Offloaded Stack Unwinding for Embedded Operating Systems

Sebastian Alexander Schott
a.schott@yale.edu

Advisor: Lin Zhong
lin.zhong@yale.edu

Working with Zhiyao Ma
zhiyao.ma@yale.edu

A Senior Thesis as a partial fulfillment of requirements for the Bachelor of Science in Computer Science

Department of Computer Science
Yale University
May 2, 2024

Acknowledgments

I want to express my gratitude to both Professor Lin Zhong and Zhiyao Ma for their support and guidance on this project.

Contents

1 Introduction 4

2 Background 6
- 2.1 Rust Panics . . . 6
- 2.2 Stack Unwinding . . . 6
- 2.3 Exception handling on Embedded Systems . . . 7

3 Implementation 8
- 3.1 Design . . . 8
- 3.2 Communication Protocol . . . 8
  - 3.2.1 Hadusos . . . 9
  - 3.2.2 USART module . . . 9
- 3.3 MCU Unwinder Design . . . 9
  - 3.3.1 Metadata Storage and UnwindInfo Struct . . . 10
  - 3.3.2 get_for_pc . . . 10
  - 3.3.3 from_bytes . . . 11
- 3.4 Server . . . 11

4 Results 13
- 4.1 Offloaded Unwind Example . . . 13
  - 4.1.1 Binary Setup . . . 13
  - 4.1.2 Program Execution . . . 14
- 4.2 Testing . . . 14
  - 4.2.1 Correctness Testing . . . 14

5 Conclusions 16

6 Future Work 17
- 6.1 Offloaded Landing Pad Code . . . 17
- 6.2 Networked Offloading . . . 17
- 6.3 Multi threaded Server, Multiple Clients . . . 18

Offloaded Stack Unwinding for Embedded Operating Systems

Sebastian Schott

Abstract

Stack unwinding is an effective way to handle and recover from Panics in Rust, but comes with a storage overhead which can be prohibitive on resource constrained systems. This paper presents an implementation of an offloaded stack unwinder implemented in Hopter, an embedded Rust operating system. By storing unwinding data only on a connected server, we can reduce the storage overhead associated with exception handling while preserving functionality. Performance evaluations show that while the offloaded unwinder incurs additional latency during unwinding, the non-panicking execution pathway remains unaffected. On average, binary size is reduced by 7.55%, demonstrating that offloading metadata is an effective strategy to address the storage limitations of embedded systems. This work highlights the potential for offloading techniques to enable more robust error handling and fault tolerance in embedded operating systems without compromising resource efficiency.

1 Introduction

Embedded systems form the backbone for many modern technologies, used in critical applications from medical devices to industrial machinery. In these domains where a single failure can have drastic consequences, reliability is as important as performance. However many embedded systems must operate without being able to rely on hardware redundancy. By implementing fault tolerance purely through software, we can enable embedded systems to recover gracefully from faults without needing excess hardware.

While software based fault tolerance is not new for embedded systems, its application has historically faced significant challenges. Many fault recovery techniques, such as checkpoint/restore involving storing safe states during execution which are then loaded after a fault. However, the performance and memory costs associated with software only methods are often prohibitive for resource constrained embedded systems. Another solution for panic recovery is stack unwinding, which returns to safe execution by cleaning up any resources which have been allocated since the system was in a safe state.

Stack unwinding offers a promising solution for fault recovery in embedded systems. Ma et al. showed that by leveraging the memory safety guarantees of the Rust programming language, stack unwinding can achieve adequate performance without increasing complexity [1]. However, even this implementation comes with a significant storage overhead of 28%.

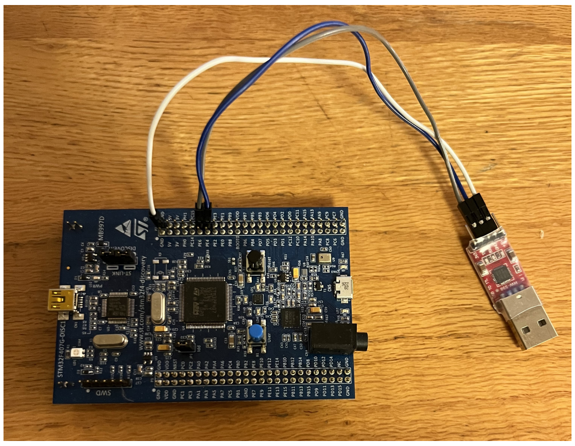


Figure 1.1: STM32F407 discovery board running Hopter

This paper introduces an offloaded stack unwinder for use in resource constrained embedded systems. By decoupling the unwinder from the local system, we reduce the local storage overhead while still maintaining full functionality. This approach sacrifices error handling performance due to the inherent delay in serial communication, however the binary size reduction makes this trade off worthwhile for systems which would otherwise lack panic recovery.

This project focuses on redesigning the unwind module of Hopter, a lightweight embedded operating system built for STM32Fxx microcontrollers (figure 1.1). Unlike larger systems, these MCUs lack virtual memory and other hardware protections, requiring alternative approaches to ensure memory safety and reliability. Hopter addresses these challenges through use of the Rust programming language to guarantee memory safety entirely through software, and implements stack unwinding to handle kernel panics. By offloading unwinder logic and metadata which are used only during panic recovery, Hopter achieves efficient error handling without incurring unnecessary overhead on the MCU.

2 Background

2.1 Rust Panics

Rust implements ”Panics” to signal a fatal runtime error, such as accessing an array out of bounds. While Rust usually prioritizes **compile time** memory safety guarantees, some issues cannot be detected at compile time, making panics a valuable tool for runtime error handling. Panics are particularly prevalent in hardware abstraction layer (HAL) crates, which interface directly with the microcontroller (MCU). These crates typically use assertions like `assert!()` and `assert_eq!()` to verify the MCU hardware state aligns with the software configuration. If an assertion fails, the program triggers a panic, halting current execution to prevent undefined behavior.

2.2 Stack Unwinding

Stack unwinding is one method of cleaning up resources when a panic occurs. When triggered, the unwinder begins at the current function and forces it to return, then continues up the call stack, cleaning up resources and forcing returns along the way. If a `catch_unwind()` statement is encountered, execution resumes after the catch. If no such block exists, the task terminates when the unwinder forces a return from the `main` function.

The unwinder relies heavily on compiler generated metadata, which is used for four primary purposes:

1. Force a return when the next instruction is not `ret`. When a panic occurs, the unwinder must exit the current function without completing its execution. This requires knowing the function’s return address to know where to jump and where callee-saved register values are located to be restored before the function returns.

2. Identify which resources need to be freed, and how to free them. When the unwinder forces a return, it bypasses essential steps. This includes invoking object destructors, which the unwinder must handle explicitly to ensure proper cleanup and prevent memory leaks.

3. Determine whether panicking code is running inside a `try` block. Try blocks are used to anticipate panicking code. If a panic occurs inside a try block, the unwinder will search for the corresponding landing pad where execution will resume. Otherwise it proceeds with cleaning up the function and forces a return.

4. Location of landing pads. Landing pads hold the actual unwinding logic for each function. Cleanup routines release resources, such as invoking destructors before

calling `_Unwind_Resume()` to return control to the unwinder. Catch blocks transfer control back to the program, resuming execution after the `catch`. The unwinder uses metadata to identify the appropriate landing pad based on where the error occurred, ensuring proper cleanup and exception handling.

The required metadata is generated at compile time, and thus enables **zero cost exception handling**, where all exception handling logic and data are placed outside the normal execution path. Although this approach can lead to slower exception handling following an error, it is best practice because it maintains optimal performance under typical circumstances [2].

The metadata is stored in the compiled ELF executable, but it's organization depends on the target architecture. In x86_64, information for storing a return (1) is stored in the `.eh_frame` section, while resource cleanup (2), `try` (3) and landing pad (4) information is found in the `gcc_except_table`. Although this project focuses on ARM, usage on x86_64 is similar.

On ARM, there are two relevant sections. `.ARM.exidx` is a compact binary searchable table containing one or more entries for each function based on where the panic occurred. This section is used as an index to located the detailed unwinding instructions and LSDA stored in the `.ARM.extab` section, although if the entry is small enough, it may be inlined into the `.ARM.exidx` section for efficiency.

2.3 Exception handling on Embedded Systems

Stack unwinding has been considered for use on embedded systems, however it's feasibility in other languages remained unclear. Ensuring memory safety and program correctness during unwinding is inherently difficult, and the storage and performance overhead are usually prohibitive on resource constrained systems. Renwick et al. [3] discouraged the use of a stack unwinder implemented in C++ for use on embedded systems due to the large storage and performance overhead. Ma et al. [1] showed that leveraging safe rust code could reduce the complexity stack unwinding introduced and achieve acceptable performance in normal code paths. However, both these implementations still had a significant storage overhead which offloading the unwinder would help to alleviate [4].

3 Implementation

3.1 Design

The implementation of the offloaded stack unwinder is composed of three main components, which collectively enable the removal of all stack unwinding metadata from the flashed binary. These components include the communication protocol, modifications to the local operating system running on the MCU, and the remote server. Our goal is to modify any sections of the local unwinder which access data stored within the `.ARM.extab` and `.ARM.exidx` sections (figure 4.1) to instead request this information from the server (figure 3.6).

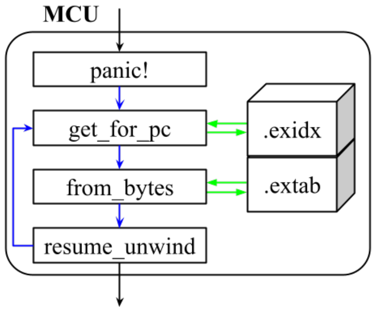


Figure 3.1: Local stack unwinder stores metadata on MCU

3.2 Communication Protocol

The USART module facilitates communication between the MCU hardware and the remote server using a half duplex serial connection, allowing bidirectional messages, though only one side transmits at a time. We utilize the Hadusos [4] protocol as an abstraction layer, to allow changes to either end independently. Hadusos only requires implementation of two functions: `read_byte_with_timeout` and `write_byte`. Once these functions are set on both the MCU and server according to corresponding hardware, Hadusos will handle higher level concerns like packetization and error checking. This protocol has multiple layers of redundancy, to ensure completely reliable communication. Transmitted data will include executable code and physical memory addresses to be used by the unwinder. Even minor changes to this data will result in undefined behavior and runtime errors. Therefore, ensuring data reliability is the number one focus of the communication protocol.

3.2.1 Hadusos

Hadusos supports sending data of arbitrary length by organizing transmissions into packets (figure 3.2) containing up to 255 bytes of user data, supplemented by a header and footer. The header and footer maintain synchronization between sender and receiver, using predefined byte sequences, referred to as the `preamble` and `postamble`, which sequences signify the start and end of a packet. Additionally, the footer contains a crc32 checksum which allows detection of up to three flipped bits which may be caused by hardware errors. Hadusos also has further built in redundancy through message acknowledgment. After receiving a packet, the receiver uses the `type` field to send an acknowledgment (ACK) or negative acknowledgment (NACK) message. This mechanism allows the sender to identify and retransmit corrupted packets, ensuring reliable communication.

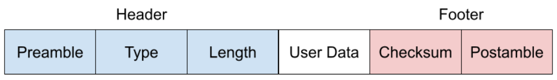


Figure 3.2: Hadusos packet layout

3.2.2 USART module

The USART module is used to set up the Hadusos session on the MCU. USART connections function as a stream of bytes, but the unwinder may not always be ready to process data as it arrives. To address this, we use a double ended queue (deque) as a buffer for incoming data and implement our Hadusos functions to interact with this data structure rather than the hardware directly. The deque allows for efficient reading and writing of data, which is essential to keep the hardware interrupt lightweight.

USART connections operate using two hardware pins: Rx (receive) and Tx (transmit). When data is received, it triggers the `usart1_handler` interrupt (fig 3.3), which stores the data into the global buffer. The buffer is wrapped in a mailbox, which increments for each byte added to the buffer, and decrements for each byte removed. This way, if a task is waiting to revive data, it can yield its processing time, and be notified when there is available data to read.

At startup, the user program configures the hardware to establish the USART-based communication interface. This requires supplying the configured read and write pins to create an instance of UsartSerial and UsartTimer which are then used to implement a Hadusos session. This setup ensures a reliable serial connection between the MCU and remote server which can also be configured to work on different hardware as needed.

3.3 MCU Unwinder Design

The local component is a modification of the Hopter Operating System which is being run on the MCU. These changes focus on the unwind module, which handles stack unwind-

```
#[handler(USART1)]
fn usart1_handler() {
    cortex_m::interrupt::free(|_| {

        // Read incoming byte
        let byte = G_UART_RX.as_mut().unwrap().read()?;

        // Push the byte into the global buffer
        let result = G_UART_RBYTE.push_back(byte);

        // Increment mailbox counter
        G_UART_MAILBOX.notify_allow_isr();
    });
}
```

Figure 3.3: Hardware interrupt function which stores data in to the global buffer

ing after a panic. The goal of these changes is to offload data accesses and move some processing tasks to the server, simplifying the MCU's implementation.

3.3.1 Metadata Storage and UnwindInfo Struct

On ARM systems all unwinder metadata is stored in the `.ARM.extab` and `.ARM.exidx` sections of the binary. In Hopter these sections are used to populate the `UnwindInfo` struct (figure 3.4), which describes how to unwind a given function. The struct includes the function address, personality routine, unwind instruction iterator and LSDA (if applicable). To decouple the unwinder metadata from the MCU, two methods - `get_for_pc` and `from_bytes` - are reworked to retrieve data from the server instead of accessing local memory.

3.3.2 get_for_pc

The `get_for_pc` method performs a binary search on the `.ARM.exidx` using a given program counter to locate the corresponding entry, represented in the OS using the `ExIdxEntry` struct. This struct either indicates that a function cannot be unwound, or contains a 32-bit unsigned integer. The integer either points to an entry in the `.ARM.extab` table or holds a compact representation of the ARM unwind instructions.

In the offloaded implementation, the method uses the global Hadusos session to request this information from the server. First the MCU sends the byte string `0xBBBB` which signals a `.exidx` request, then follows up with the program counter. The server performs a binary search through the `.exidx` table to find the matching entry, which it sends back as raw bytes. The server also sends back the entry's physical address within the binary, which is needed because the entry uses relative addressing. The retrieved data is then passed to the `from_bytes` method.

```
pub struct UnwindInfo<'a> {
    /// The address of the corresponding function.
    func_addr: u32,

    /// The personality routine of the corresponding function.
    personality: PersonalityType,

    /// The iterator that yields unwind instructions to
    /// restore register values.
    unw_instr_iter: OffUnwindInstrIter,

    /// LSDA describing cleanup routines and catch blocks.
    /// Compact model that embeds the unwind instructions
    /// into the exidx entry does not have LSDA, thus the
    /// field is optional.
    lsda: Option<LSDA<EndianSlice<'a, LittleEndian>>>,
}
```

Figure 3.4: UnwindInfo struct which is created using data from the `.ARM.exidx` and `.ARM.extab` sections

3.3.3 from_bytes

The bytes from the exidx table entry are then used to populate the `UnwindInfo` struct. If the compact representation has been used, there is no LSDA field, and no further metadata is needed. In this case the MCU can use the `get_unw_intr_iter()` and `get_-personality` methods, otherwise it must make a second request to the server. It sends another packet to the server, this time with the byte string `0xAAAA` to signal a `.extab` request and the address which was encoded in the extab entry. The server retrieves the necessary `extab` data, constructs the required fields of the `UnwindInfo` struct, and sends them as raw bytes over the Hadusos session. The MCU receives these values and uses them to populate the `UnwindInfo` struct then returns the completed struct to the unwinder for further processing.

3.4 Server

The server component emulates the unwinding functions previously handled locally by the MCU. During program execution the server waits for MCU requests and handles them using two functions - `handle_exidx` and `handle_extab`. These functions interact with the raw bytes stored on the server, and emulate OS functions previously used to handle this data.

The server component is also responsible for the conditional compilation of the program binary before it is flashed onto the MCU. This process involves linking the binary twice: once with the `.ARM.extab` and `.ARM.exidx` sections included, which is stored on the server, and once with these sections removed, which is flashed onto the MCU. The stripped binary minimizes resource usage on the MCU while allowing the server to handle unwinder metadata. Because Hopter is designed to run on devices without a memory

```
/// Get UnwindInfo for address 'pc' inside a function
pub fn get_for_pc_offloaded(pc: u32) -> UnwindInfo {
    // Aquire global hadusos session
    let session = HADUSOS_SESSION.as_mut()

    // Send 0xBBBB to indicate a exidx request
    let exidx_request = (0xBBBB as u32).to_le_bytes();
    session.send(&data, TIMEOUT_MS);

    // Send the current pc as bytes
    let pc_bytes = pc.to_le_bytes();
    session.send(&data, TIMEOUT_MS);

    // Listen for incoming data, and allocate memory
    let size = session.listen(TIMEOUT_MS).unwrap();
    let mut exidx_bytes = vec![0; size].into_boxed_slice();

    // Receive exidx_entry bytes
    session.receive(&mut exidx_entry, TIMEOUT_MS);

    // Recieve exidx entry address
    // Call from_bytes(exidx_bytes)
    // ...
}
```

Figure 3.5: Simplified code showing how a Hadusos session is used to remove local data accesses in the `get_for_pc` method

management unit, it does not utilize virtual memory. To ensure addresses are consistent between machines, the server also stores the physical address of each binary section and uses `mmap` to align the data in memory. By mapping the binary data to the same physical addresses used on the MCU, the server guarantees that all metadata references and relative addresses remain accurate during execution.

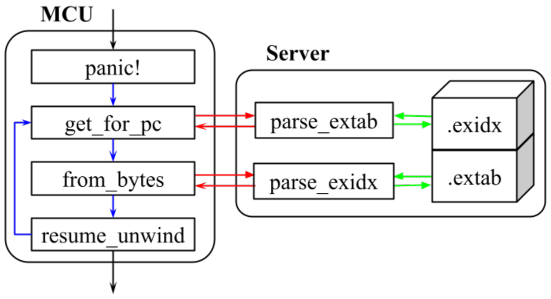


Figure 3.6: Offloaded stack unwinder stores metadata on remote server

4 Results

4.1 Offloaded Unwind Example

This section describes how the offloaded stack unwinder can be used in practice when connected to a PC using a USB cable. Testing was run on a STM32F407 discovery board simulated with QEMU. All of the code for this project can be found on public GitHub repositories [1][2].

4.1.1 Binary Setup

The first step is to compile our example project using rustc to get a full binary file which includes all the unwinder code and metadata. By using the rustc `--emit-asm` arg we can get an assembly file and perform linking ourselves. To generate the binary for use on the MCU, during linking we must remove all of the unwinder code from the assembly file. This is currently implemented to work for an arbitrary function, but requires further work to automatically find unwinder code. Then we can use a custom linker script that ignores the `.ARM.extab` and `.ARM.exidx` sections, giving us a binary without metadata or select code. For the copy stored on the server, we store the entire binary generated by rustc. By generating both versions of the binary for all of Hopter's 200 unit tests, we determined on average a 7.55% reduction in the size of the binary from removing these two sections.

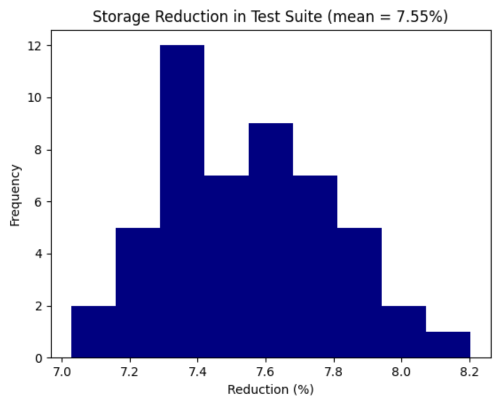


Figure 4.1: Binary size reduction caused by the removal of unwinder metadata

[1] https://github.com/sschott20/hopter

[2] https://github.com/sschott20/hopter_friend

4.1.2 Program Execution

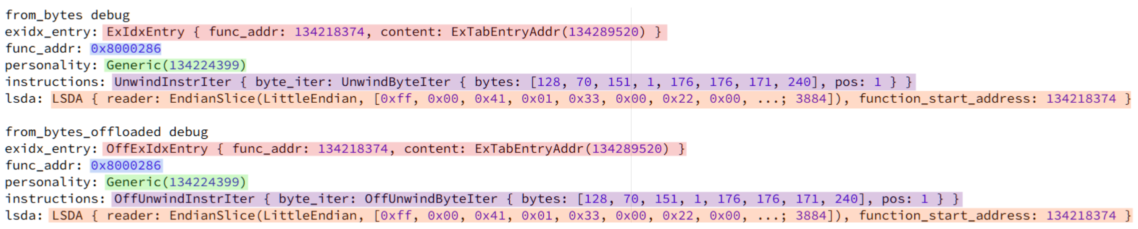
To make use of the unwinder, we first need to set up the USART connection and create a global Hadusos session. Establishing the session requires a timer and two functions: `read_byte` and `write_byte`. This requires configuration in the user program as it is hardware specific, and relies on the hardware abstraction layer crate. Once the user configures the hardware interrupt to read data into the global deque used in the hadusos session, setup is complete, and normal program execution can begin. This setup incurs a small runtime cost, typically below 1ms and only needs to be performed once. Once the Hadusos session has been initialized, functionality is identical to that of the local unwinder.

4.2 Testing

Hopter allows for two ways to catch panics: spawning restartable tasks and using the `catch_unwind` function. If a task is spawned using the `.spawn_restartable()` method, then it will be unwound and restarted upon panicking. Hopter also supports the `catch_unwind` function which performs similarly to the Rust standard library implementation[3], and is conceptually equivalent to a try-except block in other programming languages. The function takes a function as an argument and executes it. If the function executes successfully, `catch_unwind` returns `Ok(())`. If the function panics, it returns `Err(exception_type)` with the associated panic information.

4.2.1 Correctness Testing

To verify the correctness of the offloaded unwinder, we compare its results with those of the local unwinder. Since only two functions are modified—`get_for_pc` and `from_bytes`—and both are responsible for constructing the `UnwindAbility` struct, testing correctness involves ensuring that both implementations produce identical `UnwindAbility` structs for the same inputs.



```
from_bytes debug
exidx_entry: ExIdxEntry { func_addr: 134218374, content: ExTabEntryAddr(134289520) }
func_addr: 0x8000286
personality: Generic(134224399)
instructions: UnwindInstrIter { byte_iter: UnwindByteIter { bytes: [128, 70, 151, 1, 176, 176, 171, 240], pos: 1 } }
lsda: LSDA { reader: EndianSlice(LittleEndian, [0xff, 0x00, 0x41, 0x01, 0x33, 0x00, 0x22, 0x00, ...; 3884]), function_start_address: 134218374 }

from_bytes_offloaded debug
exidx_entry: OffExIdxEntry { func_addr: 134218374, content: ExTabEntryAddr(134289520) }
func_addr: 0x8000286
personality: Generic(134224399)
instructions: OffUnwindInstrIter { byte_iter: OffUnwindByteIter { bytes: [128, 70, 151, 1, 176, 176, 171, 240], pos: 1 } }
lsda: LSDA { reader: EndianSlice(LittleEndian, [0xff, 0x00, 0x41, 0x01, 0x33, 0x00, 0x22, 0x00, ...; 3884]), function_start_address: 134218374 }
```

Figure 4.2: Debug information showing both local and offloaded stack unwinder return the same unwind information

Hopter's `offload_debug` feature can be enabled to log debug information during execution. This allows us to capture and compare the values returned by the local and offloaded unwinder for a given test program. By running the same program with both

[3]https://doc.rust-lang.org/nomicon/unwinding.html

implementations, we can confirm that the offloaded unwinder accurately replicates the behavior of the local unwinder.

5 Conclusions

In this project we demonstrate the feasibility of an offloaded stack unwinder for resource constrained embedded systems. Our implementation targets the STM32fxx family of microchips and uses a USB serial connection to allow a connected PC to store unwinding metadata and handle some unwinding logic. While the offloaded unwinder introduces higher latency during stack unwinding compared to the locally implemented version, the performance during non-panicking execution remains unaffected. Our approach eliminates the need to store unwinding metadata locally on the MCU, and on average reduces the size of program binaries by 7.55%. This reduction helps to offset the storage overhead incurred by stack unwinding, making fault tolerance more accessible in embedded systems. Future work will focus on further reducing the unwinder overhead by using runtime linking to offload landing pad sections and more unwinder logic.

6 Future Work

6.1 Offloaded Landing Pad Code

The next step to fully offloading the stack unwinder is to remove the unwinder code entirely from the local MCU component. Separating the landing pad code (6.1) poses additional challenges, on top of those already associated with offloading the metadata.

Because the MCU does not use virtual memory, the landing pads cannot just be compiled separately from the rest of the binary. When the MCU wants to execute code, the address where that code is being executed matters due to relative addressing. Since the MCU cannot know where the landing pad will be executed at compile time (because if there is reserved memory for the landing pads it defeats the purpose of offloading them), the landing pads must be linked at runtime, only after knowing what address it will be placed at.

Although we implemented runtime linking, it proved surprisingly difficult to determine which sections of the code were landing pads. In 6.1 we can see this is clearly a landing pad because it finishes with a call to `_Unwind_Resume`, however this is not the case for all landing pads. The block is only given a generic compiler generated label which is not useful in determining where landing pads are located. To offload the landing pads will likely require a modification to the Rust compiler that generates specific labels, allowing the landing pads to be extracted for runtime linking.

```
.LBB48_58:
    ldr     r0, [r6]
    cmp     r0, #0
    itt     eq
    ldreq   r0, [r6, #4]
    cmpeq   r0, #1
    beq     .LBB48_60
    mov     r0, r8
    bl      _Unwind_Resume
```

Figure 6.1: Landing pad snippet, as shown by the call to _Unwind_Resume

6.2 Networked Offloading

Current builds of the offloaded stack unwinder are run using virtual machines or a wired USB serial connection. Practical applications of an offloaded operating system would benefit from using a wireless connection. By allowing wireless communication, the MCU would not need to be physically close to the server.

6.3 Multi threaded Server, Multiple Clients

The current implementation uses a single server and a single client which maintain constant communication. However, one of the main benefits of offloading operating system functions is we can allow one server to handle many clients. By improving the server design, we can allow the server to connect to multiple local components as needed. This would be especially useful for panic recovery, as it should only happen infrequently, and thus a single server would have ample downtime.

Bibliography

[1] Zhiyao Ma, Guojun Chen, and Lin Zhong. “Panic Recovery in Rust-based Embedded Systems”. In: *Proceedings of the 12th Workshop on Programming Languages and Operating Systems*. 2023, pp. 66–73. ISBN: 9798400704048.

[2] S Drew, K John Gough, and J Ledermann. “Implementing zero overhead exception handling”. In: *Faculty of Information Technology, Queensland University of Technology, Australia, Tech. Rep* (1995), pp. 95–12.

[3] James Renwick, Tom Spink, and Bjoern Franke. “Low-Cost Deterministic C++ Exceptions for Embedded Systems”. English. In: *Proceedings of the 28th International Conference on Compiler Construction*. 28th International Conference on Compiler Construction, CC 2019 ; Conference date: 16-02-2019 Through 17-02-2019. ACM, Feb. 2019, pp. 76–86. DOI: 10.1145/3302516.3307346. URL: https://cc-conference.github.io/19/.

[4] Zhiyao Ma, Samantha Detor, and Lin Zhong. “Offloading Operating System Functions to the Cloud”. In: *Proceedings of the 25th International Workshop on Mobile Computing Systems and Applications*. 2024, pp. 40–46.