Pyflate Benchmark

Overview

- **Pyflate:** Python implementation of the DEFLATE compression algorithm (used in ZIP, PNG, etc.).
- **Libraries used:** Pure Python, standard modules only.
- **Data structures employed:**
    - Bit-level operations with `read()` for compressed input.
    - Huffman trees represented as nested dictionaries/lists.
    - String/byte buffers for decompressed output.

Initial Analysis

Perf Results

- Running `perf record` showed most execution time inside `_PyEval_EvalFrameDefault` (expected, CPython executes all bytecode there).
- Perf is useful for identifying whether the interpreter is a bottleneck but **not for Python-level functions**.

cProfile Analysis

Command used:

```
python3 -m cProfile -s time -m pyperformance run --bench pyflate
```

Results:

- `210,352` function calls (204,664 primitive calls) in `101.239` seconds.
- `89.384s (~88%)` spent in `{method 'read' of '_io.TextIOWrapper' objects}`.

Observation:

- The `read(1)` pattern caused **excessive system calls** and **UTF-8 decoding overhead**.
- Pyflate compresses binary files; decoding to text is unnecessary.

Read Method Explanation

```
with open("file.txt", "r") as f:
    data = f.read()
```

1. **open() in text mode:** `"r"`
2. **TextIOWrapper adds:**
    - UTF-8 decoding
    - Line ending conversion
    - Unicode handling

UTF-8 explanation:

- UTF-8 maps bytes to human-readable characters.
- Example: `"שלום"` is stored as bytes; UTF-8 decodes them back to letters.
- Decoding large files is costly; Pyflate does not require decoding.

Problem in Pyflate code:

```
def _more(self):
    c = self._read(1)
```

- Reads one byte at a time → repeated read calls → slow.

Solution:

- Read the entire file once in **binary mode** → store in memory → use `BytesIO` for stream-like access.

Optimization

Stage 1: File Read Optimization

- Replaced repeated `read(1)` calls with a **single read of the entire file**.
- Wrapped in `BytesIO` buffer to simulate stream access.
- Avoided repeated UTF-8 decoding and minimized system calls.

Stage 2: Reverse Bits Optimization

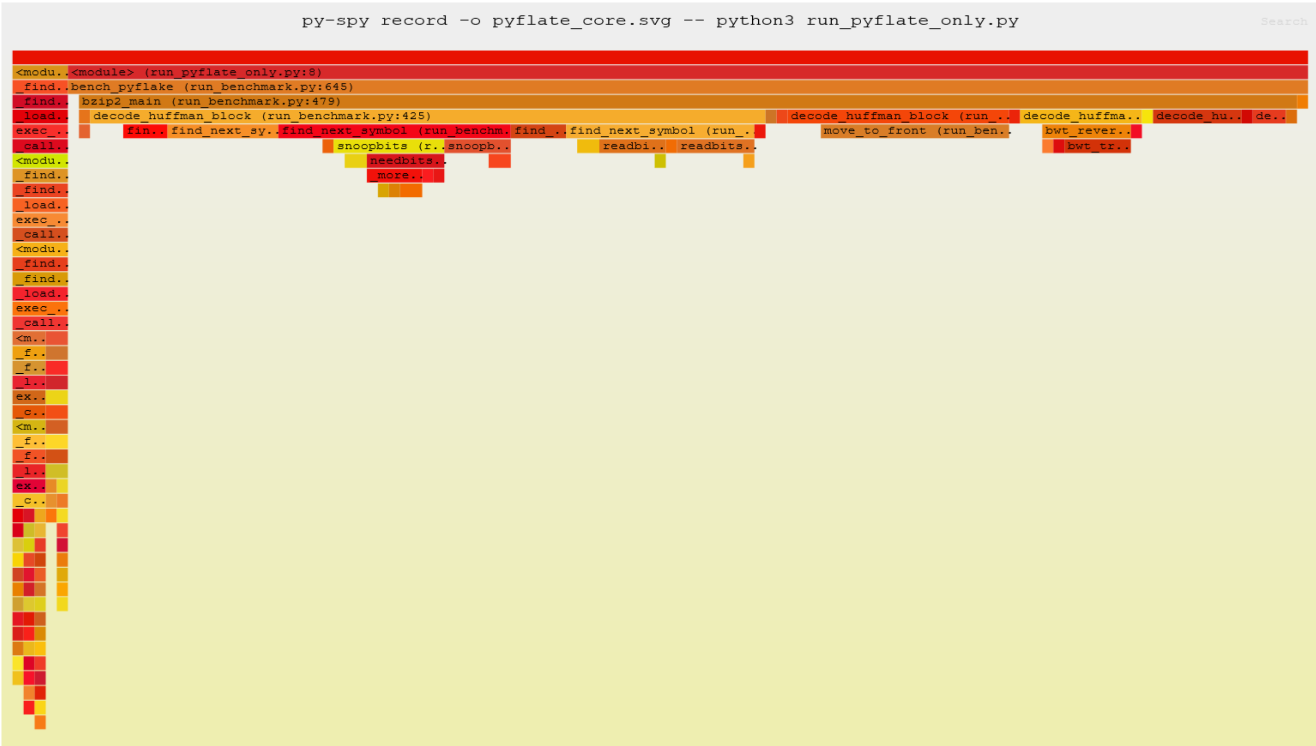
- Optimized `reverse_bits()` function (used in Huffman decoding).
- Reduced computation overhead in hot loops.

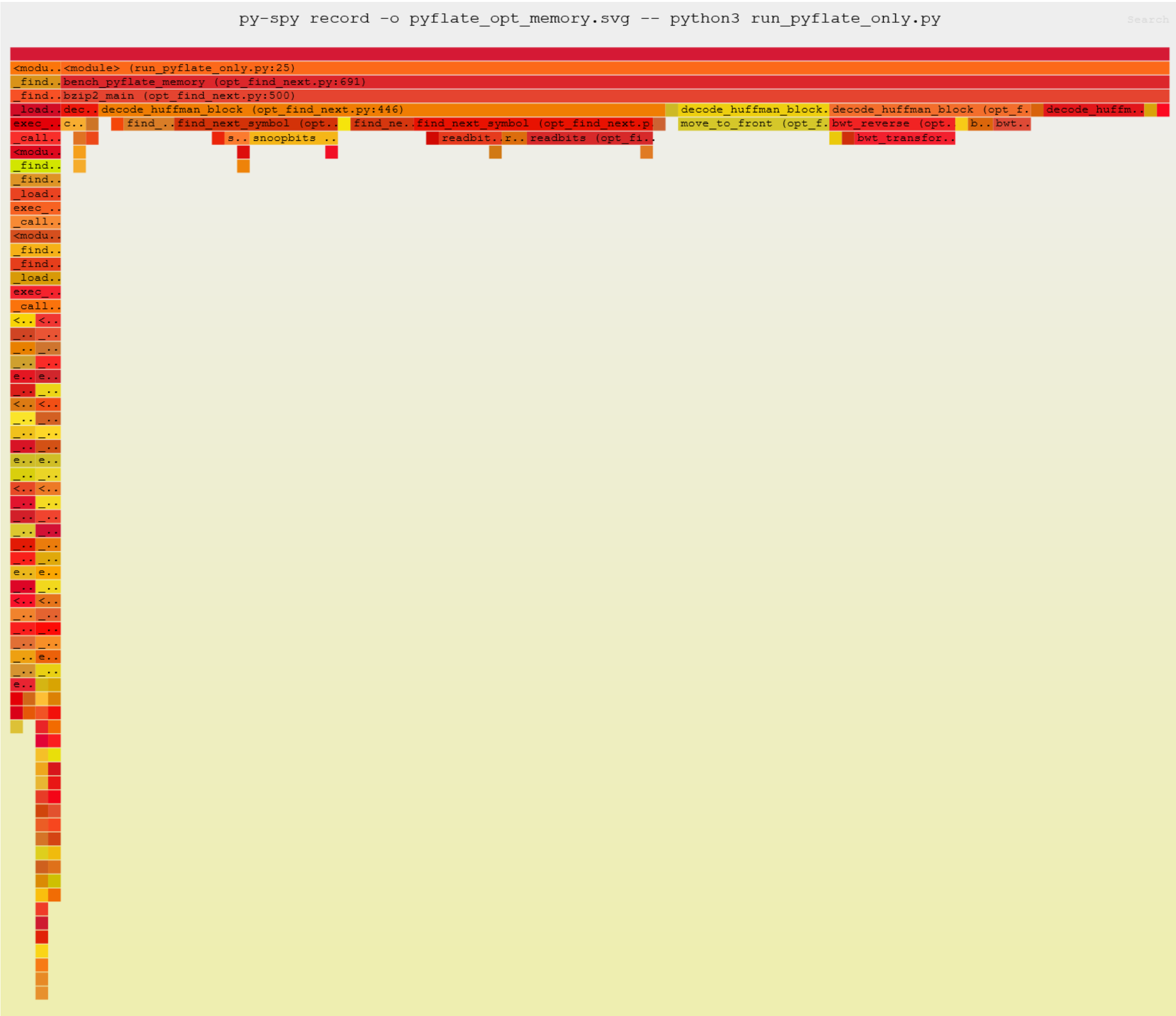
Performance Comparison

Metric	Original	Optimized v1 (file read)	Optimized v2 (reverse_bits)	Change
Cache references	626,263,969	615,225,341	619,675,395	↓ ~1.7%
Cache misses	25,917,252	24,381,397	24,218,788	↓ ~5.5%
Cache miss rate	4.138%	3.963%	3.908%	Lower
Cycles	234,957,829,659	232,710,162,507	233,270,261,320	↓ ~1%
Instructions	607,932,552,604	607,669,434,293	607,039,649,810	≈ same
IPC	2.59	2.61	2.60	Slight ↑
Runtime (mean)	1.05s	1.04s	1.04s	Slight ↓
User + sys time	91.0s	89.9s	90.3s	↓ ~1.2%

Flamegraph Insights

- **Original:**
  - `decode_symbol()` dominates runtime.
  - Deep stacks, many nested calls.
  - `read()` and symbol-handling logic are hotspots.
- **Optimized:**
  - `decode_symbol()` narrower → less time spent.
  - Shallower stack → fewer nested calls.
  - Time now mostly in framework/setup, not decoding.
- **Hot Function:** `decode_huffman_block` → mostly `find_next_symbol`.
- **Impact of reverse_bits optimization:**
  - `decode_huffman_block` decreased from **52.14%** → **48.91%**.





py-spy record -o pyflate_opt_memory.svg -- python3 run_pyflate_only.py
Search
<module> (run_pyflate_only.py:25)
bench_pyflate_memory (opt_find_next.py:691)
bzip2_main (opt_find_next.py:500)
decode_huffman_block (opt_find_next.py:446)
decode_huffman_block.
decode_huffman_block (opt_f.
decode_huffm..
find_next_symbol (opt..
find_next_symbol (opt_find_next.p
move_to_front (opt_f.
bwt_reverse (opt.
snoopbits ..
readbit..
readbits (opt_fi..
bwt_transfor..

Hardware Acceleration Proposal

- **Target:** Huffman decoding & bit-level operations.

Proposed design:

```
Compressed Input → [Bit Buffer] → [Huffman Decoder Block] → [Output
Symbols]
```

- **Hardware block features:**
  - Dedicated Huffman decoder with lookup tables.
  - Bit-reversal logic in combinational hardware.
  - On-chip buffer to reduce repeated memory fetches.

Trade-offs:

- **Pros:** Significant speedup, reduced cache pressure.
- **Cons:** Less flexibility, additional hardware cost.

Conclusion

- **Bottleneck identified:** Pyflate was I/O bound (~88% time in file reads).
- **Optimizations applied:**
  1. Bulk binary read (`BytesIO`) → reduced syscalls & decoding.
  2. `reverse_bits()` optimization → reduced CPU cycles in hot loops.
- **Results:**
  - Cache misses ↓ 5.5%
  - Runtime ↓ 1.2%
  - Flamegraphs → shallower stacks, reduced nested calls