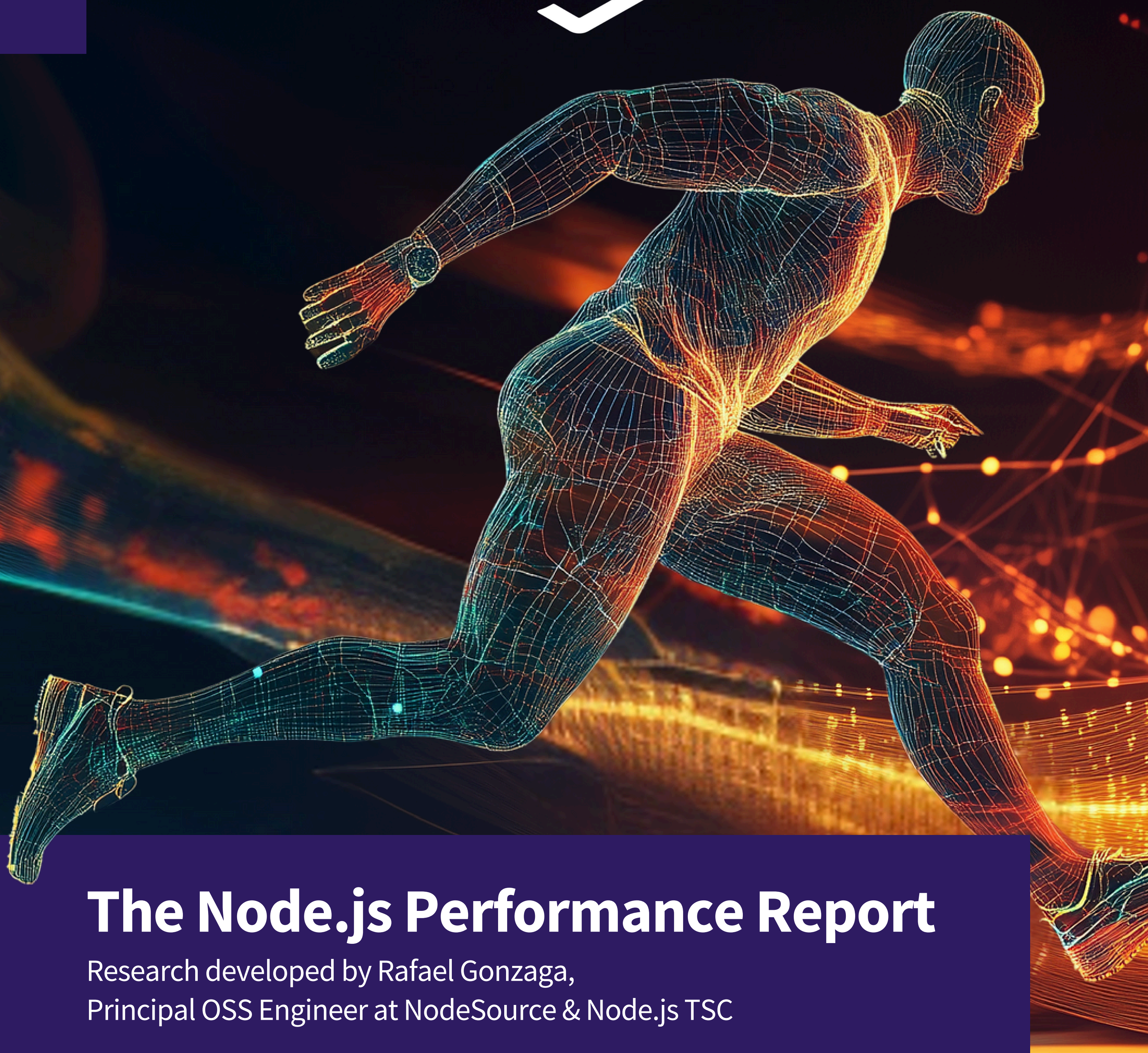
2024



The Node.js Performance Report

Research developed by Rafael Gonzaga,
Principal OSS Engineer at NodeSource & Node.js TSC

NODESOURCE

Node Reaches Version 23

The year is 2024, and Node.js has reached version 23, with two semver-majors released per year it might be difficult to keep track of all areas of Node.js. This article revisits the State of Node.js performance, with a focus on comparing versions 20 through 22. The goal is to provide a detailed analysis of how the platform has evolved over the past year.

This is a second version of "The State of Node.js Performance" series.View the 2023 version.

The report continues a commitment to rigorous benchmarking, complete with hardware details and reproducible examples. To streamline the experience, reproducible steps are collapsed at the start of each section, making it easy for readers to follow along without distraction.

This article exclusively compares Node.js versions without drawing parallels to other JavaScript runtimes. The intent is to highlight the platform's internal progress—its performance gains, regressions, and the factors driving these changes.

Benchmark Setup

This blog post will share benchmark results across different Node release lines.js using two repositories as references:

- Node.js internal benchmark suite
- nodejs-bench-operations
  - Using bench-node as the benchmark tool

Benchmarks were run on a dedicated AWS machine (C6i.xlarge) with:

- 4 vCPUs, 8GB RAM
- Ubuntu 22.04 LTS

Using the following Node.js versions:

- v20.17.0
- v22.9.0

Several key modules significantly impact Node.js performance. Any enhancements or regressions within these core components resonate across the platform. For this benchmark, we selected the following core modules:

- assert - Node.js assert operations
- buffers - Node.js Buffer operations
- diagnostics_channel - Node.js diagnostics channel module
- fs - Node.js file system
- path - Node.js path module on UNIX systems
- streams - Node.js streams creation, destroy, readable and more
- misc - Node.js startup time using child_processes and worker_threads + trace_events
- test_runner - Node.js test runner
- url - Node.js URL parser
- util - Node.js text encoder/decoder
- webstreams - Node.js WebStreams (per WHATWG spec)
- zlib - Node.js zlib API

Benchmark script and results are available at RafaelGSS/state-of-nodejs-performance-2024

How Node.js Benchmarks Are Evaluated

As mentioned in “State of the Node.js Performance 2023”, the Node.js benchmark suite by default runs each configuration 30 times to ensure accuracy, and the results undergo a statistical analysis using the Student’s t-test, which measures the confidence level of each benchmark.

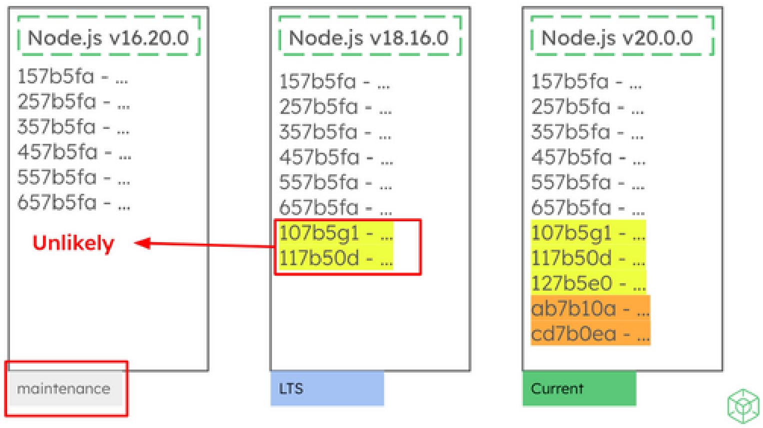
Three asterisks (***) indicate high confidence in the data as we can see in the following image:

```
                                                                                confidence
fs/readfile.js concurrent=1 len=16777216 encoding='ascii' duration=5                   ***
fs/readfile.js concurrent=1 len=16777216 encoding='utf-8' duration=5                   ***
fs/writefile-promises.js concurrent=1 size=1024 encodingType='utf' duration=5

Be aware that when doing many comparisons the risk of a false-positive result increases.
In this case, there are 10 comparisons, you can thus expect the following amount of false
  0.50 false positives, when considering a   5% risk acceptance (*, **, ***),
  0.10 false positives, when considering a   1% risk acceptance (**, ***),
  0.01 false positives, when considering a 0.1% risk acceptance (***)
```

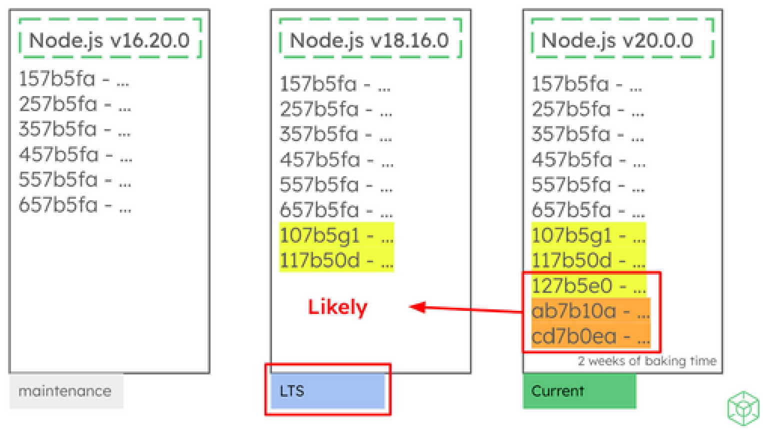
Performance Updates and Semantic Versioning

Many performance improvements arrive as semver-minor or semver-patch updates. While Node.js v22.9.0 might currently outperform Node.js v20.17.0, this can shift over time, as minor and patch-level improvements in v20 continue to be backported.

To illustrate, here's a comparison of commits across Node.js v16, v18, and v20. The latest commits, highlighted in yellow, are unlikely to land in v16, as it's in maintenance mode.



These latest commits in Node.js v20 have a high chance of being integrated into v18 since it's in Long Term Support (LTS), meaning these v20 updates can either improve or potentially degrade v18's performance.



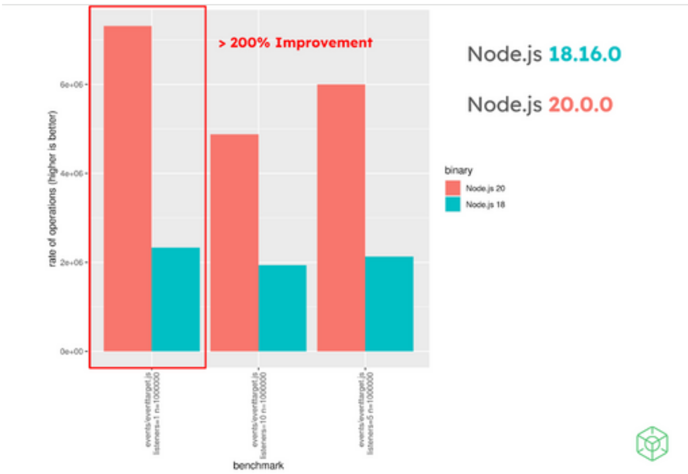
> Note: Results across release lines should be viewed with caution, except for release lines that are in End-of-Life (EOL) or Maintenance modes

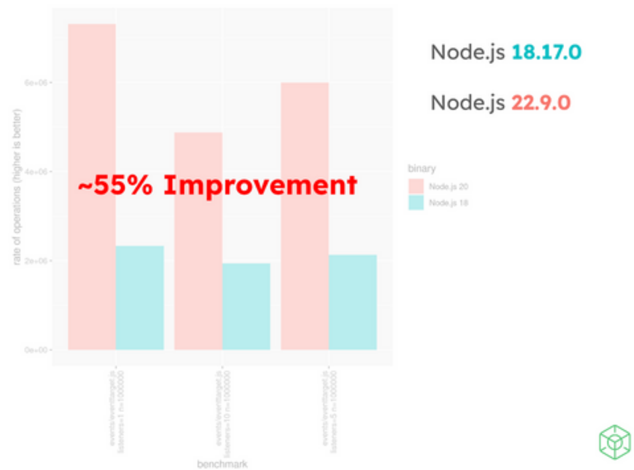
Is Newer Always Faster?

It might seem logical to expect each new Node.js version to improve performance. However, that's not always the case. For example, in ASCII encoding, Node.js v20.17.0 exhibited a ~58% regression in performance compared to v18.17.0, indicating that performance declined noticeably.

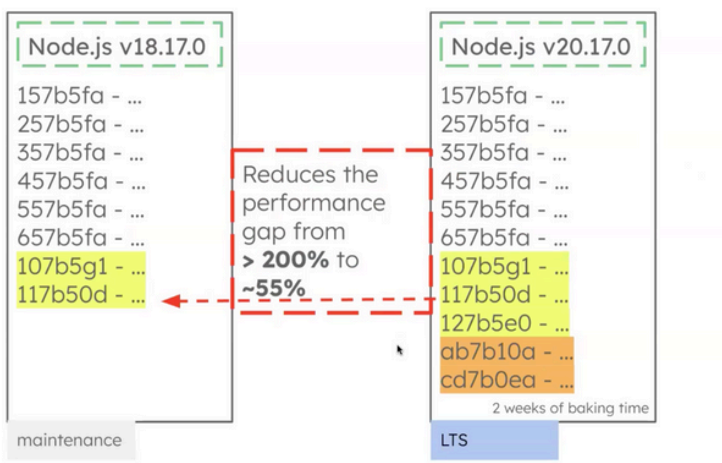
```
fs/readfile.js concurrent=1 len=1024 encoding='' duration=5              ***      1.76 %       ±0.72% ±0.96%  ±1.25%
fs/readfile.js concurrent=1 len=1024 encoding='utf-8' duration=5         ***      1.43 %       ±0.54% ±0.72%  ±0.94%
fs/readfile.js concurrent=1 len=16777216 encoding='' duration=5          ***    -58.13 %       ±4.27% ±5.71%  ±7.48%
fs/readfile.js concurrent=1 len=16777216 encoding='utf-8' duration=5       *      1.07 %       ±0.93% ±1.24%  ±1.62%
```

On the other hand, Node.js v20 demonstrated significant gains over v18 for event handling, specifically with event.target, as shown in the following benchmark. Here, v20 handles 200% more operations than v18, showing a major performance increase.





Comparing this with Node.js v22, the improvement over v18 is around 55%, not because v22 is slower, but because v18 received enhancements that closed the performance gap.



The commits in v20.17.0 effectively reduce this performance gap from 200% to ~55% in Node.js v18.17.0.

“At NASA, mission-critical code is the rule, not the exception. As we transition from a legacy environment to a modern Node-based architecture, N|Solid, along with the support of NodeSource, is proving invaluable by allowing us to scale rapidly while staying focused on our core mission.”

How to Start a Benchmarking Process

If you're new to benchmarking, this [blog post]() is a great place to begin.

1. Prepare the Environment: A golden rule for accurate benchmarking is to control your environment, as almost anything can affect results. For example, running a benchmark during a Zoom call or streaming music can introduce noise into your measurements. In one famous instance from 2004, Brendan Gregg demonstrated that even shouting near the hardware could disrupt slow disk I/O operations!

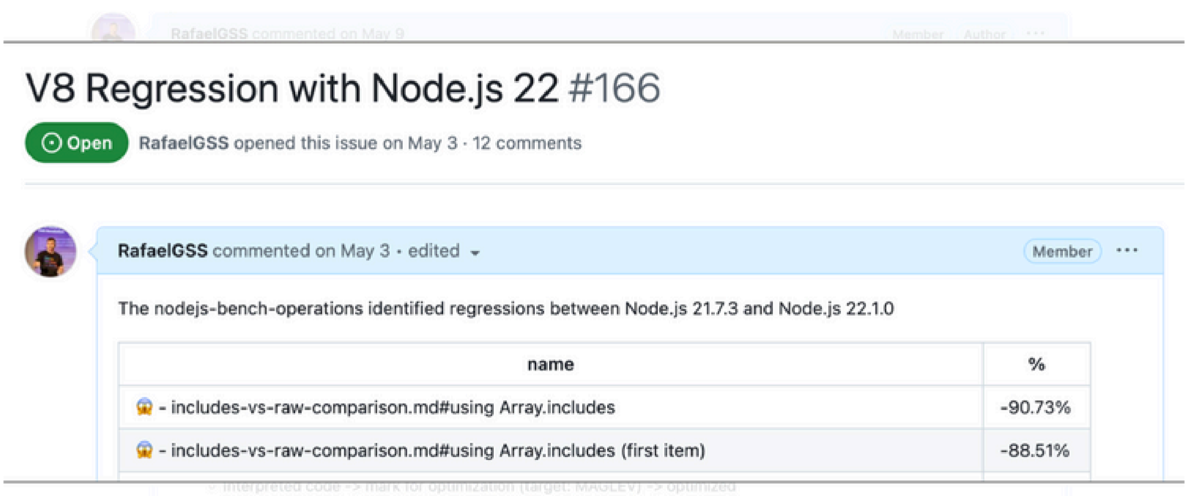
To avoid such interference, always use a dedicated machine for benchmarking. The Instant Bench Agent can help you set up an AWS dedicated machine for this purpose.

1. Isolate the Bottleneck: in order to isolate the bottlenecks, you should reduce the variability as much as you can.

Benchmark workflow:

1. Use a dedicated machine to run your benchmarks.
2. Run a benchmark before making a change.
3. Run the same benchmark after the change.
4. Compare the results.

Note: Prior to Node.js v22.9.0, Maglev, a V8 compiler, was enabled by default in the v22.x release line. This change could lead to a false-positive to regressions if you compare operations per second across different release lines. Node.js v22.9.0 has been released disabling Maglev for different reasons. Therefore, if you conduct a benchmark before Node.js v22.9.0, it may contain inaccuracies due to Maglev's influence.



V8 Regression with Node.js 22 #166

Open RafaelGSS opened this issue on May 3 · 12 comments

RafaelGSS commented on May 3 • edited Member

The nodejs-bench-operations identified regressions between Node.js 21.7.3 and Node.js 22.1.0

name	%
😱 - includes-vs-raw-comparison.md#using Array.includes	-90.73%
😱 - includes-vs-raw-comparison.md#using Array.includes (first item)	-88.51%

Open V8 Regression with Node.js 22 #166
RafaelGSS opened this issue on May 3 · 12 comments

RafaelGSS commented on May 9 — Member — Author

So, I have investigated it a bit and it's unlikely to contain a regression, but a different benchmark approach is required.

Before maglev, the benchmarks were optimized directly into TURBOFAN during the benchmark clock. Assume the following analogy:

1. benchmark.clock.start()
2. benchmark.start() -> running the bench function several times and
3. during this process V8 marks the code to optimize and optimize it in the next cycle
   - Interpreted code -> mark for optimization (target: TURBOFAN) -> optimized
4. benchmark.clock.end()

Note that, the samples were collected including portions of Interpreted code measurements + Turbofan code measurements.

Now, with MAGLEV in the middle, a few more operations are executed **during the benchmark clock** measurement.

1. benchmark.clock.start()
2. benchmark.start() -> running the bench function several times and
3. during this process V8 marks the code to optimize and optimize it in the next cycle
   - Interpreted code -> mark for optimization (target: MAGLEV) -> optimized
4. When V8 identifies the code `hot & stable` it marks the code to optimize targeting TURBOFAN
   - Maglev code -> mark for optimization (target: MAGLEV) -> optimized
5. benchmark.clock.end()

Handle JS Micro Benchmarks with Care

Although many micro-benchmarks are created and spread over the network, micro-benchmarks in JavaScript most of the time (if not all) won't represent reality and are wrong in rare scenarios. This article won't expand on why JS Micro-Benchmarks are complex to write and evaluate, but the important take is to read all these values carefully (including the ones shared in this article).

Suggestions for reading are:

- The truth about traditional JavaScript benchmarks
- Benchmarking JavaScript GOTO 2015

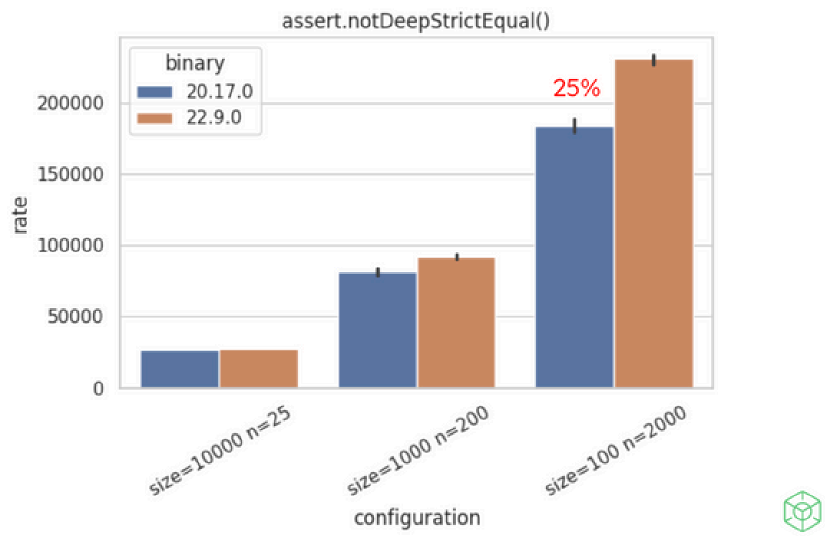
Node.js Internal Benchmark

This section shares results obtained from running the Node.js internal benchmark suite. Although Node.js contains many modules and thousands of APIs, this article will only share APIs that had a considerable performance impact during the benchmark. Therefore, if your favorite API doesn't appear on this report, assume that there's no performance change from v22.9.0 to v20.17.0.

[The Node.js Internal Benchmark](), contains code and data used to measure performance of different Node.js implementations and different ways of writing JS run by the built-in JS engine.

Assert

The **node:assert** module is widely used with test_runner and other test frameworks so making it fast will make any test suite faster. **`assert.notDeepStrictEqual`** is now 25% faster in Node.js v22 (on small-size objects)



`DeepEqual` + `Buffers` – Improved by about 20%.

```
assert/deepequal-buffer.js method='deepEqual' arrayBuffer=0 strict=0 len=100 n=20000      ***     14.15 %      ±5.59% ±7.51% ±9.91%
assert/deepequal-buffer.js method='deepEqual' arrayBuffer=0 strict=0 len=1000 n=20000     ***     18.97 %      ±2.51% ±3.38% ±4.47%
assert/deepequal-buffer.js method='deepEqual' arrayBuffer=0 strict=1 len=100 n=20000      ***     13.87 %      ±3.10% ±4.14% ±5.42%
assert/deepequal-buffer.js method='deepEqual' arrayBuffer=0 strict=1 len=1000 n=20000     ***     21.86 %      ±3.00% ±4.02% ±5.28%
assert/deepequal-buffer.js method='deepEqual' arrayBuffer=1 strict=0 len=100 n=20000      ***     10.89 %      ±2.19% ±2.92% ±3.80%
assert/deepequal-buffer.js method='deepEqual' arrayBuffer=1 strict=0 len=1000 n=20000     ***     14.72 %      ±2.19% ±2.92% ±3.80%
assert/deepequal-buffer.js method='deepEqual' arrayBuffer=1 strict=1 len=100 n=20000      ***      9.62 %      ±1.43% ±1.91% ±2.50%
assert/deepequal-buffer.js method='deepEqual' arrayBuffer=1 strict=1 len=1000 n=20000     ***     13.60 %      ±1.34% ±1.78% ±2.32%
assert/deepequal-buffer.js method='notDeepEqual' arrayBuffer=0 strict=1 len=100 n=20000   ***     22.65 %      ±1.28% ±1.70% ±2.22%
assert/deepequal-buffer.js method='notDeepEqual' arrayBuffer=0 strict=1 len=1000 n=20000  ***     24.87 %      ±2.73% ±3.63% ±4.73%
assert/deepequal-buffer.js method='notDeepEqual' arrayBuffer=1 strict=1 len=100 n=20000   ***      8.93 %      ±3.28% ±4.38% ±5.73%
assert/deepequal-buffer.js method='notDeepEqual' arrayBuffer=1 strict=1 len=1000 n=20000  ***     15.15 %      ±3.39% ±4.51% ±5.88%
```

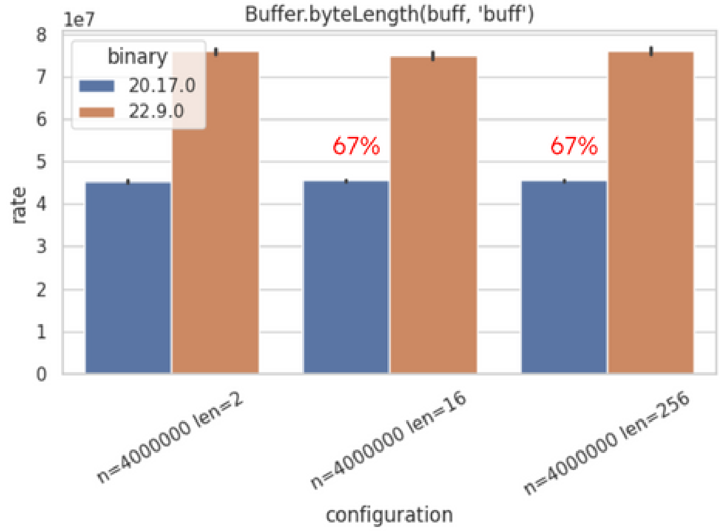
`strictEqual` – Shows a 7% slowdown based on a reliable sample size (n=200K).

```
assert/strictequal.js method='notStrictEqual' type='number' n=200000   ***    -7.65 %       ±2.48% ±3.30% ±4.30%
assert/strictequal.js method='notStrictEqual' type='number' n=25       ***   -44.37 %       ±2.84% ±3.80% ±5.01%
assert/strictequal.js method='notStrictEqual' type='object' n=200000   ***    -7.85 %       ±2.64% ±3.51% ±4.57%
assert/strictequal.js method='notStrictEqual' type='object' n=25       ***   -41.96 %       ±3.72% ±4.98% ±6.55%
assert/strictequal.js method='notStrictEqual' type='string' n=200000   ***    -6.63 %       ±2.62% ±3.48% ±4.54%
assert/strictequal.js method='notStrictEqual' type='string' n=25       ***   -44.90 %       ±2.92% ±3.90% ±5.11%

assert/strictequal.js method='strictEqual' type='number' n=200000      ***    -7.65 %       ±2.87% ±3.82% ±4.98%
assert/strictequal.js method='strictEqual' type='number' n=25          ***   -44.68 %       ±2.71% ±3.62% ±4.73%
assert/strictequal.js method='strictEqual' type='object' n=200000      ***    -7.57 %       ±2.47% ±3.29% ±4.28%
assert/strictequal.js method='strictEqual' type='object' n=25          ***   -45.20 %       ±1.80% ±2.40% ±3.13%
assert/strictequal.js method='strictEqual' type='string' n=200000      ***    -7.38 %       ±2.44% ±3.25% ±4.25%
assert/strictequal.js method='strictEqual' type='string' n=25          ***   -42.90 %       ±2.69% ±3.62% ±4.80%
```

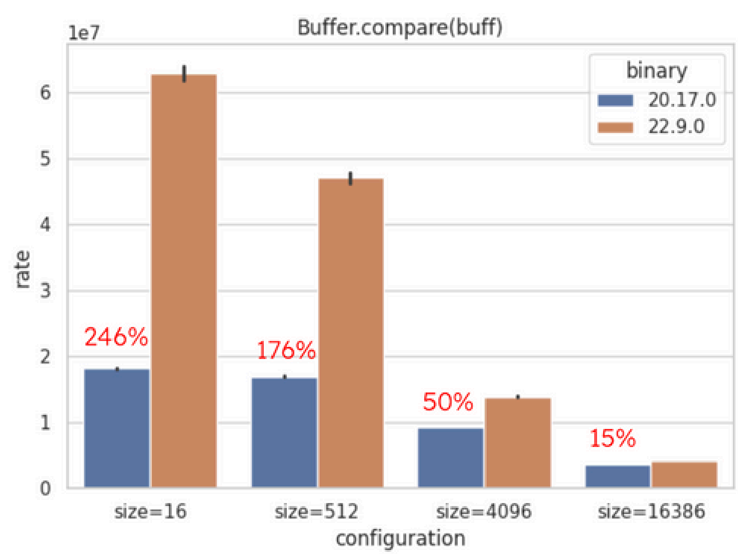
Buffers

Node.js Buffers have become significantly faster in all its APIs —except when handling base64 data.

`Buffer.byteLength` – Shows a 67% of performance improvement when compared to v20.17.0.



For **`buffer.compare(buff)`** specifically, performance has increased by over 200%, marking a substantial improvement.



Company's that trust NodeSource

The following Buffer operations are all faster:

- `` `Buffer.concat()` `` - 9% up to 33%! Combines multiple Buffers into a single Buffer efficiently.
- `` `Buffer.copy()` `` - When copying buffers using Buffer.copy(buff, 0, buffLen) 95% of improvement was identified.
- `` `Buffer.equals()` `` - Checks if two Buffers have identical byte content. Some results reach 150% improvement (see the image).

```
buffers/buffer-equals.js n=1000000 difflen='false' size=0       ***     37.91 %       ±1.98%  ±2.65%  ±3.50%
buffers/buffer-equals.js n=1000000 difflen='false' size=16386   ***     17.73 %       ±0.53%  ±0.71%  ±0.94%
buffers/buffer-equals.js n=1000000 difflen='false' size=512     ***    149.64 %       ±4.29%  ±5.78%  ±7.66%
buffers/buffer-equals.js n=1000000 difflen='true' size=0        ***     40.90 %       ±1.56%  ±2.09%  ±2.75%
buffers/buffer-equals.js n=1000000 difflen='true' size=16386    ***     42.35 %       ±1.43%  ±1.91%  ±2.51%
buffers/buffer-equals.js n=1000000 difflen='true' size=512      ***     42.12 %       ±1.58%  ±2.12%  ±2.78%
```

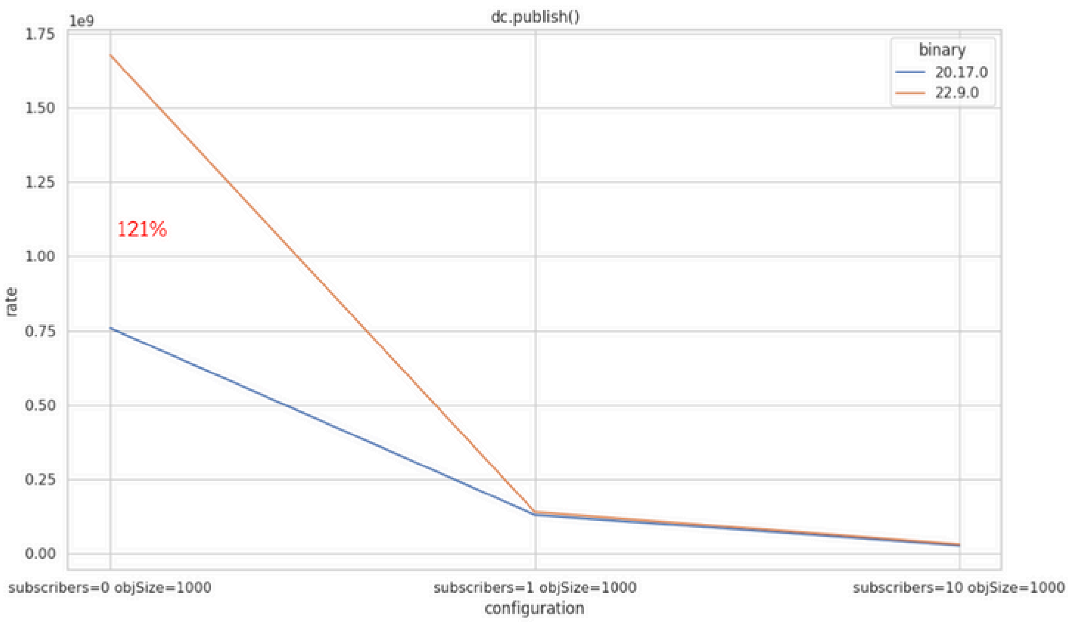
- `` `Buffer.read*(0, byteLength)` `` - From Buffer.readIntBE() to Buffer.readUIntLE() performance has been significantly boosted, and results cross the 100% barrier.
- `` `Buffer.slice()` `` - On .slice() a performance improvement of 90% has been identified on Node.js v22.9.0.
- `` `Buffer.write(X, byteLength)` `` - On .write() also received a significant boost, from 5% when dealing with BigInt64BE to 138% when dealing with FloatBE.

In general, the `` `node:buffers` `` module performs remarkably well, though Buffer.isUTF8 and Buffer.isASCII() saw slight regressions.

```
buffers/buffer-isascii.js input='hello world' length='long' n=20000000       ***    -14.84 %       ±0.83%  ±1.10%  ±1.43%
buffers/buffer-isascii.js input='hello world' length='short' n=20000000      ***    -14.39 %       ±0.64%  ±0.85%  ±1.11%

buffers/buffer-isutf8.js input='∀x∈ℝ: ⌈x⌉ = −⌊−x⌋' length='long' n=20000000  ***     -1.33 %       ±0.19%  ±0.26%  ±0.33%
buffers/buffer-isutf8.js input='regular string' length='long' n=20000000     ***    -38.85 %       ±0.45%  ±0.60%  ±0.80%
buffers/buffer-isutf8.js input='regular string' length='short' n=20000000    ***     -5.23 %       ±1.07%  ±1.42%  ±1.85%
```

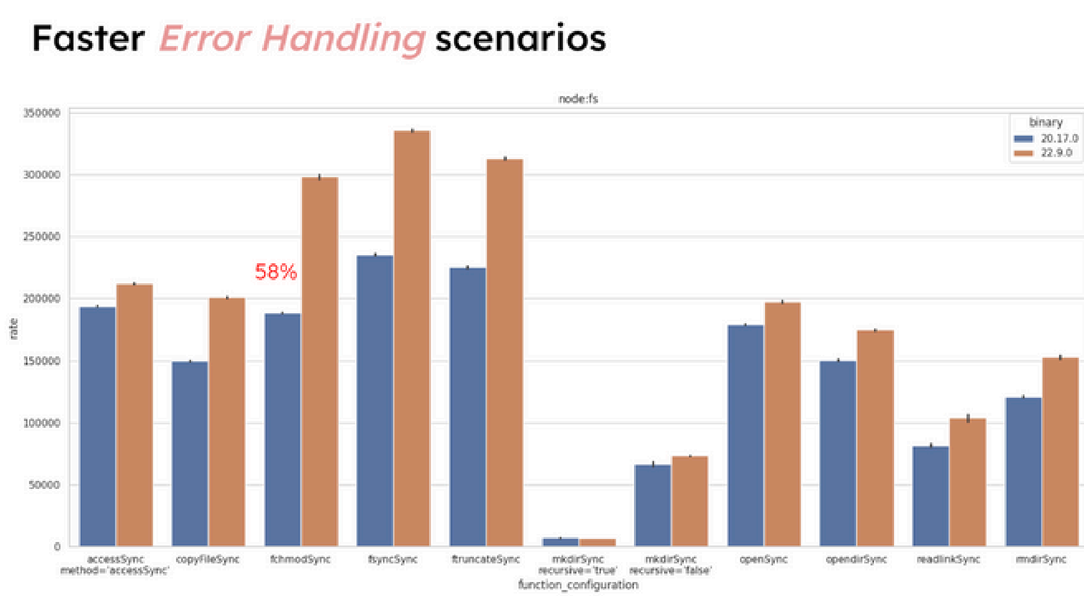
Diagnostics Channel

Diagnostics channels are now significantly faster when there are no subscribers—up to 120% faster, as shown in the graph below. This improvement is especially relevant for users who rely on diagnostic channels indirectly. At NodeSource, we leverage diagnostic channels in our APM, and this performance boost ensures that systems without subscribers remain unaffected.

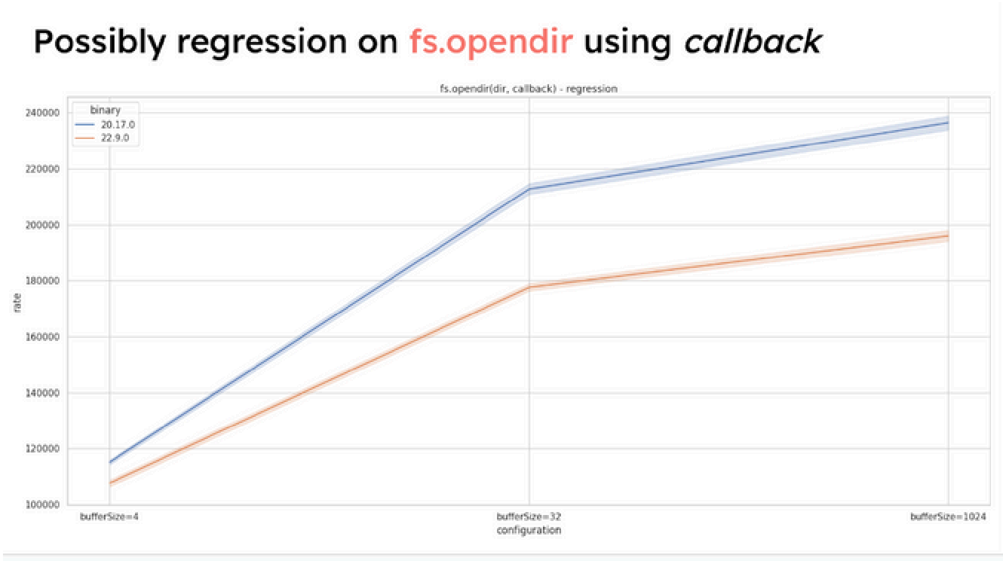


Node.js File System

Node.js has improved its handling of error scenarios within the node:fs module. For instance, attempts to open non-existent files fail ~58% faster. While this doesn't change app functionality, it speeds up error detection for processes that routinely check file availability or integrity.



A potential regression was noted for **fs.opendir** when using callbacks, so this function may perform differently in certain callback-driven cases.



Faster node:path

Node.js' node:path module has also seen performance gains. This benchmark only includes POSIX environments (Linux and macOS). Improvements are:

path.basename() – Up to 10% faster.

```
path/basename-posix.js n=100000 pathext=''                              ***    -6.67 %     ±2.94% ±3.93% ±5.16%
path/basename-posix.js n=100000 pathext='/'                                    -1.18 %     ±2.48% ±3.32% ±4.36%
path/basename-posix.js n=100000 pathext='/foo'                                  2.10 %     ±3.16% ±4.20% ±5.47%
path/basename-posix.js n=100000 pathext='/foo/.bar.baz'                 ***     8.87 %     ±2.79% ±3.72% ±4.85%
path/basename-posix.js n=100000 pathext='/foo/.bar.baz|.baz'            ***    11.84 %     ±2.99% ±3.97% ±5.18%
path/basename-posix.js n=100000 pathext='/foo/bar/baz/asdf/quux.html'   ***     7.07 %     ±2.75% ±3.67% ±4.80%
path/basename-posix.js n=100000 pathext='/foo/bar/baz/asdf/quux.html|.html' *** 7.67 %     ±3.55% ±4.73% ±6.17%
path/basename-posix.js n=100000 pathext='foo'                                   0.71 %     ±3.1% ±4.14% ±5.40%
path/basename-posix.js n=100000 pathext='foo/bar.'                      ***     5.58 %     ±2.43% ±3.24% ±4.22%
path/basename-posix.js n=100000 pathext='foo/bar.|.'                    ***    11.26 %     ±3.48% ±4.63% ±6.02%
```

path.isAbsolute() – About 38% faster.

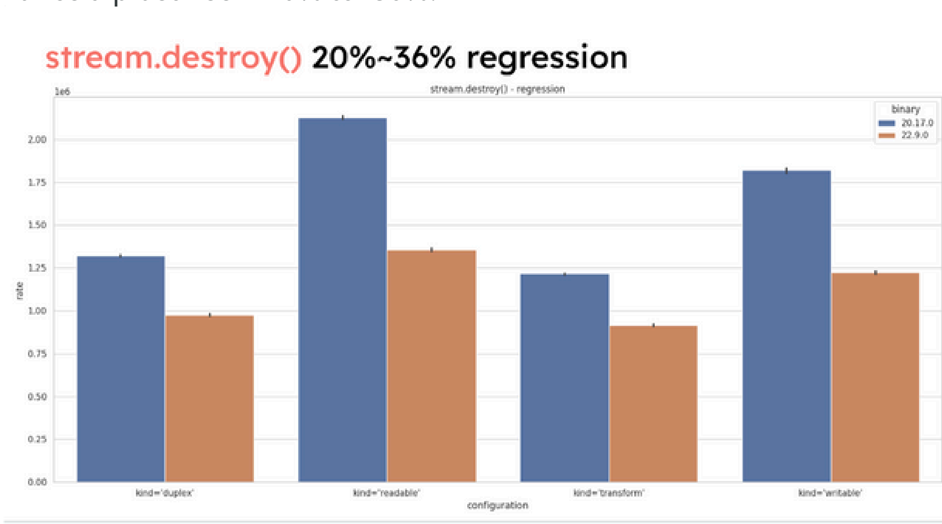
```
path/isAbsolute-posix.js n=100000 path='.'            **     4.99 %     ±3.26% ±4.34% ±5.65%
path/isAbsolute-posix.js n=100000 path=''             ***    5.93 %     ±2.63% ±3.50% ±4.56%
path/isAbsolute-posix.js n=100000 path='/baz/..'      ***    5.59 %     ±2.73% ±3.63% ±4.74%
path/isAbsolute-posix.js n=100000 path='/foo/bar'     ***   38.10 %     ±3.13% ±4.16% ±5.41%
path/isAbsolute-posix.js n=100000 path='bar/baz'        *    3.60 %     ±2.90% ±3.86% ±5.03%
```

path.resolve() – A minor ~9% boost in some cases.

```
path/resolve-posix.js n=100000 paths=''                                 ***    5.18 %     ±0.98% ±1.30% ±1.70%
path/resolve-posix.js n=100000 paths='|'                                ***    3.63 %     ±0.75% ±1.00% ±1.30%
path/resolve-posix.js n=100000 paths='a/b/c/|../../..'                  ***    8.94 %     ±0.76% ±1.01% ±1.32%
path/resolve-posix.js n=100000 paths='foo/bar|/tmp/file/|..|a/../subfile' ***  6.60 %     ±0.72% ±0.96% ±1.25%
```

Regressions in node:streams

A notable regression has been detected in **node:streams**, specifically when destroying streams, with a performance dip between -20% to -36%.



The Node.js benchmark test runner shows an approximate 10% performance boost in test creation

```
test_runner/suite-tests.js concurrency='no' testsPerSuite=10 numberOfSuites=10           ***      5.83 %       ±1.65% ±2.23%  ±2.97%
test_runner/suite-tests.js concurrency='no' testsPerSuite=10 numberOfSuites=100          ***     10.33 %       ±1.79% ±2.44%  ±3.30%
test_runner/suite-tests.js concurrency='no' testsPerSuite=100 numberOfSuites=10          ***     10.38 %       ±1.81% ±2.44%  ±3.25%
test_runner/suite-tests.js concurrency='no' testsPerSuite=100 numberOfSuites=100         ***     17.71 %       ±1.72% ±2.32%  ±3.10%
test_runner/suite-tests.js concurrency='no' testsPerSuite=1000 numberOfSuites=10         ***     17.50 %       ±1.95% ±2.63%  ±3.50%
test_runner/suite-tests.js concurrency='no' testsPerSuite=1000 numberOfSuites=100        ***     19.81 %       ±3.30% ±4.46%  ±5.95%

test_runner/suite-tests.js concurrency='yes' testsPerSuite=10 numberOfSuites=10          ***      7.23 %       ±1.32% ±1.78%  ±2.37%
test_runner/suite-tests.js concurrency='yes' testsPerSuite=10 numberOfSuites=100         ***      7.53 %       ±2.07% ±2.82%  ±3.79%
test_runner/suite-tests.js concurrency='yes' testsPerSuite=100 numberOfSuites=10         ***      7.20 %       ±1.98% ±2.68%  ±3.56%
test_runner/suite-tests.js concurrency='yes' testsPerSuite=100 numberOfSuites=100        ***     17.02 %       ±1.52% ±2.05%  ±2.73%
test_runner/suite-tests.js concurrency='yes' testsPerSuite=1000 numberOfSuites=10        ***     23.50 %       ±2.90% ±3.92%  ±5.23%
test_runner/suite-tests.js concurrency='yes' testsPerSuite=1000 numberOfSuites=100       ***     15.08 %       ±1.24% ±1.68%  ±2.24%
```

and concurrent tests benefit from an additional 12% increase in speed

```
test_runner/global-concurrent-tests.js type='async' n=100        ***    12.15 %      ±2.30% ±3.11%  ±4.18%
test_runner/global-concurrent-tests.js type='async' n=1000       ***    12.80 %      ±2.48% ±3.39%  ±4.58%
test_runner/global-concurrent-tests.js type='async' n=10000      ***    18.73 %      ±1.82% ±2.45%  ±3.26%
test_runner/global-concurrent-tests.js type='sync' n=100         ***    11.34 %      ±2.17% ±2.94%  ±3.94%
test_runner/global-concurrent-tests.js type='sync' n=1000        ***    11.44 %      ±2.38% ±3.23%  ±4.33%
test_runner/global-concurrent-tests.js type='sync' n=10000       ***    20.47 %      ±1.51% ±2.03%  ±2.70%
```

Node.js URL parser

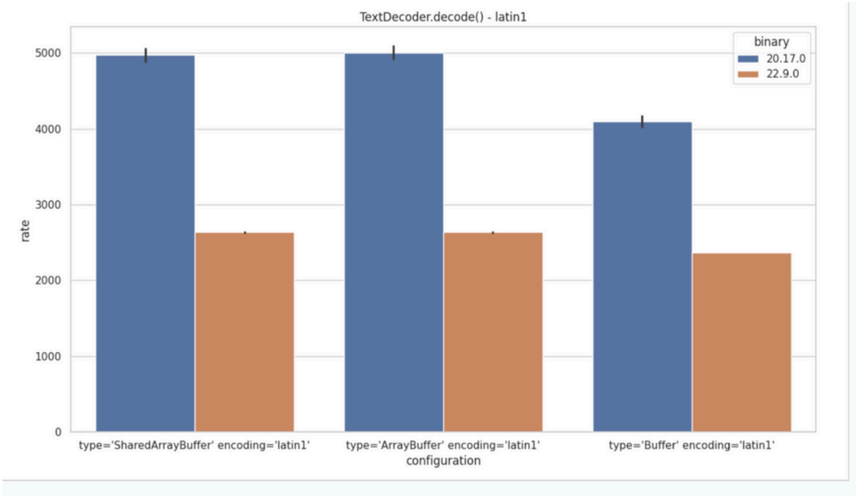
Node.js' URL parser has become even faster. URL.resolve has been optimized, bringing significant performance improvements.

```
url/url-resolve.js n=100000 path='foo/bar' href='auth'           ***     13.87 %       ±0.50% ±0.66%  ±0.86%
url/url-resolve.js n=100000 path='foo/bar' href='dot'            ***     13.97 %       ±0.62% ±0.82%  ±1.07%
url/url-resolve.js n=100000 path='foo/bar' href='file'           ***      8.59 %       ±0.69% ±0.92%  ±1.19%
url/url-resolve.js n=100000 path='foo/bar' href='idn'            ***     14.10 %       ±0.60% ±0.80%  ±1.04%
url/url-resolve.js n=100000 path='foo/bar' href='javascript'     ***      5.09 %       ±0.70% ±0.93%  ±1.21%
url/url-resolve.js n=100000 path='foo/bar' href='long'           ***      2.56 %       ±0.38% ±0.51%  ±0.66%
url/url-resolve.js n=100000 path='foo/bar' href='noscheme'       ***     12.91 %       ±0.51% ±0.68%  ±0.88%
url/url-resolve.js n=100000 path='foo/bar' href='percent'        ***      9.96 %       ±0.65% ±0.86%  ±1.13%
url/url-resolve.js n=100000 path='foo/bar' href='short'          ***     14.94 %       ±0.53% ±0.70%  ±0.92%
url/url-resolve.js n=100000 path='foo/bar' href='ws'             ***     12.64 %       ±0.52% ±0.69%  ±0.90%
url/url-resolve.js n=100000 path='sibling' href='auth'           ***     10.72 %       ±0.46% ±0.61%  ±0.80%
url/url-resolve.js n=100000 path='sibling' href='dot'            ***     13.56 %       ±0.41% ±0.55%  ±0.72%
url/url-resolve.js n=100000 path='sibling' href='file'           ***      8.87 %       ±0.56% ±0.74%  ±0.97%
url/url-resolve.js n=100000 path='sibling' href='idn'            ***     10.39 %       ±0.93% ±1.24%  ±1.61%
url/url-resolve.js n=100000 path='sibling' href='javascript'     ***      8.28 %       ±0.66% ±0.88%  ±1.15%
url/url-resolve.js n=100000 path='sibling' href='long'           ***      4.25 %       ±0.43% ±0.57%  ±0.74%
url/url-resolve.js n=100000 path='sibling' href='noscheme'       ***     10.75 %       ±0.46% ±0.62%  ±0.81%
url/url-resolve.js n=100000 path='sibling' href='percent'        ***      7.15 %       ±0.55% ±0.74%  ±0.96%
url/url-resolve.js n=100000 path='sibling' href='short'          ***     12.54 %       ±0.52% ±0.69%  ±0.91%
url/url-resolve.js n=100000 path='sibling' href='ws'             ***     10.14 %       ±0.55% ±0.74%  ±0.96%
```

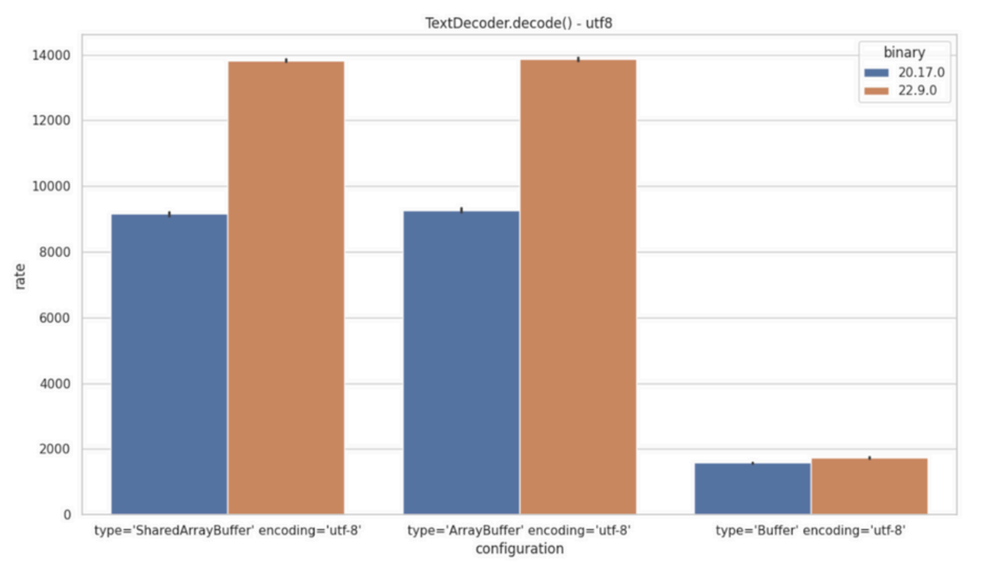
```
url/url-resolve.js n=100000 path='up' href='auth'                ***     11.68 %       ±0.53% ±0.71%  ±0.92%
url/url-resolve.js n=100000 path='up' href='dot'                 ***     14.01 %       ±0.55% ±0.73%  ±0.96%
url/url-resolve.js n=100000 path='up' href='file'                ***      8.79 %       ±0.52% ±0.70%  ±0.91%
url/url-resolve.js n=100000 path='up' href='idn'                 ***     11.99 %       ±0.44% ±0.59%  ±0.77%
url/url-resolve.js n=100000 path='up' href='javascript'          ***      5.18 %       ±0.54% ±0.72%  ±0.94%
url/url-resolve.js n=100000 path='up' href='long'                ***      4.46 %       ±0.45% ±0.60%  ±0.78%
url/url-resolve.js n=100000 path='up' href='noscheme'            ***     10.72 %       ±0.86% ±1.15%  ±1.51%
url/url-resolve.js n=100000 path='up' href='percent'             ***      6.42 %       ±0.72% ±0.96%  ±1.25%
url/url-resolve.js n=100000 path='up' href='short'               ***     15.09 %       ±0.42% ±0.56%  ±0.74%
url/url-resolve.js n=100000 path='up' href='ws'                  ***     10.97 %       ±0.46% ±0.61%  ±0.80%
url/url-resolve.js n=100000 path='withscheme' href='auth'        ***      5.24 %       ±0.58% ±0.77%  ±1.01%
url/url-resolve.js n=100000 path='withscheme' href='dot'         ***      7.24 %       ±0.84% ±1.12%  ±1.45%
url/url-resolve.js n=100000 path='withscheme' href='file'        ***     12.62 %       ±0.76% ±1.02%  ±1.33%
url/url-resolve.js n=100000 path='withscheme' href='idn'         ***     11.59 %       ±0.59% ±0.78%  ±1.02%
url/url-resolve.js n=100000 path='withscheme' href='javascript'  ***     14.21 %       ±0.67% ±0.89%  ±1.17%
url/url-resolve.js n=100000 path='withscheme' href='long'        ***      4.23 %       ±0.36% ±0.48%  ±0.62%
url/url-resolve.js n=100000 path='withscheme' href='noscheme'    ***     12.31 %       ±0.59% ±0.79%  ±1.03%
url/url-resolve.js n=100000 path='withscheme' href='percent'     ***     12.79 %       ±0.79% ±1.05%  ±1.38%
url/url-resolve.js n=100000 path='withscheme' href='short'       ***      8.86 %       ±1.05% ±1.41%  ±1.84%
```

TextDecode Regression

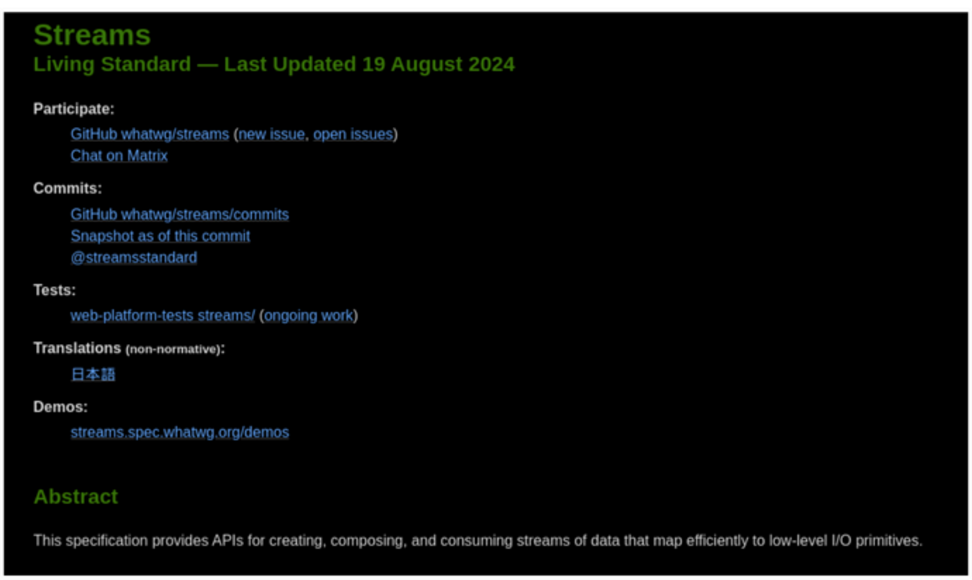
A major regression was noted in TextDecoder.decode(), specifically for Latin-1 encoding, with a nearly 100% slowdown. ISO8859-3 is similarly affected.



However, UTF-8 decoding shows a 50% speed increase, providing a marked improvement in certain use cases:



WebStreams performance has seen substantial gains, with improvements of over 100% across various stream types, including **Readable**, **Writable**, **Transform**, and **Duplex**. This is particularly impactful for `fetch`, a widely used HTTP request tool, as it relies on WebStreams by specification.



Streams

Living Standard — Last Updated 19 August 2024

Participate:
GitHub whatwg/streams (new issue, open issues)
Chat on Matrix

Commits:
GitHub whatwg/streams/commits
Snapshot as of this commit
@streamsstandard

Tests:
web-platform-tests streams/ (ongoing work)

Translations (non-normative):
日本語

Demos:
streams.spec.whatwg.org/demos

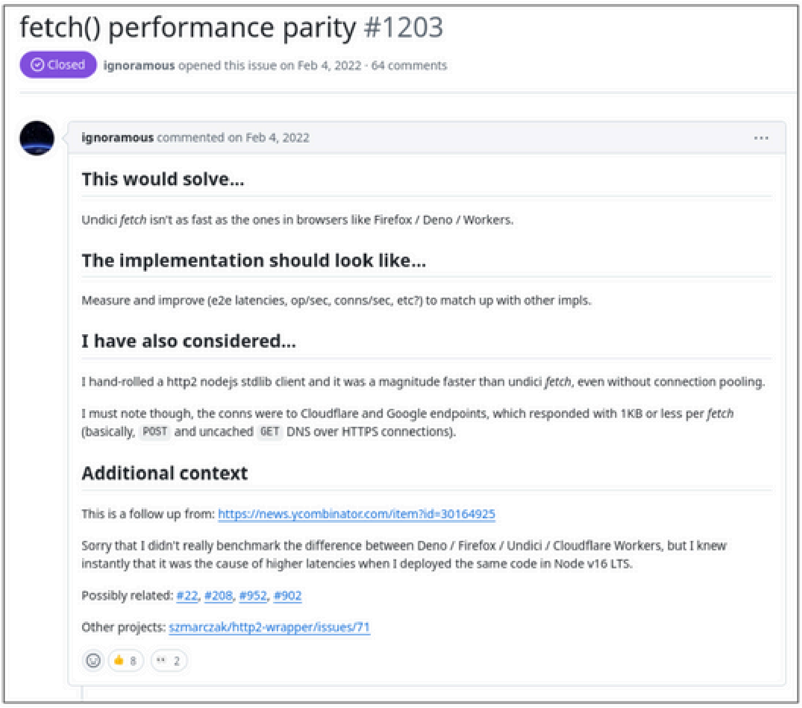
Abstract

This specification provides APIs for creating, composing, and consuming streams of data that map efficiently to low-level I/O primitives.

Fetch and WebStreams

The Fetch API is a web standard for making HTTP requests, and it requires the use of WebStreams as part of its specification. Consequently, when WebStreams are optimized, Fetch benefits directly, which is why improvements to WebStreams are so impactful.

In 2022, [there was an identified issue](#) with the Undici library's fetch implementation (used by Node.js), where fetch was notably slow compared to alternatives. [Rafael Gonzaga provided an analysis explaining that WebStreams' inherent slowness was the main reason for fetch's limited performance](#), as fetch relies on WebStreams by design.



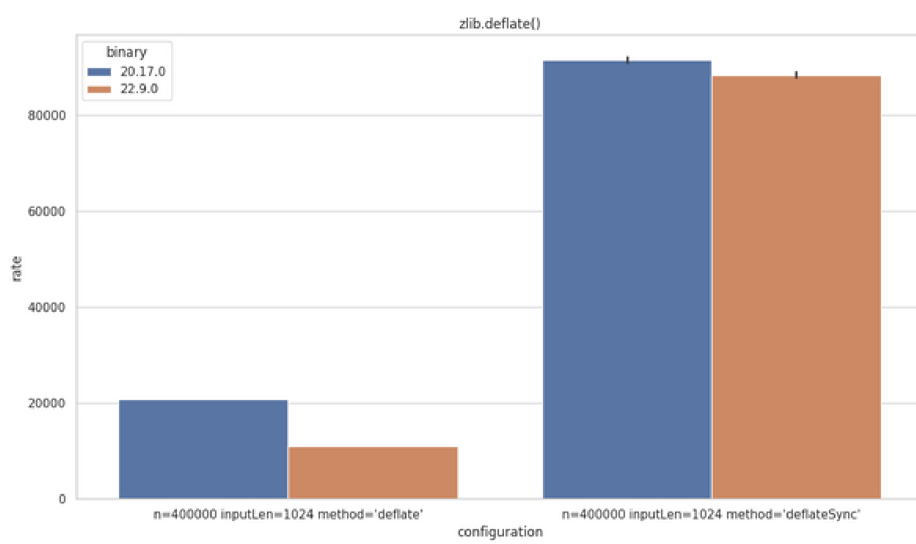
With the release of Node.js v22, improvements to WebStreams have helped Fetch jump from 2,246 requests per second to 2,689 requests per second, marking a good enhancement for an API known to be performance-sensitive.

```
→  benchmarks (main) node -v
v20.17.0

→  benchmarks (main) PORT=3001 node benchmark.js

│ (index) │ Tests                  │ Samples │ Result              │ Tolerance  │ Difference with slowest │
├─────────┼────────────────────────┼─────────┼─────────────────────┼────────────┼─────────────────────────┤
│ 0       │ 'node-fetch'           │ 1       │ '2246.81 req/sec'   │ '± 0.00 %' │ '-'                     │
```

```
→  benchmarks (main) node -v
v22.9.0

→  benchmarks (main) PORT=3001 node --no-warnings benchmark.js


│ (index) │ Tests                  │ Samples │ Result              │ Tolerance  │ Difference with slowest │
├─────────┼────────────────────────┼─────────┼─────────────────────┼────────────┼─────────────────────────┤
│ 0       │ 'node-fetch'           │ 1       │ '2689.95 req/sec'   │ '± 0.00 %' │ '-'                     │

```

Zlib Regression

The zlib module in Node.js provides compression and decompression utilities using the Gzip and Deflate/Inflate algorithms. A regression has been identified on zlib.deflate() with a higher impact on the asynchronous API (zlib.deflate()) over the synchronous call (zlib.deflateSync())



Avoiding Dead-Code elimination on Micro-Benchmarks using bench-node

As said in "Handle JS Micro-Benchmarks carefully" it's very common to see benchmarks being written in a way that after a full V8 optimization, the code will be removed as the V8 JIT compiler will flag the measured piece of code as prone to "Dead-code elimination", so you will end-up measuring a noop().

```
function benchmark() {
  const r = 1234
  const start = Date.now();
- for (let i = 0; i < N; i++) {
- }      Measuring noop()
  return (Date.now() - start) / N;
}
```

That's why bench-node has been created. This benchmark library by default tells V8 to never optimize your code

```
  beforeClockTemplate(_varNames) {
    let code = '';

    code += `
function DoNotOptimize(x) {}
// Prevent DoNotOptimize from optimizing or being inlined.
%NeverOptimizeFunction(DoNotOptimize);
`
    return [code, 'DoNotOptimize'];
  }

  toString() {
    return 'V8NeverOptimizePlugin';
  }
```

This article won't dive into the internals of `bench-node`. Instead, the next section will showcase benchmark results generated using this library. While `bench-node` excels at providing a reliable and consistent way to compare simple operations, it's important to note that these results might not reflect real-world scenarios. In production, V8 optimizations can significantly influence code performance, making it challenging to perfectly replicate runtime behavior.

nodejs-bench-operations

If you have read the "State of Node.js Performance 2023" you might know the nodejs-bench-operations repository. TL;DR It's a repository to compare simple Node.js/JS operations across multiple versions of Node.js.

RESULTS-v18.md	chore(unix-time.mjs): update benchmark results	3 days ago
RESULTS-v20.md	chore(unix-time.mjs): update benchmark results	3 days ago
RESULTS-v21.md	chore(unix-time.mjs): update benchmark results	last week
RESULTS-v22.md	chore(unix-time.mjs): update benchmark results	3 days ago

This repository also contains a *regression checker*, a GitHub action that compares results between different release lines and alerts in case of regressions/improvements greater than the 10% threshold.

```
Checking regression between v20_17_0 and v22_9_0

😱 - possible-undefined-function.md#Using if to check function existence | -21.30%
😱 - possible-undefined-function.md#Using ? operator to avoid rejection | -25.77%
😁 - string-startsWith.md#(short string) (true) String#slice and strict comparison | 78.13%
😁 - string-startsWith.md#(long string) (true) String#startsWith | 28.69%
😁 - string-startsWith.md#(short string) (false) String#startsWith | 46.22%
😁 - blob.md#new Blob (128) | 21.33%
😁 - blob.md#slice (0, 64) | 184.73%
😁 - blob.md#slice (0, 512) | 21.63%
😁 - stream-writable.md#streams.Writable writing 1e3 * "some data" | 78.90%
😁 - stream-writable.md#streams.web.WritableStream writing 1e3 * "some data" | 26.85%
```

Significant improvements were identified in `Blob.slice()` handling > 2.5x more than the previous benchmark result. The `Writable` benchmark seems to have improved both Streams and WebStreams (it could be related to the Buffer improvements we have seen in the nodejs internal benchmark suite).

`String.prototype.startsWith()` noticed another important performance improvement (due to the V8 update). The same applies to `String.prototype.endsWith()`

```
😁 - string-endsWith.md#(short string) (true) String#endsWith | 98.12%
😁 - string-endsWith.md#(long string) (true) String#endsWith | 31.11%
😁 - string-endsWith.md#(short string) (false) String#endsWith | 62.87%
😁 - string-endsWith.md#(long string) (false) String#endsWith | 66.46%
```

The nodejs-bench-operations also contains some curious benchmarks, for example, historically parsing big integers integers using `+` was faster than using `parseInt(x, 10)`.

Parsing Integer

name	ops/sec	samples
Using parseInt(x, 10) - small number (2 len)	132,214,453	66107241
Using parseInt(x, 10) - big number (10 len)	17,222,411	8618478
Using + - small number (2 len)	104,781,265	52390642
Using + - big number (10 len)	106,028,083	53015910

https://github.com/RafaelGSS/nodejs-bench-operations/blob/main/RESULTS-v18.md#parsing-integer

However, this is not true anymore since Node.js 20.

Parsing Integer

name	ops/sec	samples
Using parseInt(x, 10) - small number (2 len)	142,155,753	71077900
Using parseInt(x, 10) - big number (10 len)	89,211,357	44666124
Using + - small number (2 len)	99,812,366	49939813
Using + - big number (10 len)	98,944,329	49488636

Approaches that were utilized but not included in the article

Many other benchmark approaches were utilized while conducting this research:

- tinybench has been used instead of bench-node to certificate the accuracy of the nodejs-bench-operations results
- HTTP Benchmarks using wrk2 and different HTTP Frameworks (express, fastify) were also conducted, but no expressive differentiation was identified that was worth it to mention in this blog post.
- NodeSource/nodejs-package-benchmark a Node.js benchmark for common web developer workloads was also utilized. No expressive results were found.

Why do regressions exist? Doesn't the Node.js Team Measure Each PR for Regressions?

Achieving the benchmark results above required a dedicated machine to run the entire Node.js test suite, which took four days to complete. Imagine making a small code change to Node.js core—you might not immediately know if it introduces a regression until benchmarks are run. Running a full benchmark for every pull request, each taking days, would be highly resource-intensive and could significantly slow down development.

Given the scale of the Node.js project—with thousands of contributors and a vast codebase—tracking every possible regression is challenging. The team strives to balance thorough testing with practical resource constraints, ensuring critical areas are well-covered while prioritizing rapid development.

That said, we actively monitor performance and are always open to sponsorship programs that could expand our benchmarking capabilities, helping to identify regressions earlier and further improve the quality of releases.

Conclusion:

- A single benchmark run isn't enough to reliably measure performance changes.
- Performance improvements are typically released as semver-minor or semver-patch updates.
- Be cautious with micro-benchmarks in JavaScript.
- Highlights include:
  - Faster Buffers module
  - FastAPI additions for error handling in node:fs
  - Faster node:path (for UNIX systems)
  - Improved Node.js test runner
  - Optimized WebStreams and Fetch
  - Notable regressions in TextDecode and streams.destroy.

For more in-depth performance insights, see the full report on the State of Node.js Performance 2023 and follow NodeSource for updates.

> Looking for advanced or enterprise support for Node.js? NodeSource is the premier provider, trusted by organizations like Delta, Visa, and Kaiser, to ensure robust support for their Node.js platforms. Partner with us to keep your applications secure, performant, and reliable.