Lightweight RR for Intermittent Tests Failures

Omar S Navarro Leija[1] and Alan Jeffrey[2]

[1]University Of Pennsylvania
[2]Mozilla Corporation

Summer 2019

Abstract

TODO: Wait for everything else to be written before writing the abstract. We present lightweight RR. A record-and-replay method for dynamically lowering the nondeterminism of programs with concurrent message passing architectures. Lightweight RR allows more intermittent tests to pass, is robust to desynchronizations.

Ultimately our experimental results did not

1 Introduction

What should even go in the introduction?

2 Background

I feel like I should add some starting paragraphs here.

2.1 Concurrent Message Passing Channels

Concurrent systems are popularly implemented using threads plus shared memory with locks for exclusive write accesses. These systems make it hard to reason about data race freedom, and deadlock avoidance. the shared data and its locks are usually decoupled, and grabbing a lock before accessing data is not enforced by the language or runtime, instead it is merely a convention the programmer must be aware of, or enforced via the api + encapsulation.

Recent trends seen in languages like Go, Erlang, and Rust have popularized message passing as an alternative to shared data + locks. With message passing, isolated threads/processes communicate via channels, sending data as messages to each other. Channels come in many flavors: bounded, unbounded, blocking, non-blocking, and more.

Channels are often implemented using shared memory and locks, so they are just as fast.

2.2 Intermittent Failures In Servo

Servo is a parallel browser engine with the lotfy goals: highly concurrent, memory safe, and easily embeddable. Servo achieves these goals partly due to being written in Rust, as well as featuring a concurrent message passing architecture. This allows servo to be highly concurrent, maximizing parallelism, while maintaining maintaining components decoupled.

Nondeterminism is often inherent to concurrent systems, Servo is no exception. Debugging concurrent systems is more difficult than sequential programs, the difficulty is further exacerbated by nondeteminism.

Servo runs a large suite of browser-agnostic tests known as web-platform-tests (wpt) as part of the continuous integration system. The wpt are expected to pass before changes to the code base are accepted. Unfortunately, Servo has hundreds of wpt that fail randomly for unknown reasons, these failures are known as intermittent failures. As the tests sometimes pass/fail, there is unknown nondeterministic behavior in Servo which causes this behavior.

3 Lightweight RR

Here we present lightweight RR for channel communication: *lightweight RR*. Lightweight RR records and replays channel communication among threads. Later this same execution is replayed. Lightweight RR works well for systems were the main sources of nondeteminism are thread scheduling, and arrival time of messages.

Roughly lightweight RR handles the following sources of nondeterminism:

- Nondeteminism arising from thread scheduling. This is your garden-variety nondeterminism endemic to any concurrent system.
- Nondeteminism from message arrival order. While messages sent from the same sender thread are guaranteed to arrive in FIFO order. Multiple threads may be sending messages down the same channel. Thus, message arrival order can vary.
- Select operation. Channel implementations often have a *select* operation. A select allows a thread to block on a set of channels until a message arrives from any channel, returning one or more channel messages at a time. Which channels return a message from a select is nondeterministic.

Our approach is guaranteed to successfully replay an execution given that the above are the only sources of nondeteminism. Many programs have nondeteminism that affects the execution path of the program, for these programs we may desynchronize. If the replay execution diverges from the recorded execution, we say the execution has *desynchronized*.

Our approach is robust to desynchronizations. When a desynchonization is detected, the execution falls back to a nondeterministic "passthorugh" execution of the program.

3.1 Lightweight RR for Intermittent Failures

Lightweight RR is designed to work in the context of unit tests with intermittent failures. If some tests nondeterministically return unexpected results, timeout, or crash they can benefit from lightweight RR. First the user "loops" on a test until a successful execution is captured in a log. When unit tests are executed, we run lightweight rr in "replay" mode, replaying the successful execution recorded.

3.2 Designing Overview

For all channels, we define the receiver as the reader end, and the sender as the sending end. We assume the channels are unbounded (have no size limit), multiple producers, but single consumer channels (MPSC). This assumption fit our needs but nothing fundamentally stops us from supporting bounded or multiple producers, multiple consumer variants.

We support all common operations on channels, including:

- receive: Block until we receive a single message from the channel.
- select: Block on one or more receivers until a message arrives from any.
- try_receive: Non blocking variant of receive. If no message is available, an error is returned.
- timeout_receive: blocks until a message arrives or the specified time elapses.
- send: non-blocking send a message through a channel.

3.3 What is lightweight?

Unlike whole programs approaches to RR, lightweight RR can be implemented as a language level library, agnostic to the operating system, hardware details, etc. We highlight the following advantages between traditional RR methods and lightweight RR:

- Can be implemetnted as a language-level library.
- All threads can still run in parallel.
- As only channel communication is recorded, the logs are smaller.
- We only record several integers worth of information per event.
- Robust to desynchronization.
- The implementation "wraps" an existing channel implementation. Thus, the implementation is a straightforward "transform".

3.3.1 Assigning Deterministic IDs

In order to faithfully replay execution of a program. We must assign a deterministic thread id: DetThreadId to all threads. This thread ID must be unique and nondeterministic even in the presence of threads racing to spawn threads. Starting from the main thread, every time a new thread spawns, it is assigned an ID equal to the number of children that have spawned from this thread so far, so we need one integer per thread. Then a DetThreadId is generated as a vector of indices starting from the main thread "down the tree". For example, the main thread has a DetThreadId of [] ("empty list"), the first thread spawned by the main thread is assigned [1], while the second great-grandchild of the main thread would have a DetThreadId of [1, 1, 2]. TODO talk about log(N) size?

Every channel receiver/sender pair is assigned a unique, determinstic channeld id: DetChannelId. The DetChannelId consists of a (DetTheadId, N), where N is a per-thread integer, which is increased after generating a channel. This way, every channel gets an ID which is the same through multiple executions.

3.3.2 Recording Events

Our record log does not need to record the value of the data sent down the channel. Instead we transform every message of type T, to a tuple $(T, DetThreadId)$. Where $DetThreadId$ is the ID of the sender thread.

During "record" mode, we allow the program to execute with no changes to the execution. If the user calls a channel method, we merely allow the function to go through, and on return record some values. For all channel communication events, we record:

- EventId: A per-thread integer that increases by one on every channel communication event.
- Sender ID: The DetThreadId of the sender of this message.
- Event type: The type of event (e.g. send, recv, try_recv).
- Type of data sent across channels.
- Channel Flavor: E.g. ipc-channel, bounded channel, cross-beam channel.
- Event status.
- DetChannelId of the channel involved in the event.

Event status record the possible return values of a event. For example, try_recv may either: time out, recv error, or successfully receiver a value. In case of a success we record the DetThreadId of the sender thread. The sender thread ID is imperative for faithfully replaying executions when there are multiple writers to the same channel. Select operations have an ordering to their receivers, each receiver is given an index to its position on the select. On select events we record the index (or indices for multiple events) of ready receivers, and once the receive happens we again record the DetThreadId.

Many of the items recorded above are not strictly necessary. and are only used for robust logging (debugging) and ensuring the system is working correctly. For production releases, most of these values could be ignored at compile time.

3.3.3 Replaying Events

Unlike record, which tries to affect the program's execution as little as possible, replaying dynamically stops threads, does multiple channel receives, etc, to attempt to generate the same execution.

Using (DetThreadId, EventId) we can deterministically identify an event for any thread. For a given event we consult the log to see what actions are expected for this event.

We compare the event type, DetChannelId, and channel flavor to ensure we are still synchronized with the execution. Otherwise a desynchronization error is returned.

Next we consult the event status. If the status was say timeout, we don't bother doing the operation, and instead return a timeout error immediately.

If the event status indicated the event succeeded, we still don't do the real operation per se.

We handle receive events (recv, try_recv, timeout_recv) by implementing a rr_recv as follows. We call blocking $recv()$ on the receiver directly. The blocking recv will eventually return a message of type $(DetTheadId, T)$. We compare the received DetThreadId against the expected from the log. If they match we have found the correct message to return. Otherwise, this event came from another thread, so we add it to a per-receiver buffer. We loop on $recv()$ until the correct event is found. Next time we do a rr_recv, we first check the buffer for an event from the expected sender.

On select, we use the expected index/indices to retrieve the correct receiver and call rr_recv. Notice rr_recv takes care of buferring "wrong" entries, so we always the correct entry.

We must be careful with blocking recv as a desynchronization may cause the entire thread to deadlock. See Section 3.4.

3.3.4 Running Off the End of the Log

When replaying an execution, it is common to "fall of the end of the log". That is, there is no $(DetTheadId, N + 1)$ event in the log. In general, missing log entries means that the program did not get this far during record or desynchronization error has occurred.

The former can happen for programs that race between exiting and communicating through messages. While we could consider this a source of nondeteminism outside the scope of rr_channels, doing so would render our implementation useless for many programs.

Instead we handle getting to the end of the log by having the thread block on a conditional variable. See Section 3.4 for details on when this conditional variable is set.

3.4 Handling Desynchronizations

Our approach is robust to desynchronization events by continuing to execute in a best effort mode (See Section 7 for possible extensions). Once a desynchronization is detected, initial user input determines whether the program should be stopped (strict) or continue executing the program (keep_going).

Programs desynchronize when there is sources of nondeterminism outside of what rr_channels handles, and they affect program execution. For example network IO, timer firing, etc. These other sources of nondeteminism are outside the scope of this work. We expect many programs to have sources of nondeterminism so must be robust to desynchronizations.

Once we detect that execution has failed. We switch from the replay method described above, to merely doing the operation the user has asked for. This allows us to continue the execution but with no determinism guarantees. We notify all threads waiting on the end-of-log conditional variable, this unblocks these threads and they run on desynchronization mode.

If the program has diverged, a blocking recv may never unblock. To avoid deadlocks, we use recv_timeout, if the timeout ever elapses, we assume the program has desynchronized.

4 Implementation

We implemented lightweight RR as a Rust library. Wrapping two popular Rust libraries for channel communication: ipc-channel and crossbeam-channel. Rust does not have support for dependency injection or inheritance, so users must switch from the channel library they're using to rr-channel. To make this progress seamless, rr-channel copies the API of ipc-channel and crossbeam-channel. Therefore, the only requirement is changing the name of the library when it is being imported.

We must assign deterministic thread IDs to all threads. So a user must also use our rr-channel thread spawn function to ensure the thread receives a DetThreadId. The user may not use our thread spawn function in all cases, or a program dependency outside the control of the user may spawns threads. rr-channel handles this by giving these threads a $None$ DetTheadId. Events are still record and replayed for $None$ IDs. However, if two threads, both with $None$ IDs are writing to the same channel, we cannot distinguish these two threads, so the execution may not be deterministic. So we print a warning to users if this is detected.

We also implement the ipc-channel router. Which allows for ipc-channel and crossbeam-channel senders/receivers to be mixed. It is straightforward for our implementation to support other channel implementations as long as the underlying implementation supports simple channel operations.

5 Evaluation

As a case study, we integrate our rr-channel library into Servo. Servo is a highly complicated program featuring N foos, and M lines of Rust code. Servo represents a high target for lightweight RR, as Servo uses many complicated channels, to send many messages among its multitude of threads. Therefore Servo is the perfect program to stress-test our implementation and get a proper evaluation on the limits of our approach.

Thanks to the implementation of rr-channels, the integration was fairly simple, and a few hours of work is enough to have Servo using lightweight RR channels everywhere.

5.1 Reducing intermittent failures

5.2 Performance Overhead

5.3 Space overhead

6 Discussion

Reasons RR Doesn't work well: Lightweight RR only works as well as nondeterminism in application. Servo is a hard mark to hit. Sources on nondeterminism we're not capturing. Known sources of nondeterminsm we're not capturing =¿ rayon, tokio, runtime. Bugs in our implementation =¿ deadlocks.

7 Future Work

The current work just barely scratches the surface: providing an implementation and doing a first-pass test. We believe with some iteration we could significantly increase how well lightweight RR works.

Lightweight RR has many possible extensions and future work. Currently, when the replay execution desynchronizes, we merely fall back to executing the program natively. A more elaborate desynchronization recovery scheme could be robust to some desynchronized events. Some threads could be desynchronized while other threads continue to run synchronized. Similarly, we could look at real synchronization recovery by attempting to match the execution back at a later point.

Once a desynchronization is detected, it is indicative of a source of nondeteminism in some program thread. Using the current execution, and the log, it may be possible to help the programmer narrow down and identify the source of nondeterminsm.

When we detect a desynchronization, we could start recording this alternate program execution. Later the two executions could be compared to better understand why they diverged. Furthermore, when considering continuous integration, and as programs evolve overtime, we expect changing part of the code will have changes in the execution, therefore, as new patches are tested for

merging we could record the "new" executions, and use those for future record and replay testing.

The log we record is a full record of all communications and thread topology for a given execution. This log could create a visualization of threads, and channel communication for the program execution. This Describing the topology of programs. Visualizing channel communication of threads.

Record a new execution