The Rust Performance Book

First published in November 2020

Written by Nicholas Nethercote and others

Source code

Introduction

Performance is important for many Rust programs.

This book contains techniques that can improve the performance—speed and memory usage—of Rust programs. The Compile Times section also contains techniques that will improve the compile times of Rust programs. Some techniques only require changing build configurations, but many require changing code.

Some techniques are entirely Rust-specific, and some involve ideas that can be applied (often with modifications) to programs written in other languages. The General Tips section also includes some general principles that apply to any programming language. Nonetheless, this book is mostly about the performance of Rust programs and is no substitute for a general purpose guide to profiling and optimization.

This book also focuses on techniques that are practical and proven: many are accompanied by links to pull requests or other resources that show how the technique was used on a real-world Rust program.

This book is deliberately terse, favouring breadth over depth, so that it is quick to read. It links to external sources that provide more depth when appropriate.

This book is aimed at intermediate and advanced Rust users. Beginner Rust users have more than enough to learn and these techniques are likely to be an unhelpful distraction to them.

Benchmarking

Benchmarking typically involves comparing the performance of two or more programs that do the same thing. Sometimes this might involve comparing two or more different programs, e.g. Firefox vs Safari vs Chrome. Sometimes it involves comparing two different versions of the same program. This latter case lets us reliably answer the question “did this change speed things up?”

Benchmarking is a complex topic and a thorough coverage is beyond the scope of this book, but here are the basics.

First, you need workloads to measure. Ideally, you would have a variety of workloads that represent realistic usage of your program. Workloads using real-world inputs are best, but microbenchmarks and stress tests can be useful in moderation.

Second, you need a way to run the workloads, which will also dictate the metrics used. Rust’s built-in benchmark tests are a simple starting point, but they use unstable features and therefore only work on Nightly Rust. Criterion is a more sophisticated alternative. Custom benchmarking harnesses are also possible. For example, rustc-perf is the harness used to benchmark the Rust compiler.

When it comes to metrics, there are many choices, and the right one(s) will depend on the nature of the program being benchmarked. For example, metrics that make sense for a batch program might not make sense for an interactive program. Wall-time is an obvious choice in many cases because it corresponds to what users perceive. However, it can suffer from high variance. In particular, tiny changes in memory layout can cause significant but ephemeral performance fluctuations. Therefore, other metrics with lower variance (such as cycles or instruction counts) may be a reasonable alternative.

Summarizing measurements from multiple workloads is also a challenge, and there are a variety of ways to do it, with no single method being obviously best.

Good benchmarking is hard. Having said that, do not stress too much about having a perfect benchmarking setup, particularly when you start optimizing a program. A mediocre setup is far better than no setup. Keep an open mind about what you are measuring, and over time you can make benchmarking improvements as you learn about the performance characteristics of your program.

Build Configuration

The right build configuration will maximize the performance of your Rust program without any changes to its code. But you should check your program's performance after applying any of the following changes, because they can sometimes worsen performance.

Release Builds

The single most important Rust performance tip is simple but easy to overlook: make sure you are using a release build rather than a dev build when you want high performance. This is usually done by specifying the `--release` flag to Cargo.

A release build typically runs *much* faster than a dev build. 10-100x speedups over dev builds are common!

Dev builds are the default. They are produced if you run `cargo build`, `cargo run`, or `rustc` without any additional options. Dev builds are good for debugging, but are not optimized.

Consider the following final line of output from a `cargo build` run.

```
Finished dev [unoptimized + debuginfo] target(s) in 29.80s
```

The `[unoptimized + debuginfo]` indicates that a dev build has been produced. The compiled code will be placed in the `target/debug/` directory. `cargo run` will run the dev build.

Release builds are more optimized than dev builds. They also omit some checks, such as debug assertions and integer overflow checks. Produce one with `cargo build --release`, `cargo run --release`, or `rustc -O`. (Alternatively, `rustc` has multiple other options for optimized builds, such as `-C opt-level`.) This will typically take longer than a dev build because of the additional optimizations.

Consider the following final line of output from a `cargo build --release` run.

```
Finished release [optimized] target(s) in 1m 01s
```

The `[optimized]` indicates that a release build has been produced. The compiled code will be placed in the `target/release/` directory. `cargo run --release` will run the release build.

See the Cargo profile documentation for more details about the differences between dev builds (which use the `dev` profile) and release builds (which use the `release` profile).

Codegen Units

The Rust compiler splits crates into multiple codegen units to parallelize (and thus speed up) compilation. However, this might cause it to miss some potential optimizations. If you want to potentially improve runtime performance at the cost of larger compile time, you can set the number of units to one:

```
[profile.release]
codegen-units = 1
```

Example.

Reducing the number of codegen units can also result in a smaller compiled binary.

Link-time Optimization

Link-time optimization (LTO) is a whole-program optimization technique that can improve runtime performance by 10-20% or more, at the cost of increased build times. For any individual Rust program it is easy to see if the runtime versus compile-time trade-off is worthwhile.

The simplest way to try LTO is to add the following lines to the `Cargo.toml` file and do a release build.

```
[profile.release]
lto = true
```

This will result in “fat” LTO, which optimizes across all crates in the dependency graph.

Alternatively, use `lto = "thin"` in `Cargo.toml` to use “thin” LTO, which is a less aggressive form of LTO that often works as well as “fat” LTO without increasing build times as much.

See the Cargo LTO documentation for more details about the `lto` setting, and about enabling specific settings for different profiles.

CPU Specific Instructions

If you do not care that much about the compatibility of your binary on older (or other types of) processors, you can tell the compiler to generate the newest (and potentially fastest) instructions specific to a certain CPU architecture.

For example, if you pass `-C target-cpu=native` to rustc, it will use the best instructions for your current CPU:

```
$ RUSTFLAGS="-C target-cpu=native" cargo build --release
```

This can have a large effect, especially if the compiler finds vectorization opportunities in your code.

As of July 2022, on M1 Macs there is an issue where using `-C target-cpu=native` doesn’t detect all the CPU features. You need to use `-C target-cpu=apple-m1` instead.

If you are unsure whether `-C target-cpu=native` is working optimally, compare the output of `rustc --print cfg` and `rustc --print cfg -C target-cpu=native` to see if the CPU features are being detected correctly in the latter case. If not, you can use `-C target-feature` to target specific features.

Abort on `panic!`

If you do not need to catch or unwind panics, you can tell the compiler to simply abort on panics. This might reduce binary size and increase performance slightly:

```
[profile.release]
panic = "abort"
```

Profile-guided Optimization

Profile-guided optimization (PGO) is a compilation model where you compile your program, run it on sample data while collecting profiling data, and then use that profiling data to guide a second compilation of the program. **Example**.

It is an advanced technique that takes some effort to set up, but is worthwhile in some cases. See the rustc PGO documentation for details. Also, the `cargo-pgo` command makes it easier to use PGO (and BOLT, which is similar) to optimize Rust binaries.

Linting

Clippy is a collection of lints to catch common mistakes in Rust code. It is an excellent tool to run on Rust code in general. It can also help with performance, because a number of the lints relate to code patterns that can cause sub-optimal performance.

Given that automated detection of problems is preferable to manual detection, the rest of this book will not mention performance problems that Clippy detects by default.

Basics

Once installed, it is easy to run:

```
cargo clippy
```

The full list of performance lints can be seen by visiting the lint list and deselecting all the lint groups except for "Perf".

As well as making the code faster, the performance lint suggestions usually result in code that is simpler and more idiomatic, so they are worth following even for code that is not executed frequently.

Conversely, some non-performance lint suggestions can improve performance. For example, the `ptr_arg` style lint suggests changing various container arguments to slices, such as changing `&mut Vec<T>` arguments to `&mut [T]`. The primary motivation here is that a slice gives a more flexible API, but it may also result in faster code due to less indirection and better optimization opportunities for the compiler. **Example**.

Disallowing Types

In the following chapters we will see that it is sometimes worth avoiding certain standard library types in favour of alternatives that are faster. If you decide to use these alternatives, it is easy to accidentally use the standard library types in some places by mistake.

You can use Clippy's `disallowed_types` lint, added in Rust 1.55, to avoid this problem. For example, to disallow the use of the standard hash tables (for reasons explained in the Hashing section) add a `clippy.toml` file to your code with the following lines.

```
disallowed-types = ["std::collections::HashMap", "std::collections::HashSet"]
```

Then add the following declaration to your Rust code.

```
#![warn(clippy::disallowed_types)]
```

This is necessary because `disallowed_types` is (at the time of writing) a "nursery" (under development) lint. This may change in the future.

Profiling

When optimizing a program, you also need a way to determine which parts of the program are "hot" (executed frequently enough to affect runtime) and worth modifying. This is best done via profiling.

Profilers

There are many different profilers available, each with their strengths and weaknesses. The following is an incomplete list of profilers that have been used successfully on Rust programs.

- perf is a general-purpose profiler that uses hardware performance counters. Hotspot and Firefox Profiler are good for viewing data recorded by perf. It works on Linux.
- Instruments is a general-purpose profiler that comes with Xcode on macOS.
- AMD μProf is a general-purpose profiler. It works on Windows and Linux.
- samply is a sampling profiler that produces profiles that can be viewed in the Firefox Profiler. It works on Mac and Linux.
- flamegraph is a Cargo command that uses perf/DTrace to profile your code and then displays the results in a flame graph. It works on Linux and all platforms that support DTrace (macOS, FreeBSD, NetBSD, and possibly Windows).
- Cachegrind & Callgrind give global, per-function, and per-source-line instruction counts and simulated cache and branch prediction data. They work on Linux and some other Unixes.
- DHAT is good for finding which parts of the code are causing a lot of allocations, and for giving insight into peak memory usage. It can also be used to identify hot calls to `memcpy`. It works on Linux and some other Unixes. dhat-rs is an experimental alternative that is a little less powerful and requires minor changes to your Rust program, but works on all platforms.
- heaptrack and bytehound are heap profiling tools. They work on Linux.
- `counts` supports ad hoc profiling, which combines the use of `eprintln!` statement with frequency-based post-processing, which is good for getting domain-specific insights into parts of your code. It works on all platforms.
- Coz performs *causal profiling* to measure optimization potential, and has Rust support via coz-rs. It works on Linux.

Debug Info

To profile a release build effectively you might need to enable source line debug info. To do this, add the following lines to your `Cargo.toml` file:

```
[profile.release]
debug = 1
```

See the Cargo documentation for more details about the `debug` setting.

Unfortunately, even after doing the above step you won't get detailed profiling information for standard library code. This is because shipped versions of the Rust standard library are not built with debug info. To remedy this, you can build your own version of the compiler and standard library, following these instructions, and adding the following lines to the `config.toml` file:

```
[rust]
debuginfo-level = 1
```

This is a hassle, but may be worth the effort in some cases.

Symbol Demangling

Rust uses a mangling scheme to encode function names in compiled code. If a profiler is unaware of this scheme, its output may contain symbol names beginning with `_ZN` or `_R`, such as `_ZN3foo3barE` or `_ZN28_$u7b$$u7b$closure$u7d$$u7d$E` or `_RMCsno73SFvQKx_1cINtB0_3StrKRe616263_E`

Names like these can be manually demangled using `rustfilt`.

Inlining

Entry to and exit from hot, uninlined functions often accounts for a non-trivial fraction of execution time. Inlining these functions can provide small but easy speed wins.

There are four inline attributes that can be used on Rust functions.

- **None**. The compiler will decide itself if the function should be inlined. This will depend on factors such as the optimization level and the size of the function. Non-generic functions will never be inlined across crate boundaries unless link-time optimization is used; generic functions might be.
- `#[inline]`. This suggests that the function should be inlined, including across crate boundaries.
- `#[inline(always)]`. This strongly suggests that the function should be inlined, including across crate boundaries.
- `#[inline(never)]`. This strongly suggests that the function should not be inlined.

Inline attributes do not guarantee that a function is inlined or not inlined, but in practice `#[inline(always)]` will cause inlining in all but the most exceptional cases.

Simple Cases

The best candidates for inlining are (a) functions that are very small, or (b) functions that have a single call site. The compiler will often inline these functions itself even without an inline attribute. But the compiler cannot always make the best choices, so attributes are sometimes needed. **Example 1**, **Example 2**, **Example 3**, **Example 4**, **Example 5**.

Cachegrind is a good profiler for determining if a function is inlined. When looking at Cachegrind's output, you can tell that a function has been inlined if (and only if) its first and last lines are *not* marked with event counts. For example:

```
      .  #[inline(always)]
      .  fn inlined(x: u32, y: u32) -> u32 {
700,000      eprintln!("inlined: {} + {}", x, y);
200,000      x + y
      .  }
      .
      .  #[inline(never)]
400,000  fn not_inlined(x: u32, y: u32) -> u32 {
700,000      eprintln!("not_inlined: {} + {}", x, y);
200,000      x + y
200,000  }
```

You should measure again after adding inline attributes, because the effects can be unpredictable. Sometimes it has no effect because a nearby function that was previously inlined no longer is. Sometimes it slows the code down. Inlining can also affect compile times, especially cross-crate inlining which involves duplicating internal representations of the functions.

Harder Cases

Sometimes you have a function that is large and has multiple call sites, but only one call site is hot. You would like to inline the hot call site for speed, but not inline the cold call sites to avoid unnecessary code bloat. The way to handle this is to split the function always-inlined and never-inlined variants, with the latter calling the former.

For example, this function:

```rust
fn my_function() {
    one();
    two();
    three();
}
```

Would become these two functions:

```rust
// Use this at the hot call site.
#[inline(always)]
fn inlined_my_function() {
    one();
    two();
    three();
}

// Use this at the cold call sites.
#[inline(never)]
fn uninlined_my_function() {
    inlined_my_function();
}
```

Example 1, **Example 2**.

Hashing

`HashSet` and `HashMap` are two widely-used types. The default hashing algorithm is not specified, but at the time of writing the default is an algorithm called SipHash 1-3. This algorithm is high quality—it provides high protection against collisions—but is relatively slow, particularly for short keys such as integers.

If profiling shows that hashing is hot, and HashDoS attacks are not a concern for your application, the use of hash tables with faster hash algorithms can provide large speed wins.

- `rustc-hash` provides `FxHashSet` and `FxHashMap` types that are drop-in replacements for `HashSet` and `HashMap`. Its hashing algorithm is low-quality but very fast, especially for integer keys, and has been found to out-perform all other hash algorithms within rustc. (`fxhash` is an older, less well maintained implementation of the same algorithm and types.)
- `fnv` provides `FnvHashSet` and `FnvHashMap` types. Its hashing algorithm is higher quality than `rustc-hash`'s but a little slower.
- `ahash` provides `AHashSet` and `AHashMap`. Its hashing algorithm can take advantage of AES instruction support that is available on some processors.

If hashing performance is important in your program, it is worth trying more than one of these alternatives. For example, the following results were seen in rustc.

- The switch from `fnv` to `fxhash` gave speedups of up to 6%.
- An attempt to switch from `fxhash` to `ahash` resulted in slowdowns of 1-4%.
- An attempt to switch from `fxhash` back to the default hasher resulted in slowdowns ranging from 4-84%!

If you decide to universally use one of the alternatives, such as `FxHashSet`/`FxHashMap`, it is easy to accidentally use `HashSet`/`HashMap` in some places. You can use Clippy to avoid this problem.

Some types don't need hashing. For example, you might have a newtype that wraps an integer and the integer values are random, or close to random. For such a type, the distribution of the hashed values won't be that different to the distribution of the values themselves. In this case the `nohash_hasher` crate can be useful.

Hash function design is a complex topic and is beyond the scope of this book. The `ahash` documentation has a good discussion.

Heap Allocations

Heap allocations are moderately expensive. The exact details depend on which allocator is in use, but each allocation (and deallocation) typically involves acquiring a global lock, doing some non-trivial data structure manipulation, and possibly executing a system call. Small allocations are not necessarily cheaper than large allocations. It is worth understanding which Rust data structures and operations cause allocations, because avoiding them can greatly improve performance.

The Rust Container Cheat Sheet has visualizations of common Rust types, and is an excellent companion to the following sections.

Profiling

If a general-purpose profiler shows `malloc`, `free`, and related functions as hot, then it is likely worth trying to reduce the allocation rate and/or using an alternative allocator.

DHAT is an excellent profiler to use when reducing allocation rates. It works on Linux and some other Unixes. It precisely identifies hot allocation sites and their allocation rates. Exact results will vary, but experience with rustc has shown that reducing allocation rates by 10 allocations per million instructions executed can have measurable performance improvements (e.g. ~1%).

Here is some example output from DHAT.

```
AP 1.1/25 (2 children) {
  Total:     54,533,440 bytes (4.02%, 2,714.28/Minstr) in 458,839 blocks (7.72%,
22.84/Minstr), avg size 118.85 bytes, avg lifetime 1,127,259,403.64 instrs (5.61% of
program duration)
  At t-gmax: 0 bytes (0%) in 0 blocks (0%), avg size 0 bytes
  At t-end:  0 bytes (0%) in 0 blocks (0%), avg size 0 bytes
  Reads:     15,993,012 bytes (0.29%, 796.02/Minstr), 0.29/byte
  Writes:    20,974,752 bytes (1.03%, 1,043.97/Minstr), 0.38/byte
  Allocated at {
    #1: 0x95CACC9: alloc (alloc.rs:72)
    #2: 0x95CACC9: alloc (alloc.rs:148)
    #3: 0x95CACC9:
reserve_internal<syntax::tokenstream::TokenStream,alloc::alloc::Global>
(raw_vec.rs:669)
    #4: 0x95CACC9: reserve<syntax::tokenstream::TokenStream,alloc::alloc::Global>
(raw_vec.rs:492)
    #5: 0x95CACC9: reserve<syntax::tokenstream::TokenStream> (vec.rs:460)
    #6: 0x95CACC9: push<syntax::tokenstream::TokenStream> (vec.rs:989)
    #7: 0x95CACC9: parse_token_trees_until_close_delim (tokentrees.rs:27)
    #8: 0x95CACC9: syntax::parse::lexer::tokentrees::<impl
syntax::parse::lexer::StringReader<'a>>::parse_token_tree (tokentrees.rs:81)
  }
}
```

It is beyond the scope of this book to describe everything in this example, but it should be clear that DHAT gives a wealth of information about allocations, such as where and how often they happen, how big they are, how long they live for, and how often they are accessed.

`Box`

`Box` is the simplest heap-allocated type. A `Box<T>` value is a `T` value that is allocated on the heap.

It is sometimes worth boxing one or more fields in a struct or enum fields to make a type smaller. (See the Type Sizes chapter for more about this.)

Other than that, `Box` is straightforward and does not offer much scope for optimizations.

`Rc`/`Arc`

`Rc`/`Arc` are similar to `Box`, but the value on the heap is accompanied by two reference counts. They allow value sharing, which can be an effective way to reduce memory usage.

However, if used for values that are rarely shared, they can increase allocation rates by heap allocating values that might otherwise not be heap-allocated. **Example**.

Unlike `Box`, calling `clone` on an `Rc`/`Arc` value does not involve an allocation. Instead, it merely increments a reference count.

`Vec`

`Vec` is a heap-allocated type with a great deal of scope for optimizing the number of allocations, and/or minimizing the amount of wasted space. To do this requires understanding how its elements are stored.

A `Vec` contains three words: a length, a capacity, and a pointer. The pointer will point to heap-allocated memory if the capacity is nonzero and the element size is nonzero; otherwise, it will not point to allocated memory.

Even if the `Vec` itself is not heap-allocated, the elements (if present and nonzero-sized) always will be. If nonzero-sized elements are present, the memory holding those elements may be larger than necessary, providing space for additional future elements. The number of elements present is the length, and the number of elements that could be held without reallocating is the capacity.

When the vector needs to grow beyond its current capacity, the elements will be copied into a larger heap allocation, and the old heap allocation will be freed.

`Vec` Growth

A new, empty `Vec` created by the common means (`vec![]` or `Vec::new` or `Vec::default`) has a length and capacity of zero, and no heap allocation is required. If you repeatedly push individual elements onto the end of the `Vec`, it will periodically reallocate. The growth strategy is not specified, but at the time of writing it uses a quasi-doubling strategy resulting in the following capacities: 0, 4, 8, 16, 32, 64, and so on. (It skips directly from 0 to 4, instead of going via 1 and 2, because this avoids many allocations in practice.) As a vector grows, the frequency of reallocations will decrease exponentially, but the amount of possibly-wasted excess capacity will increase exponentially.

This growth strategy is typical for growable data structures and reasonable in the general case, but if you know in advance the likely length of a vector you can often do better. If you have a hot vector allocation site (e.g. a hot `Vec::push` call), it is worth using `eprintln!` to print the vector length at that site and then doing some post-processing (e.g. with `counts`) to determine the length distribution. For example, you might have many short vectors, or you might have a smaller number of very long vectors, and the best way to optimize the allocation site will vary accordingly.

Short `Vec`s

If you have many short vectors, you can use the `SmallVec` type from the `smallvec` crate. `SmallVec<[T; N]>` is a drop-in replacement for `Vec` that can store `N` elements within the `SmallVec` itself, and then switches to a heap allocation if the number of elements exceeds that. (Note also that `vec![]` literals must be replaced with `smallvec![]` literals.) **Example 1**, **Example 2**.

`SmallVec` reliably reduces the allocation rate when used appropriately, but its use does not guarantee improved performance. It is slightly slower than `Vec` for normal operations because it must always check if the elements are heap-allocated or not. Also, If `N` is high or `T` is large, then the `SmallVec<[T; N]>` itself can be larger than `Vec<T>`, and copying of `SmallVec` values will be slower. As always, benchmarking is required to confirm that an optimization is effective.

If you have many short vectors *and* you precisely know their maximum length, `ArrayVec` from the `arrayvec` crate is a better choice than `SmallVec`. It does not require the fallback to heap allocation, which makes it a little faster. **Example**.

Longer `Vec`s

If you know the minimum or exact size of a vector, you can reserve a specific capacity with `Vec::with_capacity`, `Vec::reserve`, or `Vec::reserve_exact`. For example, if you know a vector will grow to have at least 20 elements, these functions can immediately provide a vector with a capacity of at least 20 using a single allocation, whereas pushing the items one at a time would result in four allocations (for capacities of 4, 8, 16, and 32). **Example**.

If you know the maximum length of a vector, the above functions also let you not allocate excess space unnecessarily. Similarly, `Vec::shrink_to_fit` can be used to minimize wasted space, but note that it may cause a reallocation.

`String`

A `String` contains heap-allocated bytes. The representation and operation of `String` are very similar to that of `Vec<u8>`. Many `Vec` methods relating to growth and capacity have equivalents for `String`, such as `String::with_capacity`.

The `SmallString` type from the `smallstr` crate is similar to the `SmallVec` type.

The `String` type from the `smartstring` crate is a drop-in replacement for `String` that avoids heap allocations for strings with less than three words' worth of characters. On 64-bit platforms, this is any string that is less than 24 bytes, which includes all strings containing 23 or fewer ASCII characters. **Example**.

Note that the `format!` macro produces a `String`, which means it performs an allocation. If you can avoid a `format!` call by using a string literal, that will avoid this allocation. **Example**. `std::format_args` and/or the `lazy_format` crate may help with this.

Hash Tables

`HashSet` and `HashMap` are hash tables. Their representation and operations are similar to those of `Vec`, in terms of allocations: they have a single contiguous heap allocation, holding keys and values, which is reallocated as necessary as the table grows. Many `Vec` methods relating to growth and capacity have equivalents for `HashSet`/`HashMap`, such as `HashSet::with_capacity`.

`clone`

Calling `clone` on a value that contains heap-allocated memory typically involves additional allocations. For example, calling `clone` on a non-empty `Vec` requires a new allocation for the elements (but note that the capacity of the new `Vec` might not be the same as the capacity of the original `Vec`). The exception is `Rc`/`Arc`, where a `clone` call just increments the reference count.

`clone_from` is an alternative to `clone`. `a.clone_from(&b)` is equivalent to `a = b.clone()` but may avoid unnecessary allocations. For example, if you want to clone one `Vec` over the top of an existing `Vec`, the existing `Vec`'s heap allocation will be reused if possible, as the following example shows.

```rust
let mut v1: Vec<u32> = Vec::with_capacity(99);
let v2: Vec<u32> = vec![1, 2, 3];
v1.clone_from(&v2); // v1's allocation is reused
assert_eq!(v1.capacity(), 99);
```

Although `clone` usually causes allocations, it is a reasonable thing to use in many circumstances and can often make code simpler. Use profiling data to see which `clone` calls are hot and worth taking the effort to avoid.

Sometimes Rust code ends up containing unnecessary `clone` calls, due to (a) programmer error, or (b) changes in the code that render previously-necessary `clone` calls unnecessary. If you see a hot `clone` call that does not seem necessary, sometimes it can simply be removed. **Example 1**, **Example 2**, **Example 3**.

`to_owned`

`ToOwned::to_owned` is implemented for many common types. It creates owned data from borrowed data, usually by cloning, and therefore often causes heap allocations. For example, it can be used to create a `String` from a `&str`.

Sometimes `to_owned` calls (and related calls such as `clone` and `to_string`) can be avoided by storing a reference to borrowed data in a struct rather than an owned copy. This requires lifetime annotations on the struct, complicating the code, and should only be done when profiling and benchmarking shows that it is worthwhile. **Example**.

`Cow`

Sometimes code deals with a mixture of borrowed and owned data. Imagine a vector of error messages, some of which are static string literals and some of which are constructed with `format!`. The obvious representation is `Vec<String>`, as the following example shows.

```
let mut errors: Vec<String> = vec![];
errors.push("something went wrong".to_string());
errors.push(format!("something went wrong on line {}", 100));
```

That requires a `to_string` call to promote the static string literal to a `String`, which incurs an allocation.

Instead you can use the `Cow` type, which can hold either borrowed or owned data. A borrowed value `x` is wrapped with `Cow::Borrowed(x)`, and an owned value `y` is wrapped with `Cow::Owned(y)`. `Cow` also implements the `From<T>` trait for various string, slice, and path types, so you can usually use `into` as well. (Or `Cow::from`, which is longer but results in more readable code, because it makes the type clearer.) The following example puts all this together.

```
use std::borrow::Cow;
let mut errors: Vec<Cow<'static, str>> = vec![];
errors.push(Cow::Borrowed("something went wrong"));
errors.push(Cow::Owned(format!("something went wrong on line {}", 100)));
errors.push(Cow::from("something else went wrong"));
errors.push(format!("something else went wrong on line {}", 101).into());
```

`errors` now holds a mixture of borrowed and owned data without requiring any extra allocations. This example involves `&str`/`String`, but other pairings such as `&[T]`/`Vec<T>` and `&Path`/`PathBuf` are also possible.

Example 1, **Example 2**.

All of the above applies if the data is immutable. But `Cow` also allows borrowed data to be promoted to owned data if it needs to be mutated. `Cow::to_mut` will obtain a mutable reference to an owned value, cloning if necessary. This is called "clone-on-write", which is where the name `Cow` comes from.

This clone-on-write behaviour is useful when you have some borrowed data, such as a `&str`, that is mostly read-only but occasionally needs to be modified.

Example 1, **Example 2**.

Finally, because `Cow` implements `Deref`, you can call methods directly on the data it encloses.

`Cow` can be fiddly to get working, but it is often worth the effort.

Reusing Collections

Sometimes you need to build up a collection such as a `Vec` in stages. It is usually better to do this by modifying a single `Vec` than by building multiple `Vec`s and then combining them.

For example, if you have a function `do_stuff` that produces a `Vec` that might be called multiple times:

```rust
fn do_stuff(x: u32, y: u32) -> Vec<u32> {
    vec![x, y]
}
```

It might be better to instead modify a passed-in `Vec`:

```rust
fn do_stuff(x: u32, y: u32, vec: &mut Vec<u32>) {
    vec.push(x);
    vec.push(y);
}
```

Sometimes it is worth keeping around a “workhorse” collection that can be reused. For example, if a `Vec` is needed for each iteration of a loop, you could declare the `Vec` outside the loop, use it within the loop body, and then call `clear` at the end of the loop body (to empty the `Vec` without affecting its capacity). This avoids allocations at the cost of obscuring the fact that each iteration’s usage of the `Vec` is unrelated to the others. **Example 1**, **Example 2**.

Similarly, it is sometimes worth keeping a “workhorse” collection within a struct, to be reused in one or more methods that are called repeatedly.

Using an Alternative Allocator

Another option for improving the performance of allocation-heavy Rust programs is to replace the default (system) allocator with an alternative allocator. The exact effect will depend on the individual program and the alternative allocator chosen, but large improvements in speed and very large reductions in memory usage have been seen in practice. The effect will also vary across platforms, because each platform’s system allocator has its own strengths and weaknesses. The use of an alternative allocator can also affect binary size.

One popular alternative allocator for Linux and Mac is jemalloc, usable via the `tikv-jemallocator` crate. To use it, add a dependency to your `Cargo.toml` file:

```toml
[dependencies]
tikv-jemallocator = "0.4.0"
```

Then add the following somewhere in your Rust code:

```rust
#[global_allocator]
static GLOBAL: tikv_jemallocator::Jemalloc = tikv_jemallocator::Jemalloc;
```

`tikv-jemallocator` is a fork of the `jemallocator` crate. As of December 2021, `tikv-jemallocator` uses a jemalloc version that is newer and has better performance than the jemalloc version used by `jemallocator`.

Another alternative allocator that works on many platforms is mimalloc, usable via the `mimalloc` crate.

Avoiding Regressions

To ensure the number and/or size of allocations done by your code doesn’t increase unintentionally, you can use the *heap usage testing* feature of dhat-rs to write tests that check particular code snippets allocate the expected amount of heap memory.

Type Sizes

Shrinking oft-instantiated types can help performance.

For example, if memory usage is high, a heap profiler like DHAT can identify the hot allocation points and the types involved. Shrinking these types can reduce peak memory usage, and possibly improve performance by reducing memory traffic and cache pressure.

Furthermore, Rust types that are larger than 128 bytes are copied with `memcpy` rather than inline code. If `memcpy` shows up in non-trivial amounts in profiles, DHAT's "copy profiling" mode will tell you exactly where the hot `memcpy` calls are and the types involved. Shrinking these types to 128 bytes or less can make the code faster by avoiding `memcpy` calls and reducing memory traffic.

Measuring Type Sizes

`std::mem::size_of` gives the size of a type, in bytes, but often you want to know the exact layout as well. For example, an enum might be surprisingly large due to a single outsized variant.

The `-Zprint-type-sizes` option does exactly this. It isn't enabled on release versions of rustc, so you'll need to use a nightly version of rustc. Here is one possible invocation via Cargo:

```
RUSTFLAGS=-Zprint-type-sizes cargo +nightly build --release
```

And here is a possible invocation of rustc:

```
rustc +nightly -Zprint-type-sizes input.rs
```

It will print out details of the size, layout, and alignment of all types in use. For example, for this type:

```rust
enum E {
    A,
    B(i32),
    C(u64, u8, u64, u8),
    D(Vec<u32>),
}
```

it prints the following, plus information about a few built-in types.

```
print-type-size type: `E`: 32 bytes, alignment: 8 bytes
print-type-size     discriminant: 1 bytes
print-type-size     variant `D`: 31 bytes
print-type-size         padding: 7 bytes
print-type-size         field `.0`: 24 bytes, alignment: 8 bytes
print-type-size     variant `C`: 23 bytes
print-type-size         field `.1`: 1 bytes
print-type-size         field `.3`: 1 bytes
print-type-size         padding: 5 bytes
print-type-size         field `.0`: 8 bytes, alignment: 8 bytes
print-type-size         field `.2`: 8 bytes
print-type-size     variant `B`: 7 bytes
print-type-size         padding: 3 bytes
print-type-size         field `.0`: 4 bytes, alignment: 4 bytes
print-type-size     variant `A`: 0 bytes
```

The output shows the following.

- The size and alignment of the type.
- For enums, the size of the discriminant.
- For enums, the size of each variant (sorted from largest to smallest).
- The size, alignment, and ordering of all fields. (Note that the compiler has reordered variant `C`'s fields to minimize the size of `E`.)
- The size and location of all padding.

Alternatively, the top-type-sizes crate can be used to display the output in a more compact form.

Once you know the layout of a hot type, there are multiple ways to shrink it.

Field Ordering

The Rust compiler automatically sorts the fields in struct and enums to minimize their sizes (unless the `#[repr(C)]` attribute is specified), so you do not have to worry about field ordering. But there are other ways to minimize the size of hot types.

Smaller Enums

If an enum has an outsized variant, consider boxing one or more fields. For example, you could change this type:

```rust
type LargeType = [u8; 100];
enum A {
    X,
    Y(i32),
    Z(i32, LargeType),
}
```

to this:

```rust
enum A {
    X,
    Y(i32),
    Z(Box<(i32, LargeType)>),
}
```

This reduces the type size at the cost of requiring an extra heap allocation for the `A::Z` variant. This is more likely to be a net performance win if the `A::Z` variant is relatively rare. The `Box` will also make `A::Z` slightly less ergonomic to use, especially in `match` patterns. **Example 1**, **Example 2**, **Example 3**, **Example 4**, **Example 5**, **Example 6**.

Smaller Integers

It is often possible to shrink types by using smaller integer types. For example, while it is most natural to use `usize` for indices, it is often reasonable to stores indices as `u32`, `u16`, or even `u8`, and then coerce to `usize` at use points. **Example 1**, **Example 2**.

Boxed Slices

Rust vectors contain three words: a length, a capacity, and a pointer. If you have a vector that is unlikely to be changed in the future, you can convert it to a *boxed slice* with `Vec::into_boxed_slice`. A boxed slice contains only two words, a length and a pointer. Any excess element capacity is dropped, which may cause a reallocation.

```
let v: Vec<u32> = vec![1, 2, 3];
assert_eq!(size_of_val(&v), 3 * size_of::<usize>());

let bs: Box<[u32]> = v.into_boxed_slice();
assert_eq!(size_of_val(&bs), 2 * size_of::<usize>());
```

The boxed slice can be converted back to a vector with `slice::into_vec` without any cloning or a reallocation.

`ThinVec`

An alternative to boxed slices is `ThinVec`, from the `thin_vec` crate. It is functionally equivalent to `Vec`, but stores the length and capacity in the same allocation as the elements (if there are any). This means that `size_of::<ThinVec<T>>` is only one word.

`ThinVec` is a good choice within oft-instantiated types for vectors that are often empty. It can also be used to shrink the largest variant of an enum, if that variant contains a `Vec`.

Avoiding Regressions

If a type is hot enough that its size can affect performance, it is a good idea to use a static assertion to ensure that it does not accidentally regress. The following example uses a macro from the `static_assertions` crate.

```
  // This type is used a lot. Make sure it doesn't unintentionally get bigger.
  #[cfg(target_arch = "x86_64")]
  static_assertions::assert_eq_size!(HotType, [u8; 64]);
```

The `cfg` attribute is important, because type sizes can vary on different platforms. Restricting the assertion to `x86_64` (which is typically the most widely-used platform) is likely to be good enough to prevent regressions in practice.

Standard Library Types

It is worth reading through the documentation for common standard library types—such as `Box`, `Vec`, `Option`, `Result`, and `Rc`/`Arc`—to find interesting functions that can sometimes be used to improve performance.

It is also worth knowing about high-performance alternatives to standard library types, such as `Mutex`, `RwLock`, `Condvar`, and `Once`.

`Box`

The expression `Box::default()` has the same effect as `Box::new(T::default())` but can be faster because the compiler can create the value directly on the heap, rather than constructing it on the stack and then copying it over. **Example**.

`Vec`

The best way to create a zero-filled `Vec` of length `n` is with `vec![0; n]`. This is simple and probably as fast or faster than alternatives, such as using `resize`, `extend`, or anything involving `unsafe`, because it can use OS assistance.

`Vec::remove` removes an element at a particular index and shifts all subsequent elements one to the left, which makes it O(n). `Vec::swap_remove` replaces an element at a particular index with the final element, which does not preserve ordering, but is O(1).

`Vec::retain` efficiently removes multiple items from a `Vec`. There is an equivalent method for other collection types such as `String`, `HashSet`, and `HashMap`.

`Option` and `Result`

`Option::ok_or` converts an `Option` into a `Result`, and is passed an `err` parameter that is used if the `Option` value is `None`. `err` is computed eagerly. If its computation is expensive, you should instead use `Option::ok_or_else`, which computes the error value lazily via a closure. For example, this:

```rust
let r = o.ok_or(expensive()); // always evaluates `expensive()`
```

should be changed to this:

```rust
let r = o.ok_or_else(|| expensive()); // evaluates `expensive()` only when needed
```

Example.

There are similar alternatives for `Option::map_or`, `Option::unwrap_or`, `Result::or`, `Result::map_or`, and `Result::unwrap_or`.

`Rc`/`Arc`

`Rc::make_mut` / `Arc::make_mut` provide clone-on-write semantics. They make a mutable reference to an `Rc` / `Arc`. If the refcount is greater than one, they will `clone` the inner value to ensure unique ownership; otherwise, they will modify the original value. They are not needed often, but they can be extremely useful on occasion. **Example 1**, **Example 2**.

`Mutex`, `RwLock`, `Condvar`, and `Once`

The `parking_lot` crate provides alternative implementations of these synchronization types. The APIs and semantics of the `parking_lot` types are similar but not identical to those of the equivalent types in the standard library.

The `parking_lot` versions used to be reliably smaller, faster, and more flexible than those in the standard library, but the standard library versions have greatly improved on some platforms. So you should measure before switching to `parking_lot`.

If you decide to universally use the `parking_lot` types it is easy to accidentally use the standard library equivalents in some places. You can use Clippy to avoid this problem.

Iterators

`collect` and `extend`

`Iterator::collect` converts an iterator into a collection such as `Vec`, which typically requires an allocation. You should avoid calling `collect` if the collection is then only iterated over again.

For this reason, it is often better to return an iterator type like `impl Iterator<Item=T>` from a function than a `Vec<T>`. Note that sometimes additional lifetimes are required on these return types, as this blog post explains. **Example**.

Similarly, you can use `extend` to extend an existing collection (such as a `Vec`) with an iterator, rather than collecting the iterator into a `Vec` and then using `append`.

Finally, when you write an iterator it is often worth implementing the `Iterator::size_hint` or `ExactSizeIterator::len` method, if possible. `collect` and `extend` calls that use the iterator may then do fewer allocations, because they have advance information about the number of elements yielded by the iterator.

Chaining

`chain` can be very convenient, but it can also be slower than a single iterator. It may be worth avoiding for hot iterators, if possible. **Example**.

Similarly, `filter_map` may be faster than using `filter` followed by `map`.

Chunks

When a chunking iterator is required and the chunk size is known to exactly divide the slice length, use the faster `slice::chunks_exact` instead of `slice::chunks`.

When the chunk size is not known to exactly divide the slice length, it can still be faster to use `slice::chunks_exact` in combination with either `ChunksExact::remainder` or manual handling of excess elements. **Example 1**, **Example 2**.

The same is true for related iterators:

- `slice::rchunks`, `slice::rchunks_exact`, and `RChunksExact::remainder`;
- `slice::chunks_mut`, `slice::chunks_exact_mut`, and `ChunksExactMut::into_remainder`;
- `slice::rchunks_mut`, `slice::rchunks_exact_mut`, and `RChunksExactMut::into_remainder`.

Bounds Checks

By default, accesses to container types such as slices and vectors involve bounds checks in Rust. These can affect performance, e.g. within hot loops, though less often than you might expect.

There are several safe ways to change code so that the compiler knows about container lengths and can optimize away bounds checks.

- Replace direct element accesses in a loop by using iteration.
- Instead of indexing into a `Vec` within a loop, make a slice of the `Vec` before the loop and then index into the slice within the loop.
- Add assertions on the ranges of index variables. **Example 1**, **Example 2**.

Getting these to work can be tricky. The Bounds Check Cookbook goes into more detail on this topic.

As a last resort, there are the unsafe methods `get_unchecked` and `get_unchecked_mut`.

I/O

Locking

Rust's `print!` and `println!` macros lock stdout on every call. If you have repeated calls to these macros it may be better to lock stdout manually.

For example, change this code:

```
for line in lines {
    println!("{}", line);
}
```

to this:

```
use std::io::Write;
let mut stdout = std::io::stdout();
let mut lock = stdout.lock();
for line in lines {
    writeln!(lock, "{}", line)?;
}
// stdout is unlocked when `lock` is dropped
```

stdin and stderr can likewise be locked when doing repeated operations on them.

Buffering

Rust file I/O is unbuffered by default. If you have many small and repeated read or write calls to a file or network socket, use `BufReader` or `BufWriter`. They maintain an in-memory buffer for input and output, minimizing the number of system calls required.

For example, change this unbuffered writer code:

```
use std::io::Write;
let mut out = std::fs::File::create("test.txt").unwrap();
for line in lines {
    writeln!(out, "{}", line)?;
}
```

to this:

```
use std::io::{BufWriter, Write};
let mut out = BufWriter::new(std::fs::File::create("test.txt")?);
for line in lines {
    writeln!(out, "{}", line)?;
}
out.flush()?;
```

Example 1, **Example 2**.

The explicit call to `flush` is not strictly necessary, as flushing will happen automatically when `out` is dropped. However, in that case any error that occurs on flushing will be ignored, whereas an explicit flush will make that error explicit.

Forgetting to buffer is more common when writing. Both unbuffered and buffered writers implement the `Write` trait, which means the code for writing to an unbuffered writer and a buffered writer is much the same. In contrast, unbuffered readers implement the `Read` trait but buffered readers implement the `BufRead` trait, which means the code for reading from an unbuffered reader and a buffered reader is different. For example, it is difficult to read a file line by line with an unbuffered reader, but it is trivial with a buffered reader by using `BufRead::read_line` or `BufRead::lines`. For this reason, it is hard to write an example for readers like the one above for writers, where the before and after versions are so similar.

Finally, note that buffering also works with stdout, so you might want to combine manual locking *and* buffering when making many writes to stdout.

Reading Input as Raw Bytes

The built-in String type uses UTF-8 internally, which adds a small, but nonzero overhead caused by UTF-8 validation when you read input into it. If you just want to process input bytes without worrying about UTF-8 (for example if you handle ASCII text), you can use `BufRead::read_until`.

There are also dedicated crates for reading byte-oriented lines of data and working with byte strings.

Logging and Debugging

Sometimes logging code or debugging code can slow down a program significantly. Either the logging/debugging code itself is slow, or data collection code that feeds into logging/debugging code is slow. Make sure that no unnecessary work is done for logging/debugging purposes when logging/debugging is not enabled. **Example 1**, **Example 2**.

Note that `assert!` calls always run, but `debug_assert!` calls only run in dev builds. If you have an assertion that is hot but is not necessary for safety, consider making it a `debug_assert!`. **Example 1**, **Example 2**.

Wrapper Types

Rust has a variety of “wrapper” types, such as `RefCell` and `Mutex`, that provide special behavior for values. Accessing these values can take a non-trivial amount of time. If multiple such values are typically accessed together, it may be better to put them within a single wrapper.

For example, a struct like this:

```
struct S {
    x: Arc<Mutex<u32>>,
    y: Arc<Mutex<u32>>,
}
```

may be better represented like this:

```
struct S {
    xy: Arc<Mutex<(u32, u32)>>,
}
```

Whether or not this helps performance will depend on the exact access patterns of the values. **Example**.

Machine Code

When you have a small piece of very hot code, it may be worth inspecting the generated machine code to see if it has any inefficiencies, such as removable bounds checks. The Compiler Explorer website is an excellent resource when doing this.

Relatedly, the `core::arch` module provides access to architecture-specific intrinsics, many of which relate to SIMD instructions.

Parallelism

Rust provides excellent support for safe parallel programming, which can lead to large performance improvements. There are a variety of ways to introduce parallelism into a program and the best way for any program will depend greatly on its design.

An in-depth treatment of parallelism is beyond the scope of this book. If you are interested in this topic, the documentation for the `rayon` and `crossbeam` crates is a good place to start.

Binary Size

Sometimes you might need to minimize the size of a compiled Rust binary. In that case, you should consult the comprehensive documentation in the excellent `min-sized-rust` repository.

General Tips

The previous sections of this book have discussed Rust-specific techniques. This section gives a brief overview of some general performance principles.

As long as the obvious pitfalls are avoided (e.g. using non-release builds), Rust generally has good performance. Especially if you are used to dynamically-typed languages such as Python and Ruby.

Optimized code is often more complex and takes more effort to write than unoptimized code. For this reason, it is only worth optimizing hot code.

The biggest performance improvements often come from changes to algorithms or data structures, rather than low-level optimizations. **Example 1**, **Example 2**.

Writing code that works well with modern hardware is not always easy, but worth striving for. For example, try to minimize cache misses and branch mispredictions, where possible.

Most optimizations result in small speedups. Although no single small speedup is noticeable, they really add up if you can do enough of them.

Different profilers have different strengths. It is good to use more than one.

When profiling indicates that a function is hot, there are two common ways to speed things up: (a) make the function faster, and/or (b) avoid calling it as much.

It is often easier to eliminate silly slowdowns than it is to introduce clever speedups.

Avoid computing things unless necessary. Lazy/on-demand computations are often a win. **Example 1**, **Example 2**.

Complex general cases can often be avoided by optimistically checking for common special cases that are simpler. **Example 1**, **Example 2**, **Example 3**. In particular, specially handling collections with 0, 1, or 2 elements is often a win when small sizes dominate. **Example 1**, **Example 2**, **Example 3**, **Example 4**.

Similarly, when dealing with repetitive data, it is often possible to use a simple form of data compression, by using a compact representation for common values and then having a fallback to a secondary table for unusual values. **Example 1**, **Example 2**, **Example 3**.

When code deals with multiple cases, measure case frequencies and handle the most common ones first.

When dealing with lookups that involve high locality, it can be a win to put a small cache in front of a data structure.

Optimized code often has a non-obvious structure, which means that explanatory comments are valuable, particularly those that reference profiling measurements. A comment like “99% of the time this vector has 0 or 1 elements, so handle those cases first” can be illuminating.

Compile Times

Although this book is primarily about improving the performance of Rust programs, this section is about reducing the compile times of Rust programs, because that is a related topic of interest to many people.

Linking

A big part of compile time is actually linking time, particularly when rebuilding a program after a small change. It is possible to select a faster linker than the default one.

One option is lld, which is available on Linux and Windows.

To specify lld from the command line, precede your build command with `RUSTFLAGS="-C link-arg=-fuse-ld=lld"`.

To specify lld from a config.toml file (for one or more projects), add these lines:

```
[build]
rustflags = ["-C", "link-arg=-fuse-ld=lld"]
```

lld is not fully supported for use with Rust, but it should work for most use cases on Linux and Windows. There is a GitHub Issue tracking full support for lld.

Another option is mold, which is currently available on Linux and macOS. It is specified in much the same way as lld. Simply substitute `mold` for `lld` in the instructions above.

mold is often faster than lld. It is also much newer and may not work in all cases.

Incremental Compilation

The Rust compiler supports incremental compilation, which avoids redoing work when you recompile a crate. It can greatly speed up compilation, but the compiled binary may be larger and run more slowly. For this reason, it is only enabled by default for dev builds. If you want to enable it for release builds as well, add the following lines to the `Cargo.toml` file.

```
[profile.release]
incremental = true
```

See the Cargo documentation for more details about the `incremental` setting, and about enabling specific settings for different profiles.

Visualization

Cargo has a feature that lets you visualize compilation of your program. Build with this command (1.60 or later):

```
cargo build --timings
```

or this (1.59 or earlier):

```
cargo +nightly build -Ztimings
```

On completion it will print the name of an HTML file. Open that file in a web browser. It contains a Gantt chart that shows the dependencies between the various crates in your program. This shows how much parallelism there is in your crate graph, which can indicate if any large crates that serialize compilation should be broken up. See the documentation for more details on how to read the graphs.

LLVM IR

The Rust compiler uses LLVM for its back-end. LLVM's execution can be a large part of compile times, especially when the Rust compiler's front end generates a lot of IR which takes LLVM a long time to optimize.

These problems can be diagnosed with `cargo llvm-lines`, which shows which Rust functions cause the most LLVM IR to be generated. Generic functions are often the most important ones, because they can be instantiated dozens or even hundreds of times in large programs.

If a generic function causes IR bloat, there are several ways to fix it. The simplest is to just make the function smaller. **Example 1**, **Example 2**.

Another way is to move the non-generic parts of the function into a separate, non-generic function, which will only be instantiated once. Whether this is possible will depend on the details of the generic function. When it is possible, the non-generic function can often be written neatly as an inner function within the generic function, as shown by the code for `std::fs::read`:

```
pub fn read<P: AsRef<Path>>(path: P) -> io::Result<Vec<u8>> {
    fn inner(path: &Path) -> io::Result<Vec<u8>> {
        let mut file = File::open(path)?;
        let size = file.metadata().map(|m| m.len()).unwrap_or(0);
        let mut bytes = Vec::with_capacity(size as usize);
        io::default_read_to_end(&mut file, &mut bytes)?;
        Ok(bytes)
    }
    inner(path.as_ref())
}
```

Example.

Sometimes common utility functions like `Option::map` and `Result::map_err` are instantiated many times. Replacing them with equivalent `match` expressions can help compile times.

The effects of these sorts of changes on compile times will usually be small, though occasionally they can be large. **Example**.