Anodize: Language-Level Guarantees for Mixed Criticality Systems

Phil Fox, Grant Iraci, Sofiya Semenova
CSE 605 - Checkpoint 3
Dr. Lukasz Ziarek
December 21st, 2020

***Abstract*— Anodize is a Rust library that offers language-level guarantees for mixed criticality systems written in Rust. Anodize offers memory isolation by default between threads, strictly-enforced protocols for safely sharing memory between threads, and a mechanism to expose priority-based thread scheduling to the application programmer. Leveraging Rust's type system enables these features through catching as many traditionally runtime-caught errors at compile-time as possible. The result is a mixed criticality system that resembles multiple well-coordinated virtual machines that have tightly managed means to share memory spaces and available processing power.**

I. Introduction

The embedded and Real-Time (RT) systems communities are increasingly interested in the research and development of *mixed criticality systems* – systems that contain programs given precedence via designated priorities (i.e., criticality levels) on shared hardware. These criticality levels can be vastly different: Some programs must guarantee that they will never crash while the failure of other programs is less catastrophic. Further, mixed criticality systems contain programs with different *timeliness* guarantees, even within the same criticality level. We recognize a significant challenge as multiple programs' performance on shared hardware is inextricably interconnected. For example, a computer vision program may deadlock with flight controller software and cause it to crash, or a program that collects and reasons about current sensor data may be consistently delayed so that its observations are always stale. Further, programs within mixed criticality systems almost always share some memory in addition to CPU time. It follows that partitioning each program into strictly isolated processes or VMs negatively impacts system performance and makes managing communication between programs difficult or even infeasible. Thus, mixed criticality systems ideally have a tripartite overall goal: 1) Enforce critical process precedence (expose priority-based thread *scheduling*), 2) enable programs to *safely* share some memory, (e.g., manage *shared* memory), and 3) disallow all access to memory ineligible for sharing (i.e., guaranteeing memory *isolation*).

Traditionally, mixed criticality guarantees are enforced at a hardware and/or operating system level. Leveraging these lower-level controls lends straightforward support for process time, memory, and resource isolation to an extent that it is infeasible to violate constraints. Two platforms exist that partially achieve mixed criticality goals on a hardware/OS level: RT systems offer concrete OS support for guaranteeing that process priorities are preserved but lack strict memory isolation controls or guarantees. Conversely, Multi-VM systems segment memory to an extent that VM instances share no memory space overlap apart from the host. However, programmers stand to benefit from carefully operating outside of traditionally strict memory constraints while preserving RT prioritization.

The Rust programming language offers a feasible implementation direction achieving mixed criticality goals. Generally, mixed criticality systems governed at a *language* level prospectively allow for code sharing (relieves memory footprint), faster program switching, and granular control of resource allocation. We further recognize that Rust's system-level runtime performance naturally aligns with RT requirements coupled with finite processor power. Rust's compile-time error handling emphasis aligns with safety-critical systems with low-to-zero tolerance for runtime failure. Lastly, Rust's memory ownership feature well-emulates multi-VM systems' memory segmentation guarantees while providing newfound means to share memory on a strict basis. Thus, the goal of our project is to create a library that leverages Rust's compiler and unique concept of ownership to provide *language-level* guarantees achieving mixed criticality system goals. In particular, we offer memory isolation by default between threads with strictly-enforced protocols for sharing memory, and a way to schedule threads by priority. We have also implemented a proof-of-concept, robotics-centered mixed criticality system that provides temporal and spatial isolation guarantees, written in Rust and built on top of Anodize.

II. Background

A. Rust

The Rust language is a system level language that began private development in 2006 and open source starting in 2009. Rust is designed for software level security and enhanced concurrency performance with resource control. Mozilla uses Rust for their in-development web engine project "Servo" which touts Rust's concurrency benefits among other features. Rust is also used for its own language source (rustc) and the Rust Compiler.

A novel, central feature of Rust is the concept of memory *ownership*. Rust manages memory by assigning each variable an *owner* and automatically freeing memory when the owner goes out of scope. Ownership operations mitigate the need

for a garbage collector or requiring the programmer explicitly free memory. Rust can validate much of a program's memory usage through the ownership model at compile-time which means that many memory-based errors that frequently emerge at runtime in other languages will now manifest as compilation errors. Rust establishes strong concurrency control by leveraging both ownership and strict type checking avoiding common issues with incorrectly-managed shared memory, again, often at compile-time. Despite some perceived rigidity, Rust also offers means to either 'soften' (e.g., memory *borrowing*) or even 'override' (e.g., *unsafe* Rust) its strict features.

B. Mixed Criticality

As introduced, mixed criticality systems are distinct in virtue of rigid scheduler performance guarantees. Particular RT applications and embedded systems benefit from mixed criticality features: Proper implementation enables diverse platforms to operate on common hardware (E.g. System on a Chip designs) thanks to strong memory segmentation, and rigorous control is gained over systems subject to strict safety or fault-tolerance standards via process prioritization [3]. Thus, mixed criticality system use-cases are effective in production and industrial automation (e,g., automotive, avionics) fields where unhandled system failures could yield catastrophic results. These use-cases illustrate that runtime speed and memory overhead are also important considerations for mixed criticality deployment in order to avoid complications such as process "starvation" (a process is effectively 'shut-out' by higher priorities) and insufficient memory capacity. Today, RT systems' implementation is often facilitated by formal languages such as LUSTRE, SIGNAL and Simulink or "model-based" languages such as the Architectural Analysis and Design Language (AADL) [4].

C. Session Typing

Session typing, first introduced in [6] and expanded in [2], provides type safety guarantees over communication channels. Building on linear typing, session types introduced the concept of *sessions*, a "chain of dyadic interactions whose collection constitutes a program." Each session has a *channel*, over which the corresponding process may communicate with others. Channels support three basic communication primitives: passing values, branching between labels, and the passing of other channels (known as delegation).

The syntax of session types is given in Fig. 1. These forms offer the three basic primitives introduced above. The type $!S.\alpha$ denotes the sending of a value of sort S, followed by the execution of the session α. We note that all forms except $\bot$ (which marks the end of a session) have a session α which represents the continuation of the session. Similarly, the type $?S.\alpha$ denotes the receiving of a value of sort S. Choice comes in two forms: internal and external. Internal choice, $\oplus\{\ell_1 : \alpha_1, \ldots, \ell_n : \alpha_n\}.\alpha$, represents the behavior that selects a label ℓ_i and then behaves as α_i. External choice, $\&\{\ell_1 : \alpha_1, \ldots, \ell_n : \alpha_n\}.\alpha$, represents the behavior that accepts a label ℓ_i then behaves as α_i. Finally, $\bot$ represents the end of a session, the point at which no further communication is possible.

$$S ::= \mathbf{nat} \mid \mathbf{bool}$$
$$\alpha ::= !S.\alpha \mid ?S.\alpha \mid \&\{\ell_i : \alpha_i\}.\alpha \mid \oplus\{\ell_i : \alpha_i\}.\alpha \mid \bot \mid t$$

Fig. 1: Syntax of session types

$$\begin{aligned} ATM =& ?Int. \oplus \{ok : ATM', err : \bot\} \\ ATM' =& \& \{ \\ & deposit : ?Int.ATM, \\ & withdraw : ?Int. \oplus \{ok : !Int.ATM, fail : ATM\} \\ & balance : !Int.ATM, \\ & quit : \bot \ \} \end{aligned}$$

Fig. 2: A session type for an ATM

We illustrate the concept using the following example, first described in [2] and later adopted by many authors (including [1]). Consider an ATM which offers three services: deposit, withdraw, and balance inquiry. To use the ATM, the user must first provide an account number, which is either accepted or rejected by the ATM. If accepted, the user may then perform their desired operations and quit when done. Deposits require the amount to be deposited and always succeed, returning to the choice of operation. Withdrawals require the amount to withdraw and can either succeed or fail due to insufficient funds, both returning to the choice of operation. Balance inquiries return the current balance and always succeed, again returning to the choice. When the user is finished, they may quit using the ATM and the session ends.

Such a session type is given in Fig. 2. Here ATM represents the total session and the subsession ATM' represents the looping choice of operation. We allow these types to be recursive, allowing the expression of indefinitely long or non-terminating sessions.

III. Design Overview

Rust's advanced memory allocation control, concurrency/-parallel operation support, memory overhead, and speed give rise to the idea of using process threading to facilitate a mixed criticality system on a software basis with minimal intervention from an operating system. Anodize aims to establish a toolkit that takes advantage of the Rust language's extensive compile time error handling and means to maximize thread-safe concurrent functionality.

In our coding paradigm, programs within a mixed criticality system are realized as *groups* of Rust threads within one large Rust program. Each thread is spawned by the main thread (*the primordial thread*), which exists solely to spawn the necessary programs and construct channels for communications between programs.

Memory isolation between thread groups is largely ensured by Rust's ownership system, with the caveat that the programmer cannot use anything in unsafe Rust. Memory sharing is constrained to message passing using Rust's `mpsc` library, and its safety is also ensured through Rust's ownership system. To prevent a thread group from using too much memory and risking the performance of other thread groups, Anodize monitors how much memory each thread group is using and terminates any that exceed their allotted memory. In the future, we plan to implement priority-based thread scheduling.

A. Memory Isolation By Default

Most of Anodize's memory isolation comes 'for free' in virtue of Rust's ownership features. Rust will disallow a thread to access another thread's memory by default since this is an example of an operation marked "unsafe" (disallowed per assumptions in III-E). The ownership system generally defines the following rules: each variable must have an owner, each variable can only have one owner, and the variable is automatically dropped when the variable's owner goes out of scope. Variables are automatically owned by the scope in which they are defined, but there exist several methods for passing ownership from one variable to another. For example, when a thread is created, any variables in the current scope that are needed by the thread can have their ownership passed to the new thread by using the `move` keyword, as shown in Figure 3. The Rust compiler statically analyzes which variables are needed by the thread and does this automatically. Compliance with the ownership system's rules can be statically verified at compile-time, so many incorrect memory accesses can be found and fixed before the system runs.

B. Shared Memory Via Message Passing

So far, we've explained a system wherein threads have isolated memory regions alongside strong compile-time guarantees that thread memory accesses are safe. Our design also captures means for threads to safely *share* memory. Rust allows threads to share memory in one of two ways: *shared-state concurrency*, and *message passing*.

With the shared-state concurrency paradigm, Rust threads share memory by taking locks on the shared memory. Each thread has ownership of the lock object but not the shared memory itself. However, this memory-sharing scheme is slightly at odds with the Rust ownership model – typically, memory cannot be accessed by anyone other than its owner, but the shared-state concurrency paradigm allows *multiple threads* to *act as owners*. This paradigm can be thought of the same way as how most languages handle threads sharing memory by using locks. Thus, it is prone to the runtime-level bugginess of those languages – a problem that Rust's ownership system was specifically created to avoid.

With the message passing paradigm, Rust threads share memory by passing values over channels that must be known at the time of thread creation. While there are several channel implementations, the most commonly used one is Rust's own `mpsc` library, which denotes a *multiple producer, single consumer* channel. When a value is passed through a channel, its ownership transfers as well. Thus, the compiler can leverage the ownership system to verify the validity of shared memory accesses. Because of the distinctions between the discussed paradigms, Anodize *explicitly disallows* the use of shared-state concurrency. All threads will adhere to message passing for shared state.

```
fn main() {
    // x and y owned by main thread
    let mut x = 0;
    let mut y = 0;

    thread::spawn(move || {
    // Rust automatically moves x's ownership to the
    // spawned thread because it uses the variable
        x += 1;
    }

    // ownership of y is not moved to the spawned thread,
    // so accessing y here is allowed
    y += 1;

    // accessing x here will give a compile-time error
    // because ownership has been moved
}
```

Fig. 3: Automatically passing ownership from the spawning thread to the spawned thread, using the `move` keyword.

Because Anodize defines programs in a mixed criticality system as groups of threads, threads within a thread group may need to communicate shared state with each other. Anodize currently *only* implements intra-group shared memory, and our test mixed criticality system relies on ROS2 for *inter*-group message passing. See V-A for a discussion on our system's use of ROS2.

C. Ensuring Bounded Memory

Another problem in mixed criticality systems is that of one program using too much memory and preventing another program from accessing the resources it needs. This particularly becomes an issue when the memory-starved program is of a higher criticality level. To prevent this, we assign and enforce memory bounds for each thread, and terminate any threads that exceed their memory bounds.

Rust already exposes per-thread memory bounds for *stack-allocated* memory via the `Builder` type. However, stack-allocated memory is the lesser of a programmer's concerns, given that each thread's stack size is statically known and will not change during runtime. If the combined stack sizes of all threads in a mixed criticality system exceed the total allowed memory, this problem can be thought of as an incorrectly specified deployment. Of more concern is *heap-allocated* memory, as the size of the heap may change as the program runs and cannot be estimated statically. Unfortunately, Rust does not provide an API endpoint to bound a thread's heap-allocated memory usage.

To enforce heap-allocated memory bounds, we modify the Rust allocator to track each thread group's memory usage and panic when any thread exceeds it. The panic causes the errant thread to terminate, leaving the rest of the threads running. Currently, we have no mechanism for a thread group with

one memory-exceeded thread to automatically terminate *all* of its co-grouped threads emulating a program crash, the reasons for which are discussed in the limitations section, VI-B.

D. Priority-Based Thread Scheduling

To guarantee that higher-criticality threads take precedence over lower-criticality threads, Anodize schedules threads based on their priority. The Rust threading library exposes some thread scheduling options to the programmer, but limitations exist: First, the thread API does not provide enough fine-grained scheduling control, and second, these options ultimately serve as a 'suggestion' for the OS thread scheduler, rather than strict requirements. The resulting 'soft' performance expectations are insufficient for the needs of mixed criticality systems given the goals and sensitive applications listed prior.

There are several ways to address thread scheduling problems, but two stand out amongst competing approaches: For one, fine-grained scheduling control can be achieved by bypassing Rust's threading library and calling directly to Rust's `libc` crate, which provides Rust bindings to system-level libraries. For two, switching the mixed criticality system deployment to a RT operating system (e.g., RT-Linux) will guarantee that priorities passed to the OS scheduler are directly incorporated. While the latter does impose a limitation for the system programmer, requiring the use of RT operating systems is not a particularly unusual request for the domain of most mixed criticality systems.

Alternatively, there are several *green thread* implementations in Rust – the two most familiar are the `Tokio` library and `async/await`. Green threads are frequently called "lightweight" threads because they are scheduled by the runtime (in this case, a Rust program) rather than the operating system itself. This allows for far cheaper context switching between threads, but requires a redesign of each thread to be a cyclic operation. Further, the thread execution process is slightly different for native threads and green threads – native threads run until they are interrupted by the OS thread scheduler (starting another thread) while green threads run until they no longer can, at which point the runtime switches to executing the next time. Accordingly, green thread task scheduling is frequently called "cooperative multi-tasking". Green threads come with their own pros and cons, and are not always the ideal choice for every possible program. However, for programs that can be defined cyclically/periodically, it may be advantageous to implement green threads over native threads.

We chose to leverage OS (i.e., native) thread scheduling in lieu of green threads. This is primarily due to design decision-making in Rust language development: Native thread support is handled through `Lib.rs` crates while green threading is not well-supported in an official or particularly popular capacity. Rust's development team is explicit about advantages and disadvantages regarding their design decision. Native threads benefit from OS "visibility" and well-structured priority management (especially in RT contexts) whereas green threads are comparatively lightweight, often by an order of magnitude. However, foundational attempts at green threading in Rust required "book-keeping" overhead exceeding that of native alternatives in the first place [17]. Thus, a native threading design decision partially reduces to a Rust programming language decision less libraries such as Tokio which propose non-blocking alternatives which fit a general green thread "pattern" [18]."

What this means for Anodize is that our native thread design choice is almost practical as it is convenient. Mixed criticality constraints benefit from OS visibility and oversight as long as memory resources aren't at such a premium that a system cannot perform as desired. We gain improved preservation of scheduling guarantees and capacity for reasoning about Anodize powered behavior. However, this decision comes with inherent constraints, which we discuss in section VI-B.

```
struct GroupA;
impl GroupTag for GroupA {
    fn get_tag() -> u64 { 0x41 }
}
```

Fig. 4: Defining a thread group

```
pub fn new() -> ThreadGroup<G>

pub fn channel<V>(&self) -> (
    IntragroupSender<V, G>,
    IntragroupReceiver<V, G>)

pub fn link(&self, from: TaggedThread<G>,
    to: TaggedThread<G>) ->
        (TaggedThread<G>,
        TaggedThread<G>)

pub fn spawn_thread(&mut self,
    t: TaggedThread<G>) -> ()

pub fn spawn(&mut self, f: Thunk,
    senders: Vec<IntragroupSender<
        i32, G>>, receivers: Vec<
        IntragroupReceiver<i32,
        G>>) -> ()

pub fn wait(self) -> ()
```

Fig. 5: `ThreadGroup` interface

E. Assumptions

Anodize relies on an important set of explicit assumptions that provide important constraints that leverage or add to the guarantees provided by Rust.

1) Thread groups and membership are statically known. This allows compile time checking of thread-group associations and valid channels.
2) The primordial thread is the sole entity that maintains functionality to spawn threads. This limit is imposed for maintaining positive control over memory space and scheduler load in a centralized manner.
3) The primordial thread may not send messages to spawned threads, nor may message information be constructed therein at spawn. As a result, all messages are generated, passed, and stored by spawned thread functionality.

4) The scheduler is the only runtime entity aware of thread activity. This is due to the fact that Rust directly relies on the OS scheduler for thread prioritization (i.e., no native green thread support).
5) Thread functionality is restricted only to Anodize library functions. This preserves guarantees otherwise undermined by unsafe and native threading APIs.
6) User-implemented functions or libraries must use safe Rust (violates memory guarantees otherwise).
7) Message protocol is limited to passing `<i32>` types only. This feature allows for near-trivial channel association between threads and other complications that would arise from using generic types.

IV. Implementation

A. Anodize API

The Anodize API, shown in Fig. 5 is centered around the core concept of *thread groups*. A thread group is represented as a structure implementing the `GroupTag` trait as shown in Fig. 4. To implement the `GroupTag` trait, the group structure must also provide a unique runtime identifier in the form of a 64-bit integer. This provides a dual representation of a thread group as both a compile time construct (a type) and as a runtime construct (a unique identifier). Thus, our system can provide both static and dynamic guarantees of isolation.

`ThreadGroup::new()`: Construction of a thread group is done by passing the group tag as a type parameter to the constructor. This associates the created group with the tag and allows for compile time guarantees to be made through the type system.

B. Message Passing

Message passing between threads takes place over channels. Consider the two work functions given in Fig. 6 which implement the classic producer-consumer model. In this simple example, `produce` sends an incrementing integer over the channel once per second, while `consume` accepts the integer and prints it to the console.

`ThreadGroup<G>::link(t1, t2):` As shown in Fig. 7, we use the `link` function to connect threads to each other over a channel. The link function is a member of the `ThreadGroup<G>` structure and is thus aware of the group tag type `G`. Link accepts two threads: A sender `t1` and a receiver `t2`. It consumes the handles provided, and returns two new ones with the connection established between them. This replaces the traditional Rust channel creation API (shown in Fig. 8). Repeated invocations of link can be used to construct multiple channels and bidirectional communication, as shown in Fig. 10. We note that link allows for direct definition of the edges in a communication graph, where threads are nodes and channels are edges.

By making use of the `GroupTag` type `G`, we are able to leverage the type system at compile time to prohibit inter-group channel construction. Fig. 11 shows an instance of an attempted inter-group channel and the resulting compiler error. We find that the Rust type system is able to properly infer the appropriate thread group relationships. This preserves our isolation guarantee while freeing the programmer of the burden of explicitly tagging every value with a group.

```rust
fn produce(s: Vec<Sender<i32>>,
    r: Vec<Receiver<i32>>) {
    let tx = s.get(0).unwrap();
    let _ = r;

    let mut i = 0;
    loop {
        let _ = tx.send(i);
        i += 1;
        sleep(Duration::from_secs(1));
    }
}
```

(a) `producer`

```rust
fn consume(s: Vec<Sender<i32>>,
    r: Vec<Receiver<i32>>) {
    let _ = s;
    let rx = r.get(0).unwrap();

    loop {
        let i = rx.recv().unwrap();
        println!("Consuming {}", i);
    }
}
```

(b) `consumer`

Fig. 6: Producer-Consumer Implementation

```rust
let mut group_a = ThreadGroup::
    <GroupA>::new();
let mut group_b = ThreadGroup::
    <GroupB>::new();

let group_a_t1 = TaggedThread::
    new(produce);
let group_a_t2 = TaggedThread::
    new(consume);

let (group_a_t1, group_a_t2) =
    group_a.link(group_a_t1, group_a_t2);

group_a.spawn_thread(group_a_t1);
group_a.spawn_thread(group_a_t2);

group_a.wait();
```

Fig. 7: Link functionality

C. Thread Lifecycle

Thread management is performed exclusively through the `ThreadGroup` structure. Threads can be created as first creating a `TaggedThread`, which is in essence a handle by which the thread can be referred to before spawning. This handle can be used to create channels via the aforementioned link construction and then spawned once ready.

`TaggedThread::new(f):` Threads are created by directly constructing a `TaggedThread` structure. The provided function is used as the computation when the thread is spawned. We note that the provided function must be a true function pointer and not a closure. This requirement prevents the sharing of values captured by the closure between thread groups. When no explicit annotation of thread group is provided, the compiler will infer the connection based on the use of the resulting value. This maintains our isolation guarantees, while again reducing the annotation burden placed on the programmer.

ThreadGroup::spawn_thread(t): Once all channels are constructed, the thread can be spawned. This starts the computation provided earlier and internally tracks its completion. Doing so consumes the passed thread structure, preventing the spawning of a given thread more than once. This is necessary to preserve the point to point nature of the channels, by which we currently guarantee a single producer and single consumer per channel.

ThreadGroup::wait(): After spawning threads, the primordial thread must not terminate or the whole program will. Thus, we provide a wait function that allows work within the thread groups to take place and blocks the primordial thread until the all threads in the thread group have completed. In a typical use, this would be done sequentially across all thread groups to prevent termination of the program.

D. Ensuring Bounded Memory

We implement per thread group memory bounds at runtime by redefining the global allocator in Rust [7]. Our new allocator maintains a running total of the memory used by each thread group along with a configurable bound on allowed allocations. As a thread requests memory, the requested size is added to the current total and the new total is compared with the configured bound. If the bound is met, the allocation is passed through to the underlying system allocator. We observe that the memory safety guarantees Rust provides ensure that allocations by one group cannot corrupt allocations of another. This allows us to safely use a single heap for all thread groups. Correspondingly, the deallocator subtracts from the total allocated size and invokes the system allocator to mark the memory as freed. If the bound is exceeded, the allocator triggers a panic. By doing so, the thread is terminated and an error result returned to the thread group. We note that any resources the thread was holding at the time of the panic may be left in an inconsistent or inaccessible state. Thus, properly handling the release of shared resources in our framework is left as a problem for future work.

Rust features robust support for thread local storage, which we leverage in implementing these memory bounds. Each thread maintains a thread group identifier at runtime in its thread local storage. These identifiers are statically specified in the implementation of the `GroupID` structure. During an allocation, our custom allocator accesses the thread local storage to obtain the current thread group identifier. By storing this information in thread local storage, we decouple the allocator from the thread scheduler. The group identifier is used as a key in a map structure storing the current total allocation size and the configured bound. In the current implementation, this is done by using a single lock over the entire map structure. A trivially improved version would leverage a lock free data structure to avoid mutual exclusion in the allocator.

```
    // Create mpsc channel between threads
    let (ch1_a_tx, ch1_a_rx) =
        group_a.channel::<i32>();
    let (ch2_a_tx, ch2_a_rx) =
        group_a.channel::<i32>();

    // Spawn two threads where the second
    //    argument is a list of transmitters
    // and the third argument is a
    //    list of receivers (mpsc)
    group_a.spawn(produce, vec![ch1_a_tx],
        vec![ch2_a_rx]);
    group_a.spawn(consume, vec![ch2_a_tx],
        vec![ch1_a_rx]);
```

Fig. 8: Channel creation without `link`

```
fn intergroup_messages_with_linking() {

    let mut group_a = ThreadGroup::<GroupA>::new();
    let _group_b = ThreadGroup::<GroupB>::new();

    // Create two TaggedThreads. T1 will be in
    // group A and will produce.
    // T2 will be in group B and consume
    // T1's output.
    let group_a_t1 = TaggedThread::new(produce);
    let group_b_t2 = TaggedThread::new(consume);

    // This compiles because group_b_t2 and
    // group_a_t1 are not tied to groups yet.
    let (group_a_t1, _group_b_t2) = group_a.link(
        group_a_t1, group_b_t2);

    group_a.spawn_thread(group_a_t1);
    _group_b.spawn_thread(group_b_t2);
}
```

Fig. 9: Intergroup message passing attempt

```
fn main() {
    //... Declare group

    // Link t1 and t2
    let (group_a_t1, group_a_t2) =
        group_a.link(group_a_t1, group_a_t2);

    // Link t2 to t1
    let (group_a_t2, group_a_t1) =
        group_a.link(group_a_t2, group_a_t1);

    group_a.spawn_thread(group_a_t1);
    group_a.spawn_thread(group_a_t2);

    group_a.wait();
}

fn thread1_function(s: Vec<Sender<i32>>,
    r: Vec<Receiver<i32>>) {
    let tx = s.get(0).unwrap();
    let rx = r.get(0).unwrap();
    //...
    }
}

fn thread2_function(s: Vec<Sender<i32>>,
    r: Vec<Receiver<i32>>) {
    let tx = s.get(0).unwrap();
    let rx = r.get(0).unwrap();
    //...
    }
}
```

Fig. 10: Bidirectional `link` implementation

E. Priority-Based Thread Scheduling

We allow the application developer to specify each thread group's priority as an integer in the range `[1, 99]`, which corresponds to the allowed `libc` priority range. Then, we use the `libc` Rust crate to directly call `pthread_setschedparam` with the specified priority and scheduling policy, FIFO. These directly set the priorities of each thread in the OS scheduler.

V. TinMan: A Mixed Criticality System

To test Anodize's efficacy in providing temporal and spatial isolation guarantees, we built *TinMan*, an example mixed criticality system written entirely in Rust and built on top of Anodize. TinMan uses the Anodize library and implements a primordial thread that spawns four thread groups: a visual odometry program, a robot keyboard controller program, a path planner program that lies dormant until told to run, and a simple program that periodically tells the path planner to turn on. All four thread groups interface with ROS2, an extremely common and de facto standard library for robotics applications, by using a Rust/C++ bindings library for ROS2.

TinMan is a simple but illustrative example of a mixed criticality system; it features a computationally-heavy, soft-RT, always-on program (visual odometry), a cyclic program (path planning), RT robot control (keyboard controller), and relies upon the standard robotics library (ROS2) for message passing and program control flow. Because all of TinMan is compiled together as one package, all the Rust code (Anodize, the primordial thread, the visual odometry, the keyboard controller, the path planner, and the path planner timer) is subject to Anodize's compile-time code-checking. Further, while each logically-independent program is in its own thread group, Anodize can manage memory usage and OS-level thread scheduling for all of them.

In lieu of a real-world experimental setup, with TinMan running on a physical robot in a real-world physical space, we created two separate executables that lie outside the mixed criticality system and can be used to visualize the robot. The first executable visualizes the estimated robot trajectory and input video stream in the visual odometry module. The second executable sits on top of *Gazebo*, a popular world visualizer that is frequently interfaced with ROS/ROS2, and visualizes the robot's movement in a simulated "world". These two programs are not managed by Anodize, nor spawned in the mixed criticality system, and running them is not required for TinMan to function properly. They are both written in C++ and, like TinMan, are reliant upon ROS2, although they can interface with the ROS2 library directly.

See Fig. 13 for an overview of TinMan and its associated visualizers.

A. ROS2

ROS/ROS2 is a robotics library written in C++, with optional Python bindings, that provides a framework and

```
error[E0308]: mismatched types
  --> src/main.rs:76:27
   |
76 |     _group_b.spawn_thread(group_b_t2);
   |                           ^^^^^^^^^^ expected struct `GroupB`, found struct `GroupA`
   |
   = note: expected struct `phantom_groups::thread_groups::TaggedThread<GroupB>`
              found struct `phantom_groups::thread_groups::TaggedThread<GroupA>`
```

Fig. 11: Compile time error: Inter-group channel

```
static GLOBAL_INT: i32 = 0;
lazy_static! {
    static ref GLOBAL_MUT: Mutex<i32> = Mutex::
        new(GLOBAL_INT);
}

fn main() {
    //...
    group_a.spawn(counter1, vec![], vec![]);
    group_b.spawn(counter2, vec![], vec![]);
    //...
}

fn counter1(_s: Vec<Sender<i32>>,
    _r: Vec<Receiver<i32>>) {
    loop {
        let mut num = GLOBAL_MUT.lock().unwrap();
        *num += 1;
        println!("Thread 1 '{}'", num);
        sleep(Duration::from_secs(1));
    }
}

fn counter2(_s: Vec<Sender<i32>>, _r: Vec<
    Receiver<i32>>) {
    loop {
        let mut num = GLOBAL_MUT.lock().unwrap();
        *num += 1;
        println!("Thread 2 '{}'", num);
        sleep(Duration::from_secs(1));
    }
}
```

Fig. 12: Shared memory access via global mutexes

abstracts common tasks for programming robotics applications. Particularly pertinent to TinMan is ROS' pub/sub system, which creates a programming paradigm wherein *nodes* (programs) *publish to* (send messages to) and *subscribe to* (receive messages from) *topics* (a message queue). There can be and frequently are multiple nodes within one robot, as each node just does one logical task in the subset of tasks in the robot, and nodes can publish and subscribe to multiple topics at once. This ultimately turns ROS-based programs into event-driven programs, where nodes react to and perform computations based on their subscribed topics, and spur other nodes to begin their own work by publishing to their published topics. This pub/sub system is important to TinMan for two reasons. First, the system matches how many robotics applications are structured, and TinMan should accommodate ROS if we want to accurately represent *most* robotics applications. Second, the pub/sub system creates a convenient way for structured message-passing between threads whose use would benefit Anodize and TinMan.

Up until now, we use ROS2 and ROS seemingly interchangeably. ROS and ROS2 are technically separate libraries and applications, where ROS2 was a re-imagined successor to ROS. While most of the behind-the-hood implementation differences between the two libraries are unimportant for the purposes of TinMan and Anodize, two stand out: ROS2

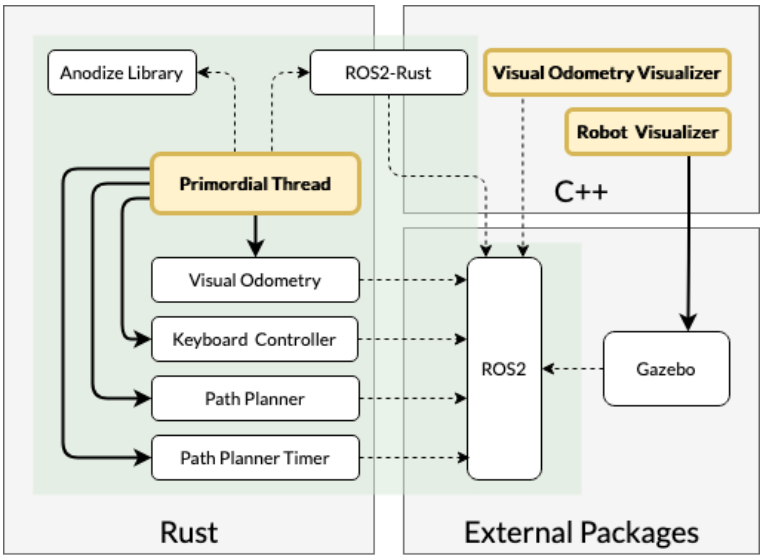

Fig. 13: The build chart for TinMan and associated visualizers.
Each rounded rectangle is a logical software module and its grey bounding box is its source code language. The software modules inside the opaque green box are part of TinMan. All other modules are external and optional system visualizers.
Yellow software modules are top-level executables – that is, programs that are independently run as separate command-line executables. A *dotted* arrow indicates a dependency on the pointed-to software module. A **bold** arrow indicates a software module spawning another.

provides hard real-time support on acceptable operating systems (while ROS does not), and ROS2 removes the need for a centralized `roscore` node for managing inter-node communication.

To incorporate ROS/ROS2 into TinMan, we used the open source library *ros2-rust*, which creates Rust-level bindings for most ROS2 C++ functions [12]. TinMan's dependency on *ROS2* specifically is borne from its dependency on ros2-rust, as ROS2 bindings cannot be trivially interchanged for ROS bindings. Using ros2-rust allows us to write code in Rust that is idiomatic to ROS/ROS2. We can create a node, a topic, and publish to the topic in our Rust code, then subscribe to the same topic *in a C++-based ROS2 node*. This allows us to very cleanly communicate between our Rust-based mixed criticality system and our C++-based visualizers, as well as to let TinMan communicate with *physical hardware* without any extra work on our part, because ROS2 already manages the interface between software nodes and hardware.

B. Visual Odometry

In robotics and computer vision applications, the visual odometry task is the process of calculating an estimated trajectory from an input video stream and is commonly deployed on mobile devices of all forms (e.g., a self-locomoting robot, an AR headset, a mobile phone, or a driverless car). Further, it is moderately resource-heavy and has soft RT requirements; while video frames can be dropped, dropping too many increases the trajectory error, and while the program can "lag" behind its RT input, the mobile device often has to *immediately use* the results of the visual odometry program, so it can't fall behind so far that its results are always late.

In most cases, the trajectory error of the visual odometry algorithm matters for the efficacy of the application. However, because TinMan is just a proof-of-concept to test the Anodize library, we have less strict requirements for the error and have some freedom with which visual odometry algorithm we use. Thus, we used an open source visual odometry application written in Rust (henceforth: *Vors*) [10]. Vors' input is a sequence from the TUM RGBD dataset, which is a commonly used visual odometry/visual SLAM dataset. Each sequence contains a series of RGB images and associated depth images which form an RGBD video when played together in order. While Vors was written to accept a TUM RGBD dataset, it can easily be modified to accept *any* RGBD-based visual odometry/visual SLAM dataset as long as the dataset provides a camera calibration file (this is customary for these datasets). To test the visual odometry application, we used the `fr1/desk2` sequence from the TUM RGBD dataset and achieved an ATE (average trajectory error) of .16m over a total trajectory of 10.161m, which is a reasonable, but not amazing, ATE for a visual odometry application.

We modified Vors slightly so it spawns in an Anodize thread group, reads the location of the RGBD associations file from the TinMan configuration file (rather than command-line arguments), creates a ROS2 node, and publishes the calculated robot pose to the `robot_pose` topic. This allows our visualizer to display the results of the visual odometry algorithm without having to run within the mixed criticality system.

C. The Path Planner and Its Timer

In robotics, path planning refers to the task of planning a robot's movement from a start position to an end position while avoiding obstacles. Unlike the visual odometry task, its RT requirement is generally more relaxed, although it still depends on the particular application. We modified *OpenRR/gear*, an open-source path planning algorithm implementation in Rust [11]. This implementation takes a starting location, a goal location, and two `urdf` files as input: one that specifies the robot's geometry and kinematics, and another that specifies the location of obstacles.

We modified OpenRR/gear so it spawns in an Anodize thread group, creates a ROS node for itself, and does nothing more until prompted thusly. Prompts are retrieved by subscription to the `execute_plan` topic and the path

planning algorithm runs on the same input every time information is published to this topic. Simple, prospective future modifications retrieve a different random input each time – a different starting/goal position, a different world of obstacles, or robot motion model per use case. However, we do not care about creating a realistic robot as much as we care about simulating a typical path planning task, so we did not spend too much time worrying about linking the path planning module to a more real-world input.

While the path planner listens for a cue to run its algorithm, the path planning timer runs in a separate Anodize thread group and cyclically publishes to the `execute_plan` topic. This allows us to use `ROS2::spin` to control the cyclic rate at which the path planner runs. However, it should be noted that ROS2 does *not* guarantee that it can meet hard RT requirements with any of its timing-based functions. However, because path planners will generally be run on the request of some other module, and not as a hard RT task, we believe this is sufficient for our use case.

D. Visualizers

Lastly, to aid in visualizing TinMan's actions, we incorporated two C++-based visualizers that interface with TinMan. These visualizers are neither necessary for TinMan to function, nor is their memory and OS-level thread scheduling managed by Anodize. They simply exist to visualize the otherwise difficult to understand output of the mixed criticality system, and verify that everything is working correctly.

The first visualizer is the *Visual Odometry Visualizer*. This program creates a node that listens to the `robot_pose` topic, which the visual odometry thread group publishes to. The visualizer shows the incoming video stream that the visual odometry thread group is processing, and shows the trajectory of the robot as a series of dots in a 2D space. This application can easily be made to visualize a RT video stream recorded from a camera (rather than one played from a file): it would merely need to subscribe to the topic associated with the camera and display those images instead of the ones from the video file. We primarily used the code from *MonoVO*, a C++ and ROS-based visual odometry visualizer, but ended up changing quite a bit of it to work with our current system [13].

The second visualizer is the *Robot Visualizer*. This program primarily spawns Gazebo, a commonly-used simulation program that can interface with ROS/ROS2, with a TurtleBot in a simulated world [14]. The TurtleBot is both a physical hardware kit to build a robot, as well as open-source software to make programming the robot easier. ROS/ROS2 provides many built-ins for TurtleBots (e.g., a urdf file that specifies the TurtleBot's geometry) that make them easy beginner robots for plug-and-play applications where the actual specific robot type may not matter. This is true for our case, where TinMan just needs to run on *any* robot.

The robot visualizer listens on the `cmd_vel` topic for changes to the robot's linear and angular velocity, then displays the current robot and world state in Gazebo's GUI. This can be swapped to a "real world" robot because ROS/ROS2 provides an abstraction for the physical TurtleBot hardware to respond to changes the `cmd_vel` topic.

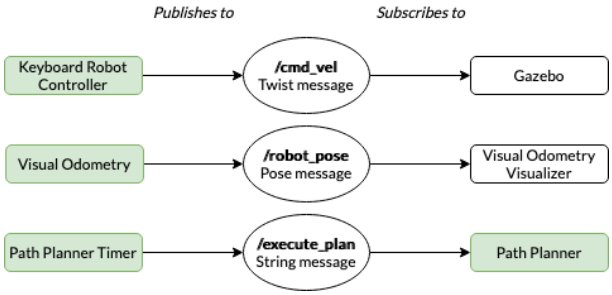

Fig. 14: The ROS computation graph for TinMan and associated visualizers. ROS nodes are rounded rectangles; ROS topics are ovals. Nodes in green are thread groups spawned by and managed by the Anodize library.

E. Deployment

TinMan is deployed on RT-Linux patched Ubuntu 20.04 with kernel 5.4.74-rt41 which was the current, stable v5 RT kernel (20.04 default) at the time of production. We describe scheduler adherence requirements met by RT-Linux in III-D. Ubuntu version 20.04 is the 'pseudo-common' development platform for distribution ROS2 Foxy Fitzroy (Foxy, current at production-time) and ROS2-Rust. Ubuntu version 18.04 LTS is unsupported by Foxy[9] and appears incapable of coordinating Foxy and ROS2-Rust given compilation issues which resolved after the upgrade to Ubuntu 20.04. Conversely, Ubuntu 20.04's commonalities with 18.04 LTS are sufficient for supporting ROS2-rust without complication despite it's 18.04 LTS development mode. Lastly, the above assortment benefits from a one-line semi-minimal package installation for Gazebo through APT (`ros-foxy-gazebo-ros-pkgs`).

F. Extended Communication Patterns

We now consider an extended version of TinMan with a more complex communication graph. The communication graph in Figure 14 outlines the exact communication graph of the threads in TinMan; however, while this graph sufficiently explains the communication pattern in TinMan, it is not a representative example of a full-fledged robotics system. Namely, a production-ready robot will likely use the output of its visual odometry algorithm to inform its path planning algorithm, implement a more robust robot control than keys on a terminal, and have some mechanism for fusing user-level robot locomotion commands with executing the results of its path planning algorithm. Thus, Figure 15 is a more accurate representation of the communication within a "real" robot, performing the same tasks and goals as the robot in TinMan.

We now examine a formalization of the extended communication graph as a multiparty session type. We adopt the notation used in [15], with the abbreviation of eliminating

the labels in communications with only one branch. As all of the given types are infinitely recursive, we also omit the outer recursive loop construct. The global type is given by Fig. 16. The rounded rectangle in the diagram represent the participants in the communication. The White ellipses are ROS topics, which serve as the typed communication channels across which our participants communicate. Thus in the global type we examine all communication interactions in the system, with the entities being the participants (rounded rectangles) and the types being determined by the topics (ellipses).

In Fig. 17, we show the *projection* of this global type on to each participant in the system. The projected local session type is similar to the binary session types shown in Sec. II-C. It shows the interactions between a single participant and the rest of the system. This specifies the partner involved in every communication, the type of value to be passed, and the direction in which it moves. This type is derived directly from the global type by the projection operation (the full definition of which is given in [15]).

VI. Evaluation and Discussion

A. Rust's Thread Memory Overhead

The Anodize library relies on a properly configured program specification for the mixed criticality system that it manages; threads should be grouped into appropriate thread groups, given appropriate memory bounds and scheduling priorities, and the communication patterns between them should be known. In order to accurately set the bounds for each thread group, an application developer would need to know about any "hidden", non-application-specific memory allocation performed in their program. In particular, since Anodize enforces a system structure wherein programs are modeled as threads within thread groups, of particular interest is the allocation required for Rust to spawn a thread. These bytes will theoretically count as memory allocated by a thread, but they are bytes that the application itself cannot directly use to move forward in its computation. Thus, it would be helpful to know about this overhead so a developer can pad their specified memory bounds with the appropriate amount of bytes. To reiterate, we primarily care about the required memory overhead for *heap-allocated* data structures, as stack-allocated data structures can be determined statically and are not managed by Anodize's memory bounding. See III-C for a longer discussion about heap- vs. stack- allocated memory and their memory bounds.

We found that the *spawning* thread (in most cases, the primordial thread is the sole spawning thread) "owns" the memory needed to spawn a thread; because of this, a *spawned* thread can use all the memory within its specified memory bounds for its own tasks, and the application developer does not need to worry about accounting for Rust's thread management overhead when assigning memory bounds to thread groups. The primordial thread does not perform any computation outside of setting up the system and letting its spawned threads run, so it would not typically be memory-

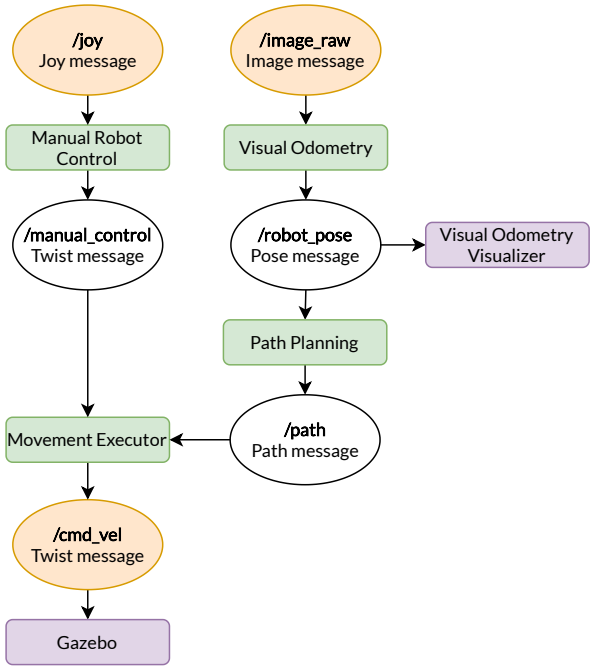

Fig. 15: Communication graph for an extended TinMan.

$$
\begin{aligned}
G = \; & Joy \to Control : Joy. \\
& Control \to Move : Twist. \\
& Camera \to VisOdom : Image. \\
& VisOdom \to Visualizer : Pose. \\
& VisOdom \to Planner : Pose. \\
& Planner \to Move : Path. \\
& Move \to Gazebo : Twist
\end{aligned}
$$

Fig. 16: Global type for an extended TinMan.

$$
\begin{aligned}
Joy &= !Control(Joy) \\
Control &= ?Joy(Joy).!Move(Twist) \\
Camera &= !VisOdom(Image) \\
VisOdom &= ?Camera(Image).!Visualizer(Pose). \\
&\quad\; !Planner(Pose) \\
Visualizer &= ?VisOdom(Pose) \\
Planner &= ?VisOdom(Pose).!Move(Path) \\
Move &= ?Control(Twist).?Planner(Path). \\
&\quad\; !Gazebo(Twist) \\
Gazebo &= ?Move(Twist)
\end{aligned}
$$

Fig. 17: Local types for an extended TinMan.

bound; thus, the application developer can safely abstract away the bytes needed for Rust's thread management.

However, in the cases where 1.) the primordial thread *does* have some computationally-rich task and an associated memory bound, and 2.) a non-primordial thread may need to spawn a thread of its own (e.g., a temporary worker thread), we would still like to know how many bytes are actually allocated for Rust's thread management overhead. To test this, we created a simple program wherein the primordial thread spawns Thread A with some memory bound, Thread A spawns Thread B with no memory bound, and Thread A allocates a `Box` containing an `i32` in an infinite loop. The Rust `Box` object is a heap-allocated data structure than can hold any data structure within it.

Given this setup, we would like to see how many bytes Thread A is capable of explicitly allocating in its infinite loop before it is shut down by the allocator. Anodize's thread management itself introduces 40 heap-allocated bytes corresponding to the handle of the spawned thread, and we found that Rust's thread management introduced another 288 bytes. We verified this for different memory bounds, as well – 2^{10}, 2^{15}, and 2^{20}. The 288-byte overhead remained constant for all of them. Thus, an application developer can safely pad their specified memory bound by **328 bytes** to account for Anodize's and Rust's thread management overhead.

B. Limitations

Global Memory Access We find that, despite Anodize's stated goal of preventing unmoderated inter-thread communications, *global* memory access across thread groups is possible. Figure 12 illustrates this issue in terms of global mutex functionality: A shared global int `num` is iterated concurrently by members of either thread group. The prospective fix to this issue relies on "lint" configuration for disallowing global memory access. Lints are managed by now-disallowed Rust compiler plugin libraries. It is not clear whether lint features will be reconfigured for future use at this time. Our mixed criticality system cannot offer strict global memory access compile-time time guarantees in the event that lints are not replaced by procedural macros or otherwise.

Message Passing The current implementation of Anodize only allows threads the capability to communicate *within* their thread group, and they can only send integers to each other. Many, if not most, production-level mixed criticality systems, need the capability for threads to communicate *outside* their thread group, with more varied and complex data structures than integers. TinMan, as an example mixed criticality system built on top of the Anodize library, highlights this shortcoming – TinMan's threads largely use ROS2 for message passing between thread groups, and don't rely on Anodize's built-in message passing at all. While the reliance on ROS2 gives TinMan the additional ability to trivially interface with hardware and leverage ROS2's inbuilt functions – benefits that Anodize, even with fully functional message passing, could not offer – it comes with its own tradeoffs and peculiarities as well.

First, the reliance on ROS2 transforms the structure of the system to be event-driven. While this should be ok for most cases, it does add a limitation that may be undesirable.

Second, ROS2 advertises its real-time functionality compared to its predecessor, ROS, which can be achieved with a RT-patched linux OS (which TinMan uses) or another real-time-compliant operating system. However, it is unclear whether the `ros2-rust` library or *any* ROS2 Rust C++ bindings library can meet these guarantees as well. Lastly, ROS2 does not come with any inherent memory problems – ROS2 itself is stable, and the Rust portion of the `ros2-rust` library is managed by Rust's ownership system. However, for some mixed criticality systems, placing trust in an external library may be unacceptable.

As stated earlier, ROS2 does provide some functionality that Anodize would not be able to (without essentially re-implementing ROS2 in Rust). The ideal version of Anodize would ideally allow for ROS2 integration, but provide purely Rust-level inter-group message passing.

Native Threads Further, some constraints arise due to our choice of OS/native threads over green threads.

First, losing green thread flexibility means that Anodize-powered mixed criticality systems are forced into preemptive thread switching rather than process-thread context switching. This cost does not come cheaply, and other mixed criticality systems, such as Fiji MultiVM (see VIII), benefit from green thread context switching. Namely, mixed criticality components (programs, or threads in our context) leverage virtual runtime memory for establishing beneficial, swift concurrent operations without *requiring* multi-core support [5].

Second, (ideal) green thread overhead benefits enable substantial thread counts that would otherwise exhaust system memory, especially when tightly bounded as in Anodize. Anodize-powered mixed criticality environments survive this worry with conservative thread deployment. For example, using (RT-constrained) threads as 'launchers' rather than computational 'workers' is a thread-minimal implementation which leaves substantial room for deeper native thread functionality. An example of a deeper TinMan implementation is writing-in native thread topic message management rather than leaving it to ROS functionality.

Thread Group Termination Lastly, native thread scheduling forces POSIX thread (pthread) library management for handling thread lifetime protocols. This causes a significant limitation due to the interactions of `pthread_cancel` with Rust's internal thread bookkeeping. Notably, Rust maintains a per thread data-structure called a `JoinHandle` which tracks the final result of a thread's computations. When a thread is terminated via a panic or completion of the body, this handle is updated with an `Err` value if the thread panicked or an `Ok(result)` if the thread completed successfully. When a thread is cancelled (via `pthread_cancel`), this data-structure is not properly updated, leaving it in an inconsistent state. This causes the main thread to panic as it attempts to access the `JoinHandle`.

Further complicating the problem, the terminated group

leaks all memory allocated. This stems from the fact that we use the underlying system allocator to manage the heap. Because of this, we track only the total number of bytes allocated by the group. To clean up the group's memory properly, we would also need to know which of the allocations belong to the group. We see two major approaches to this problem, further discussed in Sec. VII.

In summary, we find that Anodize's native thread design promotes some domain specificity preference for proposed Anodize-powered mixed criticality environments: Sparser systems that highly prioritize safety and/or fault tolerance stand to benefit from Anodize over those that require high computational complexity or raw volume in mixed criticality components (i.e., more than a few dozen ThreadGroups). This idea inspires a rough applicational comparison between a prospective Anodize deployment on chipset that controls a robotic laser cutter over that which manages a deeply interconnected automation suite of a UAV.

VII. Future Work

Anodize stands to benefit from improvements to its scheduling mechanism. Currently, Anodize uses an RT scheduler that responds to hard-coded monotonic priority assignments per thread *group* wherein individual thread priorities are undefined. Importantly, thread group task *independency* is expected. Our sparse and straightforward implementation provides benefits for initial analysis and proof of concept development. However, future applied Anodize implementation almost certainly benefits a more elaborate RT model than a 'set-and-forget' methodology simply handed over to an end user.

Initial worries include potential deadlock circumstances wherein ThreadGroup *interdependency* (i.e., inter-ThreadGroup or inter-program message passing or memory sharing) is enabled. For example, let top task-priority instance T_1 interface with respective lesser priority tasks T_2, T_3. T_1 blocks for response from T_3, but T_2 is processor intensive and T_3 starves. In the event that T_3's priority is *not* underassigned, fixed mixed criticality priorities are not sufficient for Anodize when ThreadGroups intercommunicate.

Can a straightfoward, temporary priority switch/realignment between two deadlocking tasks fix the breadth of issues within a system? We hypothesize that detection and priority switching mechanisms can be coordinated between thread groups themselves with intervention from the (chief priority) primordial thread. This project is non-trivial, and skepticism is warranted that a straightforward proposal scales well to n potentially all-interconnected tasks. Most importantly, 'thread management' tasks from the primordial thread must not starve the operation of the system or the experiment fails from the onset.

Soneoka et al. are pessimistic about evaluating RT algorithm "quality" outside of measuring simple timeliness goals. Further, "domain specificity" (e.g., medical equipment, machinery safety, UAV/UAS platforms) is prescribed wherein basic requirements for each individual system drive its evaluation criteria [16]. However, some classification helps categorize a solution set mapping to domain-specific needs. These classifications indicate instances where leveraging variable priority preemption, re-prioritizing by interruption status, and developing special OS features give a better concrete RT prescription per use case.

For Anodize, this means that the preservation of many pertinent mixed criticality guarantees are contingent on the RT adjustments that satisfy the needs of a domain. It follows that complexity in a domain's RT 'controls' such as deadlock management is inversely proportional to the likelihood that mixed criticality guarantees are either a) upheld, or b), can be reasoned about in a formal, strong, or meaningful way.

Memory management in Anodize currently lacks some desirable tracking of groups in allocations. This is necessary to implement proper clean up of failed groups or groups with exhausted memory. We see two major approaches to this problem: tagging and segmented heaps. The tagging approach modifies the allocator to attach a group tag at the beginning of every allocation, the same way current allocators track allocation size. This minimizes redundancy in the heap structure as all groups still share the same heap structure. Because of Rust's memory safety guarantees, this is safe as the language protects ill-constructed programs from overrunning memory bounds of heap allocations (subject to the typical constraint of using only safe Rust). To clean up a group in this approach, we traverse the entire heap, freeing any allocations tagged as belonging to the terminating group. While more expensive than perhaps necessary, this allows for efficient, zero-copy migration of allocations between groups.

Another approach, more similar the techniques used by multi-VM systems, is to segment the heap into per group areas. This involves keeping separate heap data structures for each group. This approach increases memory overheads related to heap management, but allows quick and easy cleanup. To clean up, we simply reset the entire heap structure at once, leaving all prior allocations behind as free space. This carries the additional consideration that we cannot overcommit the heap. With a tagging approach, the total of all memory bounds may exceed the total memory available to the system. This may be useful if the probability of all groups filling their memory bounds concurrently is minimal and acceptable. Such a system could allow low critically groups to have access to more memory while terminating them if the overall system memory is too low. We note the similarities to Out-Of-Memory (OOM) management techniques common in desktop operating systems such as Linux.

During the development of the Anodize system, we observed an interesting opportunity for for optimization of inter-group communications. Again, note the current implementation of Anodize lacks communication between groups to avoid potential issues with blocking communication. After addressing the problem of blocking, we see an interesting avenue for optimization. Leveraging Rust's borrowing system, we can be assured that once a group has passed an allocation, that memory can no longer be accessed by that group. This would allow an implementation (assuming a

tagging approach to group memory) to simply relabel the heap allocation as belonging to the destination group. This avoids the costs associated with serialization and memory copying present in a process isolated or hardware isolated system.

In summary, Anodize observes trade-offs wherein sparse use of independent ThreadGroups gives yields a best chance of mixed criticality guarantee preservation, but may fail at realizing a complicated domain-specific goal. Conversely, a high enough resolution domain-specificity may reduce a problem to one better-solved by a timelier system that faces some measure of undefined behavior. The net domain-specific benefit of using/not-using Anodize per domain is not well-drawn, but our proposal gives us good reason to believe that systems organized similar to TinMan are a good fit when a small subset of high-priority ThreadGroups introduce high-risk during unhandled failures and are mostly independent from those at lower priorities.

VIII. Related Work

Ziarek et al. present the Fiji MultiVM Architecture (Fiji MultiVM): A RT mixed criticality system written in the Java programming language. Fiji MultiVM provides time, space and resource isolation functionality at the language level which achieves the shared goal with Anodize. FijiVM was analyzed and benchmarked on embedded hardware used for cochlear implants and a UAV flight controller with a case study supporting the latter implementation compared with its 'singular' FijiVM predecessor [5].

Jorge Aparicio Rivera recently published his 2020 masters thesis on the development of a Rust API for the domain-specific language RT for the Masses (originally "layered" on top of C). Rivera's API took explicit advantage of Rust's memory safety and concurrency support while maintaining acceptable overhead in multi-core RT implementation [8].

References

[1] Thomas Bracht Laumann Jespersen, Philip Munksgaard, and Ken Friis Larsen. 2015. Session types for Rust. In Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming (WGP 2015). Association for Computing Machinery, New York, NY, USA, 13–22.

[2] K. Honda, V. T. Vasconcelos, and M. Kubo, "Language primitives and type discipline for structured communication-based programming," in Programming Languages and Systems, vol. 1381, C. Hankin, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 122–138.

[3] Burns, Alan, and Robert Davis. "Mixed criticality systems-a review." *Department of Computer Science, University of York, Tech. Rep*. 2013. https://www-users.cs.york.ac.uk/burns/review.pdf. Accessed on October 2, 2020.

[4] Pellizzoni, Rodolfo, et al. "Handling mixed-criticality in SoC-based RT embedded systems." Proceedings of the seventh ACM international conference on Embedded software. 2009. https://dl.acm.org/doi/pdf/10.1145/1629335.1629367. Accessed on October 2, 2020.

[5] Ziarek, Lukasz and Blanton, Ethan. "The Fiji MultiVM Architecture." *Concurrency and Computation: Practice and Experience*. *JTRES '15*. 2015. https://dl.acm.org/doi/10.1145/2822304.2822312. Accessed on October 9, 2020.

[6] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. 1994. An Interaction-based Language and its Typing System. In *PARLE'94 (LNCS)*, Vol. 817. 398-413.

[7] Klabnik, Steven et al. "Custom Allocators" *The Rust Book*. 2015. https://doc.rust-lang.org/1.9.0/book/custom-allocators.html. Accessed on October 10, 2020.

[8] Rivera, Jorge A. "RT Rust On Multicore Microcontrollers." Thesis. 2020. http://www.diva-portal.se/smash/get/diva2:1391552/FULLTEXT02.pdf. Accessed on October 13, 2020.

[9] ros-infrastructure, "ROS 2 Foxy Fitzroy" *ROS Index*. June 2020. https://index.ros.org/doc/ros2/Releases/Release-Foxy-Fitzroy/. Accessed on November 14, 2020.

[10] https://github.com/mpizenberg/visual-odometry-rs

[11] https://github.com/OpenRR/gear

[12] https://github.com/ros2-rust/ros2_rust

[13] https://github.com/tjchase34/MonoVO/tree/dev

[14] https://www.turtlebot.com/

[15] Coppo M., Dezani-Ciancaglini M., Padovani L., Yoshida N. (2015) A Gentle Introduction to Multiparty Asynchronous Session Types. In: Bernardo M., Johnsen E. (eds) Formal Methods for Multicore Programming. SFM 2015. Lecture Notes in Computer Science, vol 9104. Springer, Cham. https://doi-org.gate.lib.buffalo.edu/10.1007/978-3-319-18941-3_4

[16] Soneoka et al. "Highly multi-tasking real-time systems and their evaluation." 1993 Proceedings Real-Time Systems Symposium, Raleigh Durham, NC, USA, 1993, pp. 249-252, doi: 10.1109/REAL.1993.393493.

[17] Steven Klabnik. "Rust's Journey to Async/Await." *Presentations*. October 2019. https://www.infoq.com/presentations/rust-2019/. Accessed on December 20, 2020.

[18] https://docs.rs/tokio/0.3.6/tokio/task/index.html