Relocatable RISC-V Rust Applications for Embedded Systems

Hugo McNally[1], Luís Marques[1] and Jorge Prendes[1]

[1]lowRISC CIC, Cambridge, United Kingdom

Abstract

The embedded position independent code (ePIC) ABI proposal offers a solution to the challenge of generating relocatable applications for RISC-V embedded systems without an MMU. Having such an ABI is important for enabling the dynamic loading of relocatable applications in the embedded RISC-V ecosystem. This unlocks the full potential of secure platforms such as Tock, an operating system that builds on the Rust compiler guarantees to help minimize the security vulnerabilities in embedded systems. This extended abstract outlines the unique features of ePIC, particularly in comparison to the approaches used in non-embedded devices, and highlights its significance for a secure Rust platform in embedded systems.

Introduction

The RISC-V ecosystem currently lacks an Application Binary Interface (ABI) that adequately supports relocatable applications for embedded systems. This is because most embedded systems run applications using both RAM and ROM memories (e.g. SRAM and flash) and do not have a memory management unit (MMU) due to their significant area and power requirements. Adding support for application relocation in such systems is challenging and can be expensive. A major cause of this difficulty comes from not loading and relocating the entire application by a fixed offset, but instead splitting the application between RAM and ROM at variable distances. This is normally done because embedded systems tend to have a small amount of RAM available, so to conserve it code will normally be executed in place, directly from ROM or flash. This invalidates several existing approaches, such as the family of System V ABIs, that assume that all *global offset tables* (GOT) are at a known offset from the code and, with no MMU, one cannot maintain the illusion of a fixed offset. Although the family of FDPIC ABIs should be able to address this problem, there is no specification of FDPIC for RISC-V and FDPIC is not necessarily the best solution because it introduces significant overheads when considered in the context of embedded systems.

This limits the available software stacks on embedded systems, because one cannot make full use of operating systems that dynamically load applications at arbitrary addresses for improved memory efficiency. The embedded position independent code (ePIC) ABI proposal [1] opens up embedded systems to these software stacks, by providing relocatable applications with a minimal overhead and tailored to the embedded use case.

ePIC

ePIC solves the problems of making applications relocatable by computing the addresses of application symbols relative to a location in the code segment if the symbol resides in the code segment or a location in the data segment if the symbol resides in the data segment. Calculating addresses relative to the segment a symbol lives in enables the two segments to be relocated independently of one another, allowing applications to be stored in arbitrary flash locations and their data loaded at arbitrary RAM locations. Note that absolute addresses can still be used for memory locations external to the application, such as when accessing memory-mapped devices.

For addresses in the code segment, standard RISC-V PC-relative addressing is used. However, as the offset of the data section from the PC is not known at link time, PC-relative addressing cannot be used for the data section. Instead, ePIC addresses locations in the data segment relative to the *global pointer* (GP), which is a pointer to a particular location in the data segment. The linker designates this location in the binary and the program loader will set the `gp` register to the absolute address of the location once the data segment has been relocated. The ABI register name `gp` is usually specified as an alias to the `x3` register.

The segment in which a symbol resides is not always known at compile time. To ensure the relative address can be calculated for all symbols in the application, ePIC relies on an unconventional link-time mechanism. This mechanism rewrites instructions that address symbols whose residence is known only at static link time.

Comparison to other Approaches

There are many existing approaches to solving the application relocation problem. Two common mechanisms used for relocatable applications are *dynamic relocations* and the *global offset table* (GOT).

When using *dynamic relocations*, the linker provides an additional relocation section in the binary detailing the instructions and data which need to be edited once the actual address locations of symbols are known at program load time. The dynamic linker/loader can then process the relocation section and rewrite the instructions or data to use the correct addresses. The program is then able to run in its new location. One drawback is the runtime cost associated with rewriting instructions when loading the program. Another drawback to this approach is that the code segment has to be writable. This is not ideal when the code segment is in flash, as it cannot be written to like regular memory (without erasing entire pages and causing excessive wear).

The System V ABI solves this issue by using a GOT (one per shared object). The GOT has an entry for each global symbol. The content of the entry (once the program is loaded) is the absolute address of the symbol. This means, with the location of the GOT and the offset of a symbol's entry in the GOT, the address of the symbol can be found at runtime. The GOT's entries are populated when the physical addresses of symbols are known at load time. The GOT lives in the data segment of the program, so that the code segment can be read-only. This is great for when one wants to run their program from flash.

However, the System V PIC ABI relies on a known offset between the code and the GOT. Therefore, it does not work for independently relocatable code and data segments, where this offset is only known at load time. File Descriptor PIC (FDPIC) solves this issue by using the global pointer to point to the current GOT. This means a known offset between segments is not needed at link time, and so FDPIC could be used for independently relocatable segments.

One reason ePIC is being proposed is that FDPIC uses a GOT, which is not needed when shared library support is not required. The GOT can take a non-trivial amount of RAM space and introduces a run time and code size cost due to the extra indirection and table lookup. Given embedded environments are often very resource restrained, these additional costs are best avoided.

Motivation

The Rust language enforces more restrictions on what a programmer can do compared to traditional low-level languages, but in return these restrictions provide certain compiler-ensured guarantees of correctness. For example, Rust's borrow checker restricts the programmer's ability to reference memory locations, but in return the compiler can guarantee that a number of types of memory management errors are absent. Not only does this make the program more reliable, it also prevents a variety of security issues.

However, not everything can be done with these restrictions in place, for example one can not access memory-mapped devices. For situations like this, Rust has an *unsafe* dialect, which allows one to do anything one could in a language like C, such as pointer arithmetic, and opens the door to all the undefined behaviour that results. In practice, unsafe blocks clearly denote areas of the code which require particularly thorough review.

Tock [2] is an OS built to capitalise on Rust's correctness guarantees. It only uses the unsafe dialect only where necessary in the kernel core and the core is kept minimal with most functionality handled in drivers (called capsules) or processes. This provides correctness guarantees for most of the codebase.

Tock has many features expected from a modern security conscious OS. For example, processes are preemptively scheduled and isolated from one another using the Memory Protection Unit (MPU). The feature of particular relevance to this paper is that processes are dynamically created/destroyed at runtime, which means processes can be compiled and signed separately from the kernel. This allows applications to be updated or replaced without having to recompile the kernel. The code of the kernel is security critical because it runs in privileged machine mode and contains some unsafe Rust. Only having to carry out a thorough review of it when it changes, greatly reduces the effort required to ensure a high level of security and reliability. This is also a great assistance to projects spanning multiple teams/organisations, since teams can review and sign their applications individually.

However, it is not currently possible to write in Rust Tock programs that can be relocated in both RAM and ROM because the LLVM RISC-V backend only supports the RISC-V System V ABI, which assumes in its design a single relocation offset (or an MMU that can hide additional offsets). ePIC would solve this, making them more reliable and secure thanks to Rust's correctness guarantees.

References

[1] lowRISC. *ePIC Specification*. `https://github.com/luismarques/epic-spec/blob/main/epic.adoc`. 2022.

[2] The Tock Project Developers. *Tock OS*. `https://github.com/tock/tock`. 2023.