Precipice

Jack Maguire

Abstract

A programming utility for gaining knowledge on execution times via graphs and statistical information.

Contents

1 Introduction

In the modern day, many programmers are simply able to settle for the naive solution as to what is fastest, thanks to the ever-increasing computing speeds we now live and work with. However, a small sub-set of programmers still do need speed, like those working with embedded hardware (which is typically smaller, cheaper, and much less powerful), Operating Systems (which need to take up as few resources as possible) or Games (which need to run faster in real time in conjunction with many other systems).

However, in the modern day very few people code in straight Assembly and program in more user-friendly languages like C++ or Rust. These programs are written and then compiled to produce a machine-readable program which is then run by the end user. The difficulty comes with the black-box which is the compiler which has many niche optimisations and can make some portions of code run faster than others, often seemingly at random. This is the problem I aim to solve.

To work out how to best massage the compiler into giving the fastest code, programmers should be able to quickly see the impacts of how long their programs are taking. However, not many of these exist right now and those that do, like Hyperfine (TODO: CITE HF) have various issues like export formats.

2 Architecture Explanation

Having finished the project, I'll now take some time to explain the overarching architecture.

The project contains a git repository with 2 cargo crates inside - one to contain the library code and one for the binary code. I decided to separate it like this midway through the project when the binary components were more separated and I have kept it to try and provide some separation between the code that benchmarks and the code that deals with graphical user interfaces.

The library code contains the benchmarker itself, some utilities for dealing with the filesystem and the code that glues it all together.

The binary code contains 4 modules - enough for an importer and an exporter for the command line and for with a graphical user influence. Each of the individual modules take in either a *CreationContext*, or the Command-Line arguments that are specified next to the function that runs the part of the binary. The *CreationContext* provides some utilities for accessing persistent storage or the current render state, and this is used for caching things like which binaries to benchmark.

Both of the command-line applications use Clap (TODO: CITE CLAP) to print help messages and parse the arguments. Both of the graphical applications use egui (TODO: CITE EGUI) and eframe (TODO: CITE EFRAME)

2.1 Benchmarker Architecture

The library user must provide a few values to the benchmarker to start it running, like how many runs to complete, which binary to benchmark, whether or not to print out the initial run and how many warmup runs to do.

The warmup runs is used to ensure that the program is cache-local. If a program is frequently run on a computer, parts are temporarily cached inside a cache which is much faster for the CPU to access than grabbing it from the permanent storage. The warmup runs allow the user to ensure that the program is cache-local. However, the number of runs can be zero as a user might be testing a program that takes a while to run and they don't want the overhead of running it without actually using it for timing. They might also be testing a larger program which isn't as affected by cache-locality.

The benchmarker then creates a new thread to bench with, as well as a multi-threaded sender/receiver pair to send the results to the consumer with. Then it runs the warmup runs, and then does the actual runs in sets of 5. Between each set, it checks whether or not the user has sent a stop message. I only check after every set to reduce the impact of polling the stop signal receiver.

2.2 CLI Architecture

Both of the CLIs work in largely the same - they collect in arguments from the command line and then either run the benchmarker with a progress bar, or just re-export the files in the new format.

2.3 GUI Architecture

The GUI Runner works as an elaborate state machine with 3 main stages - preparation, execution, and export.

During the preparation step, the program displays a form that allows the user to input how many runs, which file to benchmark, and whether or not to do a warmup. If we do a warmup, the number of warmup runs is then chosen by the program based on the number of runs. There is also a basic list for command-line arguments which allows you to reorder and remove arguments.

Then, it does the runs and displays them to the end user, with an option to stop the runs. When it finishes, it then takes the user to a menu which allows them to customise how the runs are exported.

The GUI exporter works in largely the same way - collecting inputs, processing and exporting.

3 Code Samples

To illustrate a few of the algorithms and methods I've used, here I will explain some code samples.

I'll be removing comments as I'll be explaining the code anyways.

3.1 *EguiList*

The *EguiList* is a struct I've created for displaying lists of items, and it is used for displaying the command-line arguments, the runs and the traces for the exporter.

3.1.1 Variables

```
#[derive(Copy, Clone, Debug, PartialEq, Eq)]
pub enum ChangeType<T> {
        Removed(T),
        Reordered,
}

#[derive(Debug, Clone)]
pub struct EguiList<T> {
        is_scrollable: bool,
        is_editable: bool,
        is_reorderable: bool,
        had_list_update: Option<ChangeType<T>>,
        backing: Vec<T>,
}
```

Above you can see 1 generic enum which can act as a flag for when items are changed. The generic part means that it can contain any type that has a compile-time available size. I can use an enum here to ensure that invariants are always enforced, like making sure that if we are in the removed state, then there is always an item.

There is also the *EguiList* itself, which contains some variables which act as flags which are determined by the user. The backing list actually contains the items. The list update is then polled by the user to ensure that events are correctly dealt with. This needs to be polled to update the UI as it locks whilst there is data.

3.1.2 Trait Implementations

In Rust, structs can implement traits (TODO: CITE TRAITS), which are incredibly useful for shared behaviour. The common example is if you have a class for a circle and a class for a rectangle, you could have a trait for finding the area or a trait for drawing it to the screen. Then, other methods don't need to worry about the specifics and they can just use something that can draw or you can get the area.

```rust
impl<T> Default for EguiList<T> {
        fn default() -> Self {
                Self {
                        is_scrollable: false,
                        is_editable: false,
                        is_reorderable: false,
                        backing: vec![],
                        had_list_update: None,
                }
        }
}

impl<T> Deref for EguiList<T> {
        type Target = Vec<T>;

        fn deref(&self) -> &Self::Target {
                &self.backing
        }
}
impl<T> DerefMut for EguiList<T> {
        fn deref_mut(&mut self) -> &mut Self::Target {
                &mut self.backing
        }
}
impl<T> AsRef<[T]> for EguiList<T> {
        fn as_ref(&self) -> &[T] {
                &self.backing
        }
}
impl<T> AsMut<[T]> for EguiList<T> {
        fn as_mut(&mut self) -> &mut [T] {
                &mut self.backing
        }
}


impl<T> From<Vec<T>> for EguiList<T> {
        fn from(value: Vec<T>) -> Self {
                Self {
                        backing: value,
                        ..Default::default()
                }
        }
}
impl<T> IntoIterator for EguiList<T> {
        type Item = T;
        type IntoIter = vec::IntoIter<T>;

        fn into_iter(self) -> Self::IntoIter {
                self.backing.into_iter()
        }
}
```

Here, I implement a few main traits. Firstly, I implement *Default*, which allows users to easily create a version of this, or if they are dealing with nullable options, they can use an *_or_default* method to get the value or the default.

Then, I implement *Deref* and *DerefMut* with the target being the backing vector. This allows the users of the struct to pretend that it is just a normal vector, so they can do things like add elements or get the length without me having to copy out those methods. I also implement *AsRef<[T]>* and

AsMut<[T]> which allows the user to pretend the *EguiList* is a slice of Ts.

Then, I implement *From* which allows users to quickly turn a vector into an *EguiList* - this is often used when things are read in from persistent storage. This then uses the vector for the backing, and the rest is from the *Default*.

Finally, I implement *IntoIterator* for the *EguiList* which allows a user to iterate over all of the elements inside a for loop or apply other functional operations like mapping or filtering elements.

3.1.3 Builder Pattern

The builder pattern is a common programming pattern within the Rust language which allows users to change options for an instance of a struct, but without lengthy constructors and only the options that they find it necessary to change.

For this pattern to be useful, library designers must follow the Principle of Least Astonishment (TODO: CITE POLA) in order to know which variables to set. Here, POLA would be having all of these variables default to *false*.

```
impl<T> EguiList<T> {
        #[must_use]
        pub const fn is_scrollable(mut self, is_scrollable: bool) -> Self {
                self.is_scrollable = is_scrollable;
                self
        }

        #[must_use]
        pub const fn is_editable(mut self, is_editable: bool) -> Self {
                self.is_editable = is_editable;
                self
        }

        #[must_use]
        pub const fn is_reorderable(mut self, is_reorderable: bool) -> Self {
                self.is_reorderable = is_reorderable;
                self
        }
}
```

As you can see here, there aren't very many fields to change, and the general pattern is to change the field and then return self. This allows usage like this:

```
//A scrollable, editable eguilist of unit tuples
let mut list: EguiList<()> = EguiList::default()
        .is_scrollable(true)
        .is_editable(true);
```

Note that here, a type annotation must be added for this to compile, as otherwise the compiler wouldn't know what the generic type is. Here, it is specified to be the unit tuple. If that list were to be used later, for example:

```
list.push(());
```

Then the compiler would work out that the list contains unit tuples and automatically infer for the declaration.