ORIGINAL ARTICLE

Converting Binary Floating-Point Numbers to Shortest Decimal Strings: An Experimental Review

Jaël Champagne Gareau[1] | Daniel Lemire[1]

[1]Université du Québec (TELUQ), Montreal, Quebec, H2S 3L5, Canada

Correspondence
Daniel Lemire, Université du Québec (TELUQ), Montreal, Quebec, H2S 3L5, Canada
Email: daniel.lemire@teluq.ca

Funding information
Natural Sciences and Engineering Research Council of Canada, Grant Number: RGPIN-2024-03787
Fonds de recherche du Québec, `https://doi.org/10.69777/361128`

When sharing or logging numerical data, we must convert binary floating-point numbers into their decimal string representations. For example, the number π might become `3.1415927`. Engineers have perfected many algorithms for producing such accurate, short strings. We present an empirical comparison across diverse hardware architectures and datasets. Cutting-edge techniques like Schubfach and Dragonbox achieve up to a tenfold speedup over Steele and White's Dragon4, executing as few as 210 instructions per conversion compared to Dragon4's 1500–5000 instructions. Often per their specification, none of the implementations we surveyed consistently produced the shortest possible strings—some generate outputs up to 30% longer than optimal. We find that standard library implementations in languages such as C++ and Swift execute significantly more instructions than the fastest methods, with performance gaps varying across CPU architectures and compilers. We suggest some optimization targets for future research.

KEYWORDS
Floating-point numbers, Shortest-string algorithms, Performance benchmarking

1 | INTRODUCTION

Processor vendors have adopted 32-bit and 64-bit IEEE 754 floating-point numbers. Consequently, we typically represent numbers as IEEE 754 floating-point numbers in software. The corresponding types in Java, C, C# and C++

are `float` and `double`. JavaScript uses the 64-bit IEEE 754 floating-point format as its default number data type. These numbers take the form of a fixed-precision integer (the significand[1]) multiplied by a power of two ($m \times 2^p$). For example, a 32-bit floating-point value approximating the constant π is 13176795×2^{-22}. We often convert these binary values into decimal strings:

- when we serialize data to text formats like CSV, JSON or YAML;
- when we produce human-readable logs and telemetry;
- when we print numbers in graphical interfaces, spreadsheets and dashboards.

Producing the shortest possible string that exactly reproduces the original value can require hundreds or even thousands of CPU instructions. Since many applications convert millions or even billions of values in bulk, that per-value overhead may quickly add up to a substantial performance bottleneck.

Converting binary floating-point values into decimal strings is largely a matter of established software practices, yet it remains under-explored in the research literature. In Section 2, we formally define the shortest-string conversion problem. In Section 3, we survey the principal algorithmic families. Finally, we present an experimental comparison of key implementations in Section 4—ranging from Steele and White's 1990 Dragon4 to modern methods like Schubfach and Dragonbox—and present some directions for future work in Section 5.

Taken together, our study offers both a broad empirical perspective and new methodological insights. Our main contributions are the following:

- *A systematic empirical evaluation of major conversion algorithms*, spanning both dominant CPU families (x86-64 and ARM/AArch64) and an expanded, openly available benchmark suite including mostly real-world datasets.
- *New measurements of output behavior*, including the first detailed characterization of end-to-end string lengths—which often differ from minimal significand lengths—and instruction-level metrics that isolate algorithmic cost from microarchitectural throughput.
- *A reassessment of existing benchmarking practices*, identifying methodological limitations in prior evaluations.

2 | PROBLEM DEFINITION

Although IEEE 754 floating-point types are the default choice for representing real numbers in software—and are widely supported by commodity processors and programming languages—their use can lead to non-obvious difficulties. In this section, we formalize the problem and outline its practical implications. Table 1 describes the bit layouts of 32- and 64-bit floating-point numbers. The IEEE 754 formats dedicate a bit for the sign: accordingly, we can distinguish between -0 and 0. A positive *normal* double-precision floating-point number is a binary floating-point value whose significand is represented with 53 bits of precision: 52 bits are explicitly stored, while the leading 1 bit is implicit (not physically stored in memory). For example, the value with a significand of 1.011_2 would be stored as an implicit leading 1 and explicit bits 011. As such, the significand can be seen as a 53-bit integer m in the interval $[2^{52}, 2^{53})$ but interpreted as a number in $[1, 2)$ by dividing it by 2^{52}. The 11-bit exponent p ranges from -1022 to 1023 [1]. Values smaller than 2^{-1022} are called *subnormal* values: their special exponent code has the value 2^{-1022} and the significand is then interpreted as a value in $[0, 1)$. We can uniquely identify a 64-bit number using a 17-decimal-digit representation, although fewer digits are often needed. The 32-bit numbers are similarly defined, with a 24-bit significand m and an 8-bit exponent p ranging from -126 to 127. Numbers smaller than 2^{-126} are represented using a subnormal format. We have that 9 digits are sufficient to uniquely identify a 32-bit number.

[1] The term *mantissa* is sometimes used as a synonym for *significand*, but this usage is discouraged by the IEEE 754 standard, which reserves the term *significand* to denote the fractional component of a floating-point number. In contrast, *mantissa* historically referred to the fractional part of a logarithm.

TABLE 1 Common IEEE 754 binary floating-point numbers

name	exponent bits	significand (stored)	decimal digits (exact)
binary64	11 bits	53 bits (52 bits)	17
binary32	8 bits	24 bits (23 bits)	9

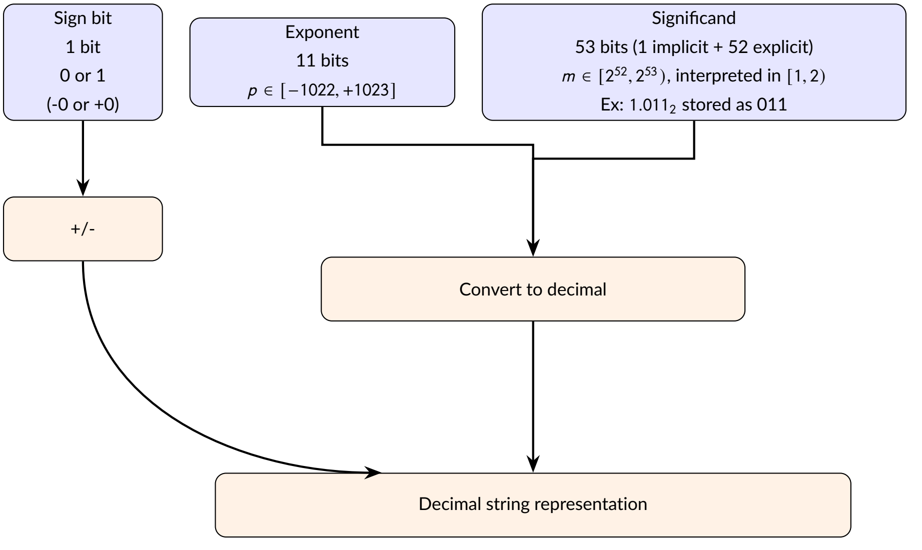

FIGURE 1 Conversion of a 64-bit number to a string

String representations are typically in decimal format. Converting a binary floating-point number to a decimal string is usually done in three steps:

1. Extracting the sign bit, exponent, and significand;
2. Converting the binary significand and exponent to their decimal counterparts;
3. Generating the string representation of the resulting decimal number.

See Fig. 1. The first step—extracting the three fields from the bit pattern—is straightforward and leaves little room for algorithmic innovation. Consequently, most research has focused on the second step: the conversion of a value from a base-2 representation ($m \times 2^p$) to a base-10 representation ($w \times 10^q$). This conversion involves determining the decimal significand w and exponent q from the binary significand m and exponent p by solving the equation $m \times 2^p = w \times 10^q$ [2].

An exact solution is always possible in this direction (from binary floating-point to decimal string). For $p \geq 0$, setting $q = 0$ and $w = m \times 2^p$ suffices, since any positive power of two is exactly representable in base ten. For $p < 0$, we can set $q = p$ and $w = m \times 5^{-q}$, making use of the identity $2^p = 10^p/5^p$.

However, the converse is not generally true: most decimal fractions cannot be represented exactly as binary floating-point numbers. For example, 0.1 cannot be expressed as $m \times 2^p$ with integer m, since $m = 2^{-p-1}/5$ would require a power of two to be divisible by 5, which is impossible.

After accounting for special cases (±0, ±∞, NaN, and subnormals) and ignoring the sign bit, the central mathematical problem becomes finding the smallest integer w such that $w \times 10^q$ maps to the original binary floating-point value and to no other representable value. In other words, the decimal representation should unambiguously represent the intended floating-point value rather than to any adjacent representable value. By finding the smallest integer w, we seek to minimize the number of *significant digits* in the decimal string. Significant digits are the digits in a number that contribute to its precision, including all non-zero digits and zeros between non-zero digits. For example, the number 1100 is generally considered to have two significant digits, namely the two 1s, as the trailing zeros are ambiguous and not counted without explicit clarification. We often use scientific notation, where a number is expressed as a coefficient multiplied by a power of 10, such as $a \times 10^b$ or in "`E`" notation as `aEb`, to clearly define the significant digits. For instance, writing 1100 as 1.1×10^3 or 1.1E3 indicates two significant digits, ensuring precision is unambiguous.

To illustrate these ideas, consider the following example using the number π. Its closest 32-bit floating-point approximation (in binary) is 13176795×2^{-22}. The corresponding exact decimal representation is:

$$31415927410125732421875 \times 10^{-22}.$$

However, it is not necessary to print the entire integer part in most applications. For 32-bit floating-point numbers, nine decimal digits are always sufficient to uniquely distinguish each value. Therefore, we can round this exact decimal to nine digits, giving 314159274×10^{-8} for our candidate decimal representation. In this case, we truncate down to nine digits: since the next digit is '1', we do not round up. However, it is actually possible to use a shorter decimal: 31415927×10^{-7}. This shorter form is still closer to 13176795×2^{-22} than to any other 32-bit float, and is therefore the shortest unique decimal representation.

After determining the decimal significand and exponent, the final step is to generate the corresponding string representation (e.g., producing `3.1415927` from w and q in $w \times 10^q$). This process is not always cleanly separable from the earlier computation: some algorithms combine string generation and significand calculation, while others treat them as distinct stages. Typically, the decimal digits of an integer value are obtained by repeatedly extracting the least significant digit ($w \bmod 10$) and dividing by ten, with various optimizations possible. For example, a common optimization is to proceed by pairs of digits ($w \bmod 100$) and use a table to map the value in $[0, 100)$ to a pair of characters (e.g., the value 10 maps to the string '`10`'). However, even if the smallest significand has been found in the previous step, further care is needed if we want to ensure that the output string is as short as possible. For instance, both `9.9E1` and `99` represent the same value with two significant digits, but the latter is shorter and arguably preferred; scientific notation should only be used when it produces a shorter overall string.

It is crucial to distinguish between two closely related but conceptually different objectives in float-to-string conversion:

- (1) *Minimal decimal significand.* Most algorithms aim to compute the shortest decimal significand w such that $w \times 10^q$ round-trips to the original binary floating-point value. This solves step 2 of the conversion pipeline.
- (2) *Minimal printed string.* We might instead wish to find the shortest *character string* that round-trips, including the choice between fixed-point and scientific notation and the placement of the decimal point.

These two goals coincide for many values but are not equivalent. Two representations with the same number of significant digits may differ in total length depending on formatting choices. For example, the binary64 value 12000000000 has a shortest significand "12", but the shortest overall string is "12e9" (4 characters), whereas the canonical scientific form "1.2e10" contains six characters. Consequently, existing algorithms typically optimize (1)—the computation of the minimal significand—but not necessarily (2).

Representation subtleties can significantly affect float-to-string conversions. Consider the integer 2 150 000 128. We can represent it exactly as a 32-bit floating-point number (8398438×2^8). The previous 32-bit floating-point number is 2 149 999 872 or 8398437 × 2^8. The number 2 150 000 000 falls between these two 32-bit floating-point values. When a decimal value lies exactly halfway between two floats, as here, IEEE 754 requires rounding to the value whose least significant bit is zero (round-to-even).[2] Hence, the string `2150000000` is parsed as the 32-bit float value `2150000128`. Thus, when converting the 32-bit floating-point number `2150000128` to a string, we should write `2.15e9` to minimize the number of digits in the significand and the string length. Such cases underscore the need for careful handling of rounding and highlight edge cases that conversion algorithms must address.

3 | RELATED WORK

This section reviews the main algorithmic families for shortest float-to-string conversion, then pinpoints why existing benchmarks leave important questions unanswered.

3.1 | Existing Algorithms

The conversion of binary floating-point numbers to their decimal string representation has been a subject of research over the years. However, there are relatively few scholarly contributions.

A crucial but long-overlooked contribution was made by Coonen in his 1980 technical report [3] and extended in Chapter 7 of his 1984 PhD Thesis [4]. Coonen was the first to articulate the modern view of the correct-rounding, formulating the requirement that a printed decimal must round back to the original binary floating-point value and introducing the interval-based reasoning later reused in algorithms such as Grisu and Ryū. Although this work circulated primarily as a technical report and was not widely cited in early literature, its importance has since been recognized—most prominently by Loitsch, who identifies Coonen's work as the earliest description of a correct floating-point–to–decimal algorithm.

In 1990, Steele and White [5] authored the foundational article that brought these ideas into the programming-language community and established the Dragon family of algorithms. They report that their work was originally conducted in the late 1970s and circulated informally for many years before publication. Their Dragon4 algorithm can thus be viewed as a practical successor to the conceptual framework introduced by Coonen. The publication of the IEEE 754 standard in 1985 [6] occurred after many implementations were already in use, and Steele and White's paper was the first widely disseminated description of a fully practical correct-rounding and shortest-decimal algorithm.

In their paper, Steele and White introduce core objectives for the conversion of floating-point values to decimal representation. They distinguish between fixed-format outputs—where the number of digits is predetermined (e.g., always printing five digits)—and free-format outputs, where the algorithm determines the number of digits dynamically.

For the free-format case, they propose three essential properties that a correct conversion algorithm from binary floating-point values to decimal should satisfy:

1. *No loss of information*: Converting back the generated decimal representation must recover the original floating-point value (i.e., round-trip conversion is the identity function).
2. *No extra information*: The generated decimal representation must not contain extraneous digits (i.e., we want the minimal significand possible).

---

[2] Round-to-even, also known as banker's rounding, is a method for rounding decimal numbers to reduce bias in calculations, often used in financial or statistical contexts. When a number is exactly halfway between two integers (e.g., 2.5 or 3.5), instead of always rounding up (as in standard rounding), round-to-even rounds to the nearest even integer. This method evens-out the frequency of rounding up and down over many operations.

3. *Correct rounding*: Multiple values may satisfy the first two properties. Among these, the algorithm should produce the one closest to the original floating-point value, using round-to-even in the case of a tie.

To address these objectives, Steele and White first present a practical, though approximate, algorithm for generating free-format output: Dragon2 (see Fig. 2 for a Python sketch). The algorithm first rescales the floating-point value so that it falls within the interval $[0, 1)$, and then uses floating-point arithmetic to compute the digits. As the authors remark, the algorithm is not accurate due to its reliance on floating-point arithmetic. Thus, some values may not be recovered exactly from the output string. Next, they present an algorithm which they named Dragon4: they omit Dragon3.[3] Several other algorithms in this domain are named after dragons. Dragon4 is based entirely on integer arithmetic, and it can therefore be exact (see Fig. 3 for a sketch in Python). Instead of scaling the floating-point integer, it scales two integers R and S representing the floating-point value. The fraction R/S represents the value which is iteratively scaled to be in a safe subinterval of $[0, 1)$. The downside of Dragon4 is that it may require big-integer arithmetic and several division operations.

```python
from math import floor

def fp3(f, n, B = 10, b = 2):
    assert 0 <= f < 1
    F, k, R, M = [], 0, f, (b ** -n) / 2
    while True:
        k += 1
        U = floor(R * B)
        R = R * B - U
        M *= B
        if R < M or R > 1 - M:
            break
        F.append(U)
    F.append(U + (R > 0.5 or (R == 0.5 and U % 2)) * 1)
    return F

def dragon2(f, n = 24, B = 10):
    if f == 0:
        return "0.0"
    x = 0
    while f < 1: x, f = x - 1, f * B
    while f > B: x, f = x + 1, f / B
    return (f"{floor(f)}.{''.join(map(str, fp3(f - floor(f), n, B)))}"
            f"{f'E{x}' if x else ''}")
```

FIGURE 2 Python sketch of the Dragon2 [5] algorithm for non-negative numbers. The algorithm supports conversions between arbitrary bases, not just from base-2 to base-10. Parameters `b` and `B` denote the input and output radix, respectively. The parameter `n` indicates the input precision, in number of significand binary digits (e.g., the default `n = 24` corresponds to binary32 precision; see Table 1).

The algorithm produces the digits one by one, stopping when the following digits would be all zeros. For producing fixed-format outputs, we can generate the number of digits and then round the final digit if needed. If fewer digits are produced by the free-format approach than we desire, we pad with zeros. The rounding up may then require changing already produced digits (e.g., if the digit 9 is rounded up). Steele and White observe that this can be avoided by an algorithm that already produces the correct digits, knowing how many are needed.

A significant amount of computation in Dragon4's algorithm is spent scaling the values: the computation of the

[3] In a retrospective article [7], Steele and White explain that the reference to a Dragon has to do with *dragon curves*, a mathematical curiosity. This curve is constructed by combining two types of steps: *Folds* and *Peaks*, whose initials ("FP") allude to floating-point numbers.

constant k in Fig. 3 requires many operations. For example, a floating-point value such as `1e-300` requires over 299 iterations, each of which entails three multiplications.

```
def fpp2(f, e, p, B = 10):
    assert 0 < f < 2 ** p
    ep = e - p
    R, S = f << max(ep, 0), 1 << max(-ep, 0)
    Mminus = Mplus = 1 << max(ep, 0)
    if f == 1 << (p - 1):
        Mplus, R, S = Mplus << 1, R << 1, S << 1
    k = 0
    while R < (S + B - 1) // B:
        k, R, Mplus, Mminus = k - 1, R * B, Mplus * B, Mminus * B
    while 2 * R + Mplus >= 2 * S:
        k, S = k + 1, S * B
    D, even = {}, True
    while True:
        k -= 1
        U, R = divmod(R * B, S)
        Mminus, Mplus = Mminus * B, Mplus * B
        low = 2 * R < Mminus
        high = 2 * R > 2 * S - Mplus
        if low or high:
            round_up = (high and not low) or (low and high and (2 * R > S or (2 * R == S and not even)))
            D[k] = U + int(round_up)
            break
        D[k] = U
        even = U % 2 == 0
    return D

def dragon4(f, B = 10):
    if f == 0: return "0.0"
    import struct
    int_val = struct.unpack('>Q', struct.pack('>d', f))[0]
    mantissa = (int_val & 0xFFFFFFFFFFFFF) | (1 << 52)
    exponent = ((int_val >> 52) & 0x7FF) - 1023 - 52
    return fpp2(mantissa, exponent + 54, 54, B)
```

FIGURE 3 Python sketch of the Dragon4 [5] algorithm for non-negative numbers.

Following Steele and White, Gay [8] presented methods for accurately converting between binary floating-point numbers and decimal strings, with a specific focus on ensuring correct rounding. He describes the `dtoa` function, which implements various modes such as free-format (shortest-string) and fixed-format output. A key insight is that instead of using the relatively slow Steele-White algorithm to compute the parameter k (see Fig. 3), we can use a faster floating-point approach, correcting it if needed. Further, Gay observes that when the number of significant digits is sufficiently small, we may use a faster algorithm based on floating-point arithmetic, producing an exact result with fewer computations. We may also check whether the floating-point value is an integer, in which case a conversion to an integer value and its printing might be faster. Gay's `dtoa` function is several times faster than Dragon4. Fig. 4 illustrates `dtoa` using Python code. For simplicity, it omits the path where the `dtoa` function uses floating-point arithmetic. Burger and Dybvig [9] contributed techniques similar to Gay's by developing a scale estimator.

Loitsch [10] explored an integer-based approach to achieve quick and accurate conversions. Their approach uses a combination of precomputed powers of ten and a `diy_fp` (do-it-yourself floating-point) representation (a 64-bit integer coupled with an integer for the exponent) to approximate the decimal output quickly. The precomputed powers of ten must hold roughly 635 precomputed values for 64-bit floating-point numbers. Starting from a suboptimal Grisu

```python
def gay(f, e, p, kp, B = 10):
    assert 0 < f < 2**p
    ep = e - p
    R, S = f << max(ep, 0), 1 << max(-ep, 0)
    Mminus = Mplus = 1 << max(ep, 0)
    if f == 1 << (p-1):
        Mplus, R, S = Mplus << 1, R << 1, S << 1
    k = kp
    if k < 0:
        R, Mplus, Mminus = R * B ** -k, Mplus * B ** -k, Mminus * B ** -k
    elif k > 0:
        S, Mplus = S * B ** k, Mplus * B ** k
    if 2 * R + Mplus >= 2 * S:
        k, S = k + 1, S * B
    D, even = {}, True
    while True:
        k -= 1
        U, R = divmod(R * B, S)
        Mminus, Mplus = Mminus * B, Mplus * B
        low = 2 * R < Mminus
        high = 2 * R > 2 * S - Mplus
        if low or high:
            round_up = (high and not low) or (low and high and (2 * R > S or (2 * R == S and not even)))
            D[k] = U + int(round_up)
            break
        D[k] = U
        even = U % 2 == 0
    return D

def dtoa(f, B = 10):
    if f == 0: return "0.0"
    import struct, math
    int_val = struct.unpack('>Q', struct.pack('>d', f))[0]
    mantissa = (int_val & 0xFFFFFFFFFFFFF) | (1 << 52)
    exponent = ((int_val >> 52) & 0x7FF) - 1023 - 52
    if exponent >= -52 and exponent <= 0 and (mantissa >> -exponent << -exponent == mantissa):
        return str(mantissa >> -exponent) # exact integer
    x, l = f * 2**(-exponent-52), exponent + 52
    kp = int(math.floor(0.301029995663981 * l + (x - 1.5) * 0.289529654602168 + 0.1760912590558))
    return gay(mantissa, exponent + 54, 54, kp, B)
```

FIGURE 4 Python sketch of the Gay's `dtoa` function for non-negative numbers.

algorithm, they have developed an algorithm called Grisu2 which is often, but not always, able to produce the shortest representation. They have also developed another algorithm called Grisu3 which detects when its output may not be the shortest. Loitsch proposes falling back on another printing algorithm like Dragon4 when Grisu3 fails.

Andrysco, Jhala, and Lerner [11] introduced the Errol algorithm for printing floating-point numbers, aiming for both correctness and speed. Initially, they claimed it was significantly faster than Grisu3; however, they later acknowledged in their repository that this evaluation was flawed: *"Our original evaluation of Errol against the prior work of Grisu3 was erroneous… Corrected performance measurements show a 2x speed loss to Grisu3."* We tested their implementation,[4] but found it contained unsafe code, so we do not consider Errol further.

Adams [12, 13] introduced the Ryū algorithm, which, like Grisu3, targets fast and correct printing of floating-point numbers, but with the crucial advantage of always guaranteeing correct output, thus eliminating the need for a fallback algorithm. The correctness of Ryū relies on several key elements described in the 2018 paper [12]. First, Ryū decodes

[4] `https://github.com/marcandrysco/Errol`

a floating-point number into a unified representation that handles both normalized and subnormal cases, written as $f = (-1)^s \cdot m_f \cdot 2^{e_f}$, where m_f is an unsigned integer. The algorithm then calculates the interval of decimal values that would be decoded back to the original binary float, by determining the midpoints to the immediately smaller and larger representable floating-point numbers. These midpoints define an interval $[u, w)$, scaled by 2^{e_2}, within which any value will round to the original float. This interval is then converted into decimal base, yielding an interval $[a, c) \cdot 10^{e_{10}}$. The core digit-generation step identifies the shortest decimal representation within this interval by iteratively removing digits from right to left. At each step, it ensures that the shortened decimal string still lies within the valid interval, thus guaranteeing that parsing this string returns the original float. To handle rounding modes correctly, Ryū analyzes the prime factorization of the significand to decide whether the result should be rounded up or down. This process is carried out using only fixed-precision integer operations, which avoids costly high-precision multiplications. A major contribution of Ryū is a reduction in the number of bits required for intermediate computations. Instead of converting the entire binary value to decimal in one high-precision operation, Ryū combines incremental decimal conversion with its digit-generation loop. This allows the use of smaller intermediate values, aided by precomputed lookup tables for multipliers such as $\lfloor 2^k/5^q \rfloor + 1$ or $\lfloor 5^{-e_2-q}/2^k \rfloor$, which enable efficient and accurate calculation. In a follow-up paper [13], Adams extends Ryū to create Ryū Printf, supporting the `%f`, `%e`, and `%g` formats with runtime-configurable precision. To maintain efficiency, Ryū Printf introduces a segmentation approach, converting the significand into segments of decimal digits (e.g., 9 digits per segment for 32-bit integers). Each segment is computed independently, so computational cost is linear in the number of digits generated, rather than superlinear as in naive approaches.

TABLE 2 Overview of binary floating-point to string algorithms. A technique is considered exact if the obtained string representation is always sufficient to recover the original binary floating-point number.

Technique	Source	Publication Date	Exact
Dragon2	Steele and White [5]	1990	No
Dragon4	Steele and White [5]	1990	Yes
`dtoa`	Gay [8]	1990	Yes
Grisu	Loitsch [10]	2010	No
Grisu2	Loitsch [10]	2010	No
Grisu3 + Dragon4	Loitsch [10]	2010	Yes
Ryū	Adams [12, 13]	2018	Yes
Schubfach	Giulietti [14]	2020 (informal)	Yes
Grisu-Exact	Jeon [15]	2020 (informal)	Yes
Dragonbox	Jeon [16]	2022 (informal)	Yes

Other algorithms not formally published in the peer-reviewed literature include Giulietti's Schubfach [14], which decomposes a floating-point value into its significand and exponent, computes tight decimal bounds, and dispatches into specialized computation paths based on the number's properties; Jeon's Dragonbox [16], inspired by the Ryū algorithm; and Jeon's earlier Grisu-Exact [15], a Grisu-family variant that guarantees shortest, correctly rounded outputs. See Table 2 for a summary of each algorithm's characteristics. For each entry, we list the original source and indicate whether it produces exact free-format strings (i.e., the strings that permit error-free round-trip parsing). The *Grisu3 + Dragon4* technique denotes Grisu3 with a Dragon4 fallback.

3.2 | Limitations of Prior Benchmarks

Several open-source benchmarks have been published to compare the performance of algorithms converting floating-point numbers to decimal strings. Notable examples include Yip's `dtoa-benchmark`[5], Lugowski's `parse-bench`[6], and Bolz's `Drachennest`[7]. Many algorithm papers—or their associated repositories—also report benchmark results. Table 3 summarizes the most relevant recent benchmarks, the algorithms they compare, and the test data they use.

TABLE 3 Overview of existing benchmarks and their limitations

Benchmark	Compared algorithms	Test data
`dtoa-benchmark`	Mostly Grisu family methods	Generates 1 000 random 64-bit values;
`parse-bench`	Principally Ryū, Dragonbox and `std::to_chars`	Three hard-coded values (`123456`, `1`, `333.323`) repeated many times;
`Drachennest`	Grisu3, Ryū, Schubfach, Dragonbox, `std::to_chars`	Random doubles in $[1, 2]$; random doubles in $[10^k, 10^{k+1}]$; random doubles in $[0; 10^{10}]$; random 64-bit bit-patterns;
`Ryū's paper`	C impl. of Ryū v. double-conversion (Grisu3 impl.); Java impl. of Ryū v. OpenJDK's native formatter and Jaffer's variant [17]	Random sampling (Mersenne Twister) of 1,000 32- and 64-bit values interpreted as floats;
`Dragonbox's preprint`	Ryū, Grisu-Exact, Schubfach, Dragonbox	100,000 random numbers (random significand and exponent) measured 1,000 times each; 1,000,000 uniformly generated floats measured 1,000 times each;

A key limitation of these benchmarks is that they do not cover all important algorithms listed in Table 2, nor do they evaluate standard or third-party libraries from other languages (e.g., Google's double-conversion or Swift's C++ `dtoa`). Another common limitation is the restricted hardware and compiler environments used for evaluation: for example, the Dragonbox preprint benchmarks were run solely on an Intel i7-7700HQ using Clang-cl. In addition, none of these benchmarks measure the length of the generated strings, even though string length is crucial for fair comparison—algorithms producing shorter outputs may not be directly comparable to those producing longer ones.

The benchmark datasets themselves also have important shortcomings. All but Ryū's paper consider only 64-bit floating-point numbers. Nevertheless, 32-bit floating-point values remain widely used in applications where reduced memory footprint or bandwidth efficiency matter. Benchmarks that focus exclusively on 64-bit numbers therefore miss important practical scenarios, e.g., in mobile applications, GPUs, and embedded systems. More importantly, all rely on synthetic data, whereas uniformly distributed floating-point numbers are rarely encountered in practice. In real-world applications—such as telemetry, finance, or science—floating-point values typically exhibit non-uniform distributions, with some values much more frequent than others. Taken together, these methodological gaps—incomplete algorithm and library coverage; lack of real-world datasets and 32-bit numbers; limited hardware and compiler con-

[5] `https://github.com/miloyip/dtoa-benchmark`
[6] `https://github.com/alugowski/parse-bench`
[7] `https://github.com/abolz/Drachennest`

figurations; and absence of any evaluation of output string lengths—motivate our empirical study.

3.3 | Parsing Numbers

A tangentially related problem consists of parsing strings to recover binary floating-point numbers. The early work was conducted by Clinger [18, 19]. His work describes an accurate decimal to binary conversion. Gay [8] improved upon Clinger's work by introducing several new optimizations. Lemire [20] provides a significantly (e.g., $4\times$) faster approach by observing that, in the common case, the significand fits in a 64-bit word and only needs to be multiplied by (at most) a 128-bit integer. Mushtak and Lemire completed the work by showing that no fallback is necessary: the core algorithm is guaranteed to succeed [21].

4 | EXPERIMENTS

This section details our experimental setup and methodology, designed to address the gaps identified previously. We first describe the systems used for benchmarking, the datasets employed, and the algorithms and libraries tested. The subsequent subsections present the results and main findings of our experiments.

4.1 | Systems

Our benchmarks were executed on the systems listed in Table 4. The Apple M4 Max results were obtained on a MacBook Pro (2024), while all other systems—except the Ryzen 9900X—were hosted on Amazon Web Services (AWS). For most systems, we compared binaries compiled with both the GNU C++ compiler (`g++`) and the LLVM Clang compiler (`clang++`), using the corresponding standard libraries (`libstdc++` and `libc++`). On the Apple M4 Max, only Clang was used. Unless otherwise noted, we used `g++` version 13 and `clang++` version 18. On the Apple M4 Max, we used Apple Clang 17. On the AMD Ryzen 9900X, we used `g++-15` and `clang++-20`.

To our knowledge, this is the first evaluation to include both contemporary x86-64 (Zen 2–5, Ice Lake, Sapphire Ridge) and ARM/AArch64 (M4 Max, Neoverse N1/V1/V2) architectures in a unified experimental framework.

4.2 | Data

We use three core datasets selected to represent distinct numerical formats and common use cases when converting floating-point numbers to strings. These datasets, summarized in Table 5, include compactly represented integers, high-precision floating-point numbers serialized as strings, and synthetically generated uniformly distributed numbers.

- The *mesh* dataset contains vertex coordinates from a triangulated 3D surface. Many values are small (typically in $[-1, 3]$) and are represented with few characters, including a large proportion of exact integers.
- The *canada* dataset is derived from a JSON file [22] from the GeoJSON project, containing 64-bit floating-point numbers serialized as strings. These values represent geographic coordinates and attributes (e.g., `83.109421000000111`), and are representative of Geographic Information Systems (GIS) and navigation pipelines.
- The *unit* dataset consists of uniformly generated floating-point numbers in the interval $[0, 1)$. While synthetic, it serves as a useful baseline for comparison with prior work, where such distributions are common (e.g., to store normalized values or probabilities).

For all benchmarks, we use arrays of either 32- or 64-bit numbers. When the original source provides only 64-bit

TABLE 4 Systems used for benchmarking

Processor	Frequency	Microarchitecture	Memory
Apple M4 Max	4.4 to 4.5 GHz	unnamed (aarch64, 2024)	LPDDR5X (7500 MT/s)
AMD Ryzen 9 9900X	4.4 to 5.6 GHz	Zen 5 (x86-64, 2024)	DDR5 (6000 MT/s)
AWS Graviton 2	2.5 to 2.5 GHz	Neoverse N1 (aarch64, 2019)	DDR4 (3200 MT/s)
AWS Graviton 3	2.6 to 2.6 GHz	Neoverse V1 (aarch64, 2022)	DDR5 (4800 MT/s)
AWS Graviton 4	2.8 to 2.8 GHz	Neoverse V2 (aarch64, 2024)	DDR5 (5600 MT/s)
AMD EPYC 7R32	2.8 to 3.3 GHz	Zen 2 (x86-64, 2019)	DDR4 (2933 MT/s)
AMD EPYC 7R13	2.7 to 3.7 GHz	Zen 3 (x86-64, 2021)	DDR4 (3200 MT/s)
AMD EPYC 9R14	3.0 to 3.7 GHz	Zen 4 (x86-64, 2023)	DDR5 (4800 MT/s)
Intel Xeon 8124M	3.0 to 3.5 GHz	Skylake-SP (x86-64, 2017)	DDR4 (2666 MT/s)
Intel Xeon 8375C	2.6 to 3.8 GHz	Ice Lake-SP (x86-64, 2021)	DDR4 (3200 MT/s)
Intel Xeon 8488C	2.0 to 3.8 GHz	Sapphire Ridge (x86-64, 2023)	DDR5 (4800 MT/s)

TABLE 5 Dataset summary. An integer value is defined as a number exactly representable by a 64-bit signed integer. The number of digits is the minimum required for exact round-trip conversion.

Name	Count	Integers	Average digits	
			32-bit	64-bit
mesh	73 019	44 557	4.7	6.6
canada	111 126	46	7.3	15.3
unit	100 000	0	7.5	16.0

numbers, we cast them to 32-bit prior to benchmarking.

To better approximate real-world workloads and address the limitations of prior benchmarks discussed in Section 3.2, we also assembled a collection of additional datasets drawn from finance, astronomy, machine learning, and meteorology [23]. These are summarized in Table 6. They consist entirely of floating-point values that arise in deployed systems and public APIs, and thus complement the three core datasets above. Specifically:

- *bitcoin*: daily closing prices of Bitcoin (USD), typical of financial APIs and market-data feeds.
- *marine*: values from a marine-robotics inverse-kinematics example, representative of control and scientific-computing workloads.
- *mobilenetv3_large*: model weights from the MobileNetV3-Large ImageNet model, characteristic of machine-learning pipelines and neural-network parameter storage.
- *gaia*: astrometric and photometric values from the ESA Gaia DR3 catalog (positions, parallaxes, fluxes), representing large-scale scientific data with substantial dynamic range.
- *noaa_global_hourly_2023*: surface-station telemetry (temperature, pressure, visibility), representative of noisy real-world measurement streams.

- *noaa_gfs*: fields extracted from NOAA GFS forecast-model output (temperature, humidity, wind components), representative of large-scale gridded scientific simulations.

TABLE 6 Additional real-world datasets used in some of our experiments. "Integers" counts values exactly representable as 64-bit signed integers.

Name	Count	Integers	Binary type
`bitcoin`	943	0	binary64
`marine`	114 950	0	binary32
`mobilenetv3_large`	5 507 432	0	binary32
`gaia`	3 879 638	0	binary64
`noaa_global_hourly_2023`	1 000 000	428 161	binary32
`noaa_gfs`	4 841 536	970 436	binary32

Results in Sections 4.4–4.7 are reported for the mesh, canada and unit datasets. Section 4.8 presents additional experiments on the datasets listed in Table 6. Complete results are available in our public benchmark data repository.

4.3 | Software Implementations

We benchmark a selection of C and C++ libraries capable of converting IEEE floating-point numbers to their shortest decimal string representations. Our benchmarking code, synthetic data generators, and datasets are all publicly available online.[8] The benchmarked libraries and algorithms are the following:

- *Grisu3 and Schubfach*: Both are evaluated using the `Drachennest` library.[9]
- *Dragon4*: Benchmarked using a dedicated library[10] rather than the Drachennest version.
- *Ryū*: Evaluated using the `Ryu` library.[11]
- *Dragonbox*: Benchmarked with the `Dragonbox` library.[12] The author of Dragonbox observes that the string generation they include is not officially part of the algorithm. They make it possible for users to provide their own algorithm to convert significands and exponents to strings.
- *Google double-conversion (Grisu3-based)*: We include Google's `double-conversion` library.[13] We use double-conversion with the default flag. There are additional flags: e.g., for forcing a trailing decimal point (and optional zero) for integer-valued floats like "123." or for emitting '+' in positive exponents. We omit Google's Abseil and `snprintf`, as they do not guarantee shortest-string output.
- *fmt (Dragonbox-based)*: Evaluated using the `fmt` library,[14] which employs a version of Dragonbox internally.

[8] `https://github.com/fastfloat/float_serialization_benchmark`

[9] `https://github.com/abolz/Drachennest`, git hash `e6714a3` (May 2021). Only 64-bit function is available for Grisu3. We exclude Grisu2, as it can produce longer-than-necessary significands. Drachennest also implements Dragon4, but due to concerning faults (see: `https://github.com/fastfloat/float_serialization_benchmark/pull/18`), we benchmark Dragon4 using a separate implementation.

[10] `https://github.com/lemire/Dragon4.git`, git hash `0ce72aa` (March 2025). Modified for portability (renamed `Math.h` to `DragonMath.h`). This is an implementation of Juckett [24] based on Burger and Dybvig's variant of Dragon4 [9], and is expected to be faster than a straightforward Dragon4. However, its 64-bit implementation is not entirely correct (e.g., `5e-324` outputs a long string of zeros).

[11] `https://github.com/ulfjack/ryu`, git hash `e6714a3` (February 2024).

[12] `https://github.com/jk-jeon/dragonbox`, version 1.1.3 (June 2022).

[13] `https://github.com/google/double-conversion`, version 3.3.1 (February 2025).

[14] `https://github.com/fmtlib/fmt`, version 11.1.4.

- *SwiftDtoa*: The Swift language implementation includes a C++ function combining ideas from Grisu2 and Ryū.[15]
- *std::to_chars*: C++17's standard floating-point to string function.

All algorithms above (except as noted) provide correct round-trips for both 32- and 64-bit floats. Drachennest's Grisu3 implementation is limited to 64-bit values. The Dragon4 implementation used is not fully correct for 64-bit; e.g., it mishandles subnormals such as `5e-324`.

When libraries expose flags or configuration options that alter the printed format (e.g., forcing a trailing decimal point, controlling exponent signs, or choosing between fixed and scientific styles), we systematically use the default settings provided by the library.

For consistency, we focus on algorithms that generate the entire output string. Though there are differences in the strings generated, we take these small differences into account in our analysis (See Section 4.5). Libraries like Gay's `dtoa`[16] and teju_jagua[17] were not benchmarked, as they compute only the decimal significand and exponent, requiring additional string-generation code. Though it is not difficult to implement the string generation, such work would have an impact on the benchmarking results. We also restrict our study to C and C++ implementations. Fair cross-language benchmarking is outside our scope, and other languages often adopt techniques originating in C or C++.

4.4 | None Provide the Shortest Strings

A key goal in floating-point to string conversion is to produce the shortest possible decimal string representations. Despite the centrality of string length in practical deployments (serialization, logging, telemetry), prior benchmark studies have never quantified end-to-end output length at scale. Our results provide, to our knowledge, the first such characterization. We define the number of significant digits by omitting leading and trailing zeros, so that strings like `1.0`, `10`, and `0.1` each have exactly one significant digit. This becomes clearer when using scientific notation: `1E0`, `1E1`, and `1E-1`. Although tested algorithms produce strings with the fewest digits required for exact round-trips (except Dragon4 for 64-bit numbers due to the aforementioned bug in the available implementation), none consistently generate the shortest strings in terms of total character length.[18] For instance, the C++17 standard library's `std::to_chars` renders the number 0.00011 as `0.00011` (7 characters), while the shorter scientific form `1.1e-4` (6 characters) is possible. Similarly, it outputs 12300 as `1.23e+04` (8 characters) rather than the shorter `1.23e4` (6 characters). Such longer outputs result from formatting rules inherited from the C standard, particularly regarding scientific notation (`%e`):

> *A double argument representing a floating-point number is converted in the style `[-]d.ddde±dd`. `[...]`, The exponent always contains at least two digits, and only as many more digits.*

These rules mandate a positive exponent sign and at least two digits for the exponent, following historical precedent. When converting floating-point numbers, `std::to_chars` chooses the shortest notation between fixed-point (`%f`) and scientific (`%e`), favoring fixed-point notation if lengths are equal.[19] Similar constraints affect libraries such

[15] `https://github.com/swiftlang/swift.git`, git hash `6a862d2` (March 2025).

[16] Gay's dtoa: `https://www.netlib.org/fp/`, retrieved January 2025.

[17] Teju Jaguá: `https://github.com/cassioneri/teju_jagua`, git hash `e62fcfc` (March 2025).

[18] We assess shortest-string behavior by cross-comparing output lengths across algorithms and against the valid decimal representation already present in the dataset. A shorter dataset string implies that none of the tested algorithms produced a minimal-length result.

[19] For example, the 32-bit value `4.27819e+09` can be equally represented by the strings `4.27819e+09` (scientific), `4278190080`, or `4278190000` (both fixed-point); all have the same string length (10 characters). The C++ standard requires choosing fixed-point notation in case of a tie. Among possible fixed-point outputs, the standard then mandates selection of the string numerically closest to the exact value. Here, `4278190080` (with 9 significant digits) is chosen over `4278190000` (6 significant digits), even though it has more digits, since significant digits are not considered, only string length and numerical proximity.

TABLE 7 Average number of characters. We use the libstdc++ `std::to_chars` implementation.

Name	mesh		canada		unit	
	32-bit	64-bit	32-bit	64-bit	32-bit	64-bit
Dragon4	5.863	7.589	8.823	16.800	9.628	18.273
fmt	5.913	7.589	8.823	16.800	9.627	18.272
Grisu3	–	7.589	–	16.800	–	18.273
Grisu-Exact	7.697	9.571	10.824	18.800	11.515	20.160
Schubfach	5.863	7.589	8.823	16.800	9.628	18.273
Dragonbox	8.005	9.879	10.824	18.800	11.515	20.160
Ryū	8.005	9.879	10.824	18.800	11.515	20.160
double_conversion	5.863	7.737	8.823	16.800	9.627	18.272
swiftDtoa	7.034	8.810	8.825	16.801	9.627	18.272
`std::to_chars`	5.863	7.589	8.823	16.800	9.627	18.272
Shortest	4.537	6.263	8.823	16.800	9.626	18.268

as fmt, Grisu3, Schubfach, double_conversion, and swiftDtoa. In contrast, Ryū, Dragonbox, and Grisu-Exact do not strictly follow these rules but frequently prefer scientific notation even when longer (e.g., printing `0.1` as `1E-1`).

Table 7 summarizes the average string length in characters across our three core datasets. For 32-bit numbers, algorithms like Dragon4, double_conversion, fmt, and `std::to_chars` consistently produce the shortest average lengths across the mesh (approximately 5.9 characters), canada (8.8 characters), and unit (approximately 9.6 characters) datasets. Conversely, Grisu-Exact, Ryū, and Dragonbox consistently yield longer strings (7.7 to 8.0 characters for mesh; 10.8 characters for canada; 11.5 characters for unit).

Similar trends emerge for 64-bit numbers: Dragon4, fmt, Grisu3, Schubfach, and `std::to_chars` produce shorter average lengths (7.6 for mesh, 16.8 for canada, and around 18.3 for unit), closely followed by double_conversion and swiftDtoa. Grisu-Exact, Ryū, and Dragonbox consistently produce longer strings (approximately 9.9 for mesh, 18.8 for canada, and 20.16 for unit).

The *Shortest* row in Table 7 illustrates the optimal achievable lengths. Notably, on the mesh dataset, `std::to_chars` produces strings that are significantly longer than optimal. For example, its outputs are roughly 30% longer than the true minimum for 32-bit numbers and about 20% longer for 64-bit numbers. These discrepancies show that differences in total character length between implementations are substantial, underscoring the importance of evaluating both digit count and final string length when benchmarking conversion algorithms.

This behavior is not a correctness issue, but rather a deliberate design decision shared by nearly all modern algorithms, which are engineered to minimize the number of significant digits of the decimal significand—not the total character length of the printed string. Formatting decisions (fixed vs. scientific notation, exponent width, trailing zeros) are delegated to the caller or the standard library. Our measurements show that this separation can yield *unexpectedly large* gaps between the shortest valid significand and the shortest possible printed representation—sometimes exceeding 30%. This highlights a practical yet previously overlooked distinction between *shortest significand* and *shortest string*.

4.5 | Performance Comparison: Schubfach and Dragonbox are faster

We compiled all functions in release mode using the default CMake parameters (`-O3 -DNDEBUG`). When possible (e.g., on x86-64 systems), we enabled hardware-specific optimizations (`-march=native`). CPU performance counters were used to record the number of completed instructions and CPU cycles, in addition to wall-clock time, to better capture microarchitectural effects.

Tables 8 and 9 present detailed benchmarking results for 64-bit float-to-string conversion on Apple M4 Max and AMD Ryzen 9900X processors. We report three key metrics: nanoseconds per float (ns/f), instructions per float (ins/f), and instructions per cycle (ins/c), to capture both algorithmic efficiency and hardware utilization. While previous work has occasionally reported wall-clock timing, instruction-level metrics for float-to-string conversion (in particular ins/f and ins/c) have not been systematically analyzed in the literature. We use these to distinguish intrinsic algorithmic cost (ins/f) from microarchitectural utilization (ins/c), exposing differences invisible to timing-only evaluations.

To assess measurement stability, each benchmark was repeated 100 times. On dedicated hardware (Apple M4 Max, AMD Ryzen 9900X), timing variability was consistently low (median 0.7%, max 2.7%). In contrast, on cloud instances, a handful of runs showed higher variability (median 0.9%, max 6.9%). These outliers likely reflect intermittent resource contention or infrastructure noise inherent to virtualized environments, but appear to affect all algorithms similarly. Cycles per float (`c/f`) tracked timing variability closely, with a median variation of 0.4%, though rare outliers again reached 6%. In all cases, instructions per float (`ins/f`) remained completely deterministic (0.0% variability), confirming that the executed instruction path is unaffected by runtime fluctuations. Observed performance jitter therefore arises solely from external factors impacting timing and cycle counts, rather than any non-determinism in the algorithms themselves.

TABLE 8 Apple M4 Max results (Apple/LLVM 17, 64-bit floats)

Name	mesh			canada			unit		
	ns/f	ins/f	ins/c	ns/f	ins/f	ins/c	ns/f	ins/f	ins/c
Dragon4	69	1500	5.3	150	3000	4.8	170	3300	4.6
fmt	22	530	5.4	29	640	5.0	30	510	3.8
Grisu3	10	260	5.6	24	440	4.2	26	470	4.0
Grisu-Exact	11	320	6.3	15	340	5.1	18	340	4.2
Schubfach	7.2	210	6.4	12	310	5.9	14	290	4.7
Dragonbox	7.7	220	6.6	9.5	240	5.6	12	230	4.2
Ryū	9.9	270	6.0	12	330	6.3	13	310	5.4
double_conversion	26	640	5.5	42	910	5.1	43	880	4.8
swiftDtoa	14	390	6.0	16	360	5.1	20	390	4.4
`std::to_chars`	13	350	5.8	15	440	6.6	16	410	5.6

Across all datasets on the Apple M4 Max processor (Table 8), Dragonbox and Schubfach consistently achieve the fastest performance, with runtime (ns/f) measures ranging from 7.2 to 14. Schubfach notably achieves the lowest runtime on the mesh dataset (7.2 ns/float), while Dragonbox leads on the canada dataset (9.5 ns/float). Both algorithms also show minimal instruction counts (210 to 310 instruction/float), reflecting efficient implementations. Ryū closely

TABLE 9 AMD Ryzen 9900X results (g++15, 64-bit floats)

Name	mesh			canada			unit		
	ns/f	ins/f	ins/c	ns/f	ins/f	ins/c	ns/f	ins/f	ins/c
Dragon4	82	2300	5.0	170	4700	5.2	190	5000	4.9
fmt	30	570	3.5	40	840	3.8	35	560	2.9
Grisu3	12	290	4.5	29	630	4.0	26	510	3.6
Grisu-Exact	18	370	3.7	24	520	3.9	21	370	3.1
Schubfach	9.9	250	4.5	24	490	3.7	19	320	3.0
Dragonbox	11	260	4.3	18	410	4.1	15	240	3.0
Ryū	14	320	4.3	24	580	4.3	20	400	3.5
double_conversion	27	610	4.0	45	1000	4.0	39	810	3.7
swiftDtoa	23	490	3.8	28	590	3.8	27	440	3.0
`std::to_chars`	18	490	4.8	30	780	4.8	25	600	4.3

follows, with runtimes ranging from 9.9 to 13 ns/float and instruction counts between 270 and 330 ins/f. On the Apple M4 Max, Dragon4 is consistently the slowest, with ns/f measures between 69 (mesh) and 170 (unit), and significantly higher instruction counts (1500 to 3300 instruction/float). The double_conversion and fmt functions exhibit moderate performance (22 to 43 ns/float), substantially faster than Dragon4 but slower than Dragonbox and Schubfach. The remaining algorithms—Grisu-Exact, Grisu3, swiftDtoa, and `std::to_chars`—occupy the intermediate performance range (10 to 26 ns/float). Performance trends are similar on the AMD Ryzen 9900X processor (Table 9). Schubfach and Dragonbox again perform best across datasets (e.g., Schubfach at 9.9 ns/float, Dragonbox at 11 ns/float for mesh). Dragon4 remains the slowest, with even higher instruction counts (up to 5000 ins/f on unit) and longer runtimes (up to 190 ns/float). The relative ranking of other algorithms (`std::to_chars`, double_conversion, fmt, etc.) remains largely consistent with the Apple M4 Max results.

Differences in algorithm outputs—particularly string lengths—must be considered when interpreting these results. Dragonbox and Ryū, for instance, often produce longer strings than Schubfach and `std::to_chars`, potentially influencing relative runtimes. Nevertheless, large performance gaps persist even among algorithms generating comparable string lengths (e.g., Schubfach vs. fmt), underscoring genuine algorithmic efficiency differences.

A consistent trend across both the Apple M4 Max and AMD Ryzen 9900X results is that all algorithms perform fastest on the mesh dataset, are slower on the canada dataset, and slowest on the unit dataset (as indicated by the ns/f column in Tables 8 and 9). This ranking reflects the differences in output string lengths reported in Table 7, with the unit dataset producing the longest outputs. While it is tempting to attribute the slowdown solely to the increased cost of formatting longer strings, the instructions-per-cycle (ins/c) results reveal an additional effect: all algorithms achieve lower ins/c on the unit dataset compared to mesh, indicating reduced pipeline throughput.

To further investigate, we profiled the execution of the `std::to_chars` algorithm on the Ryzen 9900X. On the unit dataset, 28% of executed instructions and 34% of cycles occur within the string formatting routine. For comparison, on the mesh dataset, these figures are notably lower: 19% of instructions and 21% of cycles. This increase confirms that longer outputs entail a greater share of processing within the formatting function, yet a substantial portion of instructions and cycles remains attributable to other algorithm components. Furthermore, the higher ratio of cycles to instructions spent in string formatting on the unit dataset suggests microarchitectural bottlenecks, such as increased memory latency or branch misprediction. Overall, these results indicate that the performance degradation on the unit dataset arises from both increased output costs and diminished execution efficiency within the processor pipeline.

Figures 5, 6, and 7 visualize algorithmic performance ($\log_{10}$ ns/f) across datasets, CPUs, compilers, and floating-point widths.[20] In these heatmaps, dark blue regions correspond to the fastest execution speeds, transitioning through light blue and light red to dark red, which indicates the slowest performance. This gradient clearly highlights the influence of compiler choice, CPU architecture, and numerical width (32-bit vs. 64-bit). Several notable insights emerge:

- *Compiler choice affects algorithmic performance*. On the mesh dataset, Schubfach often runs faster when compiled with clang++ (with libc++), while Dragonbox frequently benefits from g++ (with libstdc++). For example, on the Xeon 8488C CPU, Schubfach was approximately 7% faster with clang++, whereas Dragonbox was 12.5% faster with g++. This effect is observed on the majority of CPUs in our study, though the exact magnitude of the compiler advantage varies across architectures and algorithms.
- *CPU architecture influences overall performance*. The Neoverse N1 (Graviton 2), with three arithmetic units but only one capable of executing multiplications, consistently exhibits slower performance across algorithms. In contrast, recent high-end processors like AMD Zen 5 feature six arithmetic units, three of which can perform multiplications in parallel. This greater arithmetic parallelism likely contributes to Apple's M4 Max and similar CPUs typically ranking among the fastest in our tests. This interpretation is further supported by our measurements: the mean instructions-per-cycle (ins/c) across all algorithms and datasets is 3.8 on the Ryzen 9900X, compared to just 2.6 on the Graviton2. Such differences in ins/c reflect the ability of more advanced CPUs to execute a greater number of instructions in parallel.
- *Algorithmic choice dominates relative performance.* While CPU architecture and compiler matter, the number of instructions required by each algorithm varies far more, making algorithm selection the single largest determinant of speed. For example, on the Ryzen 9900X and the unit dataset, ins/f values ranged from 5000 (dragon4) to 240 (dragonbox), whereas ins/c values across all CPUs and algorithms stayed between 2.3 and 5.6. To further support our observation, figures 8, 9 and 10 show the relative performance of the algorithms compared to Dragon4 on selected CPUs. The Apple CPU achieves relatively high performance with Schubfach, Ryū and Dragonbox compared to Dragon4. In other words, it benefits more from a switch to the more recent algorithms than the other selected CPUs. The Neoverse V2 processor has slightly lower relative performance than the other selected CPUs. Yet the curves are visibly correlated: on the canada and unit dataset, Dragonbox gives the best results while on the mesh dataset, Schubfach is slightly superior to Dragonbox. The other processors have similar ratios

4.6 | Advanced CPU instructions are not exploited

We also investigated whether recent float-to-string algorithms are able to leverage advanced instructions available on modern CPUs, such as fused multiply-add (FMA) and vectorization (SIMD). On x86-64, CPUs are grouped into distinct architectural levels (x86-64-v1, v2, v3, and v4), each introducing new instruction sets and capabilities.[21] The x86-64-v1 level is the baseline introduced by AMD in 2003, and it includes foundational 64-bit capabilities like CMOV, SSE, and SSE2, compatible with early processors like AMD K8 and Intel Prescott. The x86-64-v2 level, defined in 2020 by AMD, Intel, Red Hat, and SUSE, adds instructions such as SSE3, SSE4.1, SSE4.2, and POPCNT, aligning with processors from around 2008–2011, like Intel Nehalem. The x86-64-v3 level introduces AVX, AVX2, FMA, and MOVBE, targeting

[20] Raw benchmark data for these visualizations are available on the paper's website at `https://www.jaelgareau.com/en/publication/gareau_lemire-spe25`.

[21] See `https://en.wikipedia.org/wiki/X86-64#Microarchitecture_levels`.

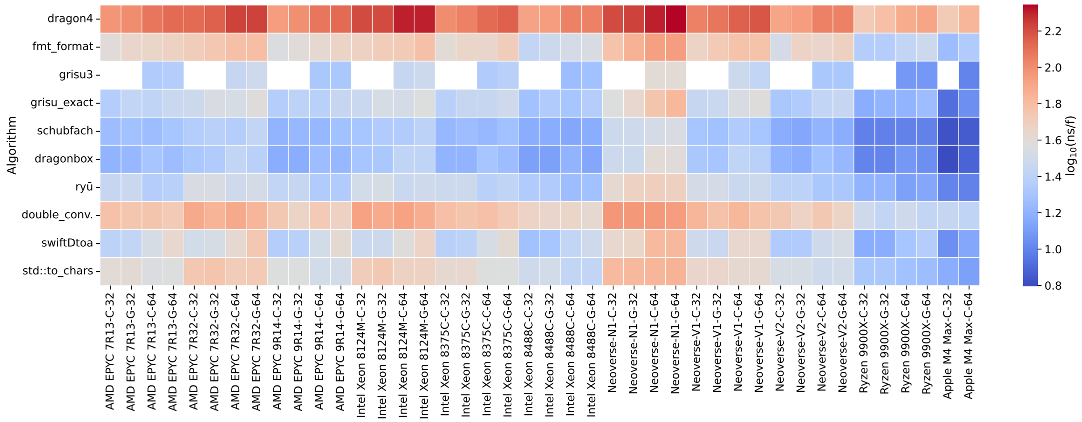

FIGURE 5 Algorithm performance ($\log_{10}$ ns/f) across CPUs, compilers, and widths (mesh dataset)

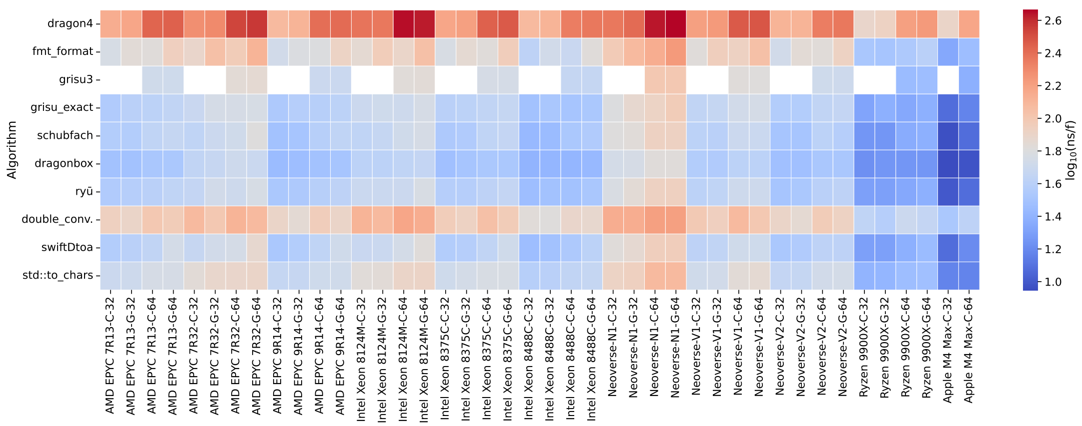

FIGURE 6 Algorithm performance ($\log_{10}$ ns/f) across CPUs, compilers, and widths (canada dataset)

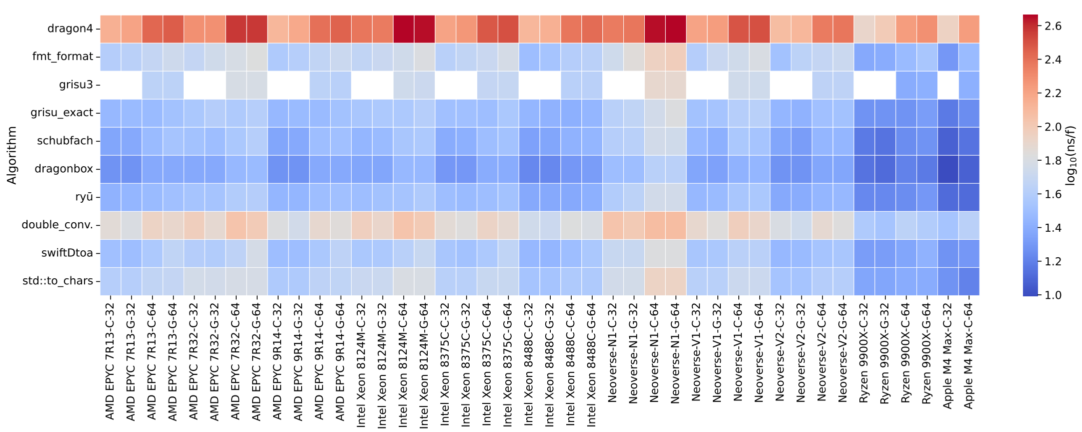

FIGURE 7 Algorithm performance ($\log_{10}$ ns/f) across CPUs, compilers, and widths (unit dataset)

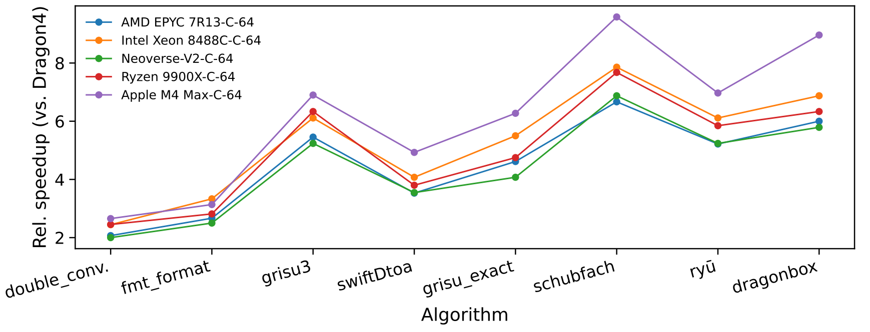

FIGURE 8 Relative speedup (vs. `dragon4`) for selected CPUs (mesh dataset)

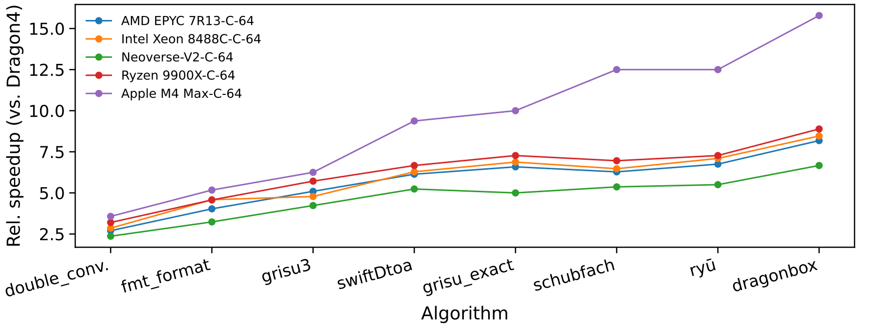

FIGURE 9 Relative speedup (vs. `dragon4`) for selected CPUs (canada dataset)

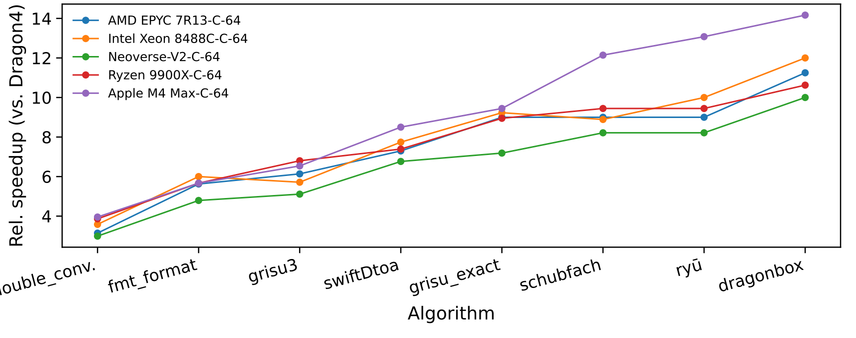

FIGURE 10 Relative speedup (vs. `dragon4`) for selected CPUs (unit dataset)

CPUs from 2013–2015, such as Intel Haswell. The x86-64-v4 level incorporates AVX-512, doubling vector instruction width to 512 bits, and is supported by newer CPUs like AMD Zen 4 and Zen 5.

Table 10 summarizes the performance of Schubfach and Dragonbox across all architectural levels on the Ryzen 9900X

TABLE 10 Schubfach vs. Dragonbox across architectural levels (g++15, 64-bit floats, Ryzen 9900X).

Level	Name	mesh			canada			unit		
		ns/f	ins/f	ins/c	ns/f	ins/f	ins/c	ns/f	ins/f	ins/c
x86-64-v1	schubfach	9.8	250	4.6	23	490	3.9	18	310	3.2
	dragonbox	11	260	4.2	18	410	4.0	15	240	3.0
x86-64-v2	schubfach	9.7	250	4.7	23	490	3.9	18	310	3.2
	dragonbox	11	260	4.2	18	410	4.1	15	240	2.9
x86-64-v3	schubfach	9.7	250	4.7	23	490	3.9	18	310	3.2
	dragonbox	11	260	4.2	18	410	4.1	15	240	2.9
x86-64-v4	schubfach	9.8	240	4.6	24	480	3.6	19	310	2.9
	dragonbox	11	260	4.2	18	410	4.1	15	250	3.0

(Zen 5). The results show that enabling newer instructions provides, at best, marginal improvements in performance. In one instance (Dragonbox on the unit data), the version of our software compiled for the most advanced (x86-64-v4) instructions required slightly more instructions. In another (Schubfach on the canada data), targeting x86-64-v4 saved a small number of instructions. This suggests that the Schubfach and Dragonbox implementations do not benefit from advanced instructions on x86-64 processors.

4.7 | Converting 32-bit numbers may be faster?

When converting floating-point numbers to strings, the output is typically shorter for 32-bit than for 64-bit values: at most nine digits suffice for 32-bit numbers, while up to seventeen are needed for 64-bit numbers. We might therefore expect that 32-bit conversions are generally faster, since less work is required for formatting and string generation.

Figure 11 confirms this expectation on the Apple M4 Max processor. The fastest implementations convert over 96 Mfloat/s for 32-bit floats (Schubfach: 109 Mfloat/s, Dragonbox: 112 Mfloat/s, Ryū: 96 Mfloat/s), while their 64-bit performance is typically lower (Schubfach: 83 Mfloat/s, Dragonbox: 106 Mfloat/s, Ryū: 83 Mfloat/s). The difference is especially pronounced for the slowest algorithm: Dragon4 processes only 12 Mfloat/s for 32-bit versus 7 Mfloat/s for 64-bit. Interestingly, some algorithms—particularly `std::to_chars` and Dragonbox—show little or no difference between 32- and 64-bit speeds (e.g., `std::to_chars` achieves 66 Mfloat/s for both widths; Dragonbox reaches 112 Mfloat/s for 32-bit and 106 Mfloat/s for 64-bit). This suggests that in certain libraries, the conversion routine is dominated by fixed overhead, or that the core bottleneck is not string length but the underlying algorithm.

Another notable point is that *character throughput* (total characters produced per second) is higher for 64-bit numbers, since their decimal representations are, on average, nearly twice as long. Thus, while more numbers can be processed per second in the 32-bit case, more textual data can be produced per second in the 64-bit case.

4.8 | Additional real-world datasets

To verify that our conclusions are not an artifact of the three core datasets, we also benchmarked all implementations on the additional real-world datasets listed in Table 6. For space reasons, we report here only a condensed view of

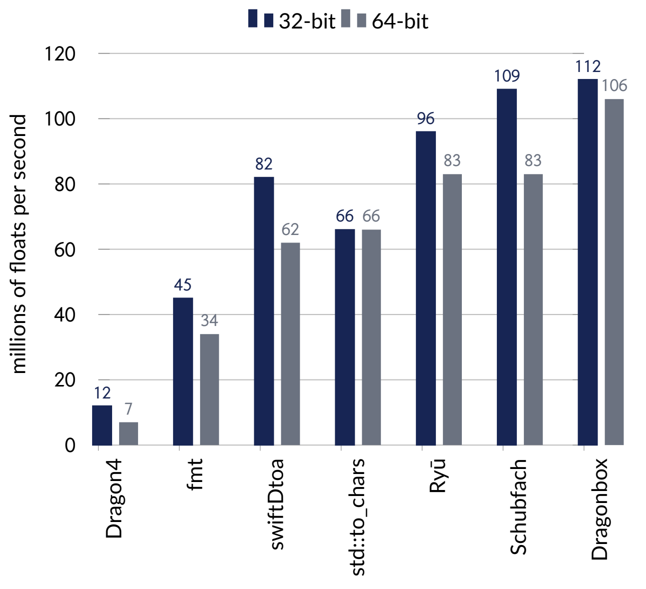

FIGURE 11 Throughput (Mfloat/s) on Apple M4 Max (canada dataset, 32-/64-bit)

the results. We ran the full benchmark suite on these datasets on both the Apple M4 Max and AMD Ryzen 9 9900X processors described in Table 4, using the same compiler configurations as in Section 4.1.

Table 11 reports detailed results for the three implementations we use in this section—Dragonbox, Schubfach, and Ryū—which are representative of the performance trends observed across all algorithms in the full benchmark. The absolute numbers vary across datasets and architectures, but the relative ordering is consistent with our findings for mesh, canada, and unit: Dragonbox and Schubfach are typically the fastest, while Ryū is consistently slower by a moderate but systematic margin.

Several patterns mirror those observed earlier. First, algorithms that emit longer strings—particularly Ryū and Dragonbox—show slightly higher instruction counts on high-dynamic-range datasets, such as `bitcoin` or `gaia`. Second, datasets dominated by exact integers or very small magnitudes (e.g., `noaa_global_hourly_2023` and `marine_ik`) yield the shortest strings and therefore the fastest conversions among the real-world datasets. Datasets that are still relatively small but contain fewer integers (e.g., `noaa_gfs`) produce moderately longer strings and slightly slower performance, though they are still faster than high-dynamic-range sets.

Crucially, none of the additional datasets reveals outliers that contradict our earlier conclusions. Across all workloads, Dragonbox and Schubfach remain among the fastest implementations, Ryū is competitive but consistently slower, and the performance gaps fall squarely within the ranges observed on the core datasets.

String-length behavior generalizes just as cleanly. None of the implementations reaches the theoretically shortest possible strings, and the excess is again frequently in the 20–30% range, particularly on datasets with large exponents such as `bitcoin` and `gaia`. This reinforces our earlier finding that minimizing the number of significant digits is insufficient to guarantee shortest-string output, and that real-world workloads tend to amplify these effects.

Overall, these results support the robustness of our conclusions across a broad range of practical numerical domains, including finance, astronomy, machine learning, and meteorology. Detailed per-dataset results for all algorithms are available in our public benchmark repository.

TABLE 11 Performance (ns/f, ins/f, ins/c) of Dragonbox, Ryū, and Schubfach on six additional real-world datasets. Results are shown for two CPUs (Ryzen 9 9900X, Apple M4 Max) and exhibit the same ranking and trends seen on the core datasets.

Dataset	Algorithm	AMD Ryzen 9 9900X			Apple M4 Max		
		ns/f	ins/f	ins/c	ns/f	ins/f	ins/c
bitcoin (f64)	dragonbox	12.0	290	4.3	8.6	270	6.4
	ryu	23.0	530	4.1	13.0	420	6.6
	schubfach	15.0	340	4.1	8.4	310	7.4
gaia (f64)	dragonbox	22.0	380	3.2	14.0	250	4.1
	ryu	26.0	540	3.7	16.0	320	4.8
	schubfach	26.0	460	3.3	16.0	310	4.8
marine_ik (f32)	dragonbox	9.3	210	4.2	5.7	180	7.1
	ryu	14.0	370	4.6	8.2	270	7.2
	schubfach	10.0	260	4.6	6.3	230	8.2
mobilenetv3_large (f32)	dragonbox	18.0	220	2.2	14.0	180	3.1
	ryu	19.0	350	3.3	14.0	250	4.3
	schubfach	17.0	270	2.8	13.0	240	4.4
noaa_gfs_1p00 (f32)	dragonbox	12.0	190	2.9	9.1	160	4.2
	ryu	15.0	310	3.8	11.0	220	5.0
	schubfach	13.0	240	3.3	9.0	210	5.5
noaa_global_hourly_2023 (f32)	dragonbox	10.0	210	3.7	7.7	170	5.1
	ryu	17.0	400	4.3	13.0	310	5.6
	schubfach	8.9	220	4.4	6.8	190	6.6

5 | CONCLUSION

Although the original Dragon4 algorithm required thousands of instructions to convert a single IEEE floating-point number to a decimal string, modern methods accomplish the same task in just a few hundred instructions. This tenfold improvement over three decades represents an average software efficiency gain of about 8 % per year—a striking reminder that performance gains in software, like those in hardware, can accumulate to significant effect over time [25].

Yet, despite these advances, our results show that widely used libraries (such as `fmt` and `libc++`) still require more instructions than state-of-the-art algorithms like Dragonbox. This suggests that these libraries may be prioritizing other trade-offs (e.g., code size or portability), or that further performance optimizations remain unexplored. Moreover, while each algorithm we tested produces a valid string representation, none consistently generates the shortest possible output. We identify two key directions for future research:

- *High performance string generation.* Historically, converting decimal significands and exponents into ASCII strings was a minor portion of the overall float-to-string process. However, as core conversion algorithms have become dramatically faster, this final string generation step can now consume a significant share of runtime—for example, only 2% of cycles in Dragon4 on the unit dataset, but up to 34% in `std::to_chars`. Future work should explore advanced optimizations for this stage, ideally decoupled from earlier steps, to further accelerate end-to-end conversion. Related to this, several modern algorithms (e.g., Dragonbox, Schubfach, and Ryu) efficiently compute the shortest decimal significand but leave the final construction of the shortest string to user-defined routines. Developing unified backends that consistently generate the shortest possible decimal string across algorithms represents a promising direction for future work.

- *Exploiting modern CPU features.* Our results show that current algorithms do not fully utilize advanced instructions such as FMA and SIMD, and that enabling SIMD extensions (e.g., AVX-512) yields, at best, marginal gains for single-value conversions. Future work should investigate whether these capabilities can be leveraged by reformulating the problem—for example, by designing algorithms that convert multiple floats to strings in parallel (batch conversion). Such an approach could better exploit vectorization and other modern CPU features, potentially unlocking significant performance gains for applications that require bulk formatting of floating-point data.

Also, as future work, we might investigate the generation of the significand and exponent independently from the generation of the string.

Author Contributions

Jaël Champagne Gareau: conceptualization; investigation; software; experimentation; writing-review and editing. Daniel Lemire: conceptualization; software; validation; experimentation; data analysis; writing-original draft; writing-review and editing.

Funding Information

This work was supported by the Natural Sciences and Engineering Research Council of Canada, Grant Number: RGPIN-2017-03910. The first author is supported by a postdoctoral grant from Fonds de recherche du Québec, https://doi.org/10.69777/361128.

Data Availability Statement

All our data and software is freely available online. The C++ benchmarking software is available online at https://github.com/fastfloat/float_serialization_benchmark. All of our test datasets are at https://github.com/fastfloat/float-data. We collected performance data and we make it available on the paper's webpage at https://www.jaelgareau.com/en/publication/gareau_lemire-spe25/.

references

[1] Goldberg D. What Every Computer Scientist Should Know about Floating-Point Arithmetic. ACM Comput Surv 1991 Mar;23(1):5–48.

[2] Matula DW. In-and-out conversions. Commun ACM 1968 Jan;11(1):47–50.

[3] Coonen JT. An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic. Computer 1980 Jan;13(1):68–79.

[4] Coonen JT. Contributions to a Proposed Standard for Binary Floating~Point Arithmetic. Thesis, University of California; 1984.

[5] Steele Jr GL, White JL. How to print floating-point numbers accurately. In: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation; 1990. p. 112–126.

[6] IEEE Computer Society Standards Committee. IEEE Standard for Binary Floating-Point Arithmetic, vol. 754. New York, NY: IEEE; 1985. ANSI/IEEE Std 754-1985.

[7] Steele GL, White JL. How to print floating-point numbers accurately. SIGPLAN Not 2004 Apr;39(4):372–389.

[8] Gay DM, Correctly rounded binary-decimal and decimal-binary conversions; 1990. AT&T Bell Laboratories Numerical Analysis Manuscript 90-10.

[9] Burger RG, Dybvig RK. Printing Floating-Point Numbers Quickly and Accurately. SIGPLAN Not 1996 May;31(5):108–116.

[10] Loitsch F. Printing Floating-Point Numbers Quickly and Accurately with Integers. In: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI '10, New York, NY, USA: Association for Computing Machinery; 2010. p. 233–243. https://doi.org/10.1145/1806596.1806623.

[11] Andrysco M, Jhala R, Lerner S. Printing Floating-Point Numbers: A Faster, Always Correct Method. In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages POPL '16, New York, NY, USA: Association for Computing Machinery; 2016. p. 555–567.

[12] Adams U. Ryū: fast float-to-string conversion. In: Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation; 2018. p. 270–282.

[13] Adams U. Ryū revisited: printf floating point conversion. Proceedings of the ACM on Programming Languages 2019;3(OOPSLA):1–23.

[14] Giulietti R, The Schubfach way to render doubles; 2020. https://drive.google.com/file/d/1luHhyQF9zKlM8yJ1nebU0OgVYhfC6CBN/view [last checked January 2026].

[15] Jeon J, Grisu-Exact: A Fast and Exact Floating-Point Printing Algorithm; 2020. https://github.com/jk-jeon/Grisu-Exact/blob/master/other_files/Grisu-Exact.pdf [last checked January 2026].

[16] Jeon J, Dragonbox: A New Floating-Point Binary-to-Decimal Conversion Algorithm; 2022. https://github.com/jk-jeon/dragonbox/blob/master/other_files/Dragonbox.pdf [last checked January 2026].

[17] Jaffer A, Easy Accurate Reading and Writing of Floating-Point Numbers; 2018. https://arxiv.org/abs/1310.8121 [last checked January 2026].

[18] Clinger WD. How to Read Floating Point Numbers Accurately. SIGPLAN Not 1990 Jun;25(6):92–101.

[19] Clinger WD. How to Read Floating Point Numbers Accurately. SIGPLAN Not 2004 Apr;39(4):360–371.

[20] Lemire D. Number parsing at a gigabyte per second. Software: Practice and Experience 2021;51(8):1700–1727.

[21] Mushtak N, Lemire D. Fast number parsing without fallback. Software: Practice and Experience 2023;53(7):1467–1471.

[22] Langdale G, Lemire D. Parsing gigabytes of JSON per second. The VLDB Journal 2019;28(6):941–960.

[23] Champagne Gareau J, Lemire D, float-data: A collection of floating-point numbers; 2025. https://github.com/fastfloat/float-data [last checked January 2026].

[24] Juckett R, Printing Floating-Point Numbers; 2014. https://www.ryanjuckett.com/printing-floating-point-numbers/ [last checked January 2026].

[25] Leiserson CE, Thompson NC, Emer JS, Kuszmaul BC, Lampson BW, Sanchez D, et al. There's plenty of room at the Top: What will drive computer performance after Moore's law? Science 2020;368(6495):eaam9744.