Panic Recovery in Rust-based Embedded Systems

Zhiyao Ma, Guojun Chen, and Lin Zhong
Yale University
{zhiyao.ma, guojun.chen, lin.zhong}@yale.edu

Abstract

Stack unwinding is a well-established approach for handling panics in Rust programs. However, its feasibility on resource-constrained embedded systems has been unclear due to the associated overhead and complexity. This paper presents our experience of implementing stack unwinding and panic recovery within a Rust-based soft real-time embedded operating system. We describe several novel optimizations that help achieve adequate performance for a flying drone with a CPU overhead of 2.6% and a storage overhead of 26.0% to recover from panics in application tasks and interrupt handlers.

1 Introduction

Fault tolerance is highly desirable for embedded systems that have to achieve high availability without hardware redundancy. A system can cope with various types of faults, ranging from language exceptions to hardware errors that manifest as such exceptions, if the system returns to some good state. For example, checkpoint/restore is popular with server systems. Despite early theoretical investigation [19, 24], it is not widely used in embedded systems due to the high performance and memory overhead.

This work considers an alternative approach: return the system to some previous state by cleaning up the resources allocated since. While this is formidably difficult for embedded systems written in popular languages like C, our key insight is that Rust, a safe language emerging in the embedded world [7, 16, 23], provides a new opportunity via its error handling mechanism, especially panic.

The occurrence of a Rust panic indicates a fatal error: the program has reached an unexpected state. In embedded systems, a Rust panic typically results in either halting the system [11], optionally with messages printed [12, 13, 15], or trapping the execution to a hardware handler [1].

We investigate the use of stack unwinding to return a panicking embedded system to a good state from which it may recover from the offending fault. Unwinding involves the forced return of active functions, releasing acquired resources along the way, until a `catch_unwind()` statement is reached. Theseus [4], a recent Rust-based OS, features a stack unwinder for resource cleanup. It shows promising results, incurring approximately 34% storage overhead, but within the context of powerful x86_64 systems.

Nevertheless, the feasibility of implementing a Rust unwinder on embedded systems remains unclear. Indeed, Renwick et al. [20] discourage the adoption of a C++ unwinder for embedded systems due to the large storage overhead. Other challenges for embedded systems include all interrupt handlers sharing one call stack, the lower performance of CPU, and the limited tolerance for memory overhead.

In this work, we address the feasibility question through three primary contributions. (*i*) We describe and evaluate the first stack unwinder implementation in the context of a soft real-time embedded OS flying a quadcopter. (*ii*) We quantify for the first time the overheads associated with stack unwinding for Rust in a microcontroller-based embedded system. (*iii*) We present multiple optimizations that mitigate the negative effects of Rust panic on other system components and improve the recovery speed.

2 Background

2.1 Rust Panic

Rust panic is a type of language exception that arises when a fatal error happens at runtime. An example of this is accessing an array out of bounds. Rust panic serves as a mechanism to enforce memory safety in cases where compile-time analysis cannot guarantee the absence of memory safety violations. It complements the compile-time analysis in ensuring memory safety by forestalling illegal memory access.

The canonical Rust programming paradigm offers numerous other scenarios where Rust panics may occur. For instance, when programmers assume that an `Option` value is not `None` or a `Result` value is not `Err`, they often use `unwrap()` to extract the contained value, which triggers a panic if the assumption is wrong. Additionally, hardware abstraction layer (HAL) crates frequently incorporate sanity check assertions such as `assert!()` and `assert_eq!()`, which fail if the hardware state mismatches the software configuration, resulting in Rust panics.

2.2 Stack Unwinding

The stack unwinder cleans up resources when a panic occurs, as shown in Figure 1. Semantically, active functions

PLOS '23, October 23, 2023, Koblenz, Germany

ACM ISBN 979-8-4007-0404-8/23/10.
https://doi.org/10.1145/3623759.3624549

Table 1. Rust panics are simpler than C++ exceptions.

	C++ exception	Rust panic
Allowed type	Any C++ class	Only `Panic` struct
Match rule	Follow subtype	No subtyping
Catch argument	By value or reference	Only by value
Nesting	Arbitrary level	No nesting
Re-throw	Supported	Not supported

return immediately upon a panic, starting from the most recent one and continuing until reaching a `catch_unwind()` statement. Then, the code resumes execution from the first statement after `catch_unwind()`. Mechanically, the code invokes the stack unwinder upon a panic. The unwinder refers to the compiler-generated static data structure known as the exception table to understand the content of each function call frame in the stack. Subsequently, it restores callee-saved registers and invokes object destructors for initialized objects. The unwinder finishes upon reaching a `catch_unwind()` statement, returning to the following statement.

The embedded world, however, has resisted handling language exceptions through stack unwinding, primarily due to concerns regarding storage/performance overhead and the added responsibility of ensuring exception safety. Our key insight is that Rust allays both concerns. First, Rust panics are semantically much simpler compared to exceptions found in other systems programming languages such as C++. Table 1 summarizes the distinctions between C++ exceptions and Rust panics. The simplicity of panic semantics reduces the logic necessary to support stack unwinding, making it an appealing option for embedded systems.

Moreover, safe Rust relieves programmers of the responsibility to ensure exception safety. Code is considered exception safe if the occurrence of an exception does not compromise its correctness. Unlike C++ in which achieving exception safety has long been known to be hard [6], in part due to the freedom to use raw pointers, safe Rust guarantees memory safety even in the event of a panic.

In summary, the simple semantics of Rust panic, together with the exception safety offered by the Rust type system, motivates us to handle Rust panics by stack unwinding in order to enhance embedded system availability.

3 Challenges to Stack Unwinding

3.1 Performance Overheads

Stack unwinding can introduce overhead to both normal and panicking code paths. We find the overhead in the normal code path is small (2.6%) when an unwinder is used to handle Rust panics. This is because Rust employs a metadata-based panic handling mechanism: the compiler statically generates the read-only exception table, which the unwinder references upon panics to analyze the call stack and execute appropriate actions. As a result, no additional logic is present in the non-panicking code path. The slight runtime overhead may result

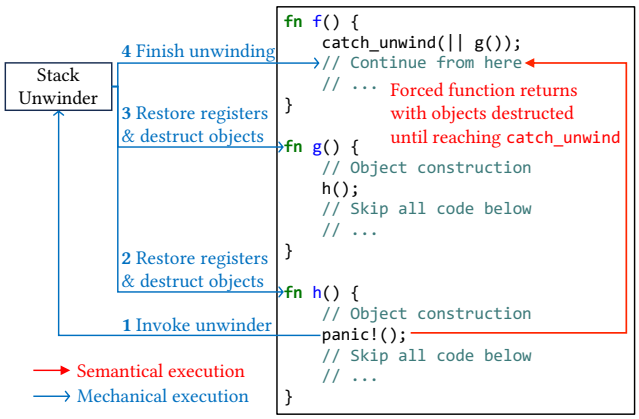


Figure 1. The stack unwinder forces a series of function returns upon a panic, with objects destructed along the returns, until reaching a `catch_unwind()` statement.

from precluded compiler optimization when functions are allowed to return through unwinding.

The unwinder does noticeably slow down the panicking code path since it interprets the exception table for each function during unwinding. We introduce several optimizations to mitigate the impact of panicking code and to expedite recovery. Moreover, panics are supposed to be infrequent, because Rust encourages explicit error handling and reserves panics solely for fatal errors.

3.2 Storage Overheads

Stack unwinding also increases code size by adding landing pads, the exception table, and the stack unwinder logic. Landing pads are compiler-generated code to be invoked when unwinding, responsible either for destructing the objects of an active function before forcing it to return or for catching the panic. The exception table, also compiler-generated, contains per-function, read-only data structures dictating the actions to be taken during unwinding. The table entry includes the operation to restore callee-saved registers and indicates which landing pad to invoke. The unwinder logic references the exception table to restore registers and invoke landing pads.

Theseus [4] has already demonstrated that the storage overhead of a metadata-based Rust unwinder is acceptable. We calculate the storage overhead resulting from the unwinding mechanism by building Theseus without it, removing unwinding related modules and setting the `-C panic=abort` compilation flag. As Table 2 shows, out of the total 3018 KB overhead, the unwinder accounts for 149 KB, representing a constant overhead. The remaining overhead arising from the landing pads, symbol table, and exception table are approximately proportional, on average 32%.

Embedded systems provide opportunities to further reduce the storage overhead. For instance, embedded ARM defines an alternative representation of the exception table

Table 2. Storage overhead for stack unwinding in Theseus. Code size is the sum of all `.text` sections. Data includes `.rodata`, `.data` sections and their corresponding `.rel.*` sections. Since Theseus performs dynamic linking for the whole system, symbol tables must be present in the final binaries, including `.symtab` and `.strtab` sections. Exception tables are only present when unwinding is enabled, including `.eh_frame` and `.gcc_except_table` sections.

	w/o Unwind	w/ Unwind	Unwinder
Code	5141 KB	6130 KB	91 KB
Data	1471 KB	1503 KB	28 KB
Symbol table	2309 KB	2849 KB	30 KB
Exception table	0 KB	1457 KB	0 KB
Total	8921 KB	11939 KB	149 KB

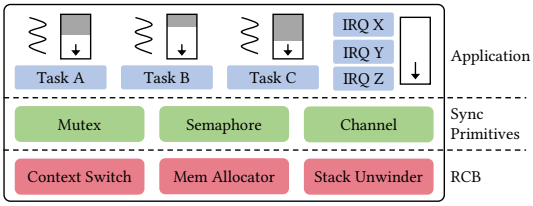


Figure 2. Hopter OS structure. It can recover from all panics except those in the Reliable Computing Base (RCB).

with the exception handling ABI (EHABI) [2]. Compared to the DWARF [9] format used by Theseus targeting x86_64, EHABI is more compact and entails fewer variations in the data structure, resulting in both smaller exception tables and simpler unwinder logic.

4 Stack Unwinding in Hopter

We report our experience of implementing stack unwinding in Hopter, an embedded OS designed for single-core embedded systems, currently implemented for ARMv7-M architecture.

4.1 Overview of Hopter

Hopter is written in Rust and requires application developers to implement tasks and interrupt handlers with safe Rust code. Hopter provides safe synchronization primitive interfaces to application developers, while the HAL library supplies safe hardware manipulation interfaces. Hopter tolerates Rust panics originating from both tasks and interrupt handlers.

Hopter supports multitasking by allocating one call stack for each task running in the thread mode, whereas it uses a single kernel stack to handle interrupts and system calls running in the handler mode of CPU, as shown in Figure 2. Hopter assigns priorities to tasks and interrupts. The scheduler or the nested vectored interrupt controller (NVIC) chooses the highest-priority task or interrupt handler to execute, respectively. Since system calls and interrupts share a single kernel stack, a higher-priority interrupt preempts a lower-priority one by placing its stack frame on top of the existing

```rust
let sender = |send: Producer<T>| {
    loop {
        // ... Prepare data ...
        send.push(data);
    }
};
let receiver = |recv: Consumer<T>| {
    loop {
        let data = recv.pop();
        // ... Work on data ...
    }
};

// Create a channel of type T with buffer size 8.
let (send, recv) = channel::create::<T>(8);

// Start tasks with stack size 256 bytes and priority 0.
create_restartable_task(sender, send, 256, 0);
create_restartable_task(receiver, recv, 256, 0);
```

Listing 1. Task synchronization using data channel.

```rust
pub fn create_restartable_task<F, A>(
    entry_closure: F, entry_argument: A,
    stack_size: usize, priority: u8,
) where
    F: FnOnce(A) + Clone + Send + Sync + 'static,
    A: Clone + Send + Sync + 'static,
{ ... }

fn restartable_task_entry_trampoline<F, A>(
    entry_closure: &F, entry_argument: &A
) where ...,
{
    // Run the entry closure with argument.
    // Catch possible panic.
    catch_unwind_with_arg(
        entry_closure.clone(),
        entry_argument.clone(),
    );
}
```

Listing 2. Restartable task requires the `Clone` trait implemented for the entry closure and argument. Unwinding terminates upon reaching `catch_unwind_with_arg()`.

ones. Hopter's call stack organization and priority design closely resemble those of other popular embedded OSes like FreeRTOS [10], ThreadX [18], and VxWorks [21].

Hopter provides various synchronization primitives, including mutexes, semaphores, and data channels. For instance, in Listing 1, two tasks synchronize over a data channel. The sender sends data with `push()` and blocks when the channel is full, similarly for the receiver with `pop()`. Additionally, a task can synchronize with an interrupt handler with either a semaphore or a data channel, where the interrupt handler must call non-blocking methods, like `force_push()` of the data channel, which discards the oldest data in the channel when it is full.

4.2 Stack Unwinding

Hopter supports recovery from Rust panics originating from both tasks and interrupt handlers. The recovery is conducted by the reliable computing base (RCB), which includes the context switch logic, the memory allocator, and the stack unwinder. The RCB itself, however, cannot recover from a panic and will loop if a panic does occur within.

The unwinder leverages metadata generated by the compiler toolchain. For example, the compiler generates an exception table entry for each function, which is later aggregated by the linker and placed in the `.ARM.extab` section within the final binary. To accelerate table entry lookup for panicking functions, the compiler also generates indices for each function. The linker then organizes these indices in ascending order of function addresses and stores them in the `.ARM.exidx` section, facilitating fast function entry lookup through binary search. Both the `.ARM.extab` and `.ARM.exidx` sections are placed in the flash memory and are read in-place, eliminating the need for them to be copied into SRAM. This approach conserves SRAM resources while ensuring a reasonable speed for table entry searches.

Hopter employs distinct panic recovery procedures for tasks and interrupt handlers due to their different execution patterns. Tasks are long-lived, often persisting throughout the entire system boot, whereas interrupt handlers are short-lived, activated solely during the handling of raised interrupts. Therefore, Hopter actively restarts a panicking task but silently returns a panicking handler. Tasks and interrupt handlers also have different stack organizations: each task enjoys a private call stack, while all interrupt handlers share a single kernel stack. As a result, a panicking task triggers the priority reduction of itself, but a panicking interrupt handler causes Hopter to reduce the priority of all active handlers.

4.3 Recover From Task Panic

Hopter provides a restartable task abstraction to tolerate task panics and applies an optimization to speed up recovery, and Rust plays a crucial role in enabling the optimization.

Upon a panic, a restartable task invokes the unwinder to reclaim the allocated resources by unwinding its stack and subsequently restarts. To restart, both the task entry closure and its argument must implement the `Clone` trait, as shown in Listing 2. This requirement enables safe duplication of the closure's enclosed environment and the argument, allowing for proper re-execution of the task. Unwinding of the panicking task terminates upon reaching `catch_unwind_with_arg()`, similar to `std::panic::catch_unwind()` [8].

When a restartable task panics, Hopter immediately restarts a new instance of it, running the restarted instance and unwinding the panicking one concurrently. Because unwinding is slow, waiting for the panicking task to finish unwinding before restarting it can violate the tight time constraints of embedded applications. Hopter concurrently restarts the panicking task by invoking the cloned entry closure with its entry argument within a new task context. The new task instance inherits the priority of the panicking task before it panics. The unwinder works within the context of the panicking task set to the lowest priority. This approach allows for a swift takeover by the new task and reclaims the resources of the panicking task by utilizing otherwise idle CPU time. Hopter allows a single concurrent restarting instance of each task in order not to exhaust system resources when panics happen frequently.

Rust plays a crucial role in enabling concurrent restart in two ways. First, the semantics of Rust panic offers the opportunity to concurrently restart a panicking task. This is because Rust reserves the panic only for fatal and unexpected conditions, in contrast to exceptions in other languages like C++ which may signal either expected errors or unexpected fatal ones. That is, the concurrent restart would be much more complicated, if possible at all, with C++. For instance, the C++ standard template library function `std::stoi`, used to convert a `std::string` to an `int`, may throw `std::invalid_argument` if the input string is not a valid integer representation or `std::out_of_range` if the number exceeds the range representable by `int`. A well-designed parser should treat these exceptions as expected occurrences. Only in the absence of a matching catch block do these exceptions escalate to fatal errors, leading to program termination by invoking `std::terminate()`. Therefore, a system capable of recovering from C++ fatal exceptions cannot apply similar optimizations to immediately restart a task upon an exception being thrown, because without unwinding the stack it remains uncertain whether the exception will be caught and resolved or lead to task termination.

Second, Rust helps Hopter to eliminate race conditions due to concurrent task restart. When Hopter restarts a task, race conditions may arise when the two instances access the same static data concurrently, even when the variable is not shared among different tasks. Fortunately, Rust's type system already enforces a constraint, disallowing safe code from accessing mutable static variables. The `Sync` trait is required to safely access static variables, where they must either be read-only or follow Rust's interior mutability, such as being an atomic type or guarded by a mutex. The overhead of having mutexes is minimal in non-panicking cases because no other task contends for the mutex protecting a static variable accessible only to a specific task, thus the lock operation always succeeds immediately. Similarly, variables reachable through the entry closure and argument are safely accessible to the concurrently restarted task enforced by the `Sync` trait.

Hopter further incorporates priority inheritance to address potential priority inversion during stack unwinding. Priority inversion occurs when the panicking task, undergoing stack unwinding at the lowest priority, holds a mutex

that other tasks, including the restarted one, attempt to acquire. To resolve the priority inversion, Hopter temporarily elevates the priority of the panicking task to match that of the highest-priority one among the blocked tasks, until the mutex is released. This approach is made feasible by running the stack unwinder within the context of the panicking task.

Hopter also provides its own `Mutex` type. In contrast to `std::sync::Mutex`, Hopter's `Mutex` does not automatically poison during unwinding. The standard one does so to propagate the panic across the threads accessing the same `Mutex`. This strategy conflicts with Hopter's goal to resolve the panic and minimize its impact. In contrast, Hopter conveys the panicking state via the `panicking()` function, similar to `std::thread::panicking()`, which returns true only during unwinding. As a result, a program can opt to change its behavior upon detecting a panic.

4.4 Recover From Handler Panic

Hopter further tolerates panics originating from interrupt handlers. These handlers react to events of peripherals. Common peripherals on microcontrollers include timers, DMA engines, and communication buses like USART, I²C, and SPI. Application developers supply most of these handlers, which often involve HAL function calls to acknowledge the interrupt and reconfigure hardware states, and may also include additional logic, such as pushing recently received data into a data channel. The complexity of these handlers necessitates Hopter addressing potential panics that may originate from them, but by applying different mechanisms and optimizations than those for tasks.

Hopter recovers from handler panics by wrapping every handler function with `catch_unwind_with_arg()`. The unwinder executes within the context of the panicking handler. The overhead of wrapping handler functions is minimal in the common non-panicking scenario, consisting of only 6 instructions on ARMv7-M. Unlike recovering from panics in tasks, Hopter does not automatically re-execute a panicking handler. Subsequent operation depends on whether the interrupt requires explicit acknowledgment: If the interrupt requires explicit acknowledgment, the pending interrupt will invoke the handler again after Hopter catches the panic; otherwise, for instance in the case of `SysTick`, the interrupt will invoke the corresponding handler at the next interval.

It is necessary for Hopter to mitigate the impact of one panicking handler on others, especially because all interrupt handlers share the same kernel stack. Specifically, when a high-priority interrupt handler panics, it will prevent the invocation of low-priority ones during unwinding. The time-consuming procedure of stack unwinding may compromise the prompt handling of low-priority hardware events. The general idea still applies that Hopter should reduce the priority of the panicking interrupt handler, but it necessitates additional logic to prevent breaking other handlers.

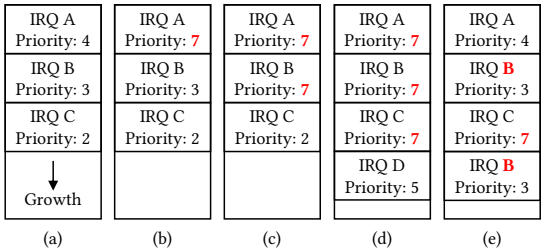


Figure 3. Reducing the priority of a high-priority handler may cause re-entrance of other active interrupt handlers. Correct management requires reducing the priority of all active handlers. A larger number represents lower priority.

Hopter reduces the priority of all active interrupt handlers when the currently running one panics. More specifically, it reduces the priorities of the handlers with lower priorities first, moving from the bottom of the stack to the top. For instance, in Figure 3(a), interrupt handler A is preempted by B, which is in turn preempted by C based on their priorities. When handler C panics, Hopter changes the priorities as depicted in Figure 3(b)-(d), and then the pending interrupt D can nest atop. After catching the panic, the priority restoration occurs in reverse order. Essentially, the relative order of the priorities of active interrupt handlers must not be inverted.

Hopter employs the above approach in order to prevent the risk of re-entrance for interrupt handlers. Consider the naive approach that simply reduces the priority of the panicking interrupt handler C. If interrupt B is not yet acknowledged to the hardware, doing so will cause its handler to be immediately invoked again, resulting in re-entrance, as shown in Figure 3(e). This situation may cause the re-entering B handler to observe inconsistent states or even lead to deadlock if it attempts to acquire an already-held lock.

Nonetheless, our optimization cannot prevent a panicking interrupt handler from affecting other active handlers. Fortunately, the probability of a panicking high-priority interrupt handler preventing the execution of preempted low-priority ones is minimal, as interrupt handlers are rarely nested at runtime. However, it should be noted that a panicking interrupt handler will block all tasks from executing since the CPU cannot return to thread mode until the handler's panic is caught. Thus, it remains advisable to prioritize the correctness of interrupt handlers.

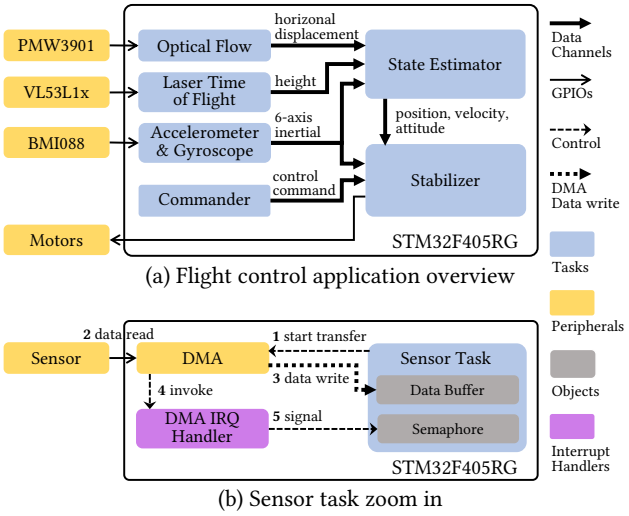
5 Panic Recovery in a Flying Drone

We demonstrate the feasibility and performance of Hopter recovering from Rust panics by deploying it to Crazyflie [3], a COTS miniature drone.

5.1 Hardware Platform

Crazyflie, shown in Figure 4, is powered by an STM32F405RG microcontroller based on ARMv7-M, and is outfitted with

Figure 4. Crazyflie 2.1, a COTS miniature drone originally with FreeRTOS.



(a) Flight control application overview

(b) Sensor task zoom in

Figure 5. Data flow of Hopter's flight control application written in safe Rust, based on that of the original Crazyflie.

various sensors: BMI088 connected through an I²C bus for 6-axis inertial measurement, VL53L1x connected through another I²C bus for height measurement using vertical laser reflection time-of-flight, and PMW3901 connected through SPI bus for horizontal movement detection based on optical flow. The CPU executes instructions directly from flash with prefetch acceleration.

5.2 Flight Control Application

Closely following Crazyflie's original flight control application (written in C for FreeRTOS), we develop a flight control application in safe Rust for Hopter. The application consists of six periodical tasks, as depicted in Figure 5(a), and also six interrupt handlers, showed as an example in Figure 5(b). We associate a dedicated task to each sensor responsible for reading raw data and converting it into a format understandable by the other tasks. The PMW3901, VL53L1x, and BMI088 sensors are read at frequencies of 50 Hz, 25 Hz, and 1000 Hz, respectively. The processed data is then sent through data channels to the state estimator and stabilizer tasks, both running at 1000 Hz. The state estimator task leverages the Kalman filter to estimate the drone's position, velocity, and attitude. The stabilizer task combines the state information with the control commands being sent at 10 Hz to generate the power signals for the four motors, aiming to keep the drone steady in the air while following the commands.

All the tasks of the flight control application run under soft real-time constraints. During flight, the application controls the motors to sustain an unstable equilibrium, wherein a minor disturbance, if left uncontrolled, can lead to a loss of balance. In practice, such disturbances may manifest as missed sensor readings or slight delays in state estimation updates. Although they may not trigger an immediate drone crash, as the error accumulates, the system will start from instability and finally end with catastrophic loss of control if correction is not applied timely by the control application.

5.3 Evaluation Results

We demonstrate Hopter's ability to recover from panics by deliberately inserting `panic!()` statements to simulate unexpected software or hardware faults while the drone is set to hover at half a meter above the ground.

We first place the `panic!()` statement within each of the six tasks, causing them to panic at one-second intervals. Specifically, we place the panic statements in deep function calls, emulating a worst-case scenario, where the unwinder must clean up many functions. The drone successfully takes off and maintains its hover, withstanding the periodical panics, without any noticeable disturbances. Only when panicking in the stabilizer task, the drone slightly rotates along the yaw axis for ten degrees each time it panics, because the stabilizer task directly manipulates the power distribution of the motors. Notably, the drone can withstand an additional 20-millisecond delay in the unwinding path of the stabilizer task, whereas, without concurrent restart as described in §4.3, the drone would tumble over immediately after taking off.

Moreover, we showcase Hopter's recovery from interrupt handler panics by introducing a `PendSV` interrupt and placing `panic!()` within its handler. The test code triggers `PendSV` every second, and we introduce a delay of two milliseconds in the unwinding path to emulate a tough scenario. We also configure `PendSV` to have the highest priority among all interrupts, which includes other four DMA interrupts, a high-precision timestamp timer interrupt, a USART interrupt, and the `SysTick` interrupt. The drone withstands the panic while hovering without noticeable disturbance. However, without reducing the priority of the panicking `PendSV`, as described in §4.4, the system quickly becomes unresponsive due to a drifted system clock arising from missing `SysTick` interrupts, resulting in a drone crash.

Finally, we find the CPU and storage overhead introduced by unwinding to be moderate for Hopter running the flight control application. The CPU usage increases slightly from 38.0% to 40.6% when there is no panic, primarily because enabling unwinding precludes certain compiler optimizations. The storage overhead is smaller than that of Theseus. Table 3 illustrates the size increase due to stack unwinding.

Table 3. Storage overhead for unwinding in Hopter. Code size includes the interrupt vector table and `.text` section. Data includes `.rodata` and `.data` section. Exception table is only present when unwinding is enabled, including `.ARM.exidx` and `.ARM.extab` section.

	w/o Unwind	w/ Unwind	Unwinder
Code	169.00 KB	199.00 KB	13.29 KB
Data	10.50 KB	12.05 KB	1.26 KB
Exception table	0 KB	15.16 KB	0 KB
Total	179.50 KB	226.21 KB	14.55 KB

Out of the total 46.71 KB overhead, 14.55 KB comes from the unwinder and the restart logic. This constant overhead is significantly smaller than that of Theseus, partially because EHABI simplifies the representation of the exception table on embedded ARM. This simplification not only facilitates the parsing logic in the unwinder but also reduces the relative overhead from the exception table, amounting to only 6.7% in the final binary compared to 12.2% in Theseus. Nevertheless, it is important to acknowledge that the overall simplicity of Hopter, when compared to Theseus, likely contributes to the smaller overhead observed. As expected, the storage overhead of unwinding in Rust is more than an order of magnitude smaller than in C++ as reported by Renwick et al. [20], because Rust panic has simpler semantics and only signals fatal error.

6 Related Work

Stack unwinding. Existing OSes running on more powerful machines have already integrated stack unwinders into the kernel. The Linux kernel incorporates a stack unwinder aimed at providing stack trace information when a kernel `oops` happens; it does not employ the unwinder for resource cleanup. Notably, Linux utilizes an alternative exception table format called ORC, which results in a 50% size increase in exchange for 20x faster execution of the stack unwinder than the DWARF format [17]. However, ORC is overly simplified in that the compiler with optimization can potentially generate code with a stack frame indescribable by ORC. Theseus, a recent Rust-based OS, adopts a stack unwinder for resource cleanup upon panics. It restarts a task after unwinding the latter's stack. However, Theseus' implementation is within the context of powerful x86_64 machines and does not take real-time constraints into account.

Wary of the overhead of stack unwinding, Renwick et al. [20] eschew a separate stack unwinder. Instead, they cleverly embed stack unwinding logic within the normal control flow by forcing all function returns to indicate whether an exception has occurred. While this improves the performance of exception handling, it adds overhead to all normal code paths, a questionable tradeoff given that exceptions are supposed to happen infrequently.

Fault recovery. Previous studies have explored various approaches for recovering from fatal errors. As mentioned earlier, *checkpoint/restore* is a popular approach with server systems, e.g., [14]. Because execution between checkpoints must be transactional, high memory overhead arises from maintaining unmodified variable copies. Another approach is *record/replay*, which requires a well-defined interface for recording, e.g., [22], and long latency due to replay, not suitable for mission-critical embedded systems.

Many systems track the resources allocated to application tasks in order to reclaim the resources when a task terminates with a fatal error. For example, the kernel code of Linux and Tock [16] tracks the resource allocated to a task (process). However, these kernels can only recognize the resources provided by themselves, like an opened file or network socket, but treat application-defined resources as raw bytes. Therefore, these kernels are unable to gracefully shut down an application session, for example. On the other hand, Microreboot [5] can track application-defined resources. However, the tracking is based on lease, which is not applicable to embedded systems with real-time constraints, because the restarting task must wait until the lease of the panicking task expires.

Acknowledgments

This work is supported in part by NSF Awards #2130257 and #2112562. The authors are grateful for the constructive feedback from the reviewers.

References

[1] Jorge Aparicio. 2018. *panic-halt*. https://crates.io/crates/panic-abort

[2] ARM Limited 2023. *Exception Handling ABI for the Arm Architecture*. ARM Limited. https://github.com/ARM-software/abi-aa/blob/844a79fd4c77252a11342709e3b27b2c9f590cf1/ehabi32/ehabi32.rst

[3] Bitcraze. [n. d.]. Crazyflie: A flying open development platform. https://www.bitcraze.io/.

[4] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. 2020. Theseus: an experiment in operating system structure and state management. In *Proc. USENIX OSDI*.

[5] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. 2004. Microreboot–a technique for cheap recovery. *arXiv preprint cs/0406005* (2004).

[6] Tom Cargill. 1996. Exception handling: A false sense of security. In *C++ gems*. 423–431.

[7] Embassy Project Contributors. 2023. *Embassy: The next-generation framework for embedded applications*. https://embassy.dev

[8] Rust Standard Library Contributors. 2023. *Rust std::panic::catch_unwind*. https://doc.rust-lang.org/std/panic/fn.catch_unwind.html

[9] DWARF Standards Committee 2010. *DWARF Debugging Information Format*. DWARF Standards Committee. http://dwarfstd.org/doc/DWARF4.pdf

[10] FreeRTOS. [n. d.]. Real Time Operating System. http://www.freertos.org/.

[11] Emil Fresk. 2018. *panic-halt*. https://crates.io/crates/panic-halt

[12] Adam Greig. 2018. *panic-itm*. https://crates.io/crates/panic-itm

[13] Adam Greig. 2018. *panic-semihosting*. https://crates.io/crates/panic-semihosting

[14] Asim Kadav, Matthew J Renzelmann, and Michael M Swift. 2013. Fine-grained fault tolerance using device checkpoints. *ACM SIGPLAN Notices* 48, 4 (2013), 473–484.

[15] knurling team. 2020. *panic-probe*. https://crates.io/crates/panic-probe/

[16] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proc. ACM SOSP*. 234–251.

[17] Linux. 2023. *Linux Kernel ORC Unwinder*. https://www.kernel.org/doc/html/next/x86/orc-unwinder.html

[18] Express Logic and Microsoft. 2023. *ThreadX*. https://azure.microsoft.com/en-us/products/rtos/

[19] Paul Pop, Viacheslav Izosimov, Petru Eles, and Zebo Peng. 2009. Design optimization of time-and cost-constrained fault-tolerant embedded systems with checkpointing and replication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2009).

[20] James Renwick, Tom Spink, and Björn Franke. 2019. Low-cost deterministic C++ exceptions for embedded systems. In *Proceedings of the 28th International Conference on Compiler Construction*. 76–86.

[21] Wind River. 2023. *VxWorks*. https://www.windriver.com/products/vxworks

[22] Michael M Swift, Muthukaruppan Annamalai, Brian N Bershad, and Henry M Levy. 2006. Recovering device drivers. *ACM Transactions on Computer Systems (TOCS)* 24, 4 (2006), 333–360.

[23] Valentine Valyaeff. 2017. *Drone OS*. https://www.drone-os.com/.

[24] Ying Zhang and Krishnendu Chakrabarty. [n. d.]. Fault recovery based on checkpointing for hard real-time embedded systems. In *Proc. IEEE Symp. Defect and Fault Tolerance in VLSI Systems*.