Offloading Stack Unwinding in Embedded Operating Systems

Sebastian Schott, Zhiyao Ma, Dr. Lin Zhong - Yale University

Background

Embedded systems are crucial in applications from medical devices to industrial machinery, where reliability and performance must be achieved without hardware redundancy. However, software-based fault recovery is often impractical due to the high performance and storage overhead.

This project builds on Hopter, a Rust embedded operating system which implements stack unwinding to recover from fatal panics. Hopter runs on the STM32Fxx family of microcontrollers, which lack virtual memory and other hardware protections common on larger systems, making efficient software-based solutions essential for reliable operation.

Stack Unwinding

> The panic!() macro is used when a fatal error happens at runtime, such as from a failed assert!(), or an out of bounds array access

> When a panic occurs, the stack unwinder is responsible for cleaning up allocated resources and returning execution to a safe state

> The unwinder starts at the current function and moves up the call stack forcing returns until reaching a catch_unwind() statement or terminating the task

> Relies heavily on compiler generated metadata to understand how each function can be unwound

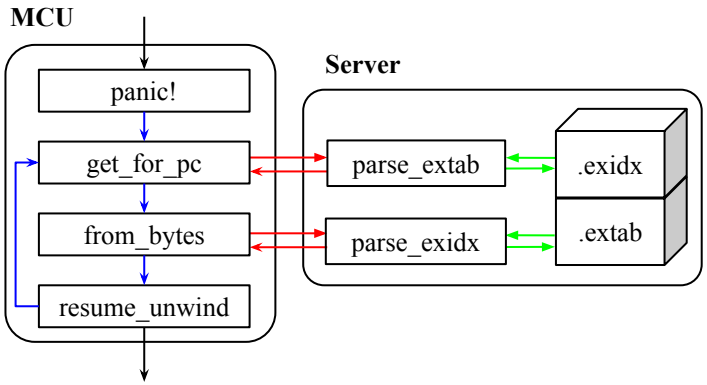
> Not typically used on embedded systems - metadata and code is rarely used, but can account for nearly 30% increase in binary size

Project Objectives

> **Enable efficient error handling on embedded systems** by offloading stack unwinder metadata and logic to an external server, reducing the local storage overhead without losing functionality.

This project focuses on redesigning the unwind module of Hopter. By moving code and metadata only used during unwinding to a connected PC, Hopter can achieve efficient panic recovery within the constraints of resource limited microcontrollers.

System Design



Unwinder Metadata

Unwinding metadata is stored in two key sections of the binary: *.ARM.extab* and *.ARM.exidx*. This data is generated by the compiler, and allows efficient runtime error handling

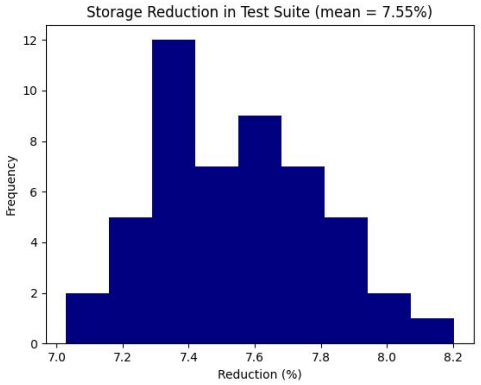
.ARM.exidx: This section is a compact, binary searchable table containing one or more entries for each function based on where the panic occurs. It acts as an index to locate the detailed unwinding instructions stored in the *.ARM.extab* section.

.ARM.extab: This section stores the corresponding unwinding instructions to restore register values, and the Language-Specific Data Area (LSDA) which describes cleanup routines and exception catch blocks.

Future Work

> **Offload Landing Pads:** Further decrease size of local binary by removing landing pads from local binary. When the unwinder reaches a landing pad it will request code from the server and provide an address where it can be placed. The server will link the snippet at runtime to be executed at that address, and transmit it back to the MCU.

Although we implemented runtime linking and code extraction, identifying landing pad sections proved harder than expected and will require a small modification to the Rust compiler to be fully implemented.



Performance on Unit Tests

Over Hopter's suite of ~200 unit tests, we observed an average of 7.55% reduction in binary size when removing unwinding metadata.

In non-panicking program paths, performance remained unaffected by the the offloaded stack unwinder.

When invoked, the offloaded stack unwinder introduced a significant delay. Tasks the local unwinder could recover from in milliseconds took up to 10 seconds with the offloaded unwinder, primarily due to the communication overhead.

Conclusions

This project demonstrates the feasibility of an offloaded stack unwinder for resource constrained embedded systems. Our implementation leverages a USB serial connection to an external PC which manages unwinding metadata and some logic. Through this, we achieved a 7.55% reduction in compiled binary size, significantly reducing the local storage overhead while maintaining full functionality of the unwinder.

Although the offloaded unwinder incurs a notable performance penalty when invoked, this trade off is acceptable, as it is only triggered during critical runtime failures, which should be rare in well designed software.