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 so-called naive solution, thanks to the ever-increasing computing speeds we now live and work with. However, a small sub-set of programmers still do need fast and efficient code, 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).

In the modern day, however, very few people code in raw Assembly and instead use more user-friendly languages like C++ or Rust. The difficulty with decreasing execution time comes with the black-box compiler which can apply 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 [5] have various issues like export formats or lack of graphical user interfaces.

2 Architecture Explanation

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 I wanted the individual binaries for the use cases 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 four 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 [4] to print help messages and parse the arguments. Both of the graphical applications use egui and eframe [3].

2.1 Benchmarker Architecture

The library user must provide a few values to the benchmarker to start it running, for example 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, if any.

The warmup runs are used to ensure that the program is cache-local. If a program is frequently run on a computer, parts can be temporarily cached inside a cache which is much faster for the CPU to access than grabbing it from the persistent storage (as the cache is wiped when power is lost). 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 do not want the overhead of running it without using it for timing. They might also be testing a larger program which is not 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. It only checks 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 way - they collect 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 I 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 have used, I will explain some code samples. I will mainly be focusing on application and general-use code, rather than the declarative UI or relatively simple console runners.

I will be removing comments to reduce the space the code takes up.

3.1 *EguiList*

The *EguiList* is a struct I 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

```rust
#[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 it is 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 removes interactivity, whilst there is input that has not been processed by the consumer.

3.1.2 Trait Implementations

In Rust, structs can implement traits [8], which are incredibly useful for shared behaviour. For example, 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 do not 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 useful traits. Firstly, I implement *Default*, which allows users to easily create a version of this, or if they are dealing with nullable variables, they can use an *_or_default* method to get the non-null 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 *[T]* [6], to do things like sort, fill, shuffle or index into.

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 stor-

age. 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: " Could there be a high astonishment factor associated with the new feature? If a feature is accidentally misapplied by the user and causes what appears to him to be an unpredictable result, that feature has a high astonishment factor and is therefore undesirable. If a necessary feature has a high astonishment factor, it may be necessary to redesign the feature". [1]

Here, this is done in order to know how to set variables and would be applied by 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 are not 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 would not know what the generic type is. Here, it is specified to be the unit tuple. If that list I re 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 type.

3.1.4 Business Code

Here is the main code which actually displays the list.

```
fn display_inner(&mut self, ui: &mut Ui, label
    : impl Fn(&T, usize) -> String) {
  if self.backing.is_empty() {
    return;
  }

  let mut need_to_remove = None;
  let mut up = None;
  let mut down = None;

  for (i, arg) in self.backing.iter().
    enumerate() {
    ui.horizontal(|ui| {
      ui.label(label(arg, i));

      if self.had_list_update.is_none() {
        if self.is_editable && ui.button("
    Remove?").clicked() {
          need_to_remove = Some(i);
        }
        if self.is_reorderable {
          if ui.button("Up?").clicked() {
            up = Some(i);
          }
          if ui.button("Down?").clicked() {
            down = Some(i);
          }
        }
      }
    });
  }

  let len_minus_one = self.backing.len() - 1;
  if let Some(need_to_remove) = need_to_remove
      {
    self.had_list_update = Some(ChangeType::
    Removed(self.backing.remove(
    need_to_remove)));
  } else if let Some(up) = up {
    self.had_list_update = Some(ChangeType::
    Reordered);
    if up > 0 {
      self.backing.swap(up, up - 1);
    } else {
      self.backing.swap(0, len_minus_one);
    }
  } else if let Some(down) = down {
    self.had_list_update = Some(ChangeType::
    Reordered);
    if down < len_minus_one {
      self.backing.swap(down, down + 1);
    } else {
      self.backing.swap(len_minus_one, 0);
    }
  }
}

pub fn display(&mut self, ui: &mut Ui, label:
    impl Fn(&T, usize) -> String) {
```

```
  if self.is_scrollable {
    ScrollArea::vertical().max_height(300.0).
    show(ui, |ui| {
      self.display_inner(ui, label);
    });
  } else {
    self.display_inner(ui, label);
  }
}
```

Since the list can be inside a scrollable area, I have to have 2 methods - one to draw the list and one that the user calls. If the user specifies that the list should be scrollable, then the UI object from the *ScrollArea* is passed into the inner display method, and if not then I give it the raw UI. The function also takes a function in as an argument which takes in the object to display and its index in the list and should return a string to display.

Inside the inner function, I first check if the list is empty, and if it is then I early exit. Then, I declare a few temporary nullable variables for edits to the list. I then iterate through the whole list, displaying each item using the passed in function. I check if I can edit and reorder and if so, then I add buttons. If those buttons are clicked, I set the relevant temporary variables. Since *egui* works as an immediate mode GUI, only one element can be clicked every frame. This means I only have to store one change, even if there are multiple buttons that could be clicked.

Then, I check those variables and make the relevant changes to the list, ensuring to account for rolling around the list if I try to move the top element up or the bottom element down.

3.2 *Runner*

The *Runner* is the struct which actually benches the program.

3.2.1 Variables

Similar to the *EguiList*, the *Runner* has a number of variables which control the options.

```
pub struct Runner {
  pub binary: PathBuf,
  pub cli_args: Vec<String>,
  pub runs: usize,
  pub stop_rx: Option<Receiver<()>>,
  pub warmup: u8,
  pub print_initial: bool,
}
```

Here, I have a few variables that need to be used for running:

- The binary to run, stored as a *PathBuf*.
- The CLI Arguments to pass to that binary
- How many timed and warmup runs to complete
- A nullable stop receiver - when the user wants to stop the benchmarking, they can send a message to the corresponding sender and the runs will stop
- Whether or not to print the initial run to the console.

3.2.2 Benchmark Function

This is the actual function which runs the benchmarks.

```
pub fn start(self) -> (JoinHandle<io::Result
    <()>>, Receiver<Duration>) {we
  let Self {
    runs,
    binary,
    cli_args,
    stop_rx,
    warmup,
    print_initial,
  } = self;

  let (duration_sender, duration_receiver) =
    channel();

  let handle = std::thread::Builder::new()
  .name("benchmark_runner".into())
  .spawn(move || {
    let mut command = Command::new(binary);
    command.args(cli_args);

    if let Ok(cd) = current_dir() {
      command.current_dir(cd);
    }


    let mut is_first = true;
    for _ in 0..warmup {
      let Output {
        status,
        stdout,
        stderr,
      } = command.output()?;

      if !status.success() {
        return Ok(());
      }

      if print_initial && is_first && !stdout.
    is_empty() {
        is_first = false;
        io::stdout().lock().write_all(&stdout)
    ?;
      }
      if !stderr.is_empty() {
        io::stderr().lock().write_all(&stderr)
    ?;
      }
    }

    command.stdout(Stdio::null()).stderr(Stdio
    ::null());

    let mut start;
    for chunk_size in (0..runs)
      .chunks(CHUNK_SIZE)
      .into_iter()
      .map(Iterator::count)
    {
      if stop_rx
```

```
      .as_ref()
      .map_or(true, |stop_recv| matches!(
    stop_recv.try_recv(), Err(TryRecvError::
    Empty)))
      {
        for _ in 0..chunk_size {
          start = Instant::now();
          let status = command.status()?;
          let elapsed = start.elapsed();

          duration_sender
          .send(elapsed)
          .expect("Error sending result");

          if !status.success() {
            warn!(?status, "Command failed");
          }
        }
      } else {
        break;
      }
    }

    Ok(())
  })
  .expect("error creating thread");
  (handle, duration_receiver)
}
```

Firstly, I destructure on the object I run this on to get all of the variables that the user set. Then I create a *channel*. Channels are a concurrency primitive that can be used to send messages between threads. There are many advantages to using channels over something like a nullable variable or list which is how lots of other languages might handle it. Rust does not allow this as it can cause data races which are, as a whole, incredibly bad as they can result in hard to diagnose errors that happen under incredibly specific and hard to replicate conditions. Channels in Rust can allow threads to send and receive messages as they are ready to handle them without blocking either side waiting, like a Mutual Exclusion could cause.

We then spawn a new thread. I do this to ensure that the runs happen on their own thread so that the user interface can still update itself using the channel without only being able to update after every run. This is useful because users often have high expectations for how quickly interfaces update (the most expensive commercially available monitors expect an update around every 2ms), and programs could take any length of time to run.

The first thing done on that new thread is the creation of the *Command* which uses the binary and command-line arguments provided. I also give it the current working directory if it exists and I have permission to read it.

Then I warmup. For every warmup run, I grab the input, output and status. If the status is not a success, then I return out of the whole benchmarking function. If it is, then I continue. Then, I check if I print the initial run and this is the first warmup run and the output is not empty. If all of those hold, then I print out the output to the console. Then, since stderr should always be printed out, I print that out to the console if it is not empty. After the warmups, I redirect the stdout and stderr of the command to nothing so that nothing is printed. This is especially important for CLI users as they may display elaborate progress bars or TUIs.

Finally, I actually get to the benching stage where I split the runs up into chunks. This is done using functional programming techniques in Rust - I create a list the length of all of the runs, then I split it into chunks, and then I map each chunk into its length. I chunk it to avoid constantly polling the stop receiver. If I do not have a stop receiver or the stop receiver is empty I then proceed to benchmark the chunk. This involves resetting the start timer. The start timer is an *Instant*, which records monotonic non-decreasing time. Whilst this means that I cannot serialise and store it, it is guaranteed to always go up and be consistent which is most useful here. I then run the command, and get how long has elapsed since the start. I then send that. If the command was not a success, I also log that, but continue on. If the stop receiver was not empty or was disconnected, then I stop all remaining chunks.

3.3 Importing Traces

3.3.1 Functional programming example

Each trace is a name and a collection of timings, stored in csvs in individual rows. This is the function used to import a collection of traces from file names and uses some cool functional programming techniques.

```
pub fn get_traces(
  trace_file_names: impl IntoIterator<Item =
    impl AsRef<Path>>,
  trace: Option<(String, Vec<u128>)>,
) -> io::Result<Vec<(String, Vec<u128>)>> {
  Ok(trace_file_names
    .into_iter()
    .map(import_csv)
    .collect::<io::Result<Vec<Vec<(String, Vec
    <u128>)>>>>()?
    .into_iter()
    .flatten()
    .chain(trace)
    .collect())
}
```

Firstly, for the list of files to import from, it does not take a list of file names and instead takes in a generic type - any type that can be iterated over where each item can be a Path. This means that users could pass in anything from a vector of user collected path buffers to a slice of static strings that are always the same. It also takes in a nullable trace for adding an extra one to the list. Also note that it returns a Result, which means that it could be a list of traces, or it could be an IO error from when I read in the files.

As for the body, it is all one line, just split for readability into several sections. It undertakes the following steps for each item

1. I map it via *import_ csv*, a function declared elsewhere which turns a file path into a result which is either a vector of traces or an IO error
2. I collect it all into a Result which is either a Vector of vector of traces or an IO Error - this goes through every item, and if they are all fine then it goes to the happy path, and if not then it becomes the IO Error that it finds and early returns with that IO Error
3. I then turn that back into an iterator
4. I then flatten it - each file item can contain multiple traces, but I do not care so I just get them all into one list
5. I then add another iterator on - the nullable argument. In Rust, nullables are iterators with one item if they are not null, and zero if they are
6. I then collect it back into a vector

3.3.2 Importing the trace from a CSV file

This takes in a file which can have any number of traces and returns either those traces or an IO Error via a Result.

```
pub fn import_csv(file: impl AsRef<Path>) ->
    io::Result<Vec<(String, Vec<u128>)>> {
  let lines = read_to_string(file)?;
  if lines.trim().is_empty() {
    return Ok(vec![]);
  }
  let no_lines = lines.lines().count();
  let lines = lines.lines();

  let mut trace_contents: Vec<(String, Vec<
    u128>)> = Vec::with_capacity(no_lines);
    for line in lines {
  let mut values = line.split(',');

  let Some(title) = values.next() else {
    error!("Missing title");
    continue;
  };
  let contents = match values.map(str::parse).
    collect() {
    Ok(v) => v,
    Err(e) => {
      error!(?e, "Error parsing CSV file");
      continue;
    }
  };
  trace_contents.push((title.to_string(),
    contents));
}


  Ok(trace_contents)
}
```

We first read in the whole file, and the *?* sigil is used for functions which can return results and returns early if an error is encountered or gives the correct value if there is not an error. If that file is empty, I then just return an empty vector to signal no traces.

We then get a count of the lines and the lines themselves - the *lines* function splits a String via newline characters into a vector. I then pre-allocate a vector to avoid reallocating and moving around data in memory with space to fit one trace per line.

Then, I go through every line, and split the whole line on commas (as it is a CSV or comma-separated-values file). I then take the first value out, and if there is not a value (as taking a value out is a nullable operation) then I continue to the next line. I then use the rest of the values to build up a list of parsed times and if I fail to parse them then I continue to the next line. I then add the title and contents to a list, which I return at the end if all is I ll.

4 Evaluation

4.1 Tests

Rust has a comprehensive unit and integration testing framework built in, but unfortunately this project is not suited to unit testing, so I am relying exclusively on user interaction and manually testing it. I also feel that testing for runtime errors is less necessary than it would have been for other languages like Python or C(++) which are much more relaxed at compile-time. I have also made effective use of the algebraic type system to make invalid states unpresentable.

I have tested the following features for the CLI and GUI, which I would qualify as my primary user needs.

- The file exports correctly
- The user interface actively displays how many runs have been completed and are left, updating at an acceptable rate.
- It is easy to combine traces, re-export and re-import to move around files and contents
- It is easy to interpret the final data that is output.
- The program is able to be easily used in both programmer and continuous-integration/continuous-deployment contexts.
- The user is not limited in terms of how they want to run their application.

4.2 Conclusion

Overall, I am incredibly happy with the final product of this project. I have successfully created a profiler that provides programmers with easily digestible statistics about their execution times and I will use it to optimise and profile future projects. The source code for both the project and this document can be found here: `https://github.com/BurntNail/Precipice`.

4.2.1 Regrets

If I were to redo the project, I would take more time to thoroughly plan out the scope and features at the start as I had to complete 3 large code refactorings. I would also try to complete more thorough documentation whilst writing the code, rather than leaving it until the end and having to remember what the code all did and how the API was expected to be used. I might make more liberal use of tools like *unimplemented!* or denying code compilation until documentation is written. Also, currently the command-line options are a bit unwieldy and can lead to huge commands.

4.2.2 Future Improvements

- Currently, the project must be run locally either via the CLI or GUI, but it could be useful to run it as a webserver to allow people to remotely run benchmarks. The main annoyances would come from how to serve to the web and ensure that people would not use it as an extremely convenient manner to deny service to other server users.
- I currently have to deal with lots of threads as the code is fully synchronous, but if I ported the code over to an asynchronous runtime like tokio [7] I could save on lots of effort. The main annoyances would come with UI frameworks as most are synchronous and would need to be specially ported over.
- Currently, the CLI version finishes by exporting and displaying a few stats to the user. It would be nice if it could write out JSON to the console as that is more easily parsed by new programs.
- Finally, I would try to improve the command-line launching experience. I would likely change to either a YAML-based configuration file for most variables (like number of runs or export) and take in the binary as an argument, or move to asking the user questions via stdin and a library like *dialoguer* [2].

References

[1] Mike Cowlishaw. "The Design of the REXX Language". In: *IBM Systems Journal* 23.4 (1984). URL: https://www.cs.tufts.edu/~nr/cs257/archive/mike-cowlishaw/rexx.pdf (visited on 06/21/2023).

[2] *Dialoguer*. console-rs. URL: https://github.com/console-rs/dialoguer (visited on 06/21/2023).

[3] *Emilk/Egui: Egui: An Easy-to-Use Immediate Mode GUI in Rust That Runs on Both Web and Native*. URL: https://github.com/emilk/egui/tree/master (visited on 06/21/2023).

[4] Kevin B. Knapp and The Clap Community. *Clap*. Version 4.3.5. June 2023. URL: https://github.com/clap-rs/clap (visited on 06/21/2023).

[5] David Peter. *Hyperfine*. Version 1.16.1. Mar. 2023. URL: https://github.com/sharkdp/hyperfine (visited on 06/21/2023).

[6] *Slice - Rust*. URL: https://doc.rust-lang.org/std/primitive.slice.html (visited on 06/21/2023).

[7] *Tokio - An Asynchronous Rust Runtime*. URL: https://tokio.rs/ (visited on 06/21/2023).

[8] *Trait - Rust*. URL: https://doc.rust-lang.org/std/keyword.trait.html (visited on 06/21/2023).