Tighten Rust's Belt: Shrinking Embedded Rust Binaries

Hudson Ayers
Stanford University, Google
USA
hayers@stanford.edu

Evan Laufer
Stanford University
USA
emlaufer@cs.stanford.edu

Paul Mure
Stanford University
USA
paulmure@stanford.edu

Jaehyeon Park
Stanford University
USA
jaehpark@stanford.edu

Eduardo Rodelo
Stanford University
USA
ehiguera@stanford.edu

Thea Rossman
Stanford University
USA
tcr6@stanford.edu

Andrey Pronin
Google
USA
apronin@google.com

Philip Levis
Stanford University
USA
pal@cs.stanford.edu

Johnathan Van Why
Google
USA
jrvanwhy@google.com

Abstract

Rust is a promising programming language for embedded software, providing low-level primitives and performance similar to C/C++ alongside type safety, memory safety, and modern high-level language features. We find naive use of Rust leads to binaries much larger than their C equivalents. For flash-constrained embedded microcontrollers, this is prohibitive. We find four major causes of this growth: monomorphization, inefficient derivations, implicit data structures, and missing compiler optimizations. We present a set of embedded Rust programming principles which reduce Rust binary sizes. We apply these principles to an industrial Rust firmware application, reducing size by 76kB (19%), and an open source Rust OS kernel binary, reducing size by 23kB (26%). We explore compiler optimizations that could further shrink embedded Rust.

CCS Concepts: • **Computer systems organization → Embedded software**; • **Software and its engineering → General programming languages**.

Keywords: embedded systems, Rust, binary size

ACM Reference Format:
Hudson Ayers, Evan Laufer, Paul Mure, Jaehyeon Park, Eduardo Rodelo, Thea Rossman, Andrey Pronin, Philip Levis, and Johnathan Van Why. 2022. Tighten Rust's Belt: Shrinking Embedded Rust Binaries. In *Proceedings of the 23rd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '22), June 14, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3519941.3535075

1 Introduction

Rust [18] has emerged as a compelling alternative to C/C++ in many systems. Rust has high-level language features, and provides memory and type safety without performance overhead. For traditional software, Rust has been shown to match or exceed equivalent software written in C/C++ in performance [27] and memory use [3]. Rust software can be easier to maintain [13] and has fewer bugs [10].

This paper is about overcoming a challenge that arises in applying Rust's benefits to embedded systems: binary size. Our motivation for addressing this problem is based on personal experience. While working on a Rust-based embedded application for a next-generation security chip we discovered that the Rust app was 79% larger than the C-based system it was replacing, even after efforts towards size optimization. These efforts included ideal compiler configurations and optimization settings from well-regarded resources on the subject, including blog and forum posts detailing Rust size growth [24, 33] and the guide on minimizing Rust size [17]. Digging into individual instances where code produced larger-than-expected assembly, we found many idiomatic Rust abstractions were only "zero-cost" with regards to runtime performance, not binary size.

§2 provides background on Rust and related work. §3 focuses on a case study of porting a high quality industry security chip application from C to Rust, including signs that the use of Rust contributes to code size increase. §4 identifies specific areas where code size increase arises in the Rust port, including many non-obvious mistakes we expect others are likely to encounter as well. The sources of this size increase are grouped into general categories. In §5 we contribute five idioms that reduce this code size increase:

1. Minimize length and instantiations of generic code,
2. Use trait objects sparingly,
3. Don't panic,
4. Carefully use compiler generated support code,

5. Don't use `static mut`.

Applying these programming idioms reclaims 76 kB of space from the 400kB Rust security chip firmware binary (19%). We also apply these idioms to an open source Tock kernel binary, reducing its 87kB size by 26%, and show much of the savings comes from reductions in embedded data. In §6, we discuss changes to the Rust compiler and language which could help rectify the remaining size difference between Rust and C, including one optimization we implemented and upstreamed into the Rust compiler. In §7 we present several tools for analysis of Rust binaries. Finally, in §8 we contextualize these growth categories and idioms by looking at another embedded Rust use case – Hubris OS [5] – and discuss the implications of our findings.

2 Background

This paper focuses on small (<1 MB flash) devices. We consider only the `#[no_std]` dialect of Rust, limited to libcore, a minimal standard library, with no use of `alloc` for heap allocation.

2.1 Rust for Embedded Systems

Low cost software techniques to improve on the safety and security of traditional, written-in-C embedded software are highly valuable. Resource constraints of embedded devices preclude the use of hardware and software security mechanisms which are costly in terms of flash, RAM, or power [15]. For example, Java is memory safe, but without modification its flash and memory footprint are unacceptable in many embedded systems. Similarly, Address Space Layout Randomization (ASLR) is unusable in systems without virtual memory.

Rust is a promising approach to software security and safety because of its strong memory safety, type safety, and thread safety guarantees. To achieve these, Rust relies on an affine type system and a compiler which automatically inserts checks into potentially dangerous code. For example, all array accesses in Rust are checked and panic when out-of-bounds. Rust is especially well suited to embedded systems because it does not rely on a garbage collector, instead using a borrow-checker and lifetimes to track allocations statically.

The `unsafe` keyword in Rust enables access to low-level mechanisms (such as raw pointer reads/writes, or unchecked array accesses) which the compiler cannot verify are safe. With extensive use of `unsafe`, it is possible to write Rust much like one would write C, but doing so removes many of Rust's benefits. We focus on idiomatic Rust, in which `unsafe` is used only in rare situations where safe alternatives do not exist (e.g. memory-mapped register accesses).

Rust provides a number of higher level language features. Several of these features have significant impacts on code size. We describe two of them here for readers unfamiliar with Rust: polymorphism and closures.

Rust supports polymorphism with *traits*. A trait defines an interface to a polymorphic type, similar to Java `interfaces`. A trait can be implemented by multiple concrete types. Rust uses traits to enable polymorphism via two different mechanisms: generics (analogous to C++ templates), and trait objects (similar to C++ abstract classes). Generics generate a method for each concrete type used to call a generic function. This is called *monomorphization*, and has performance equivalent to normal function calls. Trait objects, in contrast, enable dynamic dispatch using runtime lookups in a virtual table. Trait objects are used when the concrete type that will be passed to a function can vary, or is not known at compile time.

Idiomatic Rust uses *closures* heavily. Closures are anonymous functions, similar to C++ lambdas, but they capture their environment. Closures are used widely in Rust APIs because they interact well with how the Rust compiler statically manages memory. Closures also fit well with idiomatic Rust's functional programming style. Many functions accept closures as generic parameters, allowing callers to pass arbitrary computation to be performed within the body of a function.

Rust relies heavily on compiler optimizations to make high-level programming constructs perform on-par with low-level implementations. Most Rust optimizations rely on its' single officially supported backend: LLVM [20].

2.2 Related Work

A large body of existing work aims to provide developers with tools for reducing the code size of embedded software. This includes techniques to remove unused code (dead code elimination) [19, 38], to replace code blocks with smaller, semantically equivalent constructions [8], to remove the overhead of function calls via inlining [21], and to evaluate code segments at compile time. It also includes techniques to merge duplicate source-level code within or across functions, such as identical code folding [35], function merging [32], function outlining [39], and code factoring. Recent work has quantified the value of these optimizations to higher-level language binaries in the context of Swift [7]. Other tools include entire languages dedicated to reducing code size for particular applications, such as NesC [11].

Embedded C++ has inspired research into understanding and reducing size overhead from the use of high-level language abstractions. C++ exceptions are size-expensive, since `throw` requires dynamic allocation and `catch` requires run time type information [34]. This has inspired research into low-size-cost C++ exceptions [31]. Other research has categorized high-level C++ features into three bins – cheap, maybe-cheap, and expensive – based on the suitability for low-resouce embedded settings, similar to how this paper identifies Rust features to use/avoid [29]. EC++ specified a subset of C++ suitable for low-resource embedded development, removing multiple inheritance, virtual base classes, runtime type identification (RTTI), exceptions, and templates [28].

Table 1. Size of Cr50 (C) and Ti50 (Rust) peripheral drivers with very similar structure and functionality. Rust implementations are significantly larger than C ones.

	Cr50 (kB)	Ti50 (kB)	% increase
i2c_peripheral	1.3	2.3	83%
i2c_host	1.1	3.2	184%
spi_host	0.6	1.0	66%
gpio	4.2	6.0	43%

RustyGecko, a set of libraries for bare-metal Rust, found bare-metal Rust programs to be 1.2x - 2.2x larger than equivalent bare-metal C programs [14]. Similarly, a thesis on embedded Rust programming found a Rust application for particle filtering to be approximately twice as large as the equivalently optimized C code it was translated from [6].

3 Motivation: C to Rust

This section compares two embedded applications, one written in C and one in Rust. Directly comparable portions of the Rust binary are larger than their C counterparts. Analyzing the binaries hints at how this size increase manifests.

3.1 Cr50 and Ti50

Cr50 is an open-source C firmware implementation for the ChromeOS H1 security chip, a CortexM microcontroller. Cr50 implements secure boot for ChromeOS devices as well as system services like keyboard control and battery charging [4].

Ti50 is a Rust firmware implementation for the next-generation RISC-V based ChromeOS security chip. Ti50 is based on an extensible OS kernel, TockOS [22], and a Rust runtime for Tock applications, libtock-rs [9]. Ti50 is under development and currently closed source. Ti50 (OS + applications) needs to fit in a single 512kB bank of flash, with space reserved for early boot states, data, and future growth.

During Ti50 development, developers observed that Rust code often generates larger binaries than what they expect from similar C code, and that the mapping between Rust code and the resulting binary size was more difficult to guess. Developers also noticed small, minor changes to Rust code sometimes substantially changed the size of the binary, as a result of Rust depending on fragile code size optimizations. These issues inspired this work. While Cr50 and Ti50 serve very similar purposes, their substantial hardware and software architectural differences make direct comparisons between them difficult. Nonetheless, by looking at the compiled size of extremely similar components and their binaries themselves, we gather some evidence about whether Rust binaries are larger than similar C ones.

3.2 Measuring Code Size

To quantify the size differences between C and Rust, we examine Cr50 and Ti50. First, we measure the code size of hardware peripheral drivers that are functionally almost identical: they have have very similar APIs in C and Rust and their Rust implementations are direct ports from C. This comparison allows us to see whether functionally equivalent Rust compiles to larger binaries than C. Next, we isolate one of these peripheral drivers and examine how the size of that particular driver is broken up across sections in both systems. This comparison allows us to see where size increases manifest. Finally, we look at the whole system binaries and measure how size is distributed across different sections.

We do not directly compare the total binary sizes of Cr50 and Ti50 because they are architecturally different, e.g., Ti50 has userspace processes and a system call layer, while Cr50 does not. Rust's aggressive whole program optimization makes disentangling size from additional functionality and size from the language/compiler impossible, as new functionality is merged with pre-existing code in large blocks of assembly. Removing these optimizations drastically increases the binary size and thus does not reflect real Rust binaries.

Table 2. Size breakdown of i2c_peripheral code by section in Cr50 (C) and Ti50 (Rust). While Rust code has somewhat more instructions (`.text`), there is more growth in static initializers (`.data`) and read-only data (`.rodata`).

Section	Cr50 (Bytes)	Ti50 (Bytes)
`.text`	1,148	1,661
`.rodata`	128	483
`.data`	0	180
total flash	1,276	2,336

Table 3. Section breakdowns of Cr50 (C) and Ti50 (Rust) binaries. Ti50's `.data` and `.rodata` sections are larger despite each system defining similar amounts of data in its source code.

Section	Cr50 (kB)	Ti50 (kB)
`.text`	182.9	323.0
`.rodata`	39.1	69.2
`.data`	0.2	7.3
total flash	222.2	399.6

3.3 Size Differences

Table 1 shows the size of four peripheral drivers in the two systems. We chose these four because they are functionally very similar. We gathered values using ablation: we replaced each driver with an empty "dummy" implementation, and calculated the size reduction. Ti50 drivers are 43-184% larger.

Table 2 shows how much the the i2c_peripheral driver contributes size to each section of the binaries. In both systems, the linker is configured to place static initializers in `.data`, read-only embedded data in `.rodata`, and instructions in `.text`. While the Rust driver has 40% more instructions, it

Table 4. Preview of bloat in Ti50 by category

Category	Bloat (Bytes)
Hidden Data	27,794
Compiler Generated Support Code	24,368
Monomorphization	8,932
Poor Default Size Optimizations	13,396

introduces static initializers that do not exist in the C code, and contains almost 4 times as much read-only data.

Table 3 shows how the full Cr50 and Ti50 binaries break down into sections. We expected `.text` to be larger because of additional code from Ti50's more isolated architecture. We did not expect `.data` to substantially change, as Ti50 actually has fewer global variables than Cr50. We also did not expect `.rodata` to change, as most embedded data in Cr50 is strings and large constants defined in the source code, and Ti50 defines roughly the same strings and constants in its source. We discovered that Ti50 actually has a much larger `.data` section, indicating our Rust code introduced more non-zero static initializers. It also has a much larger `.rodata` section (in absolute terms), despite similar amounts of data defined in its source. The growth in this section suggests that Rust is generating lots of embedded data that C does not.

3.4 Making Rust Closer to C

The studies above indicate that Rust's size increases are reducible. In the drivers we measured, Rust implementations were larger by a highly variable margin (43-183%); indicating it is possible to write code that is somewhat (43%) larger or code that is much (183%) larger. §4 identifies the sources of this difference, and §5 instructs how to write code that avoids them.

4 Rust Binary Growth

We identify four major causes of binary growth:

- Deeply ingrained monomorphization,
- Suboptimal compiler generated support code,
- Hidden data structures and data,
- Fewer compiler optimizations.

We discuss the first three in this section, and discuss compiler optimizations in §6. We contextualize each by the amount we saved in the Ti50 binary by addressing it. Table 4 contains a preview of these totals. These size-reduction changes were submitted over time alongside additional development, so savings numbers are calculated by summing the size difference of individual changes. We defer an explanation of how we addressed each growth cause to §5.

4.1 Deeply Ingrained Monomorphization

Idiomatic Rust uses generic types extensively: most collections have methods which are generic over the types they hold, and many methods for interacting with objects are generic

```rust
pub struct Vec<T, A: Allocator> { ... }
impl<T, A: Allocator> Vec<T, A> {
   pub fn retain<F: FnMut(&T) -> bool>(&mut self, mut f: F)
   { /* 56 lines of code */ }
}
------------------------------------
let v1: Vec<u32> = Vec::new();
v1.retain(|e| { e >= 12 });
v1.retain(|e| { e < 7 });
```

Listing 1. Rust `std` library implementation of `Vec` and example use. `Vec` is generic over `T`, the type it holds, and `A`, an allocator. `retain` is also generic over `F`, any type implementing the `FnMut` trait (e.g. a closure). Calling `retain` with two different closures leads to the body of `retain` being duplicated in the binary.

over closure traits which can be used to interact with them. Listing 1 shows an example of this.

Rust's implementation of generics relies on monomorphization: each unique type passed as a generic parameter to a generic function will lead to an additional instance of that function in the binary. This leads to *monomorphization bloat*: many copies of the same assembly instructions in only slightly different functions. These copies are expected for basic methods on data structures. However, methods that are generic over closure types are particularly problematic for size: each caller of such a function is almost guaranteed to be passing a unique closure type, so every single call of such a function will be monomorphized. For methods with simple bodies which are always inlined this does not matter (e.g. `Option::map()` just checks an enum discriminant), but for more complex methods the cost can be high. The occurence of monomorphization is invisible to a programmer without manually inspecting the produced binary, making it easy to repeatedly call a generic function without understanding the binary size impact of doing so far exceeds the amount of added code.

In Tock, we discovered substantial monomoprhization bloat in uses of the `Grant` type. The `Grant` type is parameterized by the type of data that can be allocated within the grant (`T`). Methods to interact with `Grant` data provide arbitrary access by being generic over closure types (see Listing 2).

Every `Grant` method with a generic closure parameter, such as the `F` parameter in `Grant::enter()`, is duplicated once per call to that method, rather than just once per type `T` used to construct a new `Grant`. A single instance of the `Grant::enter()` method takes up 150 bytes of flash space. In the Ti50 Tock kernel however, we found 11 different types stored in Grants, and those types were manipulated using `Grant::enter()` on average just over 5 times each. As such there were 59 copies of the method in the final binary, adding up to almost 9kB of code for this single method, most of which is near-duplicate.

This was not an isolated occurence: `kernel::process_-map_or()` was duplicated in the binary to a similar extent, and also took a closure and return type as generic arguments. Dynamic dispatch cannot be used as a direct replacement – in order to support arbitrary return types from the closure,

```
impl<T: Default> Grant<T> {
  pub fn enter<F,R>(&self, id: AppId, f: F) -> Result<R, Error>
  where F: FnOnce(&mut GrantData<T>, &GrantUpcallTable) -> R {
    // The call below is also generic, and inlined in practice
    id.kernel.process_map_or(Err(Error::NoSuchApp), id, |p| {
        let pg = ProcessGrant::new(self, p)?;
        Ok(pg.enter(f)) // access memory, run the closure
    })
  }
}
```

Listing 2. Tock's `Grant::enter()` method. Ti50 had 59 unique calls to this method, such that copies of it contributed 9kB of code.

the concrete type being returned must be known at compile time. The Ti50 GPIO and pinmux drivers also demonstrated monomorphization bloat despite mostly small methods, as a result of the number of pins being monomoprhized across.

Fixing monomorphization bloat using the techniques in §5.1 reduced the size of Ti50 by 8932 bytes: 11.7% of the total savings we achieved.

4.2 Compiler Generated Support Code

Idiomatic Rust relies on "compiler generated support code." This is a catchall term we use to describe a variety of constructions which shorten source code, such as core-library provided procedural macros, declarative macros, and language constructions that generate code, e.g. `Drop`, `async`, `await`.

4.2.1 String formatting. Rust provides formatting machinery, including a set of macros in the core library, to allow for formatting Rust types as strings. These include the `format!()` and `write!()` macros, which are used to build `println!()` in the standard library, and used by Tock to build its equivalent, `debug!()`. These macros all use Rust's `format_args!()` internally. To easily print Rust types, developers can autogenerate printable representations for their custom types by annotating those types with `#[derive(Debug)]`.

Rust's formatting code amounts to 10kB or more of flash [24]. Worse, the code generated by `#[derive(Debug)]` is large for simple types. For example, in Ti50 we used a simple switch statement to implement a manual `to_str()` method for a simple enum with 6 variants. Doing so produced code 112 bytes smaller than that generated by `#[derive(Debug)]`, despite each having equivalent string representations. Size growth from Rust string formatting has been identified elsewhere [2, 24], our findings validate the magnitude of this problem.

In the Ti50 binary, we saved 10,416 bytes by greatly reducing our use of `#[derive(Debug)]` and other core formatting utilities. We also transitioned Ti50 applications to entirely use µfmt [2], a third-party alternative to `core::fmt`, which saved an additional 7,200 bytes. In total, these savings represented 23.1% of the total size reduction we achieved.

4.2.2 Drop trait. The `Drop` trait in Rust implements destructors for types. If `Drop` is implemented for an object, that custom destructor will be called. Additionally, rustc will generate a destructor for any type that owns a type which implements the `Drop` trait. As a result, the `Drop` trait can lead to large amounts of compiler generated code. We observed this phenomenon in two places in the Tock codebase: `AppPtr::drop()` and `Owned::drop()`. Libraries which provide access to types with destructors can contribute surprising code-size growth, since storing these types in other structs can lead to cascading compiler generated implementations.

By removing Drop implementations as described in §5.4, we reduced the size of the Ti50 kernel by 2112 bytes: 2.8% of the total savings we achieved.

4.2.3 Futures. The idiomatic Rust approach to asynchronous execution is Futures, via `core::future::Future`. The `Future` type is very full featured, including an straightforward syntax (`async`/`await`) and simple chaining of futures without allocation. Futures also support dynamic allocation of futures, inline-able executor code, the ability to handle spurious wakeups, and short-lived callbacks. In many embedded contexts, some of these features are unneeded, but their costs are unavoidable. In libtock-rs, we found that even lightweight executor logic adds 100 bytes per future, as it was monomorphized across every future. The Rust future design requires indirection through global state to deal with spurious wakeups, which adds 100-200 bytes of overhead per future, and 330 bytes per combinator used. In total, a futures-based libtock-rs app was 80% larger than an equivalent non-futures based app, with the majority of this overhead expected to scale with the number of futures and future combinators in use.

Futures bloat accounted for 4910 bytes in apps; removing futures support represented 6.5% of the savings we realized.

4.3 Hidden Data and Language Agnosticism

The final category is hidden data: embedded data in the binary added for language constructs. The size of this data is difficult to determine, because of the language abstraction boundary. We refer to this concept as *language agnosticism*: the language does not define how or when this data will appear. Three examples are panics, vtables, and static initializers.

4.3.1 Panics. Panics are how Rust handles unrecoverable errors. Panics can be explicitly forced by calling macros such as `panic!()` or `unreachable!()`, but most panics are buried in `core` library code such as out-of-bounds array access checks, or automatically inserted by the compiler. Panics are common in libcore: `unwrap()`, `expect()`, and many string formatting functions contain panics. Rust does not indicate if a given function might panic.

Rust provides panic information via the `PanicInfo` struct. This structure includes the panic message (the string passed to the explicit call to panic), as well as the source location of the panic *in application code*. In an ablation test, `PanicInfo` represented 9.5% of the Ti50 binary. In the Ti50 kernel, we found replacing panics with errors saved an average of 70 bytes per panic after removing about 50 panics.

This overhead is not unavoidable, but is all or nothing: discarding PanicInfo leaves developers with no information about the source of panics, which makes debugging any runtime panic very difficult. Using any subset of the panic information embeds all of it in the binary, as Rust/LLVM's dead code elimination is unable to optimize on the basis of only part of a struct being unused. In practice, most projects are forced to pay the price of panic overhead, as not being able to debug panics is impractical.

By addressing panic size as described in §5.3, we shrunk the Ti50 binary by 18,924 bytes, 24.9% of our total savings.

4.3.2 Dynamic Dispatch. Rust dynamic dispatch uses a vtable and fat pointers. The &dyn type in Rust takes two words of memory: one word containing a pointer to the object itself, and the other word pointing to a vtable for that trait (see figure 1). &dyn enables concise code compared to generics, but increases binary size. On a 32-bit system, every vtable takes up a fixed overhead of 12 bytes for destructor pointer / size / alignment, in addition to the 4 bytes per method implementation for pointers to the methods themselves. The destructor pointer, size and alignment are only needed to support heap-allocated trait objects. Without a heap it is impossible to store an owned trait object (items stored on the stack must have a known size), and thus it is impossible to drop a trait object without knowing its concrete type. These bytes are wasteful for systems without a heap, there exists one vtable per type used as a trait object. This 12 byte overhead can add up. In the Ti50 kernel, it adds up to 1476 bytes. Trait objects also increase register pressure by using twice as many words as normal pointers/references. For architectures like RISC-V, this increases function call overhead by requiring more stack use.

Finally, trait objects prevent inlining and some dead code elimination: the Rust compiler does not inline virtual function calls, which means dead code elimination cannot occur within methods called via vtables. These overheads are hard to predict when writing code: whether vtables will end up in a binary depends on whether LLVM can perform devirtualization, and the size impact of dead code elimination failing depends on the contents of the function in question. One example of this in Tock is the ProcessStandard implementation of the Process trait. The print_full_process() method, for example, is only needed for debugging, and takes up 3860 bytes on RISC-V. Even if it is not called, it ends up in kernel binaries, because ProcessStandard is passed as a trait object to the Kernel.

By addressing dynamic dispatch bloat using the techniques in §5.2, we reduced the size of the Ti50 binary by 8,230 bytes (10.8% of our savings).

4.3.3 Static Initializers. Global variables with an all-zero representation can be stored in .bss, rather than in .data, saving space. Unfortunately, Rust makes it difficult or impossible for some types to be initialized with underlying 0-values, forcing these initial values to consume valuable flash space. This is partially a result of language agnosticism. In Rust, the underlying bit representation of most types is hidden from the programmer, unless special representations are specified. This language agnosticism enables Rust optimizations which rely on "niches" (e.g. Rust makes types like Option<&T> occupy the same amount of memory as &T by relying on the fact that references can never point to null). One effect of this is that the initial value of global variables can be surprising. For example, in listing 3, the initial bit representation of KG is not all zeroes, despite all fields being initialized to default values which are represented as 0 on their own. In that example, Option<Result<usize, usize>> is the guilty party – None is represented as 0x2, while 0x0 corresponds to Some(Ok(0)). In some cases this problem can be fixed by hunting for a 0-valued representation, but such a value does not always exist, and if found may not be stable across compiler changes.

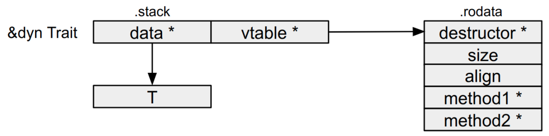

Figure 1. Rust Vtable Depiction. All blocks are word size.

```
struct KeyGuardian {
  task_pending: bool,
  hmac: Option<HmacCtx>,
  res_pending: Option<Result<usize, usize>>,
}
impl KeyGuardian {
  pub const fn new() -> Self {
    Self { task_pending: false, hmac: None, res_pending: None }
  }
}

static mut KG: KeyGuardian = KeyGuardian::new();
```

Listing 3. Example of a Rust structure and its static initialization as a global variable. Surprisingly, this variable is placed in the .data section, rather than the .bss section, as its initial representation in memory has a single non-0 bit. For larger structures this is problematic for size.

In Ti50, a global similar to listing 5 but with more fields produced a 2.1kB static initializer. Changing the default to be 0-initialized saved 2kB, 2.6% of the savings we achieved.

4.4 Ti50 Savings Summary

Table 5 details the code size reduction obtained in Ti50. Addressing all of these items involved modifying over 8500 lines of code. In total, addressing these items plus applying the improved optimizations discussed in section 6 saved 76.1 kB.

5 Best Practices for Small Rust

This section presents 5 best practices for embedded Rust code to avoid the unnecessarily large constructions in §4, and an evaluation of the practices on a snapshot.

Table 5. Savings from addressing each bloat category in Ti50.

Category	Subcategory	kB Saved	% of Savings
Hidden Data	Panic Data	18.9	24.9%
	Dyn Dispatch	8.2	10.8%
	Static Init	2.0	2.6%
	Total	**29.2**	**38.3%**
CGSC	Formatting	17.6	23.1%
	Drop	2.1	2.8%
	Futures	4.9	6.5%
	Total	**24.6**	**32.4%**
Monomorph.	**Total**	**8.9**	**11.7%**
Default Opts	**Total**	**13.4**	**17.6%**
Total Savings		**76.1**	

5.1 Minimize Length + Instantiations of Generic Code

Completely avoiding generics is impractical: they are idiomatic and the Rust standard library depends on them. Carefully minimizing the length of generic code blocks and the number of different types used can provide the benefits without the bloat.

Moving as much code as possible outside of templated scopes minimizes the amount of duplicate code, as shown in Figure 2. In figure 2a, the code to validate the process is all within the templated function. In figure 2b, all code that does not depend on the generic parameters `F` and `R` is moved into a new function, `get_process()`, so it will not be duplicated.

For methods on generic structs, this sometimes requires moving portions of those methods into standalone functions (rather than methods) which will not be monomorphized. In Tock, we applied this approach to several methods on the `Grant` type and to the `process_map_or()` method, realizing over 4kB of savings in the Ti50 binary.

Sometimes, all of the logic in a large generic block is closely tied to the generic parameters. In these cases, it is important to instead minimize the number of instantiations. When possible, these generic functions should be modified to use dynamic dispatch (trait objects), which do not cause monomorphization. Trait objects can harm usability because they prevent the use of generic return types, which provide more ergonomic interfaces. In cases where using trait objects is not possible (such as when a trait is not *object safe*), minimize unique calls to generic functions. Listing 4 provides an example.

5.2 Use Trait Objects Sparingly

While trait objects can reduce monomorphization, they are not a silver bullet. If only one implementation of a trait is used, trait objects add overhead. An example where this is common is hardware abstraction layers (HALs). HALs are used to represent the capabilities of different hardware interfaces in multi-platform codebases. In embedded Rust these are usually implemented using traits: this is true in Tock, HubrisOS [5], and in libraries used by the embedded Rust working group [26]. In a given binary, the polymorphic functions using these traits will only be instantiated for a single concrete type.

Hardware traits sit at the lowest level of the abstraction stack, so generics for hardware interfaces can bubble all the way up to application code. As a result, definitions of higher

```rust
pub fn process_map_or<F,R>(&self, d: R, id: AppId, f: F) -> R
where F: FnOnce(&dyn Process) -> R {
  let process_opt = match self.processes.get(id.index) {
    Some(Some(p)) => if p.id() == id { Some(*p) } else { None },
    _ => None,
  };
  match process_opt {
    Some(process) => f(process),
    None => d,
  }
}
```

(a) Original: each copy of `process_map_or` adds 48 bytes

```rust
pub fn process_map_or<F,R>(&self, d: R, id: AppId, f: F) -> R
where F: FnOnce(&dyn Process) -> R {
  match self.get_process(id) {
    Some(process) => f(process),
    None => d,
  }
}
// ---- Key savings from below function not being generic ----
pub fn get_process(&self, id: AppId) -> Option<&dyn Process> {
  match self.processes.get(id.index) {
    Some(Some(proc)) => {
      if proc.id() == id { Some(*proc) } else { None }
    }
    _ => None,
  }
}
```

(b) New: each copy of `process_map_or` adds 16 bytes

Figure 2. Moving logic out of generic scopes reduces binary size at the cost of readability. A single change similar to this reduced the size of the Ti50 Tock kernel by over 1%.

```rust
let res = match cmd_num {
  1 => grant.enter(id, |app,_| self.send_new(id, app, len))
          .map_err(ErrorCode::from),
  2 => grant.enter(id, |app,_| self.receive_new(id, app, len))
          .map_err(ErrorCode::from),
};
// ----- Below is 112 bytes smaller than above -----
let res = grant.enter(id, |app, _| {
  match cmd_num {
    1 => self.send_new(id, app, len),
    2 => self.recieve_new(id, app, len),
  }
}).map_err(ErrorCode::from)
```

Listing 4. Example from Tock: minimizing instantiations of generic functions to reduce monomorphization size growth.

```
// Generics lead to optimal codegen, but ugly codebases
pub struct SystemManager<A, P, U, G, UAP, UEC>
where
  A: AdcController,
  P: PmuControl,
  // ... (4 more ommitted)
{...}

// If we use a marker trait, we only need one generic
// parameter instead of 6, reducing verbosity
trait SysMgrDeps {
  type A: AdcController;
  type P: PmuControl;
  // ...
}

pub struct SystemManager<S: SysMgrDeps> {
  adc: S::A,
  // ...
}
```

Listing 5. Generic hardware interfaces can lead to a proliferation of generic parameters on higher layer methods. Developers often prevent this by using trait objects. A preferable alternative is to use a marker trait, with all dependent types stored as associated types.

level functions start to become very verbose. In Ti50, it became common to see struct definitions like that in the first half of listing 5. One approach to prevent this is to define a single trait, with several associated type parameters representing each of the dependencies. Doing so prevents many generic parameters from proliferating throughout a codebase: instead a single generic parameter containing all these dependencies can be sufficient. This approach is demonstrated in listing 5.

Finally, when trait objects are used, developers should be careful to avoid relying on inlining-dependent optimizations within the implementations of trait methods.

5.3 Don't Panic!

`panic!()` calls with location information attached cost on average ~70 bytes per call, but panics without this information are almost useless for debugging. We have found the best solution is simply panicking as little as possible. This also improves dependability. In general, panics should be replaced with `Result` based error handling whenever possible.

We performed a panic removal pass on the Ti50 codebase – both its Tock kernel capsules and its Rust applications. Here, we list the 3 most common panics we found, and the fixes we applied. First, we replaced array/slice accesses using the Index operator (e.g. `array[3]`) with calls to `get()`/`get_mut()`, which return errors rather than panicking on out-of-bounds accesses. Second, we replaced calls to `unwrap()`/`expect()` with error handling for cases where the expected value is not present. Third, we addressed panicking functions in the core library without non-panicking alternatives. `core::slice::copy_from_slice()` is one such example. In these cases, removing panics from the binary requires providing additional information to the compiler, such that it knows the panic cannot occur. For `copy_from_slice()`, we inserted branches to return an error before the call in the case that the slices being manipulated are not the same length.

Some panics are impossible to avoid. To minimize these, use the `remap_path_prefix` compiler flag to shorten file paths to just the path within the workspace being built, rather than the entire absolute path. This is especially important when using the `-Z build-std=core` to build a size optimized standard library, since doing so embeds very long paths to the toolchain source code. Further improvements require modifying the compiler. §6 discusses an optimization we contributed to the Rust compiler towards this end.

5.4 Carefully Use Compiler Generated Support Code

Avoid Rust's built in string formatting. Code in `core::fmt` uses dynamic dispatch and contains panics, which bloat binaries. Also avoid using the built in procedural macro for formatting Rust types (i.e. `#[derive(Debug)]`). Handwritten `to_string()` implementations are more size efficient. µfmt [2], a third-party alternative, is much smaller, and should be preferred. However, µfmt does not include any advanced formatting (hex, fixed-width, floating point) and lacks support in the Rust ecosystem. For Ti50, we modified µfmt to include the minimal formatting utilities required by our application (hex and limited fixed-width support). These changes have been upstreamed to a public fork of the library.

Further, avoid using Futures for asynchronous execution. The improved ergonomics of Futures are not worth the cost of an additional 200-300 bytes per future compared to simpler callback schemes. Generic runtimes should not require a futures-based API, and size-sensitive applications should use lighter weight alternatives. For example, libtock-rs uses a simple function-pointer based callback approach combined with scope guards for cleanup of resources.

Finally, be very careful when using the `Drop` trait. Types likely to be stored using trait object references should not implement `Drop`, as these destructors cannot be inlined. Require manual cleanup instead. Libraries should avoid exposing types that implement `Drop`, unless the exposed types are explicitly for use as scope guards.

5.5 Don't Use `static mut`

`static mut` is somewhat widely used in embedded Rust, as the closest analog to global variables in C and the easiest way to create objects with static lifetimes. However, `static mut` can generate surprising amounts of embedded data, or make it easy for seemingly benign changes (e.g. changing a `Result` field to an `Option`) to do so. Beyond this, `static mut` causes rustc to conservatively optimize accesses, as the compiler must assume globals could have changed since initialization. Lazily initialized, non-globally-visible statics, like those provided by `lazy_static!()` [25] or Tock's `static_init!()`,

Table 6. Breakdown of savings from applying our principles to the Tock nrf52dk kernel binary. Unlike table 5, the final column shows the % of the original binary removed by each category, rather than the % of the total savings. This is possible here because these changes were not interspersed with others.

Category	Subcategory	kB Saved	% of Original
Hidden Data	Panic Data	3.5	4.0%
	Dyn Dispatch	10.1	11.6%
	Static Init	0.5	0.5%
	Total	**14.1**	**16.3%**
CGSC	Formatting	1.2	1.4%
	Drop	1.4	1.6%
	Futures	0	0%
	Total	**2.6**	**3.0%**
Monomorph.	**Total**	**3.0**	**3.4%**
Default Opts	**Total**	**3.0**	**3.4%**
Total Savings		**22.6**	**26.4%**

Table 7. Size of sections in the Tock `nrf52dk` kernel binary before and after applying our best practices.

Section	Before (kB)	After (kB)	% reduction
`.text`	62.7	48.7	22.4%
`.rodata`	22.8	15.1	33.5%
`.data`	1.3	0.0	99.7%
total flash	86.7	63.8	26.5%

are preferable for reducing code size and ensuring memory safety.

5.6 Snapshot Evaluation

The Ti50 results in §4.4 measure a production system. However, the practices were applied alongside unrelated changes, so we cannot directly compare the final and original binaries.

This section evaluates the size effects of applying the best practices to a snapshot of a commonly used Tock kernel, the `nrf52dk`[1]. This kernel targets Nordic's `nrf52` development kit, and supports buttons, alarms, ipc, random numbers, bluetooth, sensors, and an interactive console. We applied changes similar to those in §5, targeting each growth item except for `futures`, which the Tock kernel does not use. In total, we modified 2349 lines of code.

We shrank the `nrf52dk` kernel from 86.7kB of flash to 63.8kB of flash, a total reduction in size of 26.4%. Table 6 shows a breakdown. The magnitude of savings is similar to Ti50, but the breakdown is different: different platforms and applications may realize different savings from each best practice. For example, `nrf52dk` realizes more savings from removing trait objects, and less savings from optimizing string formatting. Unlike §4.4, this evaluation allows direct comparisons of binaries before/after size optimizations. Table 7 shows the breakdown of sections before and after these size optimizations. These results show that applying our principles reduce the amount of embedded data and static initializers in the binary by more than they reduce the amount of code in the final binary, as expected based on our initial comparisons of what makes Rust code larger than C.

[1]commit `162efed` from August 17, 2020.

6 Optimizing for Size

Even following the techniques in §5, embedded Rust code could be smaller. This section analyzes some shortcomings that remain, and presents ideas for resolving them through improvements to the Rust compiler and language.

6.1 Better Defaults for Size Optimized Binaries

Size constrained Rust binaries are encouraged to pass `opt-level=s/z` for size optimization. Rust does not ship a `core` library that is built with size optimizations turned on, so even these binaries include code from `core` which is not optimized for size. Obtaining a size optimized `core` library requires rebuilding it on host, which increases compile times and requires a nightly compiler. Other defaults for size optimized binaries are also suboptimal. We propose 3 toolchain changes:

1. Ship a version of `core` compiled at `opt-level='s/z'`, and use it by default at this optimization level.
2. Apply `codegen-units=1` by default for binaries compiled with `opt-level='s/z'` and `lto='fat'`.
3. Use a lower inline threshold for `opt-level='z'`. We find that `inline-threshold=7` produces substantially smaller binaries in several embedded projects. We also found in several projects that for a fixed inline threshold `opt-level='s'` produced much smaller binaries than `opt-level='z'`, which seems like a bug.

In the Ti50 binary, the combination of these 3 optimizations saved an additional 13.3kB, 17.6% of our total savings. Superior models for size-inlining heuristics in LLVM, such as those presented in MLGO [36], could improve further.

6.2 Improved Control of Panic Location Data

Rust's `PanicInfo` stores a message and the filename, line number, and column number of the source location for each panic as embedded data in the final binary. Each panic thus has a different location argument. In a snippet like `let b = a[0] + a[1] + a[2]`, three separate panics and associated data will end up in the binary, instead of a single out-of-bounds access panic. One optimization to reduce the binary growth from using panics is to provide control of the amount of location data stored for each panic, such that less embedded data is stored, and so that more panics can be fused.

We implemented and profiled this optimization. We implemented this as a rustc flag `-Z location-detail`, which allows developers to choose any combination of filename

Table 8. Ablation analysis of panic location element size, showing the size of the Tock "Imix" binary after progressive removal of panic location fields. Ti50 does not use this flag yet.

Element Removed	Code Size (Bytes)	Total Savings
None	175,304	0%
- column numbers	174,424	0.6%
- line numbers	163,752	6.6%
- filename	158,356	9.6%

/ line number / column number to be tracked in panic locations. We built a compiler with this flag, and measured the size savings from excluding each portion of the location. Table 8 shows the results for the Tock "Imix" application. This flag requires a newer Rust toolchain, so Ti50 does not use it yet.

We contributed this optimization to the upstream Rust compiler, where it was accepted (citation omitted for anonymity).

Additional optimizations could be made. The LLVM IR type of panic locations is `{{[0 x i8]*, i32}, i32, i32}`, which could be cut in half by using variants for line/column info. Using the LLVM `char6` encoding for filenames could further decrease the size of panic locations.

6.3 Improved Dead Code Elimination

Use of third-party libraries depends on dead code elimination. LLVM's dead code elimination is generally effective, but breaks down when trait objects are used: passing a type as a trait object incurs the size cost of all methods on that trait.

This problem is worse for Rust than C++, where the ability to declare only some functions in a class as virtual significantly mitigates the size impact of this problem. LLVM's "dead virtual function elimination" (DVFE) [23] further mitigates this issue for C++. Unfortunately, rustc provides insufficient type metadata to LLVM, such that DVFE is only possible in very limited, simple cases. In the future, the compiler could walk all calls to trait methods, determine the set of trait methods which are never called, and optimize those away.

6.4 Improved Devirtualization

Even when only a single implementation of a trait exists in a statically linked binary, rustc may not devirtualize calls on trait objects for that trait. This optimization is called "whole program devirtualization" (WPD). Rust's lacking support for LLVM WPD means these optimizations are not applied in many cases where Clang could apply them for C++ code with dynamic dispatch. Future implementation work could identify traits where only a single type is ever instantiated as that trait, and replace all uses of that trait with direct calls to functions on the concrete type.

6.5 Smarter Polymorphisation

Rust's use of monomorphization for polymorphism increases code size in exchange for more predictable performance [1]. For embedded Rust, it would be better to deduplicate whenever possible. The Rust compiler can skip unneeded monomorphization: a process called polymorphisation. Unfortunately, Rust's simple polymorphisation implementation [37] only works for functions whose generic parameters are entirely unused. Ideally, polymorphisation would apply anytime two different generic function instances produce functionally identical LLVM-IR, similar to how C# can de-duplicate generic instances with different types of identical size/alignment.

6.6 Virtual Table Implementation

§4.3.2 discusses the size overhead of dynamic dispatch in embedded Rust, including the presence of 12 bytes of unused data in every vtable for binaries which do not support dynamic allocation. In the Ti50 binary, we found 123 vtables, adding up to almost 1500 bytes of waste. A compiler option to remove this unused data would provide significant savings.

7 Tools for Analyzing Rust Binaries

This work required attributing portions of an optimized binary to the source code responsible for its inclusion. One existing tool was particularly helpful: cargo-bloat [30]. Cargo-bloat outputs an estimate of the size in `.text` of each function in an ELF file. Unlike language-independent tools like bloaty [12], cargo-bloat uses `cargo` metadata to improve accuracy. Cargo-bloat is useful in identifying monomorphization bloat and in understanding the effects of optimizations like inlining. The inability to analyze sections other than `.text` limits its usefulness, since other sections like `.rodata` can be quite large. One tool which can analyze other sections is Tock's print_tock_memory_usage.py. However, its characterization of padding/data is not rigorous and at times inaccurate.

We developed 2 new tools, each open-sourced. First, we released embedded_data_analyzer, and submitted it to Tock. This tool can attribute embedded data (i.e. data in the final binary that is not instructions) to the source-level functions responsible for their inclusion. This tool is currently capable of attributing 66% of embedded data in an example Tock binary, and was useful in identifying the "hidden data" items discussed in this paper in §4.3. We also developed find_panics.py, which is a tool for tracing source locations of panics in a Rust ELF. This tool was submitted and accepted into the Tock github repository. This tool uses DWARF debug information in the ELF to backtrace panics. It is special in that it identifies the *user* code responsible for a panic being included in the binary, even if the actual panicking branch is contained in core library code. It is useful for removing panics, a key element in following our size reduction principles. Without this tool, it can be difficult to identify what user code actually generates panics in a final optimized binary.

8 Discussion

Idiomatic embedded Rust code compiles to larger binaries than comparable programs written in C. This is due to two design principles in the language. First, Rust relies heavily on compiler optimizations to provide high-level constructions as "zero-cost abstractions". Designed originally for performance critical applications such as the Firefox page rendering engine, Rust's abstractions are designed to be fast, not small. In Rust, "zero-cost", means zero-added-clock-cycles, not zero-added-size. Core library methods that are generic over closures are one example of this: monomorphization allows each invocation to be as fast as a normal function call.

Second, Rust's higher-level language features blur the relationship between source code length and resulting binary size. C was designed to function as a "high level assembler" [16], and as such there is a generally linear relationship between lines of C code and number of assembly instructions. In Rust, however, small pieces of code can introduce large blocks of instructions: §4.1 showed how invoking `Grant::enter` with a trivial closure added 150 bytes. Furthermore, size optimizations in the compiler are fragile, making predicting the size impact of changes harder. Examples of this include trait objects breaking programmers mental model of dead code elimination, or panics requiring less source code but more size than handling and propagating errors.

This paper proposes a set of principles for reducing the size of embedded Rust binaries, but examines only Tock kernels and applications. To see if these principles are generalizable, we look briefly at Hubris, a new, commercial embedded Rust OS that targets security critical applications [5]. Hubris was released after we applied the principles to Ti50: it is an independent data point. Hubis currently only has very simple example applications: we describe results from `demo-stm32h7-nucleo/app-h743` because it is one of the most complex.

Hubris has a very different architecture than Tock. Applications consist of many small, independently compiled tasks, each of which is heavily inlined. As a result, there is very little monomorphization bloat within tasks, but tasks replicate all shared code. In some cases, Hubris follows the proposed principles: it uses a single trait object, does not support `Futures`, has only two implementations of the `Drop` trait, and has very few `static mut` variables besides 0-initialized buffers.

In other cases, however, applying the principles had significant savings. The core Hubris kernel task, for example, has many panics: replacing 13 of these panics with errors shrank the task by 608 bytes, a 3% size reduction by changing 13 lines of code. Applying our recommended optimization profile[2] led to cutting 22.5kB from the 150kB application (15%). The `location-detail` compiler flag we implemented is also beneficial to Hubris: when applied on top of the optimization profile, it could reduce the app size by up to 10.5 kB (8%).

[2]`opt-level=s`, `build-std=core,compiler_builtins`, use of `remap-path-prefix` to shorten paths to `core`, and `inline-threshold=7`

9 Conclusion

Rust's memory safety and other guarantees are extremely valuable for embedded software. However, idiomatic Rust code has significantly larger binaries than C equivalents, which is problematic for flash-constrained embeddded systems. This paper examined the causes of these size increases, finding many are avoidable. It described a simple set of 5 programming principles for embedded Rust, which, combined with compiler options, reduced the size of a commercial binary by 19%. Furthermore, we outlined several directions for the Rust compiler that could further reduce binary size.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1931750 "Secure Smart Machining." This work was also supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and by the National Science Foundation under Grant No. CNS-1824277. The authors would also like to thank the entire Ti50 team at Google for all of their help with this work.

References

[1] Brian Anderson, Lars Bergstrom, Manish Goregaokar, Josh Matthews, Keegan McAllister, Jack Moffitt, and Simon Sapin. 2016. Engineering the Servo Web Browser Engine Using Rust. In *Proceedings of the 38th International Conference on Software Engineering Companion* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 81–89. https://doi.org/10.1145/2889160.2889229

[2] Jorge Aparicio. Accessed: 2022-03-03. μfmt. https://github.com/japaric/ufmt.

[3] Ashwin Kumar Balakrishnan and Gaurav Nattanmai Ganesh. 2022. Modern C++ and Rust in embedded memory-constrained systems. (2022).

[4] Vadim Bendebury. 2018. Google Security Chip H1: A member of the Titan Family. (2018). Open Source Firmware Conerence 2018.

[5] Cliff L. Biffle. 2021. On Hubris and Humility: developing an OS for robustness in Rust. (2021). Open Source Firmware Conerence 2021.

[6] Nico Borgsmüller. 2021. *The Rust Programming Language for Embedded Software Development.* Ph. D. Dissertation. Technische Hochschule.

[7] Milind Chabbi, Jin Lin, and Raj Barik. 2021. *An Experience with Code-Size Optimization for Production IOS Mobile Applications.* IEEE Press, 363–366. https://doi.org/10.1109/CGO51591.2021.9370306

[8] Saumya K Debray, William Evans, Robert Muth, and Bjorn De Sutter. 2000. Compiler techniques for code compaction. *ACM Transactions on Programming languages and Systems (TOPLAS)* 22, 2 (2000), 378–415.

[9] Tock Project Developers. Accessed: 2022-03-03. libtock-rs. https://github.com/tock/libtock-rs.

[10] Paul Emmerich, Simon Ellmann, Fabian Bonk, Alex Egger, Esaú García Sánchez-Torija, Thomas Günzel, Sebastian Di Luzio, Alexandru Obada, Maximilian Stadlmeier, Sebastian Voit, et al. 2019. The Case for Writing Network Drivers in High-Level Programming Languages. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 1–13.

[11] David Gay, Philip Levis, Robert Von Behren, Matt Welsh, Eric Brewer, and David Culler. 2003. The nesC language: A holistic approach to networked embedded systems. *Acm Sigplan Notices* 38, 5 (2003), 1–11.

[12] Google. Accessed: 2022-03-03. bloaty. https://github.com/google/bloaty.

[13] Hugo Heyman and Love Brandefelt. 2020. A Comparison of Performance & Implementation Complexity of Multithreaded Applications in Rust, Java and C++.

[14] Håvard Wormdal Høiby and Sondre Lefsaker. 2015. *RustyGecko-Developing Rust on Bare-Metal-An experimental embedded software platform.* Master's thesis. NTNU.

[15] M. M. Hossain, M. Fotouhi, and R. Hasan. 2015. Towards an Analysis of Security Issues, Challenges, and Open Problems in the Internet of Things. In *2015 IEEE World Congress on Services.* 21–28.

[16] ISO. 2007. *International Standard ISO IEC 9899:1999: Technical Corrigendum 3.* pub-ISO. 10 pages. http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=43485;http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf

[17] johnthagen. Accessed: 2022-03-03. Minimizing Rust Binary Size. https://github.com/johnthagen/min-sized-rust.

[18] Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language.* No Starch Press.

[19] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. 1994. Partial dead code elimination. *ACM SIGPLAN Notices* 29, 6 (1994), 147–158.

[20] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[21] Rainer Leupers and Peter Marwedel. 1999. Function inlining under code size constraints for embedded processors. In *1999 IEEE/ACM International Conference on Computer-Aided Design. Digest of Technical Papers (Cat. No. 99CH37051).* IEEE, 253–256.

[22] 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 *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17).* Association for Computing Machinery, New York, NY, USA, 234–251.

[23] Linaro. Accessed: 2022-03-03. LLVM Dead Virtual Function Elimination. https://llvm.org/devmtg/2019-10/slides/Stannard-DeadVirtualFunctionElimintation.pdf.

[24] James Munns. Accessed: 2022-03-03. Formatting is Unreasonably Expensive for Embedded Rust. https://jamesmunns.com/blog/fmt-unreasonably-expensive/

[25] Rust Language Nursery. Accessed: 2022-03-03. lazy-static.rs. https://github.com/rust-lang-nursery/lazy-static.rs.

[26] Embedded Rust Working Group. Accessed: 2022-03-03. embedded-hal. https://github.com/rust-embedded/embedded-hal.

[27] Anthony Perez. 2017. Rust and C++ performance on the Algorithmic Lovasz Local Lemma. *Project Report. Stanford: Stanford University, Dec* (2017).

[28] Philip J Plauger. 1997. Embedded C++: An Overview. *Embedded Systems Programming* 10 (1997), 40–53.

[29] César A Quiroz. 1998. Using C++ efficiently in embedded applications. In *Proceedings of the Embedded Systems Conference.*

[30] Yevhenii Reizner. Accessed: 2022-03-03. cargo-bloat. https://github.com/RazrFalcon/cargo-bloat.

[31] 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.

[32] Rodrigo CO Rocha, Pavlos Petoumenos, Zheng Wang, Murray Cole, Kim Hazelwood, and Hugh Leather. 2021. HyFM: function merging for free. In *22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems: Co-located with PLDI 2021.*

[33] Kang Seonghoon. Accessed: 2022-03-03. Why is a Rust executable large? https://lifthrasiir.github.io/rustlog/why-is-a-rust-executable-large.html

[34] Herb Sutter. 2019. *P0709: Zero-overhead deterministic exceptions: Throwing values.* Standard Proposal. C++ Standards Committee.

[35] Sriraman Tallam, Cary Coutant, Ian Lance Taylor, Xinliang David Li, and Chris Demetriou. 2010. Safe ICF: Pointer Safe and Unwinding Aware Identical Code Folding in Gold. In *GCC Developers Summit.* http://gcc.gnu.org/wiki/summit2010?action=AttachFile&do=view&target=tallam.pdf

[36] Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. 2021. Mlgo: a machine learning guided compiler optimizations framework. *arXiv preprint arXiv:2101.04808* (2021).

[37] David Wood. 2020. Polymorphisation: Improving Rust compilation times through intelligent monomorphisation. (2020).

[38] Hongwei Xi. 1999. Dead code elimination through dependent types. In *International Symposium on Practical Aspects of Declarative Languages.* Springer, 228–242.

[39] Peng Zhao and Jose Nelson Amaral. 2005. Function outlining and partial inlining. In *17th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'05).* IEEE, 101–108.