HW–SW Co-Design Project

json_dumps Benchmark

Overview

The json.dumps function converts Python objects into JSON formatted strings. It recursively walks dictionaries, lists, and basic types (like str, int, float, bool), applies the JSON formatting and returns a str representation whose shape can be influenced by multiple options. The json_dumps benchmark in pyperformance measures how fast Python can convert objects to JSON using json.dumps. It does this by running the conversion many times on typical nested data structures. These tests are run inside the pyperformance setup that prepares the Python interpreter, repeats the tests, and calculates performance parameters like average time and variation. Since json_dumps involves lots of computation (formatting strings and numbers, validating and encoding characters, etc.), it is generally CPU-bound.

We analyze Python's JSON serialization performance on the pyperformance json_dumps benchmark and apply a software optimization that replaces json.dumps with the orjson library implementation. Following the workflow: profile → identify bottlenecks → optimize → re-measure, we generated native flame graphs for a focused microbenchmark and collected perf stat counters for the pyperformance runs before/after the optimization.

Result: stdlib json = 62.2 ms ± 0.5 ms; orjson = 11.9 ms ± 0.1 ms

≈ 5.2× speedup (−80.9% time).

Initial Analysis

Flame graphs (microbenchmark)

We first used a deterministic microbenchmark to sample and produce the stdlib json flame graph. We later used the same microbenchmark with the same payload to produce the orjson-optimized flame graph. The helper script final_fgs.sh builds the objects to be serialized, sets the sampling parameters (like the sampling rate and duration of the program) and generates the two graphs.

Below is the stdlib json flame graph:

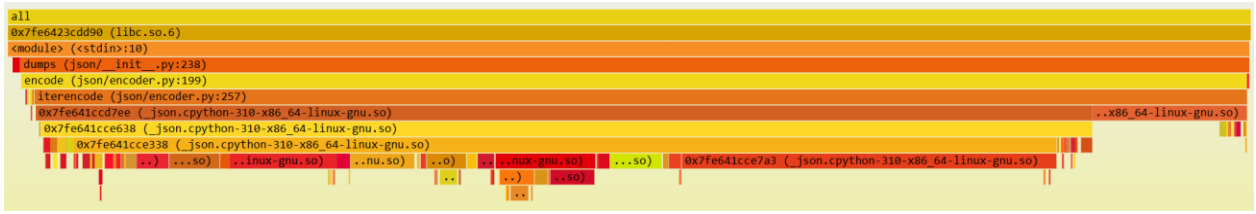


Figure 1 – Flame graph (stdlib json)

Note that there is no obvious bottleneck in the flame graph. In addition, note that the stdlib json implementation already includes default native C speedups (which are not processed line-by-line by the python interpreter). See Appendix A for the non-accelerated pure Python implementation of json dumps – it's horrendous!

In the pyperformance benchmark code, the main loop in charge of iterating and serializing each objects is:

```
def bench_json_dumps(data):
    for obj, count_it in data:
        for _ in count_it:
            json.dumps(obj)
```

The dumps function calls the JSONEncoder::encode function which calls the iterencode function, which includes many conditionals regarding the object format and the user-defined options (lots of branching) and eventually calls the corresponding fast native C function on the object, which does the actual formatting and serialization.

From the analysis of the benchmark and stdlib code and of the flame graph, we realize that the program spends lots of time in the python frames, which orjson will minimize.

Below is the orjson flame graph:

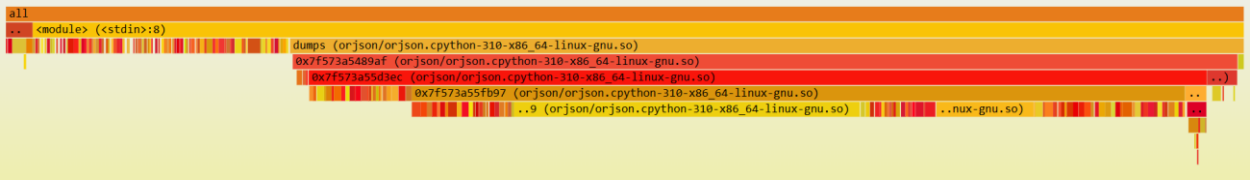


Figure 2 – Flame graph (orjson)

As desired, the Python stack is eliminated. Later in this report, we explain how we substituted the Python and C stacks for orjson's stack and how orjson accelerates the actual object serialization.

Pyperformance + perf stat

We measured hardware counters for the full pyperformance json_dumps benchmark, then repeated with the orjson patch enabled. The run stats are captured in perf_stat_baseline.txt and perf_stat_orjson.txt.

Optimization: substituting orjson for json.dumps

What is orjson

orjson is a high-performance JSON library for Python. It replaces the standard json module's dumps function with an implementation written in Rust, a compiled systems language. Because it runs in native code instead of Python, orjson avoids slow Python code and interpreter overhead. It also uses advanced techniques like efficient memory handling and SIMD (vectorized) instructions to process multiple characters at once. The result is much faster JSON serialization—often several times quicker than the standard library—while still producing correct JSON output.

Integration - how the interpreter routes calls to orjson

We use a small trick to make Python call orjson instead of the standard json.dumps. First, we create a file called sitecustomize.py and put it in a folder listed in PYTHONPATH (an environment variable that tells Python where to find extra modules). Python automatically runs this file when it starts, so inside it we import orjson and replace json.dumps with a version that uses orjson and converts its output from bytes to a string.

```
# Monkey-patch json.dumps to use orjson.dumps
PATCHDIR=$(mktemp -d)
cat > "$PATCHDIR/sitecustomize.py" <<'PY'
import json
try:
    import orjson
    json.dumps = lambda obj, *a, **kw: orjson.dumps(obj).decode("utf-8")
except Exception as e:
    import sys
    print("[sitecustomize] orjson patch failed:", e, file=sys.stderr)
PY
export PYTHONPATH="$PATCHDIR"

# perf record + report for optimized run, note the inherit flag for the
# pyperformance process to see the patch
perf record -F 999 -g -o "$OUTDIR/perf_orjson.data" -- \

    python3-dbg -m pyperformance run --bench json_dumps --inherit-
environ=PYTHONPATH
```

This approach works for the pyperformance benchmark because it runs in a controlled environment and does not affect other applications.

Why monkey-patching can be dangerous:

- It changes the behavior of a standard library function globally, which can break code that depends on the original behavior or formatting options.
- It can make debugging harder because the function name stays the same, but its implementation changes.

Why is it acceptable here:

- The benchmark is isolated and designed to measure performance, not correctness across all edge cases.
- The patch is temporary and only applies to the benchmark processes (thanks to PYTHONPATH and --inherit-environ).

Alternative approach:

- Instead of monkey-patching, you could modify the standard library's json module to call orjson internally. This would require editing the json/__init__.py file and replacing its dumps implementation.
- However, this is more invasive, harder to maintain, and not recommended for production because it changes the Python installation itself. Monkey-patching is simpler and reversible.

Why orjson improves performance

The orjson library's dumps implementation is faster because it does almost all the work in compiled Rust code instead of Python. This means it avoids slow Python loops and complicated decision-making. It uses efficient techniques like:

- Handling text and special characters quickly using lookup tables and CPU-friendly operations.
- Formatting numbers with fast and predictable algorithms, so converting floats and integers to text is very efficient.
- Building the output in memory in large, continuous chunks instead of small pieces, which helps the CPU work faster because it can keep data in its cache and avoid jumping around in memory.
- Reducing back-and-forth calls between Python and C, which saves time on interpreter overhead. Note that the Python stack is eliminated in the orjson flame graph.
- Using vectorization and SIMD instructions: Instead of processing one character at a time, orjson can process many characters in parallel using special CPU instructions (SIMD). For example, it can check or copy 16 bytes at once instead of 1, which makes tasks like processing strings and encoding much faster.

Together, these changes cut down the number of instructions and branches the CPU has to process, make the call stack shallower, and slightly improve how many instructions run per CPU cycle.

Results

Runtime measured by pyperformance

Variant	Mean ± stdev (ms)	Δ vs. baseline
stdlib json.dumps	62.2 ± 0.5	—
orjson (patched)	11.9 ± 0.1	–80.9% (~5.24×)

```
.....................
json_dumps: Mean +- std dev: 62.2 ms +- 0.5 ms
```

```
.....................
json_dumps: Mean +- std dev: 11.9 ms +- 0.1 ms
```

Hardware counters (perf stat)

Metric	stdlib	orjson	Change (orjson improvement %)
instructions	167,287,865,175	96,822,927,687	-70,464,937,488 (-42.1%)
branches	35,540,658,013	20,984,549,290	-14556108723 (-41.0%)
branch misses	335,263,393	156,514,197	-178,749,196 (-53.3%)
cache misses (% of total accesses)	17,259,734 (2.638%)	6,968,728 (2.151%)	-10,291,006 (-59.6%)
IPC	1.83	1.93	+0.10 (+5.5%)
Elapsed (s)	39.91	20.30	-19.61 s (-49.1%)

The optimization reduced total work (instructions) and branches by ~40%, halved branch misses, and modestly increased IPC. Cache (including L1D) misses also dropped.

Note that perf stats capture statistics for the full pyperformance program, which includes overhead unrelated to the orjson optimization. The orjson improvements, including eliminating the interpreter and Python stack and SIMD utilization, yield great improvements in instruction count and branching stats.

Hardware Acceleration Proposal

Motivation

JSON dumps (and orjson dumps) does a few simple but heavy tasks repeatedly. It scans every string and 'escapes' characters that would break JSON format (like quotes and backslashes which have different meanings in JSON) and handles non-English text correctly. It turns numbers into text (e.g., 3.14159 → "3.14159"). It also adds punctuation (commas, colons, braces) and writes everything into a growing output. Even with fast libraries like orjson, these steps dominate time because they touch every character and digit. A small hardware accelerator could take these repetitive chores, do them in parallel and very predictably, and write the finished JSON straight to memory. That would free the CPU and make JSON dumps significantly faster.

Implementation

We propose a fixed-function JSON Encode Accelerator (JEA) that offloads the two dominant operations in json_dumps:

- string escaping, UTF-8 validation (converting characters into code that can be safely interpreted by JSON parser, UTF-8 is variable-width encoding)
- number-to-text formatting

The software (orjson) iterates over the Python object once to produce a compact 'token' stream in memory, and a guide table to translate tokens to strings. The tokens are components of the object and are built like this like this:

For object: {"a": 1, "ab": [2, "x"]}

Token Stream:

```
OBJ_BEGIN
 KEY( id=0 )       # "a"
 VAL_INT(1)
 KEY( id=1 )       # "ab"
 ARR_BEGIN
  VAL_INT(2)
  VAL_STRING(id=2)  # "x"
 ARR_END
OBJ_END
```

Table:

0 – "a", 1-"ab", 2-"x"

The HW reads the stream and the lookup table and then can translate IDs to strings and do the whole string escaping and character encoding process, instead of SW.

The SW signals to the JEA that when there is work to do on the stream. The JEA then uses DMA-read to take the stream as inputs. It then generates the corresponding JSON string using the received lookup table (currently assuming UTF-8 encoding) and lastly DMA-writes the validated output in a buffer provided by the host and lets the host know with a completion write.

Inputs:

- Token stream in host memory
- Lookup table for strings in token stream
- A 'doorbell' to signal that it can start reading (with address and size of buffer)
- Options (as in json.dumps for user-defined handling of certain characters)

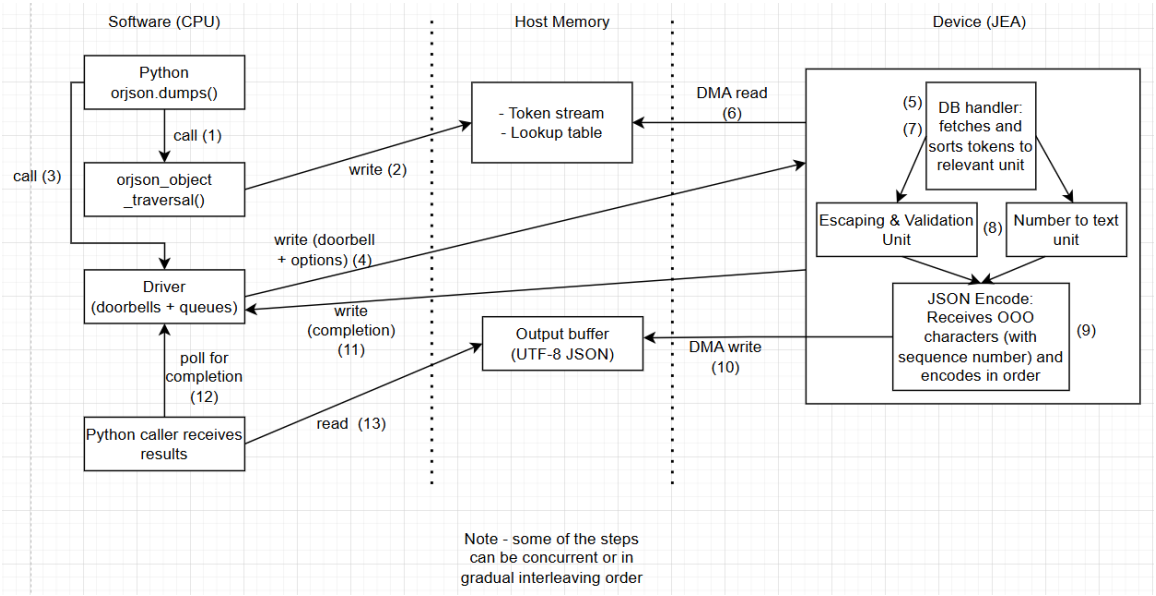
Outputs:

- Contiguous UTF-8 encoded JSON-formatted string
- Status: success/failure (from buffer overflow, unsupported options, etc.)

Hardware/Software Interface

- User-space library (part of orjson) emits the token stream, string table: prepare_tokens(obj) -> (tokens, str_table)
- Doorbell thru driver
- JEA DMA-reads tokens/strings, writes JSON into the provided output buffer
- Completion event + SW poll&read→ library method returns JSON-formatted string

Block Diagram



Performance/Area/Power Trade-offs

Performance:

- Throughput scales with datapath width (e.g., 16–32 bytes/cycle)
- Latency is mostly bound by DMA setup and streaming time
- The traversal and driver still take up CPU time
- Amdahl's law:

$$\text{Speedup}_{\text{overall}} = \frac{1}{(1 - \text{time}_{\text{optimized}}) + \frac{\text{time}_{\text{optimized}}}{\text{speedup}_{\text{optimized}}}}$$

  - We estimate the portion of time spent in JSON formatting is 70% (significantly more than half of json.dumps() as seen qualitatively in the flame graphs)
  - Note – the writes and reads of the stream, doorbells, and output take time, but we neglect them
  - The speedup, S, is determined by how many more characters the JEA can process concurrently than orjson. We assume S = 4.
  - Thus, the overall speedup of orjson with the hardware accelerator is:

$$S_{overall} = \frac{1}{\left((1 - 0.7) + \left(\frac{0.7}{4}\right)\right)} = 2.1$$

That is, ~2x acceleration over orjson → ~10x over stdlib json!

Area:

- Small: character formatting logic, DMA unit + cache
- Not negligible (multiple units, cache…)
- Niche unit (though can be adapted for other vectorized operations)

Power:

- Efficient per-character encoding
- Mostly integer, character logic with modest SRAMs for input fetching and reorder buffers

Reproducibility

Use json_generate_fgs.sh to produce stdlib and orjson flame graphs. For perf stat, run json_benchmark_perf_stat.sh which runs the pyperformance json_dumps benchmark with and without the monkey patch. See README.md in the Git repository's json_dumps directory for detailed instructions.

Conclusion

Replacing json.dumps with orjson delivered ~5× speedup (−80.9% time) on the pyperformance json_dumps benchmark. Perf analysis (which profiled pyperformance overhead too) showed ~40–60% reductions across core metrics and a small IPC increase, reflecting the usage of the orjson native code. We integrated the SW optimization using a runtime patch and suggested a future HW optimization to offload formatting.

Appendix

A – unaccelerated stdlib json.dumps()