJRefGen`	15	61184	16.428 s	0.1147 s	0.1073 s	564.02 uj	3.76 uj	18.53 uj	51.27	0.6865	0	5.63 KB
`STJSrcGen`	15	67408	14.804 s	0.1336 s	0.1250 s	508.54 uj	4.26 uj	16.60 uj	51.5	0.6379	0	5.32 KB
`Newtonsoft`	15	71888	13.952 s	0.1138 s	0.1064 s	478.67 uj	3.50 uj	15.74 uj	50.7	2.2952	0.0278	18.83 KB
UE25												
`SpanJson`	15	108186	8.906 s	0.1005 s	0.0940 s	304.39 uj	3.00 uj	10.04 uj	50.2	0.4344	0	3.55 KB
`Utf8Json`	100	25404	9.245 s	0.2570 s	0.7579 s	313.88 uj	25.94 uj	10.42 uj	50.32	0.433	0	3.57 KB
`STJRefGen`	15	67216	15.042 s	0.1303 s	0.1219 s	519.47 uj	4.11 uj	17.00 uj	51.57	0.6844	0	5.63 KB
`STJSrcGen`	15	76384	13.181 s	0.1166 s	0.1090 s	453.20 uj	3.85 uj	14.86 uj	51.43	0.6415	0	5.32 KB

continued on next page

Library	Iters	Ops	Mean	Error	StdDev	Pkg (µJ/op)	StdDev (µJ/op)	DRAM (µJ/op)	Temp (°C)	Gen0	Gen1	Allocated
`Newtonsoft`	15	68640	13.502 s	0.1106 s	0.1035 s	463.70 uj	3.74 uj	15.15 uj	50.73	2.3019	0.0291	18.83 KB
E50												
`SpanJson`	15	58145	16.569 s	0.1704 s	0.1594 s	568.64 uj	5.32 uj	18.61 uj	51	0.4644	0	3.91 KB
`Utf8Json`	15	38285	26.160 s	0.0953 s	0.0891 s	897.54 uj	3.56 uj	29.38 uj	50.73	0.4702	0	3.92 KB
`STJRefGen`	15	53232	18.409 s	0.1627 s	0.1522 s	631.93 uj	5.10 uj	20.75 uj	50.8	0.5824	0	4.77 KB
`STJSrcGen`	15	64336	15.420 s	0.1250 s	0.1169 s	529.42 uj	3.91 uj	17.40 uj	50.8	0.544	0	4.47 KB
`Newtonsoft`	15	34336	29.296 s	0.2403 s	0.2248 s	1005.54 uj	7.74 uj	33.25 uj	50.6	2.5629	0.0582	21.02 KB
U50												
`SpanJson`	15	121171	8.300 s	0.1005 s	0.0940 s	284.06 uj	3.32 uj	9.30 uj	50.63	0.5034	0	4.13 KB
`Utf8Json`	15	110033	9.154 s	0.0659 s	0.0616 s	308.80 uj	3.40 uj	10.32 uj	50.33	0.4999	0	4.15 KB
`STJRefGen`	15	44256	21.366 s	0.1745 s	0.1633 s	735.59 uj	6.28 uj	24.00 uj	51.43	0.949	0	7.87 KB
`STJSrcGen`	15	54272	18.464 s	0.1671 s	0.1563 s	634.16 uj	5.33 uj	20.69 uj	51.4	0.9213	0	7.56 KB
`Newtonsoft`	15	56432	16.737 s	0.1749 s	0.1636 s	572.40 uj	5.68 uj	19.24 uj	50.43	2.3745	0.0709	19.41 KB
UE50												
`SpanJson`	15	121659	8.326 s	0.1316 s	0.1231 s	285.49 uj	4.21 uj	9.35 uj	50.47	0.5014	0	4.13 KB
`Utf8Json`	15	109706	9.182 s	0.0948 s	0.0887 s	311.52 uj	3.27 uj	10.35 uj	50.3	0.5013	0	4.15 KB
`STJRefGen`	15	47216	21.364 s	0.1958 s	0.1831 s	734.83 uj	6.35 uj	24.14 uj	51.3	0.9531	0	7.87 KB
`STJSrcGen`	15	51968	19.316 s	0.1392 s	0.1302 s	663.98 uj	4.28 uj	21.83 uj	51	0.9236	0	7.56 KB
`Newtonsoft`	15	60832	15.957 s	0.1177 s	0.1101 s	547.42 uj	3.82 uj	18.41 uj	50.6	2.3672	0.0658	19.41 KB
E100												
`SpanJson`	15	34669	28.998 s	0.1909 s	0.1786 s	995.33 uj	6.11 uj	32.57 uj	50.83	0.5769	0	4.84 KB
`Utf8Json`	15	22086	45.448 s	0.3819 s	0.3573 s	1554.92 uj	13.26 uj	51.04 uj	50.37	0.5886	0	4.86 KB
`STJRefGen`	15	34320	28.786 s	0.2230 s	0.2086 s	987.50 uj	7.15 uj	32.41 uj	50.47	0.7576	0	6.26 KB
`STJSrcGen`	15	39200	25.504 s	0.2040 s	0.1909 s	874.87 uj	6.26 uj	28.74 uj	50.47	0.7143	0	5.95 KB
`Newtonsoft`	15	28880	34.823 s	0.2509 s	0.2347 s	1194.61 uj	7.69 uj	39.56 uj	50.27	3.8781	0.2078	31.9 KB
U100												
`SpanJson`	15	126562	7.926 s	0.0447 s	0.0418 s	271.58 uj	1.48 uj	8.91 uj	50.7	0.64	0	5.26 KB
`Utf8Json`	15	108552	9.241 s	0.0466 s	0.0436 s	309.94 uj	1.52 uj	10.36 uj	50.4	0.6356	0	5.27 KB
`STJRefGen`	15	35456	28.350 s	0.2196 s	0.2054 s	975.65 uj	7.37 uj	31.84 uj	51.67	1.523	0	12.59 KB
`STJSrcGen`	15	40368	24.882 s	0.1725 s	0.1613 s	863.21 uj	7.19 uj	27.91 uj	51.77	1.4863	0	12.28 KB
`Newtonsoft`	15	58464	16.877 s	0.1549 s	0.1449 s	578.72 uj	4.94 uj	19.39 uj	50.37	2.4973	0.0513	20.52 KB
UE100												
`SpanJson`	15	116015	8.358 s	0.0789 s	0.0738 s	286.71 uj	2.38 uj	9.44 uj	50.83	0.6378	0	5.26 KB

continued on next page

Library	Iters	Ops	Mean	Error	StdDev	Pkg (µJ/op)	StdDev (µJ/op)	DRAM (µJ/op)	Temp (°C)	Gen0	Gen1	Allocated
`Utf8Json`	15	114585	8.766 s	0.0480 s	0.0449 s	297.96 uj	1.36 uj	9.84 uj	50.23	0.6371	0	5.27 KB
`STJRefGen`	15	33008	30.418 s	0.2563 s	0.2397 s	1045.50 uj	8.20 uj	34.14 uj	51.3	1.5148	0	12.59 KB
`STJSrcGen`	15	40288	24.913 s	0.1765 s	0.1651 s	857.19 uj	5.60 uj	27.94 uj	51.8	1.4893	0	12.28 KB
`Newtonsoft`	15	59008	16.744 s	0.1606 s	0.1502 s	573.72 uj	5.29 uj	18.92 uj	50.33	2.5081	0.0508	20.52 KB

Appendix B

Text Listings

B.1 Hardware Component List

```
H/W path           Device          Class          Description
=============================================================
                                   system         Komplett PC (12345)
/0                                 bus            ROG STRIX Z390-F GAMING
/0/0                               memory         64KiB BIOS
/0/49                              memory         32GiB System Memory
/0/49/0                            memory         8GiB DIMM DDR4 Synchronous 2666 MHz (0.4 ns)
/0/49/1                            memory         8GiB DIMM DDR4 Synchronous 2666 MHz (0.4 ns)
/0/49/2                            memory         8GiB DIMM DDR4 Synchronous 2666 MHz (0.4 ns)
/0/49/3                            memory         8GiB DIMM DDR4 Synchronous 2666 MHz (0.4 ns)
/0/54                              memory         512KiB L1 cache
/0/55                              memory         2MiB L2 cache
/0/56                              memory         16MiB L3 cache
/0/57                              processor      Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz
/0/100                             bridge         8th/9th Gen Core 8-core Desktop Processor Host Bridge/DRAM
      Registers [Coffee Lake S]
/0/100/1                           bridge         6th-10th Gen Core Processor PCIe Controller (x16)
/0/100/1/0         /dev/fb0        display        TU106 [GeForce RTX 2070]
/0/100/1/0.1       card1           multimedia     TU106 High Definition Audio Controller
/0/100/1/0.1/0     input10         input          HDA NVidia HDMI/DP,pcm=9
/0/100/1/0.1/1     input7          input          HDA NVidia HDMI/DP,pcm=3
/0/100/1/0.1/2     input8          input          HDA NVidia HDMI/DP,pcm=7
/0/100/1/0.1/3     input9          input          HDA NVidia HDMI/DP,pcm=8
/0/100/1/0.2                       bus            TU106 USB 3.1 Host Controller
/0/100/1/0.2/0     usb3            bus            xHCI Host Controller
/0/100/1/0.2/1     usb4            bus            xHCI Host Controller
/0/100/1/0.3                       bus            TU106 USB Type-C UCSI Controller
/0/100/14                          bus            Cannon Lake PCH USB 3.1 xHCI Host Controller
/0/100/14/0        usb1            bus            xHCI Host Controller
/0/100/14/0/6                      bus            USB2.0 Hub
/0/100/14/0/6/4                    input          AURA MOTHERBOARD
/0/100/14/1        usb2            bus            xHCI Host Controller
/0/100/14.2                        memory         RAM memory
/0/100/16                          communication  Cannon Lake PCH HECI Controller
/0/100/17          scsi1           storage        Cannon Lake PCH SATA AHCI Controller
/0/100/17/0.0.0    /dev/sda        disk           2TB ST2000DM008-2FR1
/0/100/1b                          bridge         Cannon Lake PCH PCI Express Root Port #17
/0/100/1b.4                        bridge         Cannon Lake PCH PCI Express Root Port #21
/0/100/1b.4/0      /dev/nvme0      storage        SAMSUNG MZVLB512HAJQ-00000
/0/100/1b.4/0/0    hwmon1          disk           NVMe disk
/0/100/1b.4/0/2    /dev/ng0n1      disk           NVMe disk
/0/100/1b.4/0/1    /dev/nvme0n1    disk           512GB NVMe disk
/0/100/1b.4/0/1/1  /dev/nvme0n1p1  volume         1074MiB Windows FAT volume
/0/100/1b.4/0/1/2  /dev/nvme0n1p2  volume         2GiB EXT4 volume
/0/100/1b.4/0/1/3  /dev/nvme0n1p3  volume         473GiB EFI partition
/0/100/1c                          bridge         Cannon Lake PCH PCI Express Root Port #1
/0/100/1d                          bridge         Cannon Lake PCH PCI Express Root Port #9
/0/100/1f                          bridge         Z390 Chipset LPC/eSPI Controller
/0/100/1f/0                        system         PnP device PNP0c02
/0/100/1f/1                        system         PnP device PNP0c02
/0/100/1f/2                        communication  PnP device PNP0501
/0/100/1f/3                        system         PnP device PNP0c02
/0/100/1f/4                        system         PnP device PNP0c02
/0/100/1f/5                        system         PnP device PNP0c02
/0/100/1f/6                        system         PnP device PNP0c02
/0/100/1f/7                        system         PnP device PNP0c02
/0/100/1f.3        card0           multimedia     Cannon Lake PCH cAVS
/0/100/1f.3/0      input11         input          HDA Intel PCH Rear Mic
/0/100/1f.3/1      input12         input          HDA Intel PCH Front Mic
/0/100/1f.3/2      input13         input          HDA Intel PCH Line
/0/100/1f.3/3      input14         input          HDA Intel PCH Line Out Front
/0/100/1f.3/4      input15         input          HDA Intel PCH Line Out Surround
/0/100/1f.3/5      input16         input          HDA Intel PCH Line Out CLFE
```

```
/0/100/1f.3/6      input17         input           HDA Intel PCH Front Headphone
/0/100/1f.4                        bus             Cannon Lake PCH SMBus Controller
/0/100/1f.5                        bus             Cannon Lake PCH SPI Controller
/0/100/1f.6        eno1            network         Ethernet Connection (7) I219-V
/1                                 power           To Be Filled By O.E.M.
/2                 input0          input           Sleep Button
/3                 input1          input           Power Button
/4                 input2          input           Power Button
/5                 input6          input           Eee PC WMI hotkeys
```

B.2 Jil Deserializer Stack Trace

The following stack trace was produced when executing the `Jil_Deser` Smoke Benchmark against .NET 10.

```
System.Reflection.TargetInvocationException: Exception has been thrown by
the target of an invocation.
 ---> Jil.DeserializationException: The type initializer for
      'Jil.Deserialize.InlineDeserializer`1' threw an exception.
 ---> System.TypeInitializationException: The type initializer for
      'Jil.Deserialize.InlineDeserializer`1' threw an exception.
 ---> System.Reflection.AmbiguousMatchException: Ambiguous match found for
      'System.TimeSpan System.TimeSpan FromSeconds(Int64)'.
   at System.RuntimeType.GetMethodImplCommon(String name,
        Int32 genericParameterCount, BindingFlags bindingAttr, Binder binder,
        CallingConventions callConv, Type[] types,
        ParameterModifier[] modifiers)
   at System.RuntimeType.GetMethodImpl(String name, BindingFlags bindingAttr,
        Binder binder, CallingConventions callConv, Type[] types,
        ParameterModifier[] modifiers)
   at System.Type.GetMethod(String name, BindingFlags bindingAttr)
   at Jil.Deserialize.InlineDeserializer`1..cctor()
   --- End of inner exception stack trace ---
   at Jil.Deserialize.InlineDeserializer`1.AddGlobalVariables()
   at Jil.Deserialize.InlineDeserializer`1.BuildWithNew(Type forType)
   at Jil.Deserialize.InlineDeserializer`1
        .BuildFromStringWithNewDelegate(Int32& approximateILCount)
   at Jil.Deserialize.InlineDeserializerHelper.BuildFromString[ReturnType](
        Type optionsType, DateTimeFormat dateFormat,
        SerializationNameFormat serializationNameFormat,
        Exception& exceptionDuringBuild)
   at Jil.Deserialize.TypeCache`2.LoadFromString()
   at Jil.Deserialize.TypeCache`2.GetFromString()
   at Jil.JSON.Deserialize[T](String text, Options options)
   --- End of inner exception stack trace ---
   at Jil.JSON.Deserialize[T](String text, Options options)
   at JsonBench.Benchmarks.SmokeBenchByte.Jil_Deser()
        in JsonBench/Benchmarks/SmokeBenchByte.cs:line 42
   at BenchmarkDotNet.Autogenerated.Runnable_0
        .WorkloadActionNoUnroll(Int64 invokeCount)
   at BenchmarkDotNet.Engines.Engine.Measure(Action`1 action,
        Int64 invokeCount)
        in BenchmarkDotNet.Energy/src/BenchmarkDotNet/Engines/Engine.cs:line 242
   at BenchmarkDotNet.Engines.Engine.RunIteration(IterationData data)
        in BenchmarkDotNet.Energy/src/BenchmarkDotNet/Engines/Engine.cs:line 185
   at BenchmarkDotNet.Engines.EngineFactory.Jit(Engine engine, Int32 jitIndex,
        Int32 invokeCount, Int32 unrollFactor)
        in BenchmarkDotNet.Energy/src/BenchmarkDotNet/Engines/
        EngineFactory.cs:line 91
   at BenchmarkDotNet.Engines.EngineFactory.CreateReadyToRun(
        EngineParameters engineParameters)
        in BenchmarkDotNet.Energy/src/BenchmarkDotNet/Engines/
        EngineFactory.cs:line 45
   at BenchmarkDotNet.Autogenerated.Runnable_0.Run(IHost host,
        String benchmarkName)
   at System.Reflection.MethodBaseInvoker.InterpretedInvoke_Method(
        Object obj, IntPtr* args)
   at System.Reflection.MethodBaseInvoker.InvokeDirectByRefWithFewArgs(
        Object obj, Span`1 copyOfArgs, BindingFlags invokeAttr)
   --- End of inner exception stack trace ---
   at System.Reflection.MethodBaseInvoker.InvokeDirectByRefWithFewArgs(
        Object obj, Span`1 copyOfArgs, BindingFlags invokeAttr)
   at System.Reflection.MethodBaseInvoker.InvokeWithFewArgs(Object obj,
        BindingFlags invokeAttr, Binder binder, Object[] parameters,
        CultureInfo culture)
   at System.Reflection.RuntimeMethodInfo.Invoke(Object obj,
        BindingFlags invokeAttr, Binder binder, Object[] parameters,
        CultureInfo culture)
   at BenchmarkDotNet.Autogenerated.UniqueProgramName
        .AfterAssemblyLoadingAttached(String[] args)
```

Appendix C

Source code listings

This chapter lists the source code files referenced in the thesis that are too large to be included in the main text. The source code files can be inspected on their public GitHub repositories.

C.1 Repositories details

- BenchmarkDotNet.Energy [40].
  - GitHub HEAD: https://github.com/geovoda/BenchmarkDotNet.Energy/tree/archive/thesis-submission
  - Github diff: https://github.com/geovoda/BenchmarkDotNet.Energy/compare/master...archive/thesis-submission#files_bucket
- JSON Benchmarks [42]
  - GitHub HEAD: https://github.com/toms-vanders/json-energy-bench

C.2 Listings

```
override public List<(string, double)> openRAPLFiles()
{
    string path = "/sys/class/thermal/";
    int thermalId = 0;
    while (Directory.Exists(path + "/thermal_zone" + thermalId))
    {
        string dirname = path + "/thermal_zone" + thermalId;
        string type = File.ReadAllText(dirname + "/type").Trim();
        if (type.Contains("pkg_temp"))
            return new List<(string, double)>() {(dirname + "/temp",
                double.NaN)};
        thermalId++;
    }
    return new List<(string, double)>();
}
```

Listing C.1: CSharpRapl [55]: temperature sensor implementation.

```
private void AnalyzeMetrionDatabase(DiagnoserActionParameters parameters)
{
    ... // Safeguards
    if (config.MeasurePerIteration)
    {
        RunMetrionAnalyzeProcess(logger, latestMetrionDbFile, energyInterval);
        AnalyzeMetrionRawMeasurementsPerIteration(parameters, energyInterval);
    }
    else
    {
        // Run twice: first with exact timestamps to get Metrion's own summary energy
            (excludes
        // overlapping measurement intervals), then with extended timestamps to export
            raw measurements
        // for our own overlap-aware calculation. The two values are shown
            side-by-side in DisplayResults
        // so the user can sanity-check for large discrepancies.
        RunMetrionAnalyzeProcess(logger, latestMetrionDbFile, energyInterval);
        energyIntervals[parameters.BenchmarkCase].TotalEnergyFromSummary =
            ExtractMetrionSummaryEnergyMeasurement(logger, energyInterval.ProcessId);

        RunMetrionAnalyzeProcess(logger, latestMetrionDbFile, energyInterval,
            extendTimestamps: true);
        energyIntervals[parameters.BenchmarkCase].TotalEnergyFromRawMeasurements =
            ExtractMetrionRawEnergyMeasurements(parameters, energyInterval);
    }
    ... // Metrio Files storing / deletion
}
```

Listing C.2: Metrion Profiler: database analysis.

```
    public class MetrionEnergyProfilerConfig
{
    /// <param name="metrionBinaryPath">The path where Metrion binary is
        located.</param>
    /// <param name="metrionDatabaseDirectory">The path where Metrion exports the
        measurements' database.</param>
    /// <param name="metrionDatabaseNamePattern">The pattern of the Metrion database
        file name.</param>
    /// <param name="keepMetrionFiles">When set to true, the files generated by
        Metrion will be stored in the Artifacts folder. When set to false, the Metrion
        files will be deleted after energy extraction. False by default.</param>
    /// <param name="performExtraBenchmarksRun">When set to true, benchmarks will be
        executed one more time with the profiler attached. If set to false, there will
        be no extra run but the results will contain overhead. False by
        default.</param>
    /// <param name="timeoutInSeconds">How long should we wait for the Metrion script
        to finish exporting the measurements. 300s by default.</param>
    /// <param name="measurePerIteration">
    /// When set to true, measurements are processed separately for each iteration,
        which may introduce additional overhead.
    /// When set to false, only a single measurement of the Actual Stage is performed,
        and the energy per iteration is approximated accordingly.
    /// </param>
    /// <param name="affinityMask">A bitmask representing the processors that Metrion
        can run on. If omitted, Metrion may run on any processor.</param>
    public MetrionEnergyProfilerConfig(string metrionBinaryPath, string
        metrionDatabaseDirectory, string metrionDatabaseNamePattern="monitor*.db",
        bool keepMetrionFiles = false, bool performExtraBenchmarksRun = false, int
        timeoutInSeconds = 300, bool measurePerIteration = false, IntPtr? affinityMask
        = null )
    {
        MetrionBinaryPath = new FileInfo(metrionBinaryPath);
        MetrionDatabaseDirectory = new DirectoryInfo(metrionDatabaseDirectory);
        MetrionDatabaseNamePattern = metrionDatabaseNamePattern;
        KeepMetrionFiles = keepMetrionFiles;
        RunMode = performExtraBenchmarksRun ? RunMode.ExtraRun : RunMode.NoOverhead;
        Timeout = TimeSpan.FromSeconds(timeoutInSeconds);
        MeasurePerIteration = measurePerIteration;
        AffinityMask = affinityMask;
    }
    public TimeSpan Timeout { get; }
    public RunMode RunMode { get; }
    public FileInfo MetrionBinaryPath { get; }
    public DirectoryInfo MetrionDatabaseDirectory { get; }
    public string MetrionDatabaseNamePattern { get; }
    public bool KeepMetrionFiles { get; }
    public bool MeasurePerIteration { get; }
    public IntPtr? AffinityMask { get; }
}
```

Listing C.3: Metrion Profiler: configuration.

```
private void AnalyzeMetrionRawMeasurementsPerIteration(DiagnoserActionParameters
    parameters, EnergyInterval energyInterval)
{
    var logger = parameters.Config.GetCompositeLogger();

    if (energyInterval.IterationTimestamps.Count % 2 != 0)
    {
        logger.WriteLineError($"{nameof(MetrionEnergyProfiler)}: The number of
            iteration timestamps is odd ({energyInterval.IterationTimestamps.Count}),
            unable to calculate energy per iteration.");
        return;
    }

    var rawMeasurements = ReadRawMetrionCpuMeasurements(parameters);
    if (rawMeasurements.Length == 0)
        return;

    energyInterval.IterationTimestamps.Sort();
    logger.WriteLineInfo($"{nameof(MetrionEnergyProfiler)}: Start processing the
        energy for each iteration.");

    for (int i = 0; i < energyInterval.IterationTimestamps.Count / 2; i++)
    {
        var iterStart = energyInterval.IterationTimestamps[i * 2];
        var iterEnd = energyInterval.IterationTimestamps[i * 2 + 1];

        double iterationEnergy = rawMeasurements
            .Sum(m => CalculateOverlappingEnergy(m.Item1, m.Item2, m.Item3, iterStart,
                iterEnd));

        energyInterval.EnergyPerIteration.Add(iterationEnergy);
    }
}
```

Listing C.4: Metrion Profiler: per iteration energy calculation. in Per iteration mode.

```
private double ExtractMetrionRawEnergyMeasurements(DiagnoserActionParameters
    parameters, EnergyInterval energyInterval)
{
    var rawMeasurements = ReadRawMetrionCpuMeasurements(parameters);

    if (rawMeasurements.Length == 0)
        return 0;

    return rawMeasurements.Sum(m => CalculateOverlappingEnergy(m.Item1, m.Item2,
        m.Item3, energyInterval.StartTimestamp, energyInterval.EndTimestamp));
}
```

Listing C.5: Metrion Profiler: Extracting energy from Metrion's raw measurements calculating the total energy. Used in Per stage mode.

```
private (DateTime, DateTime, double)[] ReadRawMetrionCpuMeasurements(DiagnoserActionParameters parameters)
{
    var logger = parameters.Config.GetCompositeLogger();
    DirectoryInfo measurementsDirectory = new DirectoryInfo(Path.Combine(config.MetrionDatabaseDirectory.FullName,
          "metrion/energy_attribution/output/"));

    if (!measurementsDirectory.Exists)
    {
        logger.WriteLineError($"{nameof(MetrionEnergyProfiler)}: Unable to find measurements directory:
              {measurementsDirectory.FullName}");
        return [];
    }

    var latestMetrionOutputFile = GetLatestFileMatchingPattern(measurementsDirectory, "raw_cpu*.csv");

    if (latestMetrionOutputFile == null)
    {
        logger.WriteLineError($"{nameof(MetrionEnergyProfiler)}: The measurements were not found in the directory
              {measurementsDirectory.FullName}");
        return [];
    }

    // Read the content of latestMetrionOutputFile
    var csvFile = File.ReadAllLines(latestMetrionOutputFile.FullName);

    if (csvFile.Length < 2)
    {
        logger.WriteLineError($"{nameof(MetrionEnergyProfiler)}: The measurements file {latestMetrionOutputFile.FullName} does
              not contain enough data.");
        return [];
    }

    var headers = csvFile[0].Split(',');

    int INTERVAL_START_INDEX = Array.FindIndex(headers, h => h == "interval_start");
    int INTERVAL_END_INDEX = Array.FindIndex(headers, h => h == "interval_end");
    int TOTAL_ENERGY_INDEX = Array.FindIndex(headers, h => h == "total_energy_j");

    int MAX_INDEX = Math.Max(Math.Max(INTERVAL_START_INDEX, INTERVAL_END_INDEX), TOTAL_ENERGY_INDEX);

    (DateTime, DateTime, double)[] measurements = new (DateTime, DateTime, double)[csvFile.Length - 1];

    for (int i = 1; i < csvFile.Length; i++) // Skip header
    {
        var line = csvFile[i];
        var parts = line.Split(',');

        if (parts.Length <= MAX_INDEX)
        {
            logger.WriteLineError($"{nameof(MetrionEnergyProfiler)}: The line '{line}' does not contain enough columns.");
            continue;
        }

        if (!DateTime.TryParseExact(parts[INTERVAL_START_INDEX], "yyyy-MM-dd'T'HH:mm:ss.ffffff",
                CultureInfo.InvariantCulture, DateTimeStyles.None, out var startDateTime) ||
            !DateTime.TryParseExact(parts[INTERVAL_END_INDEX], "yyyy-MM-dd'T'HH:mm:ss.ffffff",
                CultureInfo.InvariantCulture, DateTimeStyles.None, out var endDateTime))
        {
            logger.WriteLineError($"{nameof(MetrionEnergyProfiler)}: Could not parse dates in line '{line}'.");
            continue;
        }

        var totalEnergy = double.TryParse(parts[TOTAL_ENERGY_INDEX], out double energy) ? energy : double.NaN;

        measurements[i - 1] = (startDateTime, endDateTime, totalEnergy);
    }

    return measurements;
}
```

Listing C.6: Metrion Profiler: Reading the raw measurements extracted from a Metrion database.

Appendix D

Per-Workload Factorial Detail Figures

This appendix provides the per-workload factorial views referenced in §5.1.

Figures D.1 and D.2 show the rank and ratio achieved by each library on every factorial workload. The three stacked panels correspond to the Textual, Numeric, and Boolean content types.

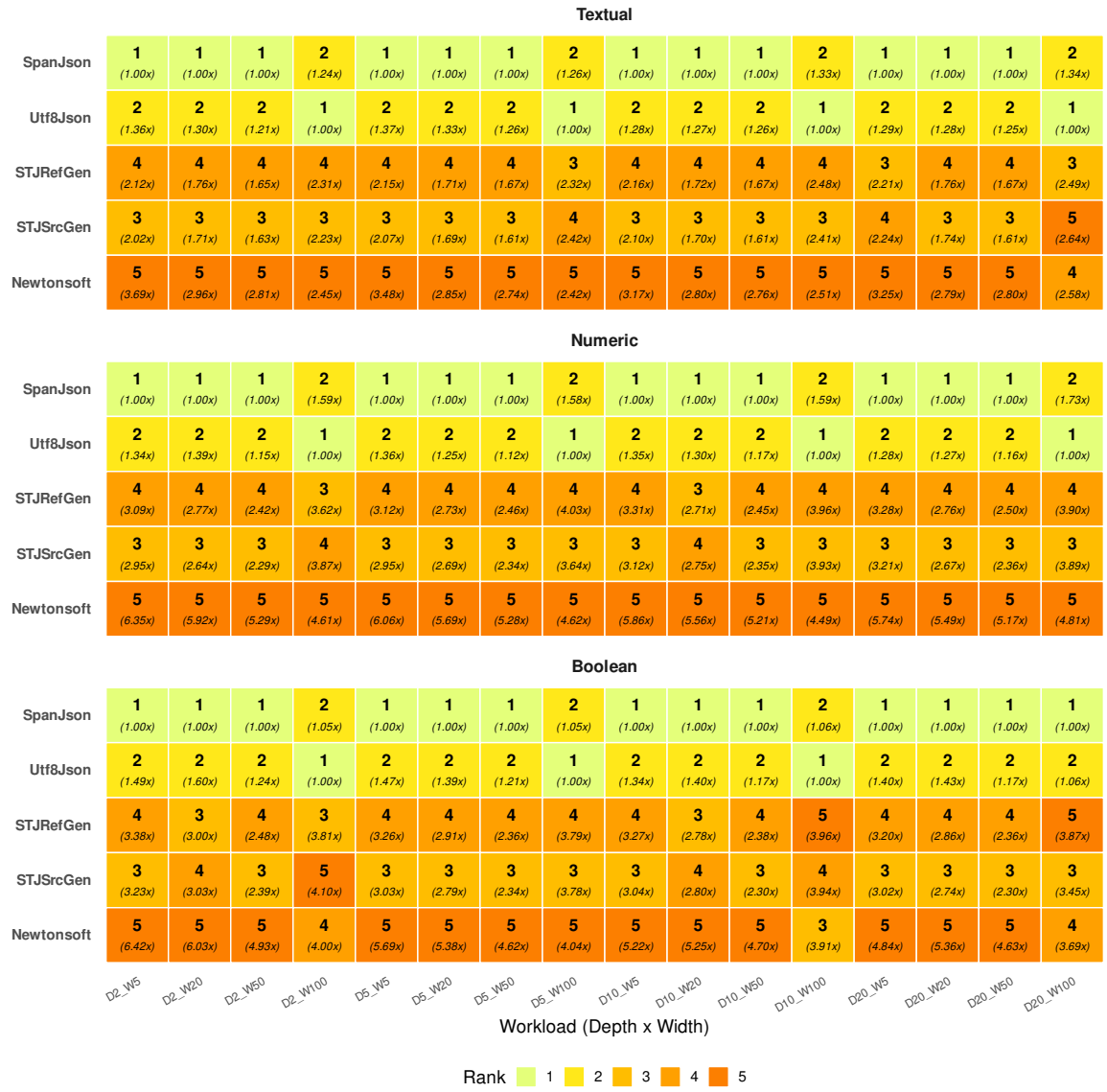


Figure D.1: Per-workload library rank and ratio-to-best, deserialisation. Rank 1 = lowest energy; Rank 5 = highest energy.

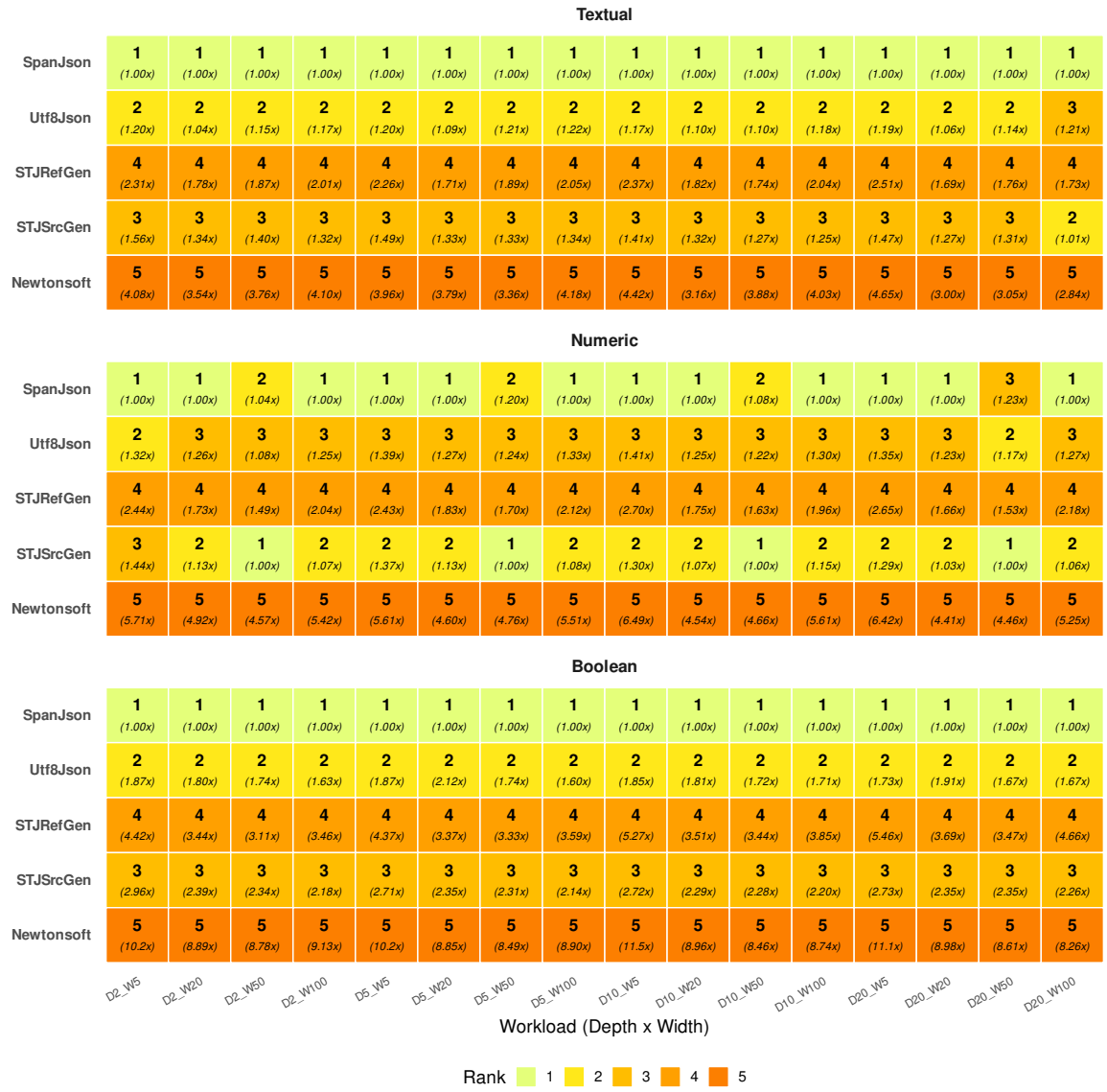


Figure D.2: Per-workload library rank and ratio-to-best, serialisation. Rank 1 = lowest energy; Rank 5 = highest energy.

Appendix E

Noisy Environment Effects Detail Figures

E.1 Depth Environment Effects

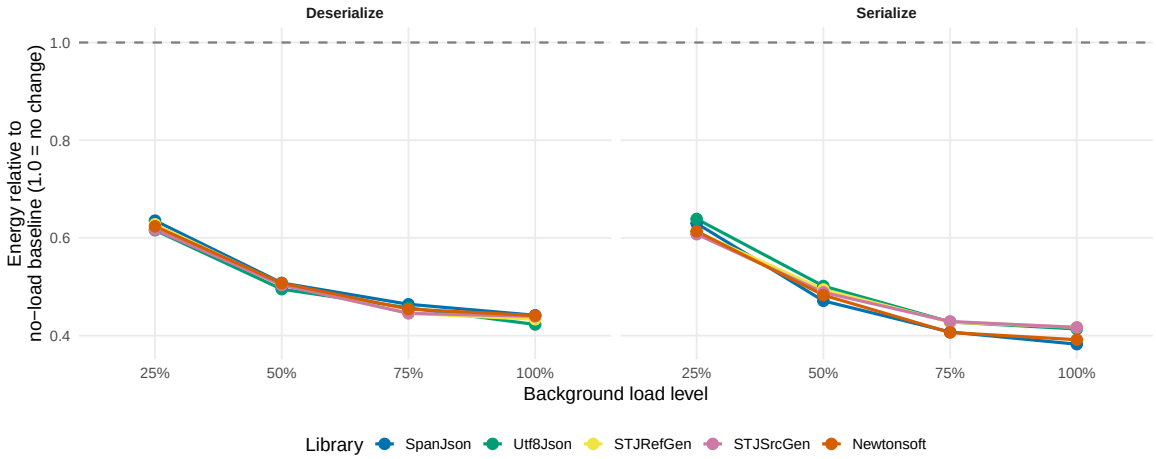


Figure E.1: Energy consumption of each library relative to its no-load (0%) Metrion baseline, averaged across all nesting depth levels. Values above 1.0 indicate overhead from background CPU noise; diverging lines reveal which libraries are most sensitive to environmental interference.

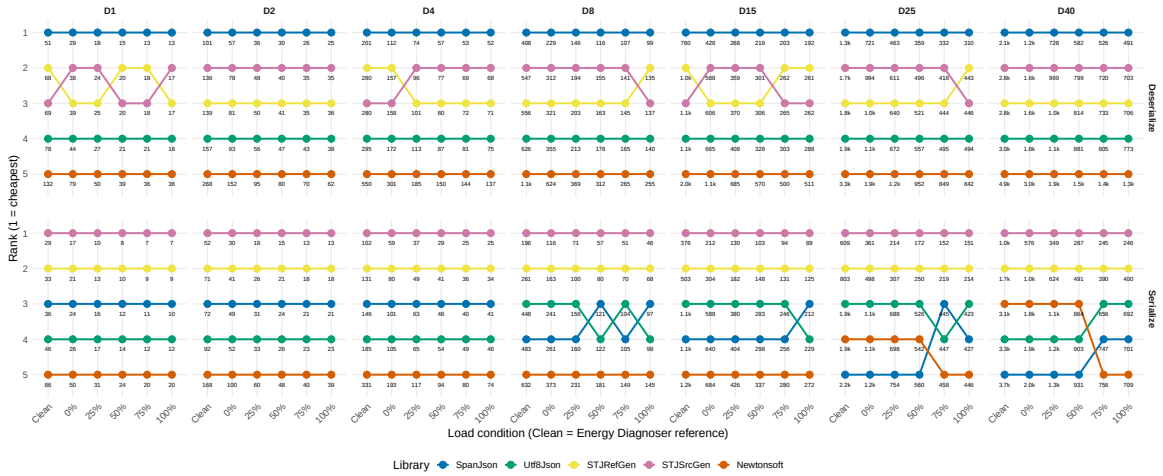


Figure E.2: Library energy rank trajectories across load conditions for each nesting depth level. Rank 1 (top) denotes the most energy-efficient library; crossing lines indicate rank inversions induced by background load.

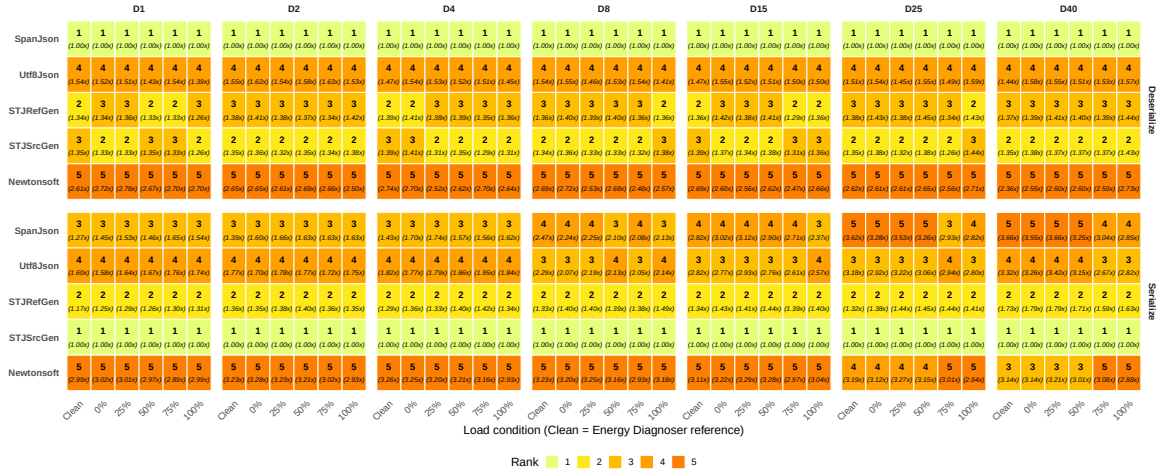


Figure E.3: Energy ranking heatmap across nesting depth levels and load conditions. Cell fill encodes rank (1 = lowest energy); bold text shows rank and italic text shows the ratio to the cheapest library within that condition and depth level.

E.2 Size Environment Effects

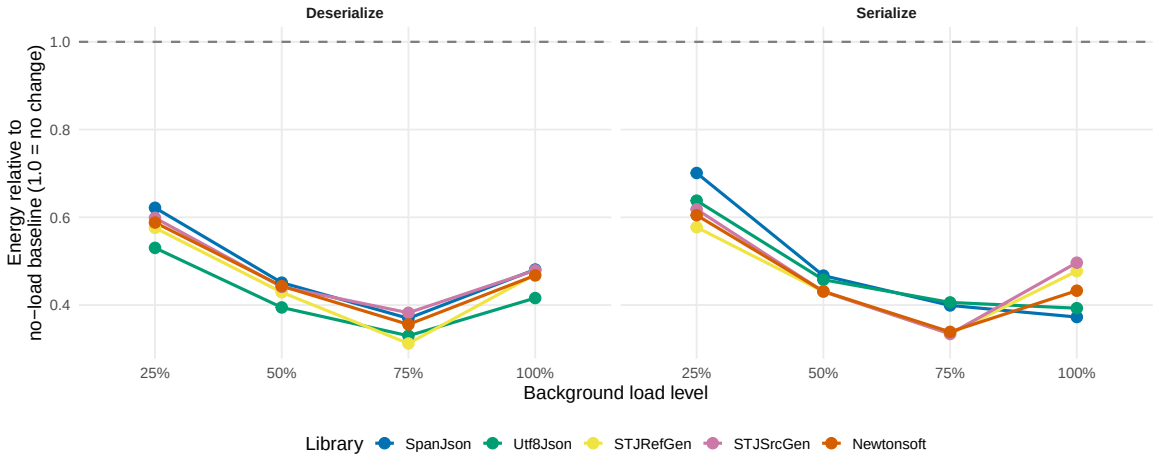


Figure E.4: Energy consumption of each library relative to its no-load (0%) Metrion baseline, averaged across all object count levels. Values above 1.0 indicate overhead from background CPU noise; diverging lines reveal which libraries amplify energy usage most under load.

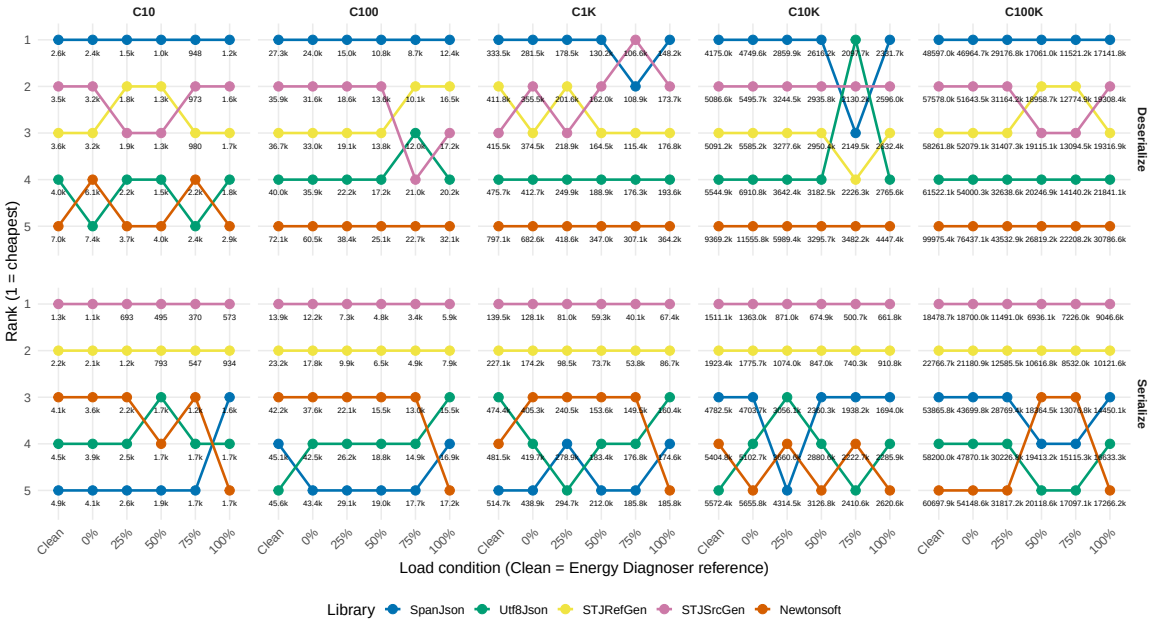


Figure E.5: Library energy rank trajectories across load conditions for each object count level. Rank 1 (top) denotes the most energy-efficient library; crossing lines indicate rank inversions induced by background load.

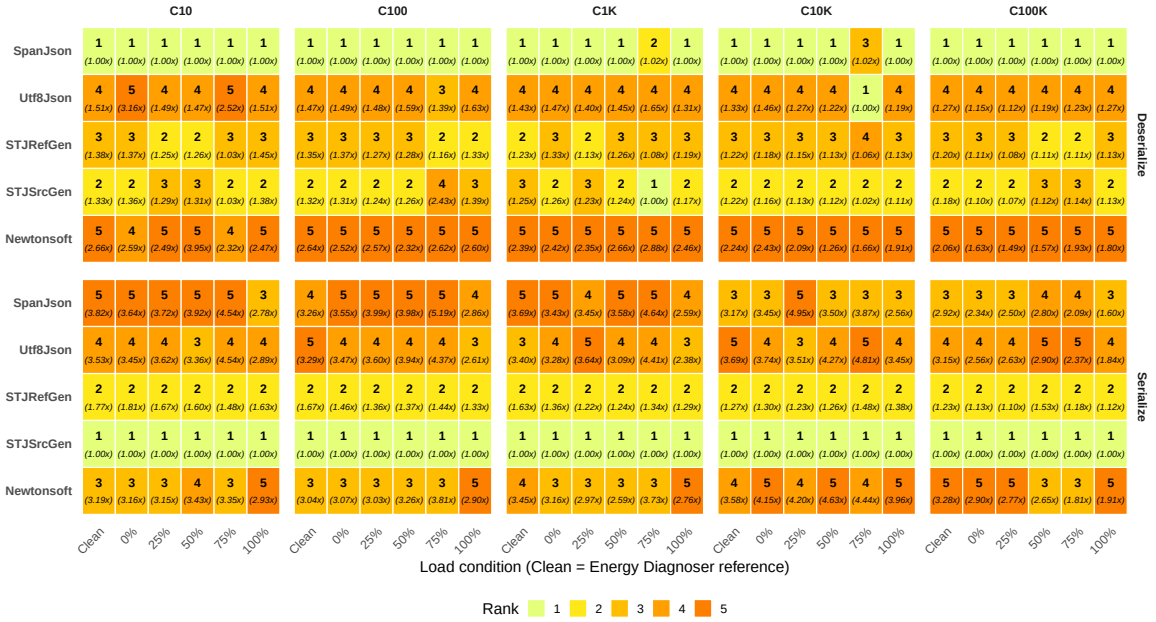


Figure E.6: Energy ranking heatmap across object count levels and load conditions. Cell fill encodes rank (1 = lowest energy); bold text shows rank and italic text shows the ratio to the cheapest library within that condition and object count level.

E.3 Width Environment Effects

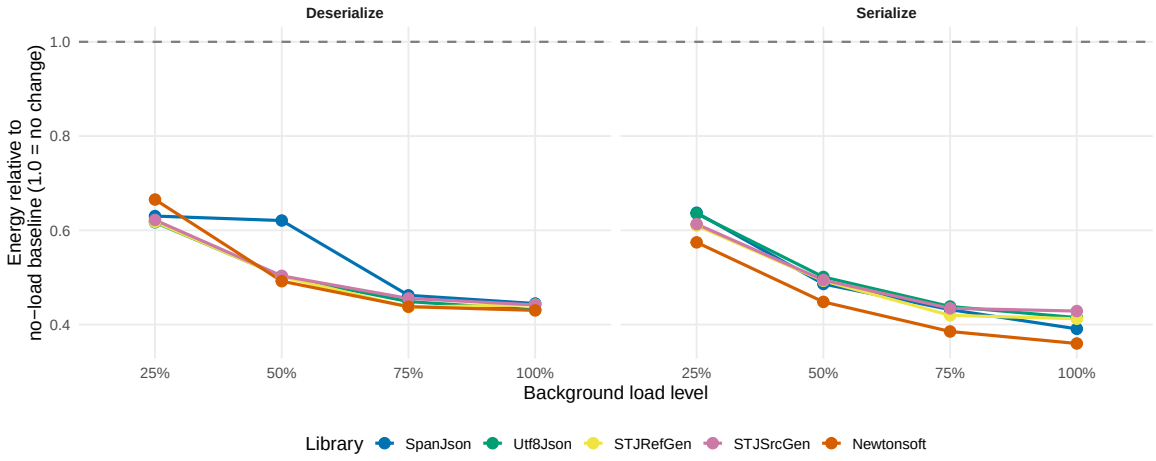


Figure E.7: Energy consumption of each library relative to its no-load (0%) Metrion baseline, averaged across all object width levels. Values above 1.0 indicate overhead introduced by background CPU noise; diverging lines between libraries reveal differential sensitivity to the noisy environment.

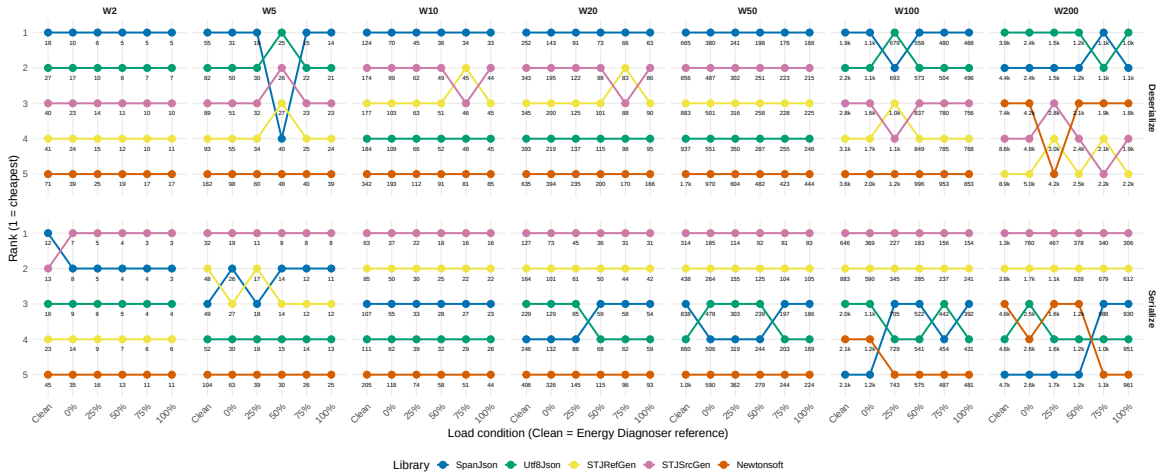


Figure E.8: Library energy rank trajectories across load conditions for each object width level. Rank 1 (top) denotes the most energy-efficient library; crossing lines indicate rank inversions induced by background load.

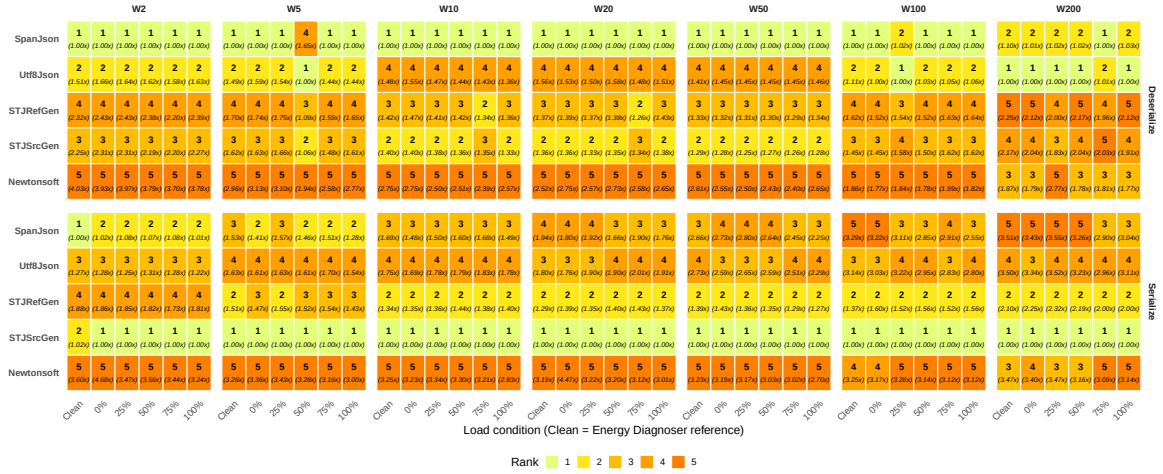


Figure E.9: Energy ranking heatmap across object width levels and load conditions. Cell fill encodes rank (1 = lowest energy); bold text shows rank and italic text shows the ratio to the cheapest library within that condition and width level.