CAT: Context Aware Tracing for Rust Asynchronous Programs

Tsung-Yen Wang[†], Shao-Hua Wang[†], Chia-Heng Tu[†], Wen-Yew Liang[§]
[†]National Cheng Kung University, Tainan, Taiwan
[§]ADLINK Technology Inc., New Taipei, Taiwan
chiaheng@ncku.edu.tw

ABSTRACT

Modern programming languages, such as Rust, have adopted the *coroutine* concept to better utilize computation resources and to improve program execution efficiency by allowing the overlap of the execution for asynchronous tasks. These programming languages often use the concept of userland thread library to dispatch the asynchronous tasks defined by the programmers. Nevertheless, it is often the case that the task scheduling on a user-space library is non-preemptive and would lead to unbounded execution time of a task.

In this work, we aim to develop a tracing methodology to capture unbounded execution time of asynchronous tasks in Rust programs. Based on the analyses of the Rust standard library, we identify several execution contexts of asynchronous computation in Rust, and develop a portable context aware tracing methodology that is able to trace the execution time of nested asynchronous computation work across different Rust runtimes. We develop a framework, called CAT, to collect and visualize the asynchronous runtime activities. The results show that CAT can help pinpoint the asynchronous computation exhibiting prolonged execution time. We believe that CAT is a complement of existing tools to improve the execution efficiency of asynchronous operations in Rust.

CCS CONCEPTS

• **Software and its engineering** → **Application specific development environments**; Runtime environments; • **Computing methodologies** → *Concurrent programming languages*;

KEYWORDS

Rust Programming Language, Performance Profiling/Tracing, Asynchronous Programming Model

ACM Reference Format:
Tsung-Yen Wang[†], Shao-Hua Wang[†], Chia-Heng Tu[†], Wen-Yew Liang[§]. 2023. CAT: Context Aware Tracing for Rust Asynchronous Programs. In *The 38th ACM/SIGAPP Symposium on Applied Computing (SAC '23), March 27 – March 31, 2023, Tallinn, Estonia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3555776.3577669

SAC '23, March 27 – March 31, 2023, Tallinn, Estonia

ACM ISBN 978-1-4503-9517-5/23/03…$15.00
https://doi.org/10.1145/3555776.3577669

1 INTRODUCTION

The *coroutine* concept [7] has gradually been adopted as a mainstream abstraction in modern programming languages to improve the computation efficiency. Programming languages, such as C#, Go, Javascript, Python, and Rust offer their abstractions for *asynchronous functions* to improve execution concurrency by allowing the programmers to give *hints* to the languages runtimes to better schedule the execution of the functions. The hints are often existed as the *keywords* to the languages, like `async` and `await`, to express the dependencies of the asynchronous functions. The languages runtimes are able to *yield* the executions of the `async` functions if the *awaited* results are not available yet, so as to improve the execution efficiency.

Most of the languages' runtimes adopt a *green thread* model approach, where a scheduler is present and is responsible for dispatching computation work specified by the programmers to the worker threads managed by the underlying operating system. This context switching scheme is considered *user-space threading* as the underlying operating system is unaware of the computation work (task). That is, the languages runtimes often act similarly to userland thread libraries, responsible for the scheduling of the asynchronous tasks without help from the underlying operating system.

Cooperative scheduling can be considered a non-preemptive system and computation tasks run on such a system may have unbounded task execution time, which would further lead to performance issues or unexpected execution behaviors. Research works have been proposed to tackle the unbounded execution time issue. For example, the Go language provides the `runtime.Gosched` function for its programmers to manually yield the execution at the call sites [3]. A thread library, *libturquoise*, is proposed to provide an abstraction for calling a function with a timeout [1].

Witnessing the important issue, instead of proposing a programming facility to prevent the unbounded execution time issue from happening (as described above), this work aims to capture the occurrences of the asynchronous tasks that may have the unexpected behavior. To this end, we develop a tracing methodology to automatically track the runtime behaviors of asynchronous operations for Rust programs. With the tracing methodology, we have built a software framework, *CAT*, for performance diagnosing/debugging to detect the excessive execution time of Rust tasks, caused by either a blocking call or an excessively-tight loop. Our framework can be considered as a complement to existing tools to identify the potential tasks that would downgrade program performance.

The remainder of this paper is organized as follows. Section 2 gives the background of the Rust programming language and the

asynchronous programming in Rust, which is followed by the motivation and contributions of this work. The proposed tracing methodology is described in Section 3, and the built framework, CAT, is further introduced in Section 4. Section 5 presents the experimental results of the proposed framework to demonstrate the effectiveness and efficiency of the proposed framework. Section 6 lists the prior performance tools for analyzing asynchronous activities in modern programming languages. Section 7 concludes this work and provides future research directions.

2 BACKGROUND AND MOTIVATION

2.1 Rust

Rust is a programming language designed by Mozilla, and its first stable release (v1.0) was announced in 2015. The important feature of Rust include performance, safety, and the support of concurrent programming, and these features are critical for network applications. Unlike Python and Java, whose runtimes run some background processes for maintenance, such as memory management via garbage collection process, Rust has a relatively lightweight runtime. In addition, Rust is able to access the underlying hardware. Its performance is similar to C/C++, and Rust does not have the memory safety concern (i.e., double-free error and dangling pointer in C/C++). With the support of asynchronous programming, it makes programmers easy to write asynchronous operations without dealing with complicated IO callbacks. The asynchrony support makes Rust good for building IO bound applications.

2.2 Asynchronous Programming in Rust

The support of Rust asynchronous programming is achieved by the three key components: 1) `Future` trait, 2) `async/await` syntax, and 3) runtime library. The first two components help facilitate the development of asynchronous programs, whereas the runtime library makes the asynchronous programs run smoothly. The components are described in more detail in the following subsections.

2.2.1 The `Future` *trait.* A `trait`, which is similar to an `interface` feature in other languages (but with some differences), defines a particular *type* with specific functionality. Specifically, a `trait` is used to define an abstracted shared behavior, and it contains the methods to define the desired behavior. A *future* refers to an asynchronous computation, and the `Future trait` is the core of asynchronous programming in Rust.

The prototype of an example future trait is listed in Figure 1, where line 1~4 is the prototype of the example ExFuture trait and the rest of the code is the actual implementation of `ExFuture`. The progress of ExFuture can be made by its `poll` function as possible[1]. The `poll` function returns `Poll::Ready(result)` when the desired computation completes. Otherwise, it registers the callback function with the `wake` function and returns `Poll::Pending` to indicate the future is not completed yet. When the socket data is ready, the `wake` is called, and the Rust runtime (i.e., the *executor*) will call the poll function again to make more progress on the ExFuture. The above concept of asynchronous execution is illustrated in Figure 2.

[1]In this example, the progress is made by performing the socket read operation.

```
1  trait ExFuture {
2      type Output;
3      fn poll(&mut self, wake: fn()) -> Poll<Self::Output>;
4  }
5
6  pub struct SocketRead<'a> {
7      socket: &'a Socket,
8  }
9
10 impl ExFuture for SocketRead<'_> {
11     type Output = Vec<u8>;
12
13     fn poll(&mut self, wake: fn()) -> Poll<Self::Output> {
14         if self.socket.has_data_to_read() {
15             Poll::Ready(self.socket.read_buf())
16         } else {
17             self.socket.set_readable_callback(wake);
18             Poll::Pending
19         }
20     }
21 }
```

Figure 1: The prototype and implementation an example `ExFuture trait`.

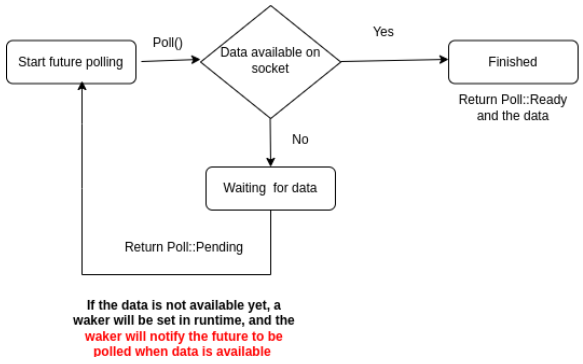

Figure 2: The execution flow of the ExFuture example.

2.2.2 The `async/await` *syntax.* The syntax is a built-in tool that helps transform a block of synchronous code into its asynchronous version. When the code is decorated with the `async` keyword (e.g., the `hello_world` function as shown in Figure 4), the Rust compiler helps turn the decorated code into a future trait by returning a future object instance when the function is executed. In particular, `async` can be used to decorate a function, a code block, and a *closure*. When `async` is added before a code block (as shown in Figure 4), the execution of the code block will return an anonymous future. A closure is an anonymous function, which is similar to a lambda function in other languages, allowing it to save in a variable or to pass as arguments to some functions. At runtime, a blocked `async` function will *yield* control of the future itself, allowing some other futures to run. It is achieved by using the `.await` keyword. The `.await` example is shown in Figure 4.

2.2.3 Asynchronous execution and Rust runtime. The asynchronous executions of futures are supported by Rust runtime. In fact, futures are grouped into *tasks*. A task is considered an *outermost future* and is a lightweight, non-blocking execution unit manipulated by

an *executor*. A task is created by the `task::spawn` function, with the exception of the task for the `main` function of a Rust program, which is created by the *attribute* (e.g., `#[async_std::main]` when using the async-std runtime library). Created tasks can be executed sequentially by a single thread (i.e., the single-threaded execution by the `LocalPool` executor) or by a thread pool (i.e., a small set of worker threads handling a set of spawned tasks via the `ThreadPool` executor). It is important to note that the schedule of the tasks is determined by the executor, instead of the operating system (OS) scheduler. This style of the logical threading model (done by the executor) is often referred to as *green threads* [10] and renders the OS is not aware of the Rust tasks.

Rust runtime library contains the implementation of the asynchronous execution support. In addition to the task management and execution, the runtime library has interfaces to the underlying I/O, file system, and network system. It is important to note that in addition to the rust-lang and the standard library, one has to leverage the asynchronous runtime to handle the `async` programs. Currently, there are three most popular asynchronous Rust runtimes, tokio, async-std, and smol.

2.3 Motivation and Contribution

In a Rust program, different types of futures are allowed to be created and *nested* to fulfill desired application logic. While this improves the programmability, it complexes the performance analysis of the built Rust programs since it is difficult to identify the performance bottlenecks. Our observations are summarized as follows.

- The *green thread* model is adopted in the Rust design. The asynchronous execution of tasks/futures on top of the executor makes it hard for general-purpose profiling tools, such as Perf [14], to identify the *hot* future in a Rust program since these tools are unaware of the executed tasks/futures. Actually, these tools are useful for reporting the function-level time distribution of the executed threads, but they cannot provide insight into the future-level information of the tasks. Taking the timeline diagram of the futures in Figure 3 as an example, there are two *interior futures* nested within the task at the beginning of the program execution. It is often the case that the anonymous futures may be used to handle asynchronous operations (e.g., the two interior futures may be created by the *async blocks*), and the existing performance tools cannot help discriminate the two anonymous contexts (i.e., the two interior futures) from the task. In such a case, it will be hard for the programmer to know that the deepest anonymous future is responsible for prolonging the execution of the task (since the anonymous future is at the bottom of the figure blocks the execution of the task).
- The *cooperative scheduling* in Rust relies heavily on the proper use of `.await`. If `.await` is missing unintentionally or if there is a blocked loop in a future for more than, for example, one hundred micro-seconds, it will impact the delivered performance and would sometimes lead to an unexpected execution error. For example, when the runtime is running out of worker threads and there are blocked futures waiting for the result produced by some other future, it will result in a deadlock in the program. Unfortunately, in such a case, it is quite a challenging task using the existing Rust tracing tool [11] to identify the root cause because the tool requires its users to instrument the interested futures manually to observe the performance of the instrumented futures. However, it is a cumbersome and challenging job to instrument all futures in a program, especially for those anonymous and nested futures, to pinpoint the problematic future.

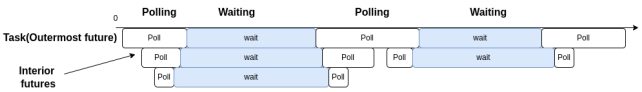

Figure 3: Illustration of the execution of nested futures.

In this work, we propose a cross-runtime tracing methodology and a software framework to automatically track the activities of asynchronous operations of a Rust program. Our contributions are listed as follows.

- We propose a context aware tracing methodology to trace the runtime behaviors of Rust programs with nested asynchronous operations. As the methodology is built on top of the APIs of the Rust standard library and the runtime libraries [2], it is able to be applied on Rust applications cross different Rust runtimes.
- We develop an open-source software framework based on the proposed methodology, where a dynamic instrumentation-based approach is adopted to alleviate the tracing overhead. The software project is available via the github repository: https://github.com/asrlabncku/Rust-Async-Context-Tracer.
- We evaluate the effectiveness and efficiency of our proposed methodology and framework on the Rust programs with the commonly seen patterns and on the Rust program of a consensus algorithm for autonomous driving vehicles to cross an intersection. We also evaluate the proposed methodology on the two popular Rust runtimes: async-std and tokio, which demonstrates the cross-runtime capability.

3 CAT: CONTEXT AWARE TRACING

With different programming constructs available for building asynchronous Rust programs (described in Section 2.2), various types of execution contexts may appear during program execution. To characterize the runtime behaviors of the asynchronous programs, it is important to identify the types of the execution contexts. Based on the involved programming constructs, the future execution contexts can be divided into the following four types.

- ⓣ Task context. The execution context of a *spawned* task.
- ⓒ Compiler-generated future context. The execution context of an asynchronous computation is usually created by the `async/await` syntax.
- ⓤ User-implemented future context. The execution context of a custom future that is implemented by programmers (e.g., using the `impl` keyword to implement a future trait.).

[2]The async-std library and tokio are considered as most common implementation of the asynchronous support for Rust

TABLE 1: The future context tracing APIs.

Notation	Future Context Tracing Function
ⓣ	[entry] _<async_std..task..builder..SupportTaskLocals<F> as..
ⓣ	[exit] _<async_std..task..builder..SupportTaskLocals<F> as..
ⓒ	[entry] _<core..future..from_generator..GenFuture<T> as..
ⓤ	[entry] USER_IMPLMENTED_FUTURE as core..future..future..Future
ⓐ	[entry] ASYNC_FUNC_NAME::_{{closure}}

ⓐ Anonymous context. The execution context of an anonymous struct that is executed by the compiler-generated future context, which is transformed by general futures, for example, an *async block*[3] or an *async function* expression.

To discover the execution contexts, we further analyze the APIs of the Rust library, future-rs crate (future 0.3.21); the crate was the first implementation of the future type to enable the `async/await` syntax, and the future type of the Rust Standard Library can be seen as a minimal subset of the future implementation in the future-rs library. In addition, the task creation APIs of the async_std library are analyzed as well to track the task context creation.

The execution contexts can be disclosed by tracking the activities of the future context tracing APIs that create the different types of futures. The APIs used to identify the four types of execution contexts are listed in TABLE 1. The first row gives the API(symbol) that is used to create tasks in Rust programs. When a task is spawned with a task-creating method (`task::spawn` or the attribute `#[async_std::main]`),

When tracing the future contexts, both entry and exit of a future context tracing API are recorded. That is, the performance tracing probes are inserted at the two program points to track the boundary of a function execution context. To demonstrate the concept, the second rows in the table represent the exit points for the first APIs, and it is denoted by the italic circled text *ⓣ*. More about the code instrumentation and the post-process of the trace data are elucidated in Section 4. In order to save space, the exits of the rest APIs are removed from TABLE 1.

Note that the APIs of the future-rs library are compatible with the popular Rust runtimes, such as async-std and tokio. Our implementation has been validated on the two runtimes, and the related results are provided in Section 5. Another advantage of using context aware tracing is to reduce the tracing overhead since only the selected APIs are tracked. Detailed information on how our proposed tool can help lower the tracing overhead is further discussed in Section 4.

4 CAT FRAMEWORK

With the built state machine in mind, we develop a software framework for analyzing the performance of Rust programs with asynchronous contexts. The workflow of the proposed framework is illustrated in Figure 6, where it should undergo the four steps to obtain the performance profile of a Rust program: dynamic binary instrumentation (Section 4.1), runtime data collection (Section 4.2),

[3]A *block* expression contains a sequence of statements.

```
async fn hello_world() {
    println!("Hello, world.");
    let fut = async {
        println!("The value is {}.", x);
    }
    fut.await;
}

#[async_std::main]
async fn main() {
    hello_world().await;
}
```

Figure 4: A Rust asynchronous program example.

post-process trace data (Section 4.3), and data visualization. A popular data visualization tool, Trace-viewer [4], is selected to reveal the runtime activities (with the timeline diagram) of execution contexts in a Rust program.

4.1 Dynamic Binary Instrumentation

This work adopts the *uftrace* toolkit [6] for performance profiling. It is an open-source function graph tracer for C/C++ and Rust and is able to trace the relationships between the callers and callees, as well as the statistical performance for each function of a program. By default, all invoked functions are traced by uftrace for analyzing their execution times. In this work, uftrace is dictated to add the performance probes only on those context tracing functions defined in our built state machine.

The *dynamic tracing* feature of uftrace is adopted to provide the opportunity of lowering the instrumentation overhead. The dynamic tracing requires the support of a more recent compiler, which can insert a sequence of NOP instructions on the entry points of the selected functions; the NOPs can be inserted via adding the `patchable_function_entry` attribute to decorate the interested functions or compiling the source program with the `patchable_function_entry` attribute. During the program load time, uftrace patches only the NOPs into the performance probes (i.e., the `mcount` functions) to collect the performance data. With the dynamic tracing feature, the tracing overhead can be further reduced when users turn on the tracing for some tasks (e.g., the one with the blocking issue) and the tracing overhead can be greatly reduced since the performance probes are activated for the handful tasks.

4.2 Runtime Data Collection

The inserted performance probes are responsible for generating the data during the program execution. A data tuple is generated upon each function invocation, i.e., the timestamp, the thread identifier, the string of the tracing program point (i.e., *entry* or *exit*), the function name with its virtual address, and the depth of the stack frame. In order to avoid the I/O operations blocking the program execution, we can create a RAM disk to buffer the trace data and write back to the secondary storage in the background. Note that the performance probes are inserted at the entry/exit of the functions associated with the APIs listed in TABLE 1, and the runtime execution context can be revealed by performing stack walks during

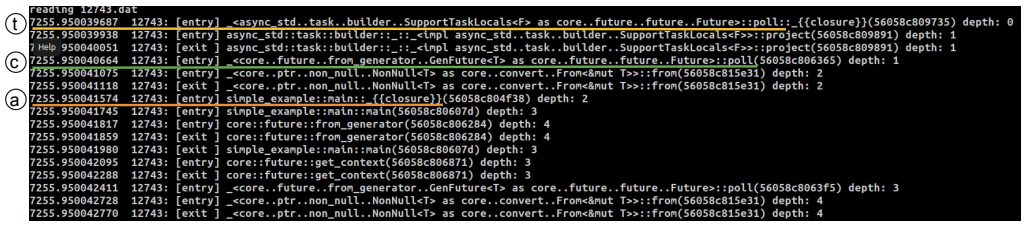

Figure 5: The collected trace data for the code listed in Figure 4, and the circled texts refer to the context tracing functions.

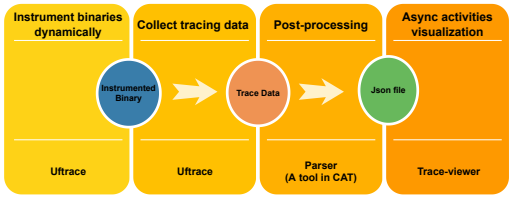

Figure 6: The workflow of the proposed tracing framework.

the parsing of the trace data, which is elucidated in the following subsection. An example of human readable generated trace data is illustrated in Figure 4.

4.3 Post-Process Trace Data

To reveal the runtime behaviors of the asynchronous execution contexts, a converter has been built to convert the collected trace data into the JSON format accepted by Trace-viewer. The converter performs *stack walks* while parsing the trace data. In particular, if the converter gets the data tuple representing an entry of a context tracing function, it performs a stack push operation, where the context tracing function is now at the top of the stack. On the contrary, if there is an occurrence of an exit for a matched context tracing function, the converter does a stack pop operation. The execution time of a task (or an execution context) can thus be logically represented by the difference of the timestamps of the entry and exit for the same context tracing function.

Figure 5 depicts the collected trace data of the Rust program listed in Figure 4. The circled texts label the occurrences of the context tracing functions. For example, ⓣ indicates the invocation of the context tracing function leading to the entering of the task context. When the converter encounters the sequence of the invoked functions ⓣ, ⓒ, and ⓐ, which indicates that it is in the anonymous context enclosed by the task and compiler-generated future contexts. Note that Figure 5 gives only the parts of the trace file. The trace is collected using the default setting of uftrace, i.e., uftrace instruments all functions' entries and exists during the program execution, and it contains unnecessary data tuples. With the dynamic tracing feature enabled, only the context tracing functions listed in TABLE 1 are tracked so as to minimize the tracing overhead.

5 EXPERIMENTAL RESULTS

To evaluate the effectiveness of our proposed methodology, we have built Rust programs with the commonly seen patterns, which are the built-in Rust primitive types offered by the async-std library, and used the developed framework to trace these programs to validate the correctness of the captured behaviors against the existing tools. The experiments are done on the machine with the Intel multicore processor 11th Gen Intel(R) Core(TM) i5-1135G7 2.40GHz with 8GB memory. On top of the machine, the Rust programs run on the Ubuntu 20.04.1 with the support of the Rust standard library (v. rustc 1.65.0-nightly), future-rs library (v. 0.3.21), the async-std library (v. 1.12.0) and the tokio library (v. 1.21.0). The GCC compiler (v. 9.4.0), the uftrace framework (v. v0.12-70-gd784a), and Trace-viewerUse the `about::tracing` in a Chrome browser to visualize the trace data. are used to run with our proposed tracing framework. Note that to use our proposed framework, the Rust program should be compiled and run in debug mode.

We have tested the asynchronous programs in Rust built with the following patterns: the timer operation[4], the file operation[5], the mutex operation[6], the multi-producer and multi-consumer operations for inter-task synchronization[7], and the client/server code pattern[8]. Considering the page length limitation, we present the visualization results of the client/server program to demonstrate the capability of our framework in Section 5.1. As for the rest of the tested programs, please refer to the source repository of this project, which is listed in Section 2.3.

We use our built framework to analyze a consensus program [5] that can be used by connected, autonomous driving vehicles when approaching an intersection to determine the sequence of the vehicles passing through the intersection without traffic signals. Through our experience of searching for the root cause of the execution failure of such a multi-node (multi-vehicle) application[9], we demonstrate that our framework is able to help expedite the process of the identification of performance issues. We also describe the limitations of the prior works when using them to do the same job. The results are detailed in Section 5.2.

The portability of our methodology is demonstrated in Section 5.3. The program with the file I/O pattern is used as an example to show that our tracing methodology is portable across the async-std and tokio runtimes. Besides, the tracing overhead

[4]The example involves the Rust type: `async_std::task::sleep()`.
[5]The example involves the Rust types: `async_std::fs::open` and `async_std::stream`.
[6]The example involves the Rust type: `async_std::sync::Mutex`.
[7]The example involves the Rust type: `async_std::channel`.
[8]The example involves the Rust types: `async_std::stream`, `async_std::net::TcpListener`.
[9]During the program development stage, the multi-node application runs on the Intel machine described above, and there are multiple processes (nodes) running concurrently on the same machine.

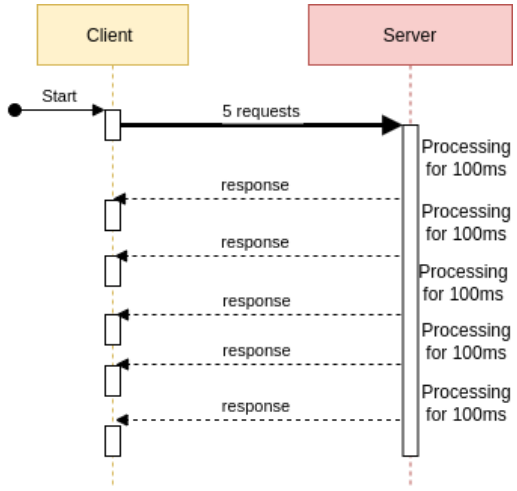

Figure 7: The sequence diagram of the sequential request handling for the client/server program.

of our proposed work is measured and compared with the prior work. The results shown in Section 5.4 provide evidence that our tracing methodology is superior to the prior work for incurring lower tracing overhead without human intervention.

5.1 Analysis of the Client/Server Program Behaviors

The built client/server program has one server task and one client task. The client task sends five requests to the server at once after its task initialization. The server side implements two strategies: the *sequential* request handling and the *concurrent* request handling. In this example, we demonstrate that our framework is capable of displaying the future activities clearly and correctly.

The sequential version uses the same spawned task to handle the requests one by one, where each request takes 100ms to complete (the computation is handled by the `process` function). The sequence diagram for the sequential request handling is illustrated in Figure 7. The visualization representation of the runtime behavior of the sequential handling of the requests is in Figure 8. The figure shows that the `process` function (the green span) is invoked five times within the task, and at the end of each function invocation, there is a bar (highlighted by the red oval) representing the response to the client.

The concurrent implementation, on the other hand, spawns the five *asynchronous tasks*, each of which is an instance of the `process` function and handles a request from the client. As shown in Figure 9, the five spawned tasks run concurrently to handle the received requests. The activities made by each task are plotted in a separated region and each task is labeled with its task identifier (e.g., "`1720654`" refers to the activities made by the first spawned task). Comparing the contents in Figures 8 and 9, it is obvious that when the number of worker threads is more than that of the requests made by the client, the concurrent scheme performs better. This example indicates that our framework users can use the visualization data to weight different design alternatives.

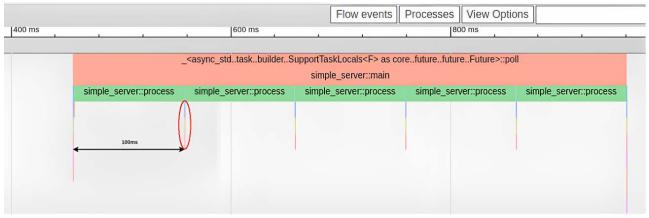

Figure 8: The timeline diagram produced by our tracing tool for the sequential request handling in Figure 7.

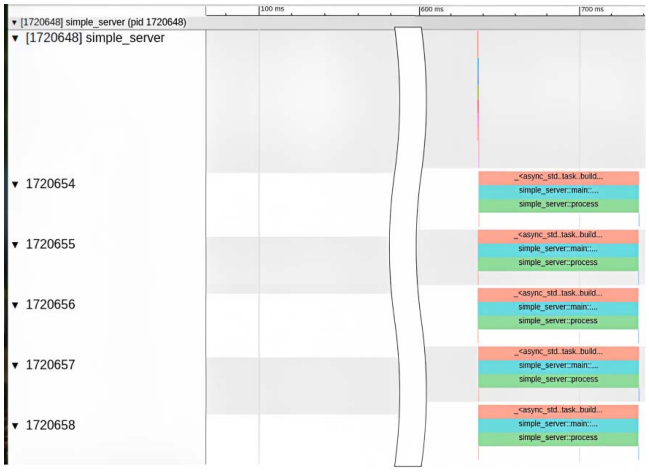

Figure 9: The timeline diagram of the concurrent request handling.

5.2 Analysis of a Consensus Program for Autonomous Driving Vehicles

The subsection provides our performance profiling experiences while developing a consensus program in Rust. The consensus program is built on top of *zenoh* [15] that is a Rust-based software library; the Eclipse zenoh is an open-source project blending traditional publish/subscribe programming model with the features of geo-distributed storage, queries, and computation. The consensus program is dedicated to autonomous vehicles connected via the vehicle-to-everything network to cross an intersection without the information provided by traffic signals. It is a multi-node program, where each node represents a Rust program instance running on an autonomous vehicle, and each node needs to send/receive messages to/from other nodes within a specific time window. During the code development process, we found that the execution terminated unexpectedly without a clear clue of what happens, except for the fact that we found some of the asynchronous tasks are `polled` later than we expected. The program starts to run abnormally when the number of nodes is beyond eight. Our experiences of searching for the root cause of the unexpected termination of the existing tools and CAT are listed below.

Perf. When the program terminates unexpectedly, our first choice of the performance tool is Perf [14]. We profile the program with Perf and examine performance results with

the visualization tool, flamegraph. However, the flamegraph reports that the function calls from the Rust runtime (e.g., `executor::block_on`[10]) are responsible for the occupying the CPU times, which obviously does not provide any insight for us to debug the program, since the sampling-based profiling tool monitors the CPU-intensive *hot* functions, and it is not suitable for analyzing the asynchronous I/O activities.

uftrace. We turn into the tracing framework, uftrace, to trace the runtime behaviors with the *static instrumentation* and try to locate the abnormal behaviors. At first, the trace data dumped by the uftrace is too large to be handled by the visualization tool, Trace-viewer, since by default, the static instrumentation scheme logs the activities of every invoked function. Hence, we use the built-in text-based data report to browse the trace data. As shown in Figure 10, one can easily realize that the function-level performance data logging the time distribution of the invoked functions cannot provide useful information for the program with asynchronous operations (a blocked future does not spend much CPU time); the performance profile is similar to that reported by Perf.

tokio-tracing. We then use tokio-tracing to analyze the program behavior. Nevertheless, the tool requires the manual code instrumentation in the source code on the interested future. While tokio-tracing can provide detailed information of the instrumented futures, it requires the programmer to pinpoint the exact futures to be analyzed, which is not suitable for the debug purpose when the program acts unexpectedly. Furthermore, the consensus program is built upon the third-party Rust library, zenoh, and it is difficult for the programmer to insert the trace probes to the supporting library. As a result, tokio-tracing is not suitable in this case study.

CAT. The proposed CAT framework is used to trace the runtime activities. The visualization data shown on Trace-viewer indicate that it spends a significant amount of time in the third-party library. As shown in Figure 11, the network connection function[11] from the zenoh library takes about 300ms to complete. With the clue of the problematic function, we use `uftrace-graph` to further extract the function calls made within the `new` function from the zenoh runtime. We find out that due to security concerns, the RSA key generation is necessary for each new zenoh network connection object, and the adopted implementation for the RSA key generation[12] takes too long to complete, which leads to the timeout for the consensus program during the program initialization. And it also caused the blocking issue(polling time should not larger than 100ms in practice), as suggested in [9]. Later, the program is function correctly after we use a faster compiler option.

As the asynchronous operations may be blocked (pending to be processed), the conventional profiling tools are good for measuring the performance of CPU-bound tasks and synchronous operations,

[10]This function is used by the executor to block and wait for the result produced by some asynchronous operation.
[11]The `zenoh::net::runtime::Runtime::new` function is called by each node to establish the vehicle-to-vehicle communication network.
[12]The `rsa::key::RsaPrivateKey::new` function in the RSA package.

```
Total time   Self time       Calls  Function
==========  ==========  ==========  ====================
 13.024  m   84.676 us         329  std::panic::catch_unwind
 13.024  m   90.704 us         329  std::panicking::try
 13.024  m   64.532 us         329  std::panicking::try::do_call
  8.051  m   28.607 us         119  _<core..panic..unwind_safe..AssertUnwindSafe<F> as core..ops..function..FnOnce<()>>
  7.051  m   18.554 us          67  std::sys_common::backtrace::__rust_begin_short_backtrace
  7.042  m  461.333 us         108  std::thread::Builder::spawn_unchecked_::_{{closure}}
  7.042  m   20.495 us          66  core::ops::function::FnOnce::call_once{{vtable.shim}}
  7.042  m   15.305 us          66  std::thread::Builder::spawn_unchecked_::_{{closure}}::_{{closure}}
  7.036  m    3.043 ms       10957  std::thread::local::LocalKey<T>::try_with
  7.028  m    1.535 ms        4356  std::thread::local::LocalKey<T>::with
  7.014  m    7.014  m        57491  linux:schedule
  6.048  m  176.139 ms         522  std::sync::condvar::Condvar::wait
  5.017  m   62.248 us         111  async_global_executor::reactor::block_on
  5.017  m  125.379 us         111  async_global_executor::reactor::block_on::_{{closure}}
  5.017  m    1.511 ms         111  async_io::driver::block_on
  4.041  m    4.225 ms       33240  core::ops::function::FnOnce::call_once
```

Figure 10: The function-level time distribution profile provided by `uftrace-report` for the consensus program.

Figure 11: The timeline diagram provided by the CAT framework for the blocked function from the zenoh library.

but they are not suitable for the performance evaluation of the blocked asynchronous tasks/futures. The proposed CAT framework can be used to fill the gap in the search for performance bottlenecks of asynchronous operations, together with the conventional tools for the synchronous part in a Rust program. It is important to note that we did not use the tokio-console profiling tool in this case study because it requires the adoption of the Tokio runtime, and the consensus program is built upon the async-std runtime.

5.3 Tracing Programs Built with Tokio Runtime

As Rust programs can be built and run with different runtime libraries, it is important that CAT can be run across different runtimes. In order to validate the portability of the CAT framework, we analyzed the tokio runtime [**?**] and updated the APIs listed in TABLE 1. We port the programs with the timer, file, and mutex operations, as described at the beginning of Section 5, to the tokio runtime version. We compare the timeline diagrams obtained from the executions of the async-std and tokio runtimes. We find that the high-level behaviors are identical for the same programs running on two different runtimes.

The test program exercising asynchronous file operations is used as an example to demonstrate the CAT framework on the tokio runtime. The source code of the file read program using the async-std library is listed in Figure 13, and the corresponding runtime activities of the program are depicted as the sequence diagram based on the collected trace data. From the sequence diagram, it is obvious that the `open().await` is called twice to open the file

to read. Furthermore, the `read_to_string().await` is invoked three times to read the contents from the file, where the function implementation is done with the help from the `stream` type and is omitted to save space.

The timeline diagram of the execution of the ported version (the file manipulation program using the tokio runtime) is illustrated in Figure 12. There are four blocks of the spans in the figure, where the first block contains the initiation of the file open operation, denoted as ①, and ② represents the second time that the `tokio::fs::file::File::open` function is called to finish the file open. The marks ③, ④, and ⑤ represent the invokations of `read_to_string().await` (the pink spans). The visualization data show that the high-level behaviors are identical. Our tool is able to trace the activities of asynchronous operations across the two runtimes because our tracing methodology is established on the future-rs library, which is responsible for defining the asynchronous functionality in Rust.

5.4 Analysis and Discussion of Tracing Overhead

We port the client/server program to the tokio runtime (the concurrent request handling version, as described in Section 5.2). The performance of the ported program is evaluated by the tokio-console [12], tokio-tracing [11], and CAT. In this section, the tracing overhead is considered as the instrumentation latency. As the tracing overhead of the client is higher than that of the server, we report the incurred overheads observed on the client side as listed in TABLE 2.

The tokio-console tool has the least overhead since it only tracks the task-level activities (but not the interior futures) and the tracing probes are inserted in the tokio runtime. On the other hand, the tokio-tracing tool has the highest tracing overhead as several probes have been inserted manually to the client program to observe the runtime activities of the asynchronous operations taking place in the client program.

The proposed CAT framework sits in between the tokio-console and tokio-tracing, in terms of the overhead for tracing asynchronous operations. It is worth noting that the overhead incurred by the CAT framework can be further lowered by selecting the interested functions (by specifying the function names in the command line) or by tracking the task-level futures only (without recording the activities for the interior futures). As shown in the table, CAT′ represents the overhead when the configuration is applied and the overhead is compatible with the overhead incurred by tokio-console. The comparisons of the features provided by the tokio-console, tokio-tracing, and CAT.

- **CAT vs. tokio-console.** CAT incurs more overhead than tokio-console since CAT covers the tracking of different types of execution contexts, as listed in Section 3. On the other hand, tokio-console tracks only the task-level contexts. In addition, tokio-console embeds the tracing facilities in the executor of the tokio runtime, which renders it of low portability.
- **CAT vs. tokio-tracing.** CAT performs the dynamic tracing automatically without the presence of the source code, which has lower instrumentation overhead. On the contrary, tokio-tracing requires the developers to insert the probes manually on the interested futures, which will incur a significant overhead when the developers want to have a general idea of the performance profile of the entire program. In addition, it will be a challenging task for its users to analyze the performance of the third-party library, which comes with the binary format.

TABLE 2: Tracing overhead (%) for the client/server program.

tokio-console	tokio-tracing	CAT	CAT′
5	98	45	30
2.75	-	-	1.05

6 RELATED WORK

6.1 General Purpose Profiling Tool

Perf [14] is a remarkable example of general-purpose performance tools that can provide application-specific and/or system-wide performance profiling. It integrates sampling and instrumenting-based tools to analyze performance issues for programs running with synchronous operations (e.g., measuring the functions that spend most of the CPU time). It is capable of profiling Rust programs; however, it is not suitable for analyzing the asynchronous activities in the Rst programs. The experiences of using Perf to profile the asynchronous program are detailed in Section 5.2.

6.2 Profiling Non-Rust Programs

Profiling C++ and Java programs. Based on our preliminary study, the asynchronous programming in C++ and Java is supported by the multithreading model. This implies that the task execution model (available in Rust) is not present in C++ and Java. As the asynchronous execution contexts are represented as *threads*, the asynchronous behaviors in C++ and Java can be analyzed with existing profiling tools for multithreaded programs, such as Intel Vtune.

Profiling C# programs. Similar to Rust, C#, Python, Go, and Javascript programming languages adopt the *green thread* model according to our analysis. Nevertheless, the performance profiling tools for the programming languages have their limitations, and it is difficult to borrow their profiling techniques for Rust. C# programs can use the .NET Async tool to track the start and end time of each occurred asynchronous function. Nevertheless, C# programs run on the .NET runtime, which is proprietary software, and it is hard to learn the developed techniques for tracking the asynchronous activities in the closed source implementation.

Profiling Python programs. Python developers leverage the `asyncio` library [13] for writing concurrent code with the `async/await` syntax. The asynchronous support comes with high-level and low-level APIs to control the concurrent executions. Nevertheless, part of the library implementation is not thread-safe (e.g., `asyncio.Future` is not thread-safe), and to support `asyncio`, a single-thread event loop is adopted to schedule asynchronous operations in a Python program [2]. In addition to the unified runtime that provides debugging support for the asynchronous operations, there is a tool, OpenTracing [8], for analyzing the asyncio Python programs via

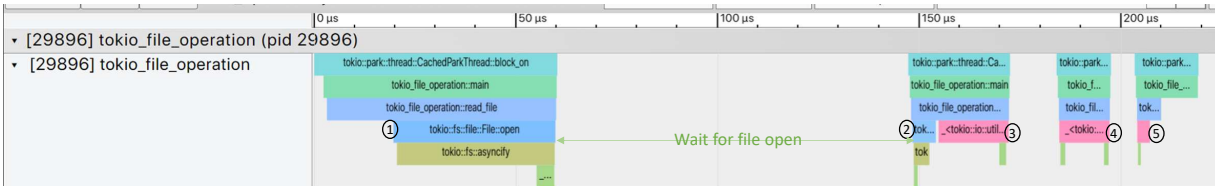

Figure 12: The timeline diagram of the asynchronous file operations using the tokio runtime.

```rust
use async_std::fs::File;
use async_std::io::{self, prelude::*};

async fn read_file() -> Result<String, io::Error>{
    let mut f = File::open("./hello.text").await.unwrap();
    let mut s = String::new();
    f.read_to_string(&mut s).await.unwrap();
    Ok(s);
}

#[async_std::main]
async fn main() {
    if let Ok(s) = read_file.await {
        println!("Read_from_file:_{:?}", s);
    }
}
```

Figure 13: The example of asynchronous file operations with the async-std library.

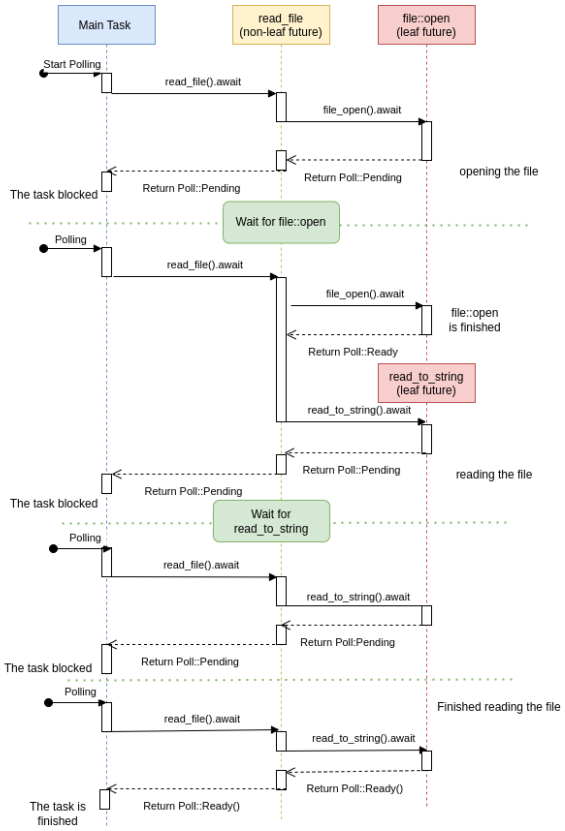

Figure 14: The sequence diagram of the Rust program in Figure 13.

static instrumentation. However, it supports only a handful of application types, such as Amazon Web Services, asynchronous SQL programs, and asynchronous HTTP client/server programs.

Profiling Go and Javascript programs. Go programming language, designed by Google, provides asynchronous programming support. The *goroutines* in Go are like the asynchronous *tasks* in Rust. The Go language has built-in support to track the activities of network, synchronous, system calls, and scheduler operations. Users are allowed to run the target program to log the runtime activities with the tracing switch being turned on. Nevertheless, the runtime-specific solution cannot be adopted in the cross-runtime design in Rust. Note that Javascript has a similar design, which relies on a Javascript engine (e.g., V8 or SpiderMonkey) to provide native support for debugging/performance profiling functionalities.

6.3 Profiling Rust Programs

The two most popular profiling tools for Rust programs are introduced below. In addition, we compare the CAT framework against the two tools.

tokio-tracing. The *tracing* framework is maintained by the Tokio project, which develops the *tokio* runtime. It is important to note that the tracing facility is a static instrumentation-based framework that can be used to trace Rust applications and libraries for different Rust runtimes, such as async-std and tokio runtimes. Developers are required to insert the *tracing* probes in a Rust source program, and the probes log the timestamps, polling time (execution time), and idle time of a *traced* future. The collected data can be visualized by some visualization tools.

tokio-console. It is a tool of diagnostics and debugging for asynchronous programs in Rust. Similar to tokio-tracing, tokio-console is also maintained by the tokio project team. It provides the capability of real-time monitoring execution of Rust programs by adding one line of the instrumentation code to the program using the tokio runtime. The real-time performance status reporting functionality is similar to the `top(1)`. The reported data include the statistical data of the *polling* time, which is very useful for analyzing the blocking issues for tasks. Note that the polling time information is only for the tasks (outermost futures).

In comparison, the CAT framework is capable of doing the dynamic instrumentation of Rust programs and libraries automatically. The framework is able to trace the activities of nested futures. Our

results show that the CAT framework can be run across Rust runtimes. More about the major difference between CAT and the above two Rust-based tools is presented and discussed in Section 5.4.

7 CONCLUSION AND FUTURE WORK

We propose a context aware tracing methodology and the CAT framework to trace the asynchronous contexts of Rust programs. The CAT framework can automatically and dynamically instrument a given Rust program to collect the nested asynchronous activities with reasonable overhead and has the ability to work across different Rust asynchronous runtimes. Our experimental results show that the developers can use CAT to observe the runtime behaviors of the asynchronous activities and find out the potential performance issues. The results render that the CAT framework can be used as a complement to the existing tools to diagnose the blocking behaviors caused by the asynchronous operations in Rust. In the future, we explore the potential way to integrate CAT with the existing Rust tools to provide a seamless experience for profiling Rust programs, and to further demonstrate the potential of the contexts aware profiling.

ACKNOWLEDGEMENT

This work was supported in part by National Science and Technology Council under grant no. 111-2221-E-006-116-MY3. In addition, this work is financially supported in part by the "Intelligent Manufacturing Research Center" (iMRC) from The Featured Areas Research Center Program within the framework of the Higher Education Sprout Project by the Ministry of Education (MOE) in Taiwan.

REFERENCES

[1] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. 2020. Lightweight Preemptible Functions. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC'20)*. USENIX Association, USA, Article 31, 13 pages.
[2] Matthew Fowler. 2022. *Python Concurrency with asyncio* (1st ed.). Manning Publications.
[3] Google. 2010. Go language. Retrieved Jun 2, 2022 from https://go.dev/
[4] Google. 2014. Trace-viewer - Analysis and Visualization tool. Retrieved Jun 2, 2022 from https://github.com/catapult-project/catapult
[5] Pankaj Khanchandani and Roger Wattenhofer. 2020. Brief Announcement: Byzantine Agreement with Unknown Participants and Failures. In *Proceedings of the 39th Symposium on Principles of Distributed Computing (PODC '20)*. Association for Computing Machinery, New York, NY, USA, 178–180. https://doi.org/10.1145/3382734.3405740
[6] Namhyung Kim. 2014. Uftrace - Function graph tracer for C/C++/Rust. Retrieved Jun 2, 2022 from https://github.com/namhyung/uftrace
[7] Donald E. Knuth. 1997. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms* (third ed.). Addison-Wesley, Reading, Mass.
[8] Jessie A. Morris. 2019. python_opentracing_async_instrumentation. Retrieved Jun 2, 2022 from https://gitlab.com/midigator/python_opentracing_async_instrumentation
[9] Alice Ryhl. 2020. Async: What is blocking? (2020). https://ryhl.io/blog/async-what-is-blocking/
[10] Minyoung Sung, Soyoung Kim, Sangsoo Park, Naehyuck Chang, and Heonshik Shin. 2002. Comparative performance evaluation of Java threads for embedded applications: Linux Thread vs. Green Thread. *Inform. Process. Lett.* 84, 4 (2002), 221–225.
[11] Tokio. 2019. tokio-tracing. Retrieved May 31, 2022 from https://github.com/tokio-rs/tracing
[12] Tokio. 2021. tokio-console. Retrieved May 31, 2022 from https://github.com/tokio-rs/console
[13] Guido van Rossum. 2015. asyncio. Retrieved Jun 2, 2022 from https://docs.python.org/3/library/asyncio.html
[14] Vincent M. Weaver. 2015. Self-monitoring overhead of the Linux perf_ event performance counter interface. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 102–111.
[15] zettascale. 2020. zenoh. Retrieved Jun 2, 2022 from https://zenoh.io/