Mechanical Sympathy in Rust Performance Optimization

Brian Pane <brianp@brianp.net>

Presented to Seattle Rust User Group 2025-10-23

Agenda

Share some ways to make fast software faster

by enabling the hardware to run more efficiently

based on my experiences contributing optimizations to zlib-rs

My Background

2023: Retired from a career in software and networking

Goal: do a project each year to give back to the community

- 2024: Write a book on edge network strategy ✔️
- 2025: Help make foundational Internet software fast & safe

Finding Somewhere to Contribute

ISRG → PROSSIMO

zlib

Let's create a memory safe zlib compression library with great performance.

View initiative

zlib

zlib-rs

fix deflate performance #18

Open

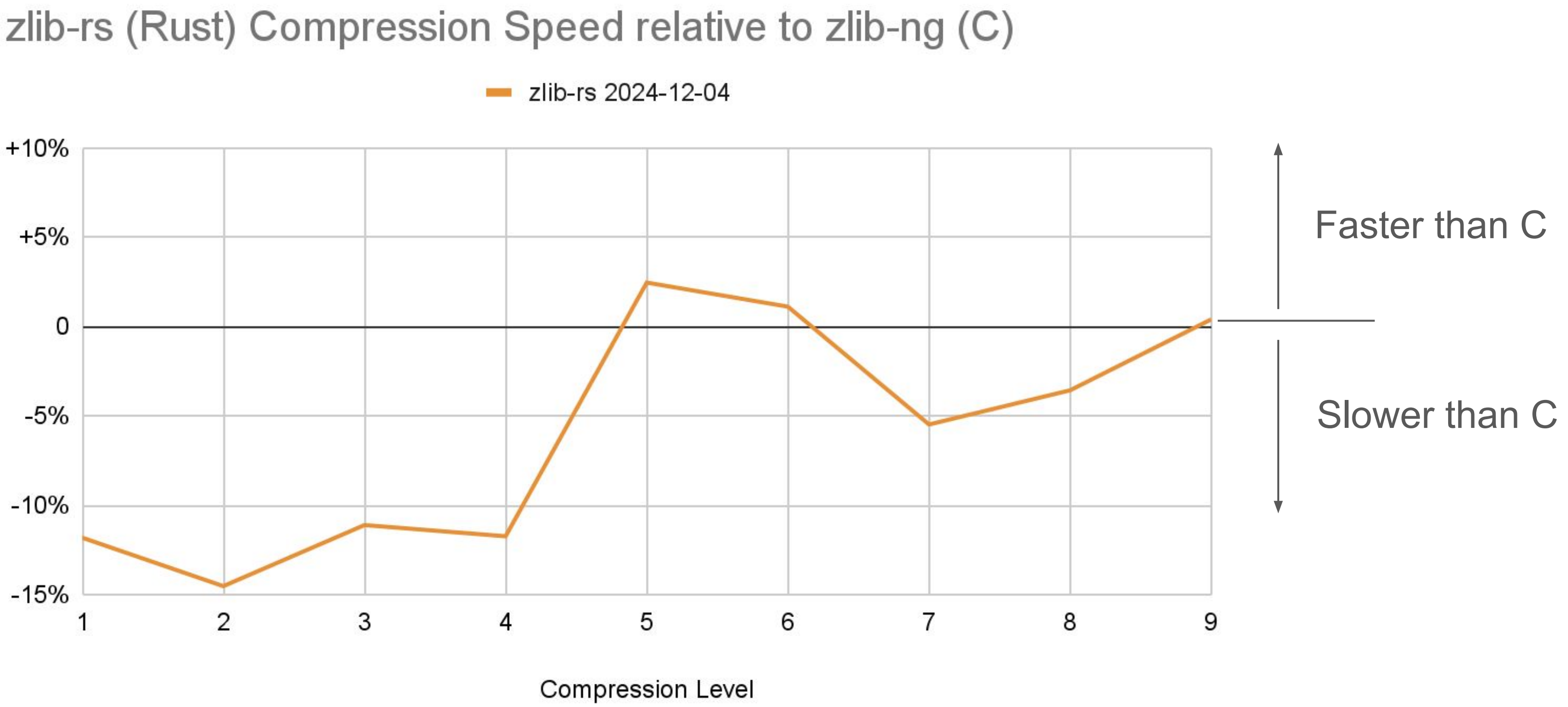
folkertdev opened on Feb 1, 2024 · Collaborator

we are consistently ~10% slower than zlib-ng on deflate. This fluctuates with the different compression levels, but currently none of then are on-par with zlib-ng.

There is so far no obvious reason for this slowdown, so it's likely a "death by a thousand papercuts" sort of thing.

👍 2 ❤️ 1

Initial Benchmark Results



Initial Findings from Profiling zlib-rs

Well-optimized implementation

- Rust + inline assembly for the SIMD parts

Smart compiler

- Micro-optimizations, zero-cost abstractions

Relatively flat profile

- Few low hanging fruit

Optimization Strategy

Mechanical Sympathy:

Adapt the software to the strengths and weaknesses of the target hardware

Primary target hardware: modern smartphone through server CPUs

+ 64-bit registers, superscalar cores, out-of-order execution, fast clock
  - Deeply pipelined
+ Lots of memory
  - Dependent on cache hierarchy for speed

Example 1: Cache Locality

Example read latencies, from an x86_64 desktop system:

Layer	Read latency (cycles)	Size	Notes
L1D cache	3	80 KB	64 byte cache lines
L2 cache	10	1 MB	
L3 cache	32	18 MB	
Main memory	168	Up to 128 GB	

zlib compression is a challenge for the small L1 cache

- Lots of string matching against a 32KB sliding window of input data

Example 1: Cache Locality

The profiler can tell us where the cache misses are happening.

```
$ perf record -F max -e L1-dcache-load-misses
./target/release/examples/blogpost-compress ...

$ samply import perf.data
```

The supported counter names are CPU-specific. Run `perf list` to see what your processor supports.

Example 1: Cache Locality

The cache misses are scattered all over the codebase

… but many involve reads of the state data structures used for bookkeeping.

Observation: longest-match processing seems to be displacing this state from the cache often.

Total (samples)		Self		
39%	4,844	506	zlib_rs::deflate::longest_match::longest_match	/home/brian/code/zlib-rs/zl
34%	4,176	2,901	inl zlib_rs::deflate::longest_match::longest_match_help	/home/brian/cod
1.2%	148	148	inl zlib_rs::deflate::longest_match::longest_match_help	/home/brian/cod
0.1%	14	14	inl zlib_rs::deflate::longest_match::longest_match_help	/home/brian/.rus
29%	3,577	2,089	zlib_rs::deflate::algorithm::medium::deflate_medium	/home/brian/code/zli
9.6%	1,192	132	inl zlib_rs::deflate::algorithm::medium::emit_match	/home/brian/code/zli
4.7%	580	122	inl zlib_rs::deflate::State::tally_dist	/home/brian/code/zlib-rs/zlib-rs/src/
2.9%	367	314	inl zlib_rs::deflate::State::tally_lit_help	/home/brian/code/zlib-rs/zlib-rs/
0.4%	55	55	inl zlib_rs::deflate::window::Window::filled	/home/brian/code/zlib-rs/zli
0.2%	30	—	inl <core::slice::iter::Iter<T> as core::iter::traits::iterator::Iterator>::next	
0.2%	28	—	inl core::slice::index::<impl core::ops::index::Index<I> for [T]>::index	/ho
1.2%	148	148	inl zlib_rs::deflate::State::quick_insert_string	/home/brian/code/zlib-rs/zli
0.8%	97	69	inl zlib_rs::deflate::algorithm::medium::insert_match	/home/brian/code/z
0.4%	44	44	inl zlib_rs::deflate::algorithm::medium::emit_match	/home/brian/code/zli
0.1%	7	7	inl zlib_rs::deflate::algorithm::medium::emit_match	/home/brian/.rustup/
11%	1,340	1,039	zlib_rs::deflate::hash_calc::Crc32HashCalc::quick_insert_string	/home/brian
1.3%	159	159	inl zlib_rs::weak_slice::WeakArrayMut<T,_>::as_slice	/home/brian/code/zli
1.1%	140	140	inl zlib_rs::weak_slice::WeakSliceMut<T>::as_mut_slice	/home/brian/code
0.0%	1	—	inl core::slice::index::<impl core::ops::index::Index<I> for [T]>::index	/home
0.0%	1	—	inl zlib_rs::deflate::window::Window::filled	/home/brian/code/zlib-rs/zlib-
9.9%	1,228	825	zlib_rs::deflate::hash_calc::Crc32HashCalc::insert_string	/home/brian/code/

Example 1: Cache Locality

```rust
#[repr(C)]
pub(crate) struct State<'a> {
    status: Status,

    last_flush: i8, /* value of flush param for previous deflate call */

    pub(crate) wrap: i8, /* bit 0 true for zlib, bit 1 true for gzip */

    pub(crate) strategy: Strategy,
    pub(crate) level: i8,

    /// Whether or not a block is currently open for the QUICK deflation scheme.
    /// true if there is an active block, or false if the block was just closed
    pub(crate) block_open: u8,

    bit_writer: BitWriter<'a>,

    /// Use a faster search when the previous match is longer than this
    pub(crate) good_match: usize,

    /// Stop searching when current match exceeds this
    pub(crate) nice_match: usize,

    // part of the fields below
    //    dyn_ltree: [Value; ],
    //    dyn_dtree: [Value; ],
    //    bl_tree: [Value; ],
    l_desc: TreeDesc<HEAP_SIZE>,             /* literal and length tree */
    d_desc: TreeDesc<{ 2 * D_CODES + 1 }>,   /* distance tree */
    bl_desc: TreeDesc<{ 2 * BL_CODES + 1 }>, /* Huffman tree for bit lengths */

    pub(crate) bl_count: [u16; MAX_BITS + 1],

    pub(crate) match_length: usize,   /* length of best match */
    pub(crate) prev_match: u16,       /* previous match */
    pub(crate) match_available: bool, /* set if previous match exists */
    pub(crate) strstart: usize,       /* start of string to insert */
    pub(crate) match_start: usize,    /* start of matching string */

    /// Length of the best match at previous step. Matches not greater than this
    /// are discarded. This is used in the lazy match evaluation.
    pub(crate) prev_length: usize,

    /// To speed up deflation, hash chains are never searched beyond this length.
    /// A higher limit improves compression ratio but degrades the speed.
    pub(crate) max_chain_length: usize,

    // TODO untangle this mess! zlib uses the same field differently based on compression level
    // we should just have 2 fields for clarity!
    //
    // Insert new strings in the hash table only if the match length is not
    // greater than this length. This saves time but degrades compression.
    // max_insert_length is used only for compression levels <= 3.
    // define max_insert_length  max_lazy_match
```

```rust
    /// Attempt to find a better match only when the current match is strictly smaller
    /// than this value. This mechanism is used only for compression levels >= 4.
    pub(crate) max_lazy_match: usize,

    /// Window position at the beginning of the current output block. Gets
    /// negative when the window is moved backwards.
    pub(crate) block_start: isize,

    pub(crate) window: Window<'a>,

    pub(crate) sym_buf: ReadBuf<'a>,

    /// Size of match buffer for literals/lengths.  There are 4 reasons for
    /// limiting lit_bufsize to 64K:
    ///   - frequencies can be kept in 16 bit counters
    ///   - if compression is not successful for the first block, all input
    ///     data is still in the window so we can still emit a stored block even
    ///     when input comes from standard input.  (This can also be done for
    ///     all blocks if lit_bufsize is not greater than 32K.)
    ///   - if compression is not successful for a file smaller than 64K, we can
    ///     even emit a stored file instead of a stored block (saving 5 bytes).
    ///     This is applicable only for zip (not gzip or zlib).
    ///   - creating new Huffman trees less frequently may not provide fast
    ///     adaptation to changes in the input data statistics. (Take for
    ///     example a binary file with poorly compressible code followed by
    ///     a highly compressible string table.) Smaller buffer sizes give
    ///     fast adaptation but have of course the overhead of transmitting
    ///     trees more frequently.
    ///   - I can't count above 4
    lit_bufsize: usize,

    /// Actual size of window: 2*wSize, except when the user input buffer is directly used as sl
    pub(crate) window_size: usize,

    /// number of string matches in current block
    pub(crate) matches: usize,

    /// bit length of current block with optimal trees
    opt_len: usize,
    /// bit length of current block with static trees
    static_len: usize,

    /// bytes at end of window left to insert
    pub(crate) insert: usize,

    pub(crate) w_size: usize,    /* LZ77 window size (32K by default) */
    pub(crate) w_bits: usize,    /* log2(w_size)  (8..16) */
    pub(crate) w_mask: usize,    /* w_size - 1 */
    pub(crate) lookahead: usize, /* number of valid bytes ahead in window */

    pub(crate) prev: WeakSliceMut<'a, u16>,
    pub(crate) head: WeakArrayMut<'a, u16, HASH_SIZE>,

    ///  hash index of string to be inserted
```

Compression state:

- Giant struct containing counters, flags, etc - in basically random order
- Small fields, compared to cache line size
- Already using `#[repr(C)]` so we control memory layout…

Hypothesis: We can speed up the program by ensuring that fields commonly used together are grouped into the same cache line.

Example 1: Cache Locality

First cache line
(64 bytes)

Contains fields commonly used together in the code.

```rust
#[repr(C, align(64))]
pub(crate) struct State<'a> {
    status: Status,

    last_flush: i8, /* value of flush param for previous deflate call */

    pub(crate) wrap: i8, /* bit 0 true for zlib, bit 1 true for gzip */

    pub(crate) strategy: Strategy,
    pub(crate) level: i8,

    /// Whether or not a block is currently open for the QUICK deflation scheme.
    /// 0 if the block is closed, 1 if there is an active block, or 2 if there
    /// is an active block and it is the last block.
    pub(crate) block_open: u8,

    pub(crate) hash_calc_variant: HashCalcVariant,

    pub(crate) match_available: bool, /* set if previous match exists */

    /// Use a faster search when the previous match is longer than this
    pub(crate) good_match: u16,

    /// Stop searching when current match exceeds this
    pub(crate) nice_match: u16,

    pub(crate) match_start: Pos,      /* start of matching string */
    pub(crate) prev_match: Pos,       /* previous match */
    pub(crate) strstart: usize,       /* start of string to insert */

    pub(crate) window: Window<'a>,
    pub(crate) w_size: usize,    /* LZ77 window size (32K by default) */
    pub(crate) w_mask: usize,    /* w_size - 1 */

    _cache_line_0: (),

    /// prev[N], where N is an offset in the current window, contains the offset in the window
    /// of the previous 4-byte sequence that hashes to the same value as the 4-byte sequence
    /// starting at N. Together with head, prev forms a chained hash table that can be used
    /// to find earlier strings in the window that are potential matches for new input being
    /// deflated.
    pub(crate) prev: WeakSliceMut<'a, u16>,
    /// head[H] contains the offset of the last 4-character sequence seen so far in
    /// the current window that hashes to H (as calculated using the hash_calc_variant).
    pub(crate) head: WeakArrayMut<'a, u16, HASH_SIZE>,

    /// Length of the best match at previous step. Matches not greater than this
    /// are discarded. This is used in the lazy match evaluation.
    pub(crate) prev_length: u16,
```

Start the struct on a 64 byte boundary

Zero-length markers to denote cache line boundaries

Next cache line
(64 bytes)

Fields commonly used together.

Example 1: Cache Locality

Result: 4% decrease in CPU cycles at compression level 1

Example 1: Cache Locality

Verify layout in unit tests to help maintainers avoid regressions

```rust
#[cfg(any(target_arch = "x86_64", target_arch = "aarch64"))]
mod _cache_lines {
    use super::State;
    // FIXME: once zlib-rs Minimum Supported Rust Version >= 1.77, switch to core::mem::offset_of
    // and move this _cache_lines module from up a level from tests to super::
    use memoffset::offset_of;

    const _: () = assert!(offset_of!(State, status) == 0);
    const _: () = assert!(offset_of!(State, _cache_line_0) == 64);
    const _: () = assert!(offset_of!(State, _cache_line_1) == 128);
    const _: () = assert!(offset_of!(State, _cache_line_2) == 192);
    const _: () = assert!(offset_of!(State, _cache_line_3) == 256);
}
```

Example 2: Inlining

This “send_bits” function is small and is called often, but isn’t inlined.

Total (samples)		Self		
25%	3,464	327	zlib_rs::deflate::longest_match::longest_match	/home/brian/c
24%	3,306	1,453	zlib_rs::deflate::algorithm::fast::deflate_fast	/home/brian/coc
17%	2,338	1,861	zlib_rs::deflate::hash_calc::Crc32HashCalc::quick_insert_string	
1.9%	253	253	inl zlib_rs::weak_slice::WeakArrayMut<T,_>::as_slice	/home/
1.5%	210	210	inl zlib_rs::weak_slice::WeakSliceMut<T>::as_mut_slice	/hon
0.1%	8	—	inl zlib_rs::deflate::window::Window::filled	/home/brian/coc
0.0%	5	—	inl core::slice::index::<impl core::ops::index::Index<I> for [T]>:	
0.0%	1	1	inl zlib_rs::deflate::hash_calc::Crc32HashCalc::update_hash	/
6.0%	814	772	zlib_rs::deflate::BitWriter::send_bits	/home/brian/code/zlib-rs
5.7%	780	250	zlib_rs::deflate::Heap::pqdownheap	/home/brian/code/zlib-rs/
5.3%	718	627	zlib_rs::deflate::BitWriter::emit_dist	/home/brian/code/zlib-rs
5.2%	706	—	zlib_rs::deflate::flush_block_only	/home/brian/code/zlib-rs/zli
4.4%	595	189	zlib_rs::deflate::compare256::avx2::compare256	/home/brian/
2.2%	299	228	zlib_rs::deflate::hash_calc::Crc32HashCalc::insert_string	/hom
1.2%	167	—	zlib_rs::deflate::slide_hash::avx2::slide_hash_chain_internal	/h

Example 2: Inlining

```rust
fn send_bits(&mut self, val: u64, len: u8) {
    debug_assert!(len <= 64);
    debug_assert!(self.bits_used <= 64);

    let total_bits: u8 = len + self.bits_used;

    self.send_bits_trace(val, len);
    self.sent_bits_add(len as usize);

    self.bit_buffer |= val << self.bits_used;
    if total_bits < Self::BIT_BUF_SIZE {
        self.bits_used = total_bits;
    } else {
        self.pending.extend(&self.bit_buffer.to_le_bytes());
        self.bit_buffer = val >> (Self::BIT_BUF_SIZE - self.bits_used);
        self.bits_used = total_bits - Self::BIT_BUF_SIZE;
    }
}
```

Debug-only

Very lightweight operations

Less common case

Hypothesis: we should add `#[inline(always)]`

Example 2: Inlining

No, inlining that function makes the performance ***worse***

… instruction cache is a scarce resource

… but how about just inlining the fast path?

```rust
#[inline(always)]
fn send_bits(&mut self, val: u64, len: u8) {
    debug_assert!(len <= 64);
    debug_assert!(self.bits_used <= 64);

    let total_bits : u8  = len + self.bits_used;

    self.send_bits_trace(val, len);
    self.sent_bits_add(len as usize);

    if total_bits < Self::BIT_BUF_SIZE {
        self.bit_buffer |= val << self.bits_used;
        self.bits_used = total_bits;
    } else {
        self.send_bits_overflow(val, total_bits);
    }
}
```

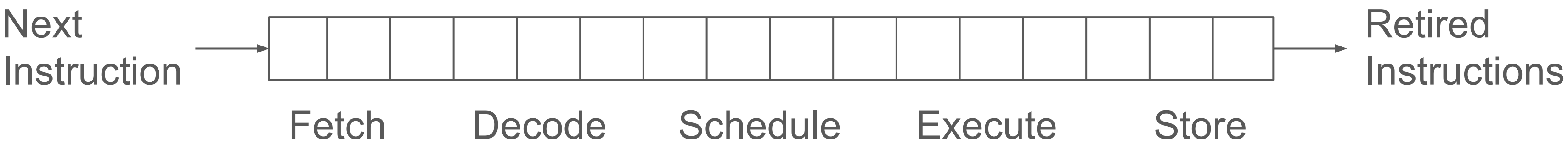
Less common case: not inlined

Example 2: Inlining

Result: 5% reduction in CPU cycles at compression level 1

Example 3: Branch Prediction

High-performance CPUs tend to have deep pipelines




```
mov dword [rdi + rcx * 4 + 0x10], r8d
lea r8, qword [r9 + r9 * 1]
mov rcx, r9
cmp r8, r11
jna 0x10c50
mov dword [rdi + r9 * 4 + 0x10], eax
```

Proceed to next instruction

?

Example 3: Branch Prediction

The CPU’s branch predictor tries to guess which way each branch will go

… by observing past executions of the same branch instruction

… but getting this prediction wrong means a pipeline stall.

We can find mispredictions with a profiler.

```
$ perf record -F max -e branch-misses
./target/release/examples/blogpost-compress ...

$ samply import perf.data
```

Example 3: Branch Prediction

This binary heap code looks interesting. Sorted by primary & secondary keys - each element comparison requires up to two unpredictable branches.

Hypothesis: doing fewer conditional branches will be a win, even if it requires more total instructions.

Total (samples)		Self	
40%	4,635	1,144	zlib_rs::deflate::algorithm::fast::deflate_fast /home/brian/code/zlib-rs/zli
27%	3,189	90	zlib_rs::deflate::longest_match::longest_match /home/brian/code/zlib-rs
14%	1,656	—	zlib_rs::deflate::flush_block_only /home/brian/code/zlib-rs/zlib-rs/src/de
10%	1,216	239	zlib_rs::deflate::Heap::pqdownheap /home/brian/code/zlib-rs/zlib-rs/src/
8.4%	977	493	inl zlib_rs::deflate::Heap::smaller /home/brian/code/zlib-rs/zlib-rs/src/d
4.1%	484	484	inl core::cmp::impls::<impl core::cmp::Ord for u16>::cmp /home/brian
4.2%	495	48	zlib_rs::deflate::compare256::avx2::compare256 /home/brian/code/zlib-r

/home/brian/code/zlib-rs/zlib-rs/src/deflate.rs

Total	Self	
		`fn smaller(tree: &[Value], n: u32, m: u32, depth: &[u8]) -> bool {`
		`    let (n, m) = (n as usize, m as usize);`
950	466	`    match Ord::cmp(&tree[n].freq(), &tree[m].freq()) {`
		`        core::cmp::Ordering::Less => true,`
20	20	`        core::cmp::Ordering::Equal => depth[n] <= depth[m],`
		`        core::cmp::Ordering::Greater => false,`
		`    }`
		`}`

Example 3: Branch Prediction

1. Pack the primary and secondary sort keys into a single register

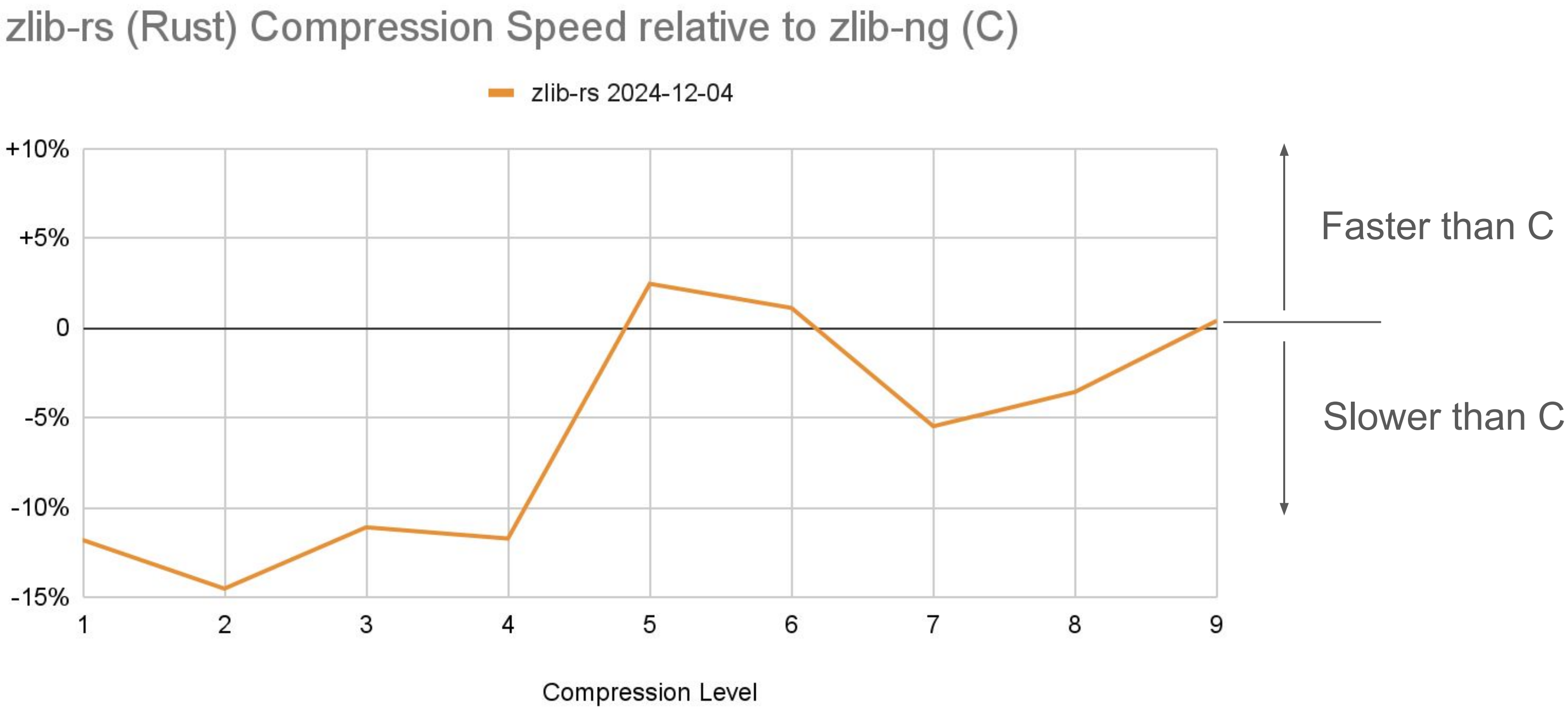
```
macro_rules! freq_and_depth {
    ($i:expr) => {
        (tree[$i as usize].freq() as u32) << 8 | self.depth[$i as usize] as u32
    };
}
```

2. Compare the packed values (turning two conditional branches into one)

Example 3: Branch Prediction

Result: 3% reduction in CPU cycles at compression level 2

Before



zlib-rs (Rust) Compression Speed relative to zlib-ng (C)
zlib-rs 2024-12-04
+10%
+5%
0
-5%
-10%
-15%
1
2
3
4
5
6
7
8
9
Compression Level
Faster than C
Slower than C

After


zlib-rs (Rust) Compression Speed relative to zlib-ng (C)
zlib-rs 2024-12-04
zlib-rs 2025-06-27
+10%
+5%
0
-5%
-10%
-15%
1
2
3
4
5
6
7
8
9
Compression Level
Faster than C
Slower than C

Takeaways

Instrumentation over intuition

The bottlenecks are surprising sometimes

Correctness first

Regression testing & test coverage measurement to avoid breaking things

Complement the compiler

You know things it doesn’t know, and vice versa

Q&A

Appendix

Example 4: Uncommon Subexpression Elimination

```
loop {
    // Make sure that we always have enough lookahead, except
    // at the end of the input file. We need STD_MAX_MATCH bytes
    // for the next match, plus WANT_MIN_MATCH bytes to insert the
    // string following the next match.
    if stream.state.lookahead < MIN_LOOKAHEAD {...}

    let state : &mut &mut State  = &mut stream.state;

    // Insert the string window[strstart .. strstart+2] in the
    // dictionary, and set hash_head to the head of the hash chain:

    if state.lookahead >= WANT_MIN_MATCH {
        let hash_head : u16  = StandardHashCalc::quick_insert_string(state, state.strstart);
        dist = state.strstart as isize - hash_head as isize;

        /* Find the longest match, discarding those <= prev_length.
         * At this point we have always match length < WANT_MIN_MATCH
         */
        if dist <= state.max_dist() as isize && dist > 0 && hash_head != 0 {
            // To simplify the code, we prevent matches with the string
            // of window index 0 (in particular we have to avoid a match
            // of the string with itself at the start of the input file).
            (match_len, state.match_start) =
                crate::deflate::longest_match::longest_match(state, hash_head);
        }
    }

    if match_len >= WANT_MIN_MATCH {...} else {
        /* No match, output a literal byte */
        let lc : u8  = state.window.filled()[state.strstart];
        bflush = state.tally_lit(lc);
        state.lookahead -= 1;
        state.strstart += 1;
    }
```

Main input-processing loop

1. Peek at the first 4 bytes of remaining input and hash them to find potential matches against data seen earlier

2. Peek at the first 8+ bytes to find the longest match

3. If no match, fetch the first byte and output it uncompressed

Example 4: Uncommon Subexpression Elimination

```
loop {
    // Make sure that we always have enough lookahead, except
    // at the end of the input file. We need STD_MAX_MATCH bytes
    // for the next match, plus WANT_MIN_MATCH bytes to insert the
    // string following the next match.
    if stream.state.lookahead < MIN_LOOKAHEAD {...}

    let state : &mut &mut State = &mut stream.state;

    // Insert the string window[strstart .. strstart+2] in the
    // dictionary, and set hash_head to the head of the hash chain:

    let lc: u8; // Literal character to output if there is no match.
    if state.lookahead >= WANT_MIN_MATCH {
        let val : u32 = u32::from_le_bytes(
            state.window.filled()[state.strstart.. < state.strstart + 4]
                .try_into() : Result<[u8; ?], TryFromSliceError>
                .unwrap(),
        );
        let hash_head : u16 = StandardHashCalc::quick_insert_value(state, state.strstart, val);
        let dist : isize = state.strstart as isize - hash_head as isize;

        // Find the longest match for the string starting at offset state.strstart.
        if dist <= state.max_dist() as isize && dist > 0 && hash_head != 0 {
            // To simplify the code, we prevent matches with the string
            // of window index 0 (in particular we have to avoid a match
            // of the string with itself at the start of the input file).
            let mut match_len : usize ;
            (match_len, state.match_start) =
                crate::deflate::longest_match::longest_match(state, hash_head);
            if match_len >= WANT_MIN_MATCH {...}
        }
        lc = val as u8;
```

Read first 4 bytes to use in hash calculation.

Keep track of 1st of those bytes in case we need to output it.

Example 4: Uncommon Subexpression Elimination

Result: 2.5% decrease in CPU cycles at compression level 2

Profiling Tools

Linux `perf` utility - perfwiki.github.io

- Sampling profiler for userspace and kernel
- Can read hardware performance counters
- Identifies hotspots at the source line level

Samply - github.com/mstange/samply

- Profile visualizer based on Firefox profiling UI
- Can read the traces recorded by `perf`

Benchmarking Tools

hyperfine - github.com/sharkdp/hyperfine

+ Finds statistically significant differences
+ Cross-platform
− Measures elapsed time only

Performance Optimizer Observation Platform - github.com/andrewrk/poop

+ Measures elapsed time, CPU cycles, instructions, cache misses, more
+ Finds statistically significant differences
− Linux only