

Rust for Morello: Always-On Memory Safety, Even in Unsafe Code

Sarah Harris ✉

University of Kent, Canterbury, UK

Simon Cooksey ✉ 🏠 

University of Kent, Canterbury, UK

Michael Vollmer ✉ 🏠 

University of Kent, Canterbury, UK

Mark Batty ✉ 🏠

University of Kent, Canterbury, UK

Abstract

Memory safety issues are a serious concern in systems programming. Rust is a systems language that provides memory safety through a combination of a static checks embodied in the type system and ad hoc dynamic checks inserted where this analysis becomes impractical. The Morello prototype architecture from ARM uses capabilities, fat pointers augmented with object bounds information, to catch failures of memory safety. This paper presents a compiler from Rust to the Morello architecture, together with a comparison of the performance of Rust's runtime safety checks and the hardware-supported checks of Morello. The cost of Morello's always-on memory safety guarantees is 39% in our 19 benchmark suites from the Rust crates repository (comprising 870 total benchmarks). For this cost, Morello's capabilities ensure that even unsafe Rust code benefits from memory safety guarantees.

2012 ACM Subject Classification Software and its engineering → Compilers; Software and its engineering → Software safety; Software and its engineering → Object oriented languages

Keywords and phrases Compilers, Rust, Memory Safety, CHERI

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.39

Category Experience Paper

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact):*

<https://doi.org/10.4230/DARTS.9.2.25>

Software (Source Code): <https://github.com/kent-weak-memory/rust>

archived at `swh:1:dir:966327cc0ecb3fb4d2196b6f0912d775392fafa5`

Funding This work has been supported by the EPSRC under the DSbD Software Ecosystem grant programme EP/X021173/1.

Acknowledgements This paper was greatly improved thanks to the responses of anonymous reviewers. We extend our thanks to Jessica Clarke for her invaluable help with CHERI LLVM.

1 Introduction

Low-level programming entails delicate use of memory in a setting where common mistakes can lead to serious bugs and security vulnerabilities in critical code. *Memory safety* is the absence of these errors – where only correctly allocated regions of memory are accessed and freed. Unsafe uses of memory are the most critical software flaws today, they create security vulnerabilities, and they are widespread: out-of-bounds writes are the most dangerous security flaw in the Mitacs Common Weakness Enumeration [8]; Microsoft found that 70%



© Sarah Harris, Simon Cooksey, Michael Vollmer, and Mark Batty;
licensed under Creative Commons License CC-BY 4.0
37th European Conference on Object-Oriented Programming (ECOOP 2023).
Editors: Karim Ali and Guido Salvaneschi; Article No. 39; pp. 39:1–39:27



Leibniz International Proceedings in Informatics
LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



of the security bugs in Windows were as a result of unsafe use of memory [23]; and in the Chromium browser 36% of bugs are caused by use-after-free errors [33], with a further 33% stemming from other unsafe uses of memory [33].

It is possible to automatically enforce memory safety, avoiding all the attendant bugs and security flaws. This enforcement comes at a cost, however, and how exactly the cost is levied is a design choice. In this paper we compare the runtime performance of the mechanisms that enforce memory safety in *Rust* and in the *Morello* architecture, and ultimately combine them, improving the coverage of Rust's memory safety guarantee.

Rust provides a guarantee of memory safety to all well-typed code. Much of the cost of this guarantee is handled by a static analysis in the type system, but it also uses runtime checks when proving safety statically would be costly or impossible. One can forgo the safety guarantee and its checks by designating a block of code as **unsafe**: memory accesses within this block are not required to pass the full rigour of the type system. Unsafe code is used sparingly for interoperability with non-Rust components and for performance. **unsafe** annotations highlight code where memory errors can survive.

Rust alleviates memory errors in common code while remaining flexible enough to support systems programming through the provision of **unsafe** blocks. Rust imposes two costs: programmers must adhere to a more restrictive type system, and there are runtime costs to support the safety guarantee. The combination of safety, pragmatism, and performance is why Rust is now the official second language of the Linux kernel alongside C [24].

Morello is a prototype ARM processor that provides *capabilities*: fat pointers, augmented with permissions and bounds information. Morello processors use this metadata to enforce memory safety at run time, halting programs when safety is violated – for example if a program makes an out-of-bounds memory access. Programs that protect every memory access are described as *Purecap*, but one can forgo the safety checks by accessing memory in *Hybrid* mode. In contrast with Rust, full-scale systems programming is possible in Purecap mode: there are Purecap Morello ports of BSD, Linux and Android [34, 4, 3]. Programming in Purecap mode ensures memory safety from the first, albeit with the possibility of runtime errors where safety would otherwise be violated. Further development effort improves the stability of the system.

The safety guarantees provided by Rust and Morello are subtly different. Morello's capabilities track only approximate bounds information (see §2.3), recording the size of objects as a floating-point number; Rust applies compiler optimizations to the runtime checks that it emits, removing unnecessary checks and improving performance. Even so, Rust and Morello provide similar guarantees that may be used in a complementary way. Purecap-Morello will check pointers are used safely even in **unsafe** blocks, for example. This is where Rust and capability hardware mesh neatly, where Rust cannot validate the safety of memory use the underlying Morello hardware can.

In this paper we will explore the interplay between the Morello prototype hardware and the Rust programming language with a focus on runtime performance. We find the performance cost of always-on hardware memory safety checks to be quite high but not prohibitively so, and the design philosophies of Rust's static memory safety guarantees and Morello's dynamic memory safety to be well-matched. This comparison will also serve as a way to benchmark the real-world performance of improvements to the Morello architecture in the context of cutting edge memory-safe programming practices. We present the following contributions:

1. A Rust compiler that targets the Purecap- and Hybrid-mode Morello prototype capability hardware (§3).
 - This includes a reasoned choice for the semantics of `usize` on a machine where pointers are not just integers (§3.1).
2. A ported Rust standard library with fixes to incorporate capabilities (§3.6).
3. A performance analysis of Rust on Morello (§4, §5):
 - Our benchmark suite of 19 crates from the Rust crates repository (§4.6, Appendix A).
 - Analysis of the benefits of bounds checking elision on capability hardware (§5.1).
 - Comparison of Rust on Hybrid-Morello to Rust on Purecap-Morello (§5.2).
4. Our artefacts which will be made public upon publication.

2 Background

For a language to provide practical memory safety, it must present a usable interface to the programmer and have acceptable performance. *Memory safety* comes with a number of requirements:

- ① values must be initialized before reading, especially pointers,
- ② values must not be accessed after deallocation,
- ③ reads and writes must be within the bounds of an object’s memory allocation,
- ④ values must be deallocated exactly once.

There are a number of approaches to enforcing these requirements: from garbage collection as used by Java, Go, OCaml and many others; assorted static and dynamic validators – Rust is one such system, using linear types, lifetimes and dynamic bounds checking; and now hardware schemes like Morello.

2.1 Rust

Rust [21] is a relatively young programming language, version 0.1 was released in 2012 [27], and is notable for incorporating a number of features geared to providing memory safety, covering the desiderata above. Minimising runtime cost while providing powerful features to programmers is a central design aim for current Rust [35], so the majority of these features are applied statically during compilation. There are however still some cases where it is regarded as impractical to infer bounds statically.

The most relevant features to Rust’s memory safety are [18, 28]:

- uninitialised values are not normally¹ allowed ①;
- move semantics, the `Drop` trait, and references prevent access to deallocated values ②;
- array and slice indices are the only pointer arithmetic normally¹ available, and these are bounds checked at runtime ③;
- move semantics, the `Drop` trait, and careful API design protects against double free ④;
- move semantics and the `Drop` trait provide some protection against memory leaks by making the default behaviour that objects are freed when their lifetime ends ④, but can be defeated by functions like `std::mem::forget()` and `Box::leak()` and by other issues²;

The most important of these safety features is the combination of move semantics, the `Drop` trait, references, and lifetimes, which together provide much of Rust’s memory safety guarantees.

¹ In `unsafe` it is possible to break these conditions, but this is not the default.

² Reference counted pointers can leak if used improperly, and some edge cases in exception handling can cause `Drop` not to execute.

2.1.1 Move and Drop

The simplest way to allocate memory in Rust is to use the stack. This works much like C, with a value being allocated memory on entry to a block, and deallocated on exit when the stack frame is popped:

```
struct Data { a: i32, b: i32 }
fn automatic_memory() {
    // data automatically allocated on the stack here:
    let data = Data{a: 1, b: 2};
    // ...
    // data falls off stack here
}
```

Large values and data with a lifetime that doesn't match that of a scope require dynamic memory allocation, which in C would be provided via `malloc()` and `free()`. In Rust this is provided using a combination of move semantics and the `Drop` trait. `Drop` allows a type to provide a method that will be called automatically when an instance of it leaves scope, which means that memory allocations can be managed semi-automatically by so-called “smart pointers”. The simplest implementation of this in Rust is the standard library type `Box`, which allocates memory on the heap when instantiated and uses `Drop` to ensure that it will be deallocated when the `Box` leaves scope. The `Box` value itself acts as a handle, tracking the lifetime of the allocation and providing access to the allocated memory while remaining a technically separate value.

```
fn heap_memory() {
    // data allocated on the heap here:
    let data = Box::new(Data{a: 1, b: 2});
    // ...
    // Box falls out of scope here, heap allocation automatically freed
}
```

Move semantics expand the utility of this approach by allowing the `Box` to be moved to a different name or out of scope, while preventing it from being duplicated or deallocated, which might otherwise cause the allocation to be freed twice or left to leak [4](#).

```
fn inner_scope() -> Box<Data> {
    // data allocated on the heap here:
    let data = Box::new(Data{a: 1, b: 2});
    // ...
    data // data is moved out of the function here
}
fn moved_box() {
    // data moved to outer scope here:
    let data = inner_scope();
    // ...
    let new_name = data; // data moved to new_name here
    assert!(data.a == 1); // compiler error: use of moved value: `data`
    assert!(new_name.a == 1); // ok
    // data falls out of scope here, heap allocation automatically freed
}
```

`Box` covers a wide range of use cases, and the same general design can be used to build more complex, more powerful tools to cover more demanding problems. The standard library provides a number of options, including dynamically resizable arrays via `Vec`, and reference counted allocations via `Rc` using the same mechanism.

2.1.2 References and lifetimes

While move semantics and `Drop` cover many uses of dynamic memory allocation, values can only be in one place at a time, and the resulting passing around can quickly become inconvenient. The solution to this problem is Rust’s reference types. These behave similarly to ordinary pointers, but enforce extra rules that are checked by the compiler ¹:

- a reference must point to a value, i.e. there are no null references ①,
- values pointed to must be currently allocated and correctly aligned ①,
- a value can either be referenced once mutably, or multiple times immutably,
- values can only be mutated via a mutable reference ², and
- values cannot be moved or deallocated while referenced ②.

The compiler statically checks these rules using a system of inferred lifetimes, which allow references to be moved around and copied in non-trivial ways while maintaining safety. This concept has its roots in region-based memory management [14, 11], and it prevents access to values after deallocation while still providing power and flexibility ②.

In the example below, the two `&mut data` references are not permitted to have overlapping lifetimes. Rust infers that an object’s lifetime ends when the last reference to it goes out of scope – in this example, at the end of `referenced_box()`.

```
fn referenced_box() {
    // data allocated on the heap here:
    let mut data = Box::new(Data{a: 1, b: 2});
    // two immutable references exist during this call:
    use_data(&data, &data);
    // and two mutable references, which causes a compiler error:
    // cannot borrow `data` as mutable more than once at a time
    use_data(&mut data, &mut data);
    // reference only exists for duration of expression:
    *get_field(&mut data) = 3;
    // no references exist by here, so data freed without errors
}

fn get_field(data: &mut Data) -> &mut i32 {
    // compiler infers lifetime of return from argument
    &mut data.a
}

fn use_data(a: &Data, b: &Data) {
    assert!(a.a == b.a);
}
```

Dangling references, i.e. references to values which would be de-allocated at the end of a scope, are forbidden in Rust.

¹ Rust doesn’t currently have a formal specification, so the best sources for this information (besides the compiler source code) are the Rust Book [18] and the Rust Reference [28]

² Though this can be circumvented via a mechanism called “interior mutability”.

```
fn dangle() -> &u32 {
    let value = 0;
    &value
}
// this function's return type contains a borrowed value, but there is no
// value for it to be borrowed from
// help: consider using the `static` lifetime
```

Rust instead provides a mechanism for overriding the point at which a lifetime ends by explicit annotation:

```
fn dangle2<'a>() -> &'a u32 {
    let value = 0;
    &value
}
```

The *borrow checker* is the name of the machinery in Rust which statically detects invalid uses of references and keeps track of when objects' lifetimes end.

2.1.3 unsafe

Rust includes a mechanism that allows programmers to choose to break the rules described above. This is useful for performance-critical hand optimisation and working around the limitations of the compiler's static checking. Use of `unsafe` indicates that something odd is afoot, it should be used sparingly and serves as an explicit marker to signal to programmers and auditing tools alike that these pieces of code require additional scrutiny.

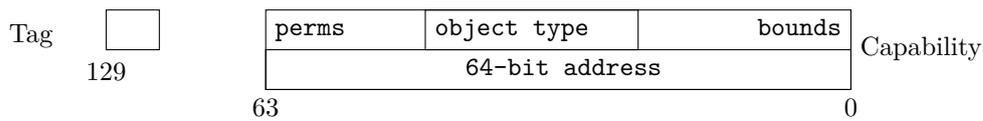
`unsafe` isn't a free pass to do anything – only a very specific set of extra privileges are available within these sections (see the Rust Book [18, §19.1]):

- ordinary C-like pointers called *raw pointers*, which don't have the limitations of references, can be dereferenced ¹²³;
- functions and types marked `unsafe` can be used, allowing access to additional library APIs ¹³ ²⁴ ³⁵ ⁴⁶;
- static variables can be accessed (which come with thread safety issues);
- traits marked `unsafe` can be implemented, automatically creating more `unsafe` code;
- unions can be accessed, which can be used to bypass type checking ¹³;
- external C/C++ functions may be called, importing the memory safety concerns of those languages ¹²³⁴.

In return, the programmer promises not to break any of the language's invariants.

The most important of these are the ability to use raw pointers and to call `unsafe` interfaces. Raw pointers are the most significant hole in the safety guarantees that Rust can provide, but *they are the primary point of compromise between full safety and a usable systems programming language*. Raw pointers are motivated by interoperability with non-Rust components, either through the operating system ABI or through linking C components into Rust programs, or Rust components into C programs. Raw pointers allow all the usual trickery, including creating them from integers, arbitrary arithmetic, null pointers, dangling

³ `std::mem::MaybeUninit`
⁴ `std::slice::from_raw_parts()`
⁵ `slice::get_unchecked()`
⁶ `Box::from_raw()`

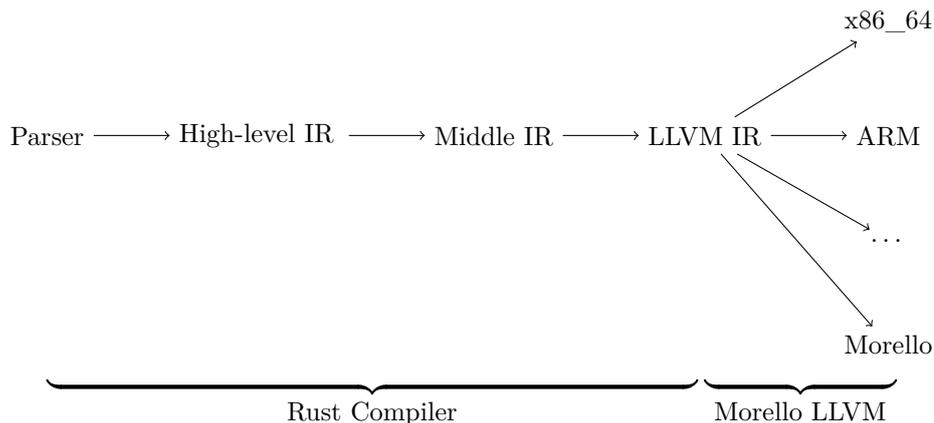


■ **Figure 1** The structure of a 128-bit capability.

pointers, freedom to cast between types, and so on. Access to `unsafe` interfaces enables use of a number of standard library features, notably `std::mem::transmute()`, which allows casting between any pair of types with the same size, and `std::mem::MaybeUninit`, which allows the creation of uninitialised values [\[1\]](#).

2.2 The Rust Compiler

The Rust compiler is mostly self-hosting, i.e. Rust is implemented in Rust. The front-end, type checking, middle intermediate representation (MIR), and an increasing number of optimisations are implemented in Rust. LLVM provides the backend and remaining optimisations, consuming LLVM IR and compiling to a range of targets including ARM and x86_64. This is all very convenient for porting Rust to Morello, as there is an existing LLVM implementation for Morello [20]. A sketch of the compiler’s structure is drawn below.



The majority of the compiler changes outlined in §3 are over the Middle IR portion of the compiler.

2.3 Capability hardware

CHERI is a generic instruction set extension to introduce *capabilities*, an extended pointer representation designed to add hardware security to memory accesses [36]. Capabilities add validity and permission information to pointers, expanding them to 128-bits on 64-bit platforms. Each capability is attached to a 1-bit *validity tag*, and this is required to be set to successfully perform memory accesses via the capability. Further, valid capabilities can only be constructed from other valid capabilities, and only in ways which don’t exceed the permissions of the parent capability. This property allows software to be separated into *compartments* which have limited, controlled access to one another.

Capabilities add four pieces of metadata to plain pointers: bounds, permissions, flags, and object type. An example of the structure of a 128-bit capability can be seen in Figure 1.

Bounds detail the range of memory which a capability is allowed to address; the hardware uses this information to perform automatic bounds checking. To be effective, the bounds of capabilities must be appropriately restricted to the object or buffer they point to. The

CHERI LLVM project extends the LLVM compiler infrastructure to do exactly this [20]. Most C and C++ programs, as well as a number of other languages with an LLVM backend, benefit because they automatically gain bounds checking with minimal to no modifications. It is worth noting, however, that CHERI's bounds checking has a limitation: to keep capability sizes reasonable, bounds information uses a floating point encoding. This means that bounds for larger regions become increasingly approximate, and marginally out of bounds accesses may succeed if regions are not padded to fill the extra space.

Permissions can be used to provide fine-grained read, write, execute, and capability manipulation protections for each capability.

Flags provide space for architecture-specific controls for each capability.

Object types are used as part of a mechanism to *seal* capabilities, such that they can only be dereferenced by specifically and deliberately chosen pieces of code. This opens up possibilities for compartmentalising programs and moving data and permissions through untrusted compartments without degrading security properties.

As a result of the requirement for capabilities to be derived from other valid capabilities, the provenance of each capability must be well-defined. This is a critical part of controlling the access rights of different parts of programs. For these limitations to hold, validity bits in uninitialised memory must be cleared before access is granted. This is expected to be provided by hardware or low level system software [37, §3.6.2]. This ensures that any uninitialised pointer will be invalid, and therefore impossible to dereference accidentally. The downside of the provenance property is that integer-to-pointer casts in existing code are likely to become invalid, though the changes needed to fix this are often minor. This is unsurprising given that integer-to-pointer casts are something of a headache for provenance analyses already [22].

Interpreted generously, bounds checking and provenance guarantees cover points ❶ (no use of uninitialised pointers) and ❸ (bounds checking) of §2. Research into ways that CHERI might be used to provide temporal safety guarantees is ongoing [39]. There is research investigating implementations of `free()` that can sweep the memory and invalidate any pointers into the free'd memory region [41] ❷. Even with temporal safety, memory leak bugs remain a problem to be solved by other tools ❹.

2.3.1 Morello prototype

Morello is a prototype platform for exploring capabilities, and is based on an application-class ARM SOC, the same sort that is found in modern smartphones and ARM-based personal computers. It is supported by a suite of open source software, including a C/C++ compiler based on LLVM [20], a FreeBSD port [34], and custom build automation tooling [9]. Software for Morello can be compiled in two different modes, both of which are supported by the CHERI BSD operating system we use [34].

Hybrid mode

In this mode of compilation, pointers are stored using a plain integer representation instead of capabilities, and normal ARM load and store operations are used to dereference them. Pointers are transparently restricted by the bounds of a single default capability provided by the operating system, with software running as if it were using normal ARM hardware. Rust code compiled with the non-CHERI compiler target triple `aarch64-unknown-freebsd` will run on CHERI BSD in this mode. While this mode is defined by the use of capability unaware load and store operations it is still possible to explicitly use capabilities, if the programmer desires and the facility is exposed by the language.

Purecap mode

In *pure capability* (Purecap) mode, all pointers are represented using capabilities, and capability instructions are used to access memory. Our modified compiler adds a new target triple to support this mode called `aarch64-unknown-freebsd-purecap`. This target configures the LLVM backend to emit capabilities instead of plain pointers, and the Rust compiler to use data layouts appropriate for capabilities. The details of the changes necessary to enable Rust for Purecap Morello are outlined in the next section.

Morello uses 128-bit capabilities, throws hardware exceptions if an invalid capability is dereferenced, and the maximum bounds size precisely representable is 4 KiB [1].

3 Adjustments to the Rust compiler and standard library

We describe the changes to accommodate capabilities in the Rust compiler. Recall that capabilities are 128-bit with one invisible tag bit which is maintained transparently by the hardware. To be able to use capabilities to represent pointers, a number of modifications to the compiler and standard library are necessary. Our port of Rust is based on release 1.56.0 (Edition 2021 Rust).

3.1 Rust semantics open question: `usize`

`usize` is an integer type, ambiguously defined by the documentation as the “[...] pointer-sized unsigned integer type.” [26] This is a straightforward definition on conventional architectures with integer pointer representations, but on Morello the meaning becomes unclear. There are two obvious interpretations for the semantics of `usize`:

- `usize` should be an integer and contain only an address, i.e. the lower word of a capability – a 64-bit number, or
- `usize` should behave like an integer and contain a whole capability, i.e. a Morello double-word sized 128-bit number.

We chose to explore the 64-bit approach for a number of reasons. First, the Rust community have sought to resolve this, and there are ongoing discussions that are leaning towards word-sized `usize` [31, 25]. Secondly, the previous work by Sim [32] explored 128-bit `usize` and was left with a handful of technical limitations which would be side-stepped by using machine-word-sized `usize`. Finally, there are several technical benefits to 64-bit `usize`, which we describe below.

Efficiency

`usize` is the only type of integer that can be used in array indexing, and is also used to represent lengths of arrays and sizes of types. These uses are very common, and only require that `usize` be able to represent the full range of addressable memory locations. In comparison, pointer-integer casts that would only function if `usize` stored a complete capability are rare. Making `usize` large enough to hold a capability would leave extra space that would be wasted in the vast majority of uses.

Robustness

While allowing `usize` to hold a valid capability would let simple pointer-integer-pointer round-trip casts work unmodified, it would also introduce inconsistent behaviour in many other cases. The tighter provenance model applied by CHERI invalidates capabilities derived only from integers, and also those produced by many arithmetic operations, including bitwise

logic. Given that bitwise logic is a common use case for integer-pointer casting, this would likely cause many unexpected capability faults. Ensuring instead that no `usize` can hold a valid capability guarantees that the compiler will always flag these cases, saving confusion and debugging time. This is traded against the cost of needing to make minor changes to simpler cases that might otherwise have worked unmodified, which we believe to be an acceptable sacrifice. Making this choice negates the major advantage of a 128-bit representation: being able to hold a valid capability.

Data compatibility

Under architectures currently supported by Rust, a cast from pointer to `usize` is expected to yield an integer containing the address being pointed to. Capabilities contain more information than this. Making `usize` a 64-bit integer containing only the address portion of the capability retains the expected behaviour.

3.2 Target specification

The Rust compiler has records of the size of various types for each platform it supports. This includes the size of pointers, which also decides the size of `usize`. To support Morello Purecap mode, the compiler needs more fine-grained information about the layout of pointers, and the size of `usize` needs to be decoupled from the in-memory size of pointers.

To do this, we have implemented *pointer width* and *pointer range*, where the compiler previously only had a single pointer size. Under mainstream architectures, these two values are all equal and redundant, but on Morello they are differentiated. *Pointer width* describes the in-memory size of pointers. Under Purecap, this will be the total size of a capability (128-bit), excluding the validity tag which is stored separately by the hardware. *Pointer range* describes the size of the address portion of pointers. Under Purecap, this will be the size of a plain pointer (64-bit), and the subset of a capability that contains the target address. Pointer range will also be the size of `usize`.

```

1 pub fn target() -> Target {
2     Target {
3         llvm_target: "aarch64-unknown-freebsd".to_string(),
4         pointer_range: 64,
5         pointer_width: 128,
6         data_layout: /* ... */,
7         arch: "aarch64".to_string(),
8         options: TargetOptions {
9             features: "+morello,+c64".to_string(),
10            llvm_abiname: "purecap".to_string(),
11            max_atomic_width: Some(128),
12            atomic_pointers_via_integers: false,
13            merge_functions: MergeFunctions::Disabled,
14            ..super::freebsd_base::opts()
15        },
16    }
17 }

```

compiler/rustc_target/src/spec/aarch64_unknown_freebsd_purecap.rs:3

■ **Figure 2** The Rust target options for the target triple `aarch64-unknown-freebsd-purecap`.

Information about a target is stored within the compiler using the structure shown in Figure 2. The target specification defines fundamental properties of the architecture and operating system. Morello inherits its base properties from the Aarch64 FreeBSD target, and then overrides some specifics. The new pointer entries can be seen on lines 4 and 5. Line 6 gives the standard data layout string describing the Morello architecture to LLVM, for brevity we do not expand on the details of this. For Purecap, this uses an extension specific to Morello LLVM [20] to specify that pointers be stored in address space 200, meaning that they should be represented using capabilities. Lines 9 and 10 specify the Purecap ABI, and are required to enable relevant features in Morello LLVM. Lines 12 and 13 disable some optimisations that are not yet compatible with Morello.

3.3 Constant evaluation

An unexpected source of problems for our changes to the compiler was Rust’s constant evaluation feature. Constant evaluation allows a subset of Rust expressions to be interpreted during compilation, as is demonstrated by the snippet below.

```

1  const MAGIC: u32 = long_multiply(3, 5)*7;
2  const fn long_multiply(a: u32, b: u32) -> u32 {
3      let mut a_shifted = a;
4      let mut b_shifted = b;
5      let mut result = 0;
6      while a_shifted != 0 {
7          if a_shifted & 1 == 1 {
8              result |= b_shifted;
9          }
10         a_shifted >>= 1;
11         b_shifted <<= 1;
12     }
13     result
14 }
```

This snippet contains only **consts**, and the compiler will fully evaluate the value of **MAGIC** at compile time. `long_multiply(3,5)` will be evaluated to 15, and then **MAGIC** will be evaluated to 105. The constant evaluator has an internal representation of memory so that it can run constant code even when it contains mutable values, as on lines 3, 4, and 5 of this example. Constant evaluation uses the same data layout as the rest of the compiler, and the subset of the language allowed includes support for pointers. Capabilities add extra non-address components to pointers, so constant evaluation must be modified to take these into account. While it might be possible to enforce the full set of capability rules during interpretation, we currently believe that Rust’s semantics already enforces them. For the time being we simply leave the unused space uninitialised, but the changes needed to the interpreter are still wide-reaching.

Values during interpretation can be represented either as large contiguous allocations, or single values represented directly⁷. Memory allocations are represented as arrays of data bytes, with auxiliary information about which bytes have been initialised. The compiler relies on type information to describe the structure of the data contained in the allocation.

⁷ Single values are handled separately as a performance optimisation.

39:12 Rust for Morello: Always-On Memory Safety, Even in Unsafe Code

Memory allocations themselves remain unchanged, the unused metadata bytes of capabilities are left uninitialised. References to subsets of memory allocations are passed around inside the compiler as the `AllocRange` type, so this must be extended to include width and range information. Without the extra information, the compiler does not know which bytes will be uninitialised metadata when operating on a referenced value. The extra information is also needed to allow conversion to types representing numeric values, which will need somewhere to inherit width and range information from. The modification to `AllocRange` is shown below.

```
pub struct AllocRange {
    pub start: Size,
    // Replacing: pub size: Size,
    pub range: Option<Size>,
    pub width: Size,
}
```

compiler/rustc_middle/src/mir/interpret/allocation.rs:75

Single numeric values are passed around as the types `Scalar` and `ScalarInt`. Because pointers are in some cases stored using these types, they must also carry width and range information. These changes are shown below.

```
pub enum Scalar<Tag = AllocId> {
    Int(ScalarInt),
    // Replacing: Ptr(Pointer<Tag>, u8),
    Ptr(Pointer<Tag>, u8, u8),
}
```

compiler/rustc_middle/src/mir/interpret/value.rs:124

```
pub struct ScalarInt {
    data: u128,
    // Replacing: size: u8,
    range: u8,
    width: u8,
}
```

compiler/rustc_middle/src/ty/consts/int.rs:122

The changes to the compiler to propagate and update the extra information on these three types are fairly simple, but very widespread, including changes to object layout, constant evaluation, and vtable construction.

3.4 Pointer code generation

Code generation for atomic pointers makes some unsound assumptions about pointers for the Morello platform. To work around the limitations of pointer operations on some targets, Rust generates code which casts the pointer to an integer – this is not permissible on Morello and yields capability faults at runtime. Thankfully, the fix here is simple: we have added an option to the target settings to disable this cast on the Morello target, which is shown in line 12 of Figure 2. The new option is then checked in the code generation pass, as shown below. On Morello, which supports 128-bit atomics, this avoids down-casting pointers to a pair of `isize` (64-bit) integers.

```
// Replacing: if ty.is_unsafe_ptr() {
if ty.is_unsafe_ptr() && bx.target_spec().atomic_pointers_via_integers {
    let ptr_llty = bx.type_ptr_to(bx.type_isize());
    ptr = bx.pointerCast(ptr, ptr_llty);
    val = bx.ptrtoint(val, bx.type_isize());
}
}
compiler/rustc_codegen_ssa/src/mir/intrinsic.rs:471
```

3.5 Tweaks to LLVM

The port of LLVM for Morello [20] is mature, and we have encountered few bugs.

Some LLVM optimisations currently cause incorrect code generation when compiling Rust for Morello, so we have disabled them until they can be debugged. The optimisations currently disabled are function merging, visible in line 13 of Figure 2; and the Scalar Replacement Of Aggregates optimisation, which requires minor changes inside LLVM.

We also encountered an edge case in LLVM code generation that caused 6 byte structures to be emitted as 96 bit integers, which then triggered a miscompilation. This ultimately lead to spurious capability faults during execution of affected parts of programs. While the underlying bug has now been fixed in upstream Morello LLVM [6], we found it could be worked around by padding affected structures to larger sizes.

The remaining adjustments that were needed were to the Rust standard library.

3.6 MPSC

Rust includes a large suite of tests for the compiler and libraries. Running the standard library tests on Morello has been our primary means of detecting code generation bugs and standard library compatibility issues. We had initially anticipated that the standard library would need many changes, given its size, low level nature, and need for performance. In actuality, we have so far only needed to make minor changes, and the modifications to the MPSC component are by far the most involved.

The Multi-Producer Single Consumer primitive in Rust (MPSC) is part of the standard library’s concurrency module, `std::sync`. It provides communication channel types that can pass objects between threads. The implementation of MPSC demonstrates exactly the sort of problem one might expect to see on Morello: pointers are passed between threads, but converted to integers and back as part of the trip. The same storage is also used to hold non-pointer signalling values. The problem stems from the code below which directly converts a `usize` into a pointer type (the type of `inner`).

This isn’t compatible with our changes to the compiler because the `usize` used by MPSC can no longer be used to carry a valid pointer, and the `usize` type is no longer the same size as a pointer. This causes casting to fail during compilation, but would still cause run time errors if it *did* compile. In CHERI C, one might use `uintptr_t`, but as no equivalent type is currently defined in Rust we have simply replaced the integer types with pointers. MPSC has no need to perform complex arithmetic or any other integer-specific operations, so this doesn’t create any problems. The signalling values can simply be cast into pointers before use, and because they should never be dereferenced it doesn’t matter that the resulting capabilities are invalid.

```
#[inline]
pub unsafe fn cast_to_ptr(self) -> *mut () {
```

39:14 Rust for Morello: Always-On Memory Safety, Even in Unsafe Code

```
// Replacing: pub unsafe fn cast_to_usize(self) -> usize {
    mem::transmute(self.inner)
}

#[inline]
pub unsafe fn cast_from_ptr(signal_ptr: *mut ()) -> SignalToken {
// Replacing: pub unsafe fn cast_from_usize(signal_ptr: usize) -> SignalToken {
    SignalToken { inner: mem::transmute(signal_ptr) }
}

library/std/src/sync/mpsc/blocking.rs:53
```

It is interesting to note that since our changes to MPSC for the port to Morello, very similar changes have happened upstream. The upstream changes are the product of work on pointer provenance in the MIR interpreter project (MIRI) [17].

3.7 FFI types

Rust's standard library makes use of the C standard library via a wrapper called `libc`. All use of C APIs from Rust requires a wrapper or *Foreign Function Interface*. In testing on Morello we found a number of incompatible type definitions, where the original API expected a pointer or pointer sized value, and the Rust wrapper declared the value to be `usize`. This is easy to fix by simply replacing the types appropriately. An example is drawn below, where we have replaced integer types with Rust pointer types.

```
pub type off_t = i64;
pub type useconds_t = u32;
pub type blkcnt_t = i64;
pub type socklen_t = u32;
pub type sa_family_t = u8;
// Replacing: pub type pthread_t = ::uintptr_t;
pub type pthread_t = *mut PThread;
pub type nfds_t = ::c_uint;
pub type regoff_t = off_t;

#[allow(missing_copy_implementations)]
pub struct PThread { _opaque: [u8; 0] }

library/libc-0.2.93/src/unix/bsd/mod.rs:1
```

There are likely to be interfaces not covered by tests that will require further work, we have taken a conservative approach to all the changes we have made in the compiler making the minimal alterations to get correct compilation of the tests we have. A different approach might be sensible here, where all `ptr_t` types should natively be Rust pointer types for Morello targets. It is not clear how significant the knock-on effects of this change would be on external libraries which expect integer semantics for C pointer types.

```
pub type size_t = usize;
pub type ptrdiff_t = isize;
pub type intptr_t = isize;
// TODO: Perhaps on Morello this should be `* ::c_void`
pub type uintptr_t = usize;
pub type ssize_t = isize;
```

```
pub type pid_t = i32;
pub type uid_t = u32;
pub type gid_t = u32;
pub type in_addr_t = u32;
pub type in_port_t = u16;
// Replacing: pub type sighandler_t = ::size_t;
pub type sighandler_t = *const ();
pub type cc_t = ::c_uchar;

library/libc-0.2.93/src/unix/mod.rs:19
```

4 Performance analysis methodology

We outline our method for measuring Rust programs running on the prototype Morello platform.

4.1 Test hardware

We are using the Morello Prototype hardware [2]. The Morello CPU is a 7nm quad-core “Neoverse N1” based Armv8-A processor clocked at 2.5 GHz, connected to 16 GiB of 2933 MT/s DDR4 memory. The firmware is updated to Release 1.3 [2]. The prototype is packaged as an ATX-style motherboard in a standard ATX computer case.

4.2 Operating system

As described earlier, we use the port of FreeBSD to Morello, called CHERI BSD [34]. CHERI BSD is very stripped back, with minimal system utilities running for network connectivity and multi-user support, making it ideal as a benchmarking platform. The operating system was compiled with the CTSRD port of LLVM for Morello [20] using the `cheribuild.py` utility [9].

4.3 Disabling bounds checking

Rust provides automatic checking of array accesses, ensuring that subscripting will be in-bounds. This covers a subset of the bounds checking provided by Morello, which checks the bounds of all pointer accesses. To compare Morello hardware bounds checking to the software bounds checking emitted by `rustc`, we added a code generation flag to the compiler which disables software bounds checking: `-C drop_bounds_checks`. This has the effect of making all array accesses the same as Rust’s `unsafe fn slice::get_unsafe()`. Enabling this option on normal hardware makes the compiler unsound, while on Purecap Morello soundness is mostly, though not entirely, restored by hardware bounds checking. Bounds checks on Morello are precise up to 4 KiB blocks of memory, becoming imprecise beyond that as a result of the floating point representation used to store bounds information. For the sake of the performance comparisons made in this paper, we think that this approach is reasonable to give some indication of the cost of bounds checks relative to the cost of CHERI extensions.

This method of disabling software bounds checking is similar to previous work on measuring the runtime cost of Rust’s safety checks by Zhang et al., although not identical [42].

39:16 Rust for Morello: Always-On Memory Safety, Even in Unsafe Code

```
fn bounds_check(
    &mut self,
    block: BasicBlock,
    slice: PlaceBuilder<'tcx>,
    index: Local,
    expr_span: Span,
    source_info: SourceInfo,
) -> BasicBlock {
    // We do return an un-modified access when the -C drop_bounds_check
    // flag is enabled
    let gcx = *self.tcx;
    if gcx.sess.opts.cg.drop_bounds_checks {
        return block
    }

    // Otherwise, generate MIR code to check the bounds of an access
    /* ... */
}
```

compiler/rustc_mir_build/src/build/expr/as_place.rs:666

Our modification adds an early return to the code generation of bounds checking assertions, which would normally be inserted when array subscripting. In Zhang's work they modify a later stage of code generation to disable the lowering of these assertions to LLVM IR, as well as assertions that check for integer-overflow. We ran benchmarks with both approaches and found similar results, but we present the results for tests with the modification listed above only, as we are not investigating the cost of integer-overflow checks on Morello.

4.4 cargo bench

We ported the standard Rust benchmarking infrastructure to Morello. Programs are cross-compiled on a normal Apple M1 or x86_64 system. Rust's infrastructure includes a remote test harness which sends binaries to a remote system under test using network sockets. We built a port of the receiving part of this software, called `remote-test-server` in C, which runs on the Morello prototype. We can then use the standard `cargo bench` command to orchestrate building and running tests, which runs benchmarks repeatedly and reports a time per iteration in nanoseconds, and the variance between runs in \pm nanoseconds. For each benchmark in the suite we produce four results to complete a comparison matrix by varying two parameters. The first parameter is the hardware mode which can be one of two values: Hybrid or Purecap, as described in §2.3.1, this is varied using the `--target` option and can be either `aarch64-unknown-freebsd` for Hybrid, or `aarch64-unknown-freebsd-purecap` for Purecap mode. The second parameter is the bounds checking mode, using our `-C drop_bounds_checks` compiler flag, which can be either: bounds checking disabled (Rust_{DBC}), or enabled (Rust).

For example the results below are produced by a test from the crate `hashbrown`. Time is in nanoseconds (ns).

<code>hashbrown-0.11.2/clone_from_large</code>				
	Rust		Rust _{DBC}	
	Time/iter	±	Time/iter	±
Purecap	15,779	8	15,818	59
Hybrid	15,557	53	15,601	16

Before each run in the test matrix, Rust and the Rust standard library is also recompiled with flags to enable/disable software bounds checking appropriately for Rust/Rust_{DBC}.

From this data, we calculate Relative Error (R_E) for each test using the formula below, and count the variance into several bins in Table 1.

$$R_E = \frac{\pm \text{time/iter (ns)}}{\text{Mean benchmark time/iter (ns)}}$$

For example, the R_E of Hybrid with bounds checking for `hashbrown-0.11.2/clone_from_large` is 0.34%. The benchmarks with high relative error are tests which are very fast running, and are limited by the measurement precision of `cargo bench` in nanoseconds.

4.5 Line counts

We include a count of the number of lines of code with the table of benchmarked projects in Appendix A. This is intended as a rough guide to the scale of the benchmarks. The table also includes a count of lines of `unsafe` code, as an indication of how much Morello’s safety guarantees might add to the soundness of the specimen code. Both counts are gathered using the `cargo count` tool [19]. In total across the 19 projects, there are 108k lines of Rust source of which 1041 are unsafe.

The lines of code count gives the number of non-blank, non-comment lines of Rust source code in the repository of each project, for every project we use a benchmark suite from. The lines of `unsafe` count gives the number of non-blank, non-comment lines inside `unsafe` blocks and `unsafe` functions.

It should be noted that both of these measures are approximate and meant as a guide only. The number of lines of code is given for the whole of each project repository, and will include the library code under test, the benchmark suite, and any additional tests and Rust build scripts present, but will exclude any code in dependencies (of which each project generally has several). The number of lines of `unsafe` is particularly approximate, for two reasons. Firstly, the effects of `unsafe` are not well quantified by counting lines of code; the impact of a single line in a frequently and widely called function may be much higher than a large `unsafe` block that executes only once during the lifetime of the program. Further, a single line of `unsafe` can call an arbitrary amount of external unsafe code through the foreign function interface. Secondly, there are some known edge cases in the counting tool that may cause slightly inaccurate counts [19].

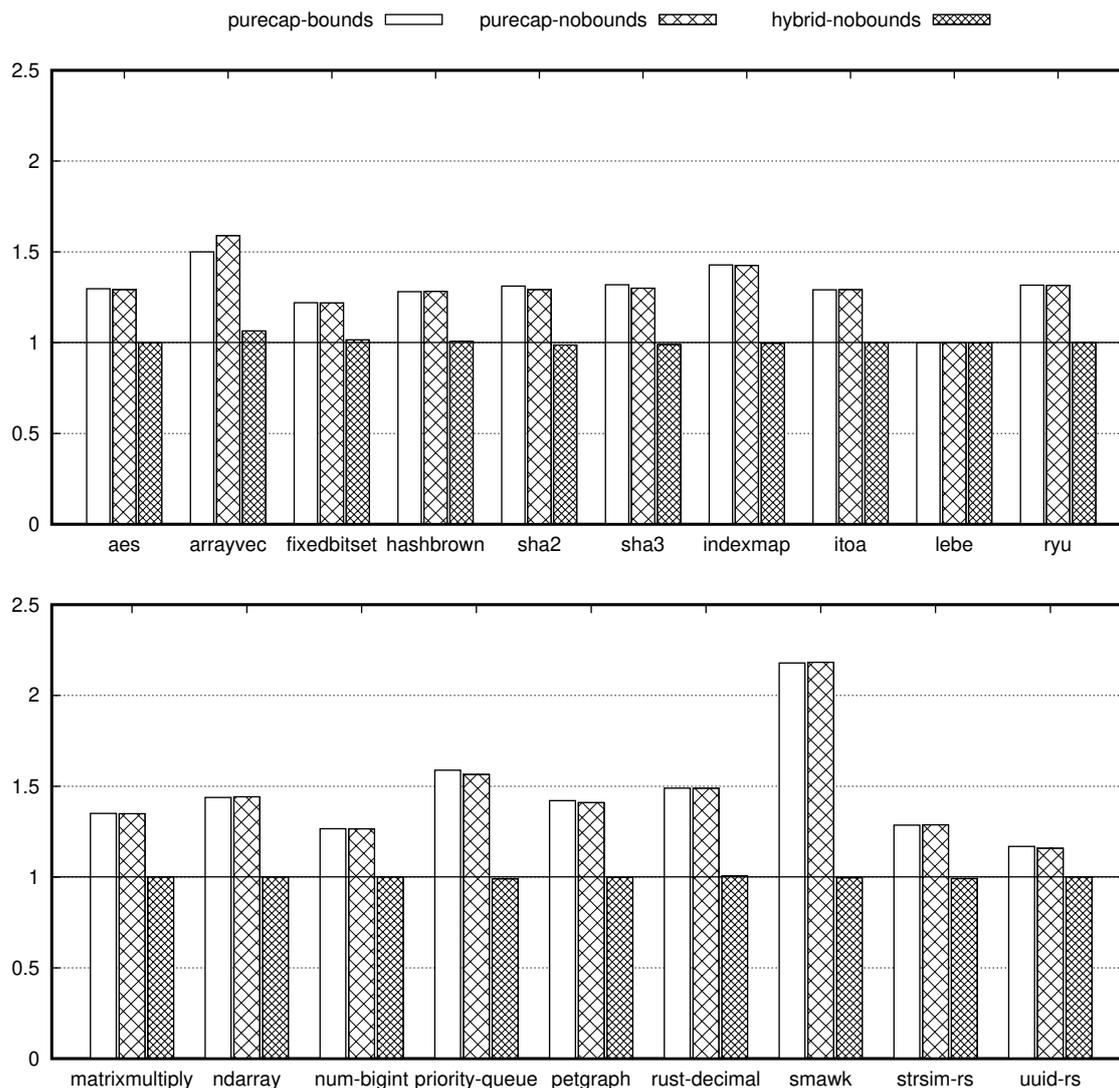
4.6 Test suites

A full list of test suites is available in Appendix A. These were picked for having low-level implementations of fast data structures, and having a small set of external dependencies. The suites cover arithmetic, array computations, cryptography, data-structures, Fourier analysis, hashing, and graph algorithms. Each test was taken directly from the standard Rust package repository (<https://crates.io>) and checked out to a version which is compatible with our Rust compiler. Cargo will satisfy dependencies with the latest available package which meets the requirements, but frustratingly that doesn’t include the Rust edition, so

39:18 Rust for Morello: Always-On Memory Safety, Even in Unsafe Code

for several packages we had to pin dependencies at older compatible versions too. Our test script automatically collects repositories from Git, and applies the minor patches to the `Config.toml` to pin dependencies where necessary. In total there is a little over 108k lines of Rust source code in these suites, and plenty more in the dependencies. There are 870 individual benchmarks which are run for each of the four modes described above.

5 Results



■ **Figure 3** Performance analysis of Rust programs in each of the modes described in §4, normalised to Hybrid-mode.

We present the aggregate execution times of each mode of Rust on Morello in the table below. They are normalised to Hybrid mode with software bounds checking enabled. Lower numbers indicate higher average performance.

■ **Table 1** Count of benchmark relative error into categories.

Relative error	Count
0 – 1%	3300
1 – 5%	142
≥ 5%	38

	Rust	Rust _{DBC}
Purecap	1.39	1.38
Hybrid	1.00	0.99

Individual benchmark suite results are shown in Figure 3. Again, for each of our benchmark results, we normalise against Hybrid-mode Rust with the Rust compiler’s software bounds checks enabled. We then take the geometric mean of each mode to aggregate the normalised performance change of the benchmarks within a benchmark suite, *i.e.* **hashbrown**, which internally contains 41 benchmarks. Bars above 1.0 are slower than Hybrid Rust with software bounds checks enabled, bars below are faster. These numbers represent the change in performance versus Rust on a modern Aarch64 machine today. Rust on Purecap Morello is approximately 39% slower than Rust on the equivalent Aarch64 machine.

In this section we will contrast the cost of bounds checking on the software and hardware, and consider the types of workload whose performance is affected the most by hardware bounds checking.

5.1 The cost of software bounds checking

Toggling software bounds checks (Rust_{DBC} vs. Rust) makes minimal difference in the performance we observe in our benchmark suites. The performance is similar because compiler optimisations remove most unnecessary bounds checks from code before runtime, and the performance cost of what remains is extremely low. Note that we are dropping bounds checks as aggressively as possible, even dropping them where Morello would provide imprecise bounds (as discussed in §4.3) – so we have here an upper-bound on the performance gain achievable in a sound implementation of Rust. We therefore conclude that re-engineering the Rust compiler to drop bounds checks on Morello is without merit.

5.2 The cost of hardware bounds checking

By-and-large the cost of hardware bounds checking is very significant, with only **lebe** showing a negligible difference. The performance hit, though large, is still relatively small compared to other techniques for always-on bounds checking for arbitrary binaries – such as running a program under **valgrind** or **purify** [13, 10]. For many applications this 39% cost for running on Morello is acceptable, and will likely only improve with any future hardware designs. This does open an interesting point for future work: off-loading as much bounds-checking to software, ideally statically enforced, and maintaining hardware bounds-checks where these guarantees cannot be enforced by software alone, could provide performance *and* strong safety guarantees.

5.3 Benchmarks

We present the results aggregated by benchmark suite in Figure 3. This shows consistent slowdowns in Purecap modes, but reveals some variation depending on the workload. The worst slowdowns are in `arrayvec` and `smawk` which perform a substantial number of array accesses, maximally exercising the capability protections afforded by Purecap Morello. `lebe`, which does endianness conversions for integers, is unaffected by differences under Purecap Morello, or whether bounds checks are enabled or not. The story is similar for other arithmetic heavy crates, like `fixedbitset`; code that leans less heavily on memory access suffers the lowest performance hit when running on Purecap Morello.

5.4 Validity of results

We address potential concerns for the validity of our results, and how we believe we have mitigated these.

5.4.1 Prototype hardware

The Morello prototype is just that, a prototype. There are known issues with real-world performance characteristics of the first prototype when operating in Purecap mode, which are reflected in our performance analysis. When the details of these limitations are released, or when a future revision of the hardware is released (to which ARM have *not* committed themselves), re-running our performance analysis to see if the overhead of capabilities can be brought in-line with normal performance of Aarch64 would be of significant interest. The comparison to Hybrid-bounds in this section gives the clearest picture of the behaviour of a comparable Aarch64 machine with no hardware bounds checking: in Hybrid mode the Morello prototype allows regular loads and stores using 64-bit pointers. That being said, the implementation is still customised compared to normal Aarch64 and micro-architectural quirks might be present which mean this comparison is invalid compared to “wild” Aarch64 implementations. Table 1 shows that our measurements are precise and run-to-run variance is very small in the vast majority of benchmarks.

5.4.2 Choice of benchmark suite

We have used benchmark suites included with large popular projects from the Rust package repository `crates.io`, spanning a range of different applications: hashing and cryptography (arithmetic heavy); tree and graph like data structures (indirection heavy); and big integer and matrix operations (array heavy). This is different to previous approaches to measuring Rust runtime performance, notably in recent work by Zhang et al., where synthetic micro-benchmarks were used [42]. We have chosen to use benchmarks that test real world Rust code as this is more likely to give a representative picture of how the cost of bounds checking will be felt in regular Rust programs. We found some benchmarks (for example, `RustFFT` and `itertools`) would compile for Morello, but not run correctly on Purecap modes because of assumptions about pointers which are invalid in Purecap mode.

6 Related Work

In this section we discuss the context for Morello and the necessary related work that enabled us to build the Rust for Morello compiler.

6.1 Rust and type-safe systems programming

Rust itself has been a subject of significant academic research. On the topic of formally specifying Rust, notable work includes Rust Belt [16] and Oxide [38]. We are interested in the formal treatment of Rust, especially in the context of the formal specification of Morello, but for this paper we consider this to be future work.

Additionally, there has been a history of other attempts at designing safe systems programming languages which, like Rust, use some notion of lifetimes to manage memory allocation and prevent memory errors. Cyclone [14] is one such language, which uses type-level *regions* among other type-system machinery to ensure safety while still admitting low-level systems code. The specifics of preventing memory errors in safe systems programming languages is discussed more in §6.3.

6.2 Prior work porting Rust to CHERI

There is existing work that explores extending the Rust compiler to target CHERI MIPS hardware. Nicholas Sim’s MSc thesis [32] describes the initial steps of targeting CHERI in the Rust compiler. Crucially, Sim chose the 128-bit representation of `usize`, whereas we opted for 64-bit `usize` (discussed in §3.1). This has cascading effects on the rest of the compiler and leads to various issues.

A major concern of Sim was divergence from upstream, which we agree is a real consideration. The wording of the documentation does not make it totally clear how large `usize` should be on Morello, it states [29]:

`usize`

The pointer-sized unsigned integer type.

The size of this primitive is how many bytes it takes to reference any location in memory. For example, on a 32 bit target, this is 4 bytes and on a 64 bit target, this is 8 bytes.

There is clearly room for argument one way or the other. We are of the opinion that 64-bit `usize` on Morello is compatible with the spirit of this definition, Morello is a 64-bit platform with 64-bit addresses, so in order to range over all memory addresses it suffices for `usize` to be 64-bit. Conversely, supporting Sim’s initial choice, to reference a location in memory on Morello it is necessary to use a capability which is 128-bit.

Sim also reported a number of technical issues resulting from 128-bit `usize`, including performance problems, and the need for inserting integer-truncate and integer-extend operations when calling LLVM intrinsics like `memcpy` and `inttoptr`. We have not had to contend with these issues in our implementation, and agree with Sim’s assessment that 64-bit `usize` is preferable for this reason.

6.3 Bounds checking

There has been significant prior research on enforcing memory safety by preventing out-of-bounds accesses, either by statically proving accesses will be in-bounds, or by augmenting programs with dynamic bounds checking (or a combination of both).

A key result on statically eliminating bounds violations comes from Xi and Pfenning, who demonstrate that, with a *dependent type system*, array accesses may be accompanied by *proofs* that the computed index is within the array bounds [40]. This gives a compile-time guarantee that no bounds violations will occur on any array access, and thus safety can be maintained without the need for any dynamic checks. This was done in the context of a high-level, ML-like language, so there was no need to deal with gnarly C-like pointer arithmetic.

Prior work also explores static elimination of *some* bounds checking as a compiler optimisation. An optimising compiler or just-in-time compiler may seek to reduce the number of required bounds checks in a program by proving that some subset of accesses will always succeed. Early work demonstrated how this may be done with dataflow analysis [12]. This has been of particular interest to implementors working with the Java programming language, as the specification states that out-of-bounds array accesses must be caught at runtime, but bounds checks are not expressible in standard Java bytecode. To address this problem, lightweight techniques for eliminating some bounds checks at runtime within a Just-in-Time compiler have been developed [7].

In addition to minimising or eliminating bounds checks when compiling high-level languages like Java, there has been interest in enforcing memory safety in existing C and C++ programs. Doing automatic bounds checking in C or C++ is difficult because of the need to track, at run-time, what object each pointer value is intended to point to [15]. Unlike languages like Java, C and C++ allow programmers to do *pointer arithmetic* in order to compute (for example) an index into the middle of an array. These pointers into the interiors of objects may be written to a data structure or passed to a function within the application; in order to check the latter dereferencing of such pointers, it is necessary to keep track of the *intended referent* of the pointer as the pointer value flows through the program. There have been several approaches to solving this problem and preventing or catching out-of-bounds errors in existing systems code.

One heavyweight approach is *binary instrumentation*. Binary instrumentation tools like `purify` and `valgrind` are able to arbitrarily add metadata to pointers and detect a wide range of memory referencing errors [13]. Of course, as these tools are designed for debugging, they impose a significant runtime overhead that makes them impractical for use in production software. Additionally, `purify` can end up missing some memory safety violations if pointer arithmetic happens to yield a pointer to a different but still valid object.

Another approach involves changing pointer representations. Systems like SafeC [5] and Cyclone [14] use an extended pointer representation (“fat pointers”) to record information about the intended referent. These fat pointers work similarly to Morello’s capabilities, but their enforcement mechanisms are implemented in software rather than in hardware. SafeC and Cyclone allow for dynamic checking, but also produce code that is incompatible with external, unchecked code. A different approach, which maintains backwards compatibility with legacy C code, was proposed by Jones and Kelly: store the address ranges of live objects and ensure that pointer arithmetic never crosses out of the one object and into another valid object [15]. In this approach, address ranges are stored in a global table, and the table is referenced before every pointer arithmetic operation. Unsurprisingly, this introduces a large amount of overhead at run-time – over 5x overhead on many programs and, in subsequent work that extended this approach to cover a larger class of C programs, over 11x overhead [30]. Later work by Dhurjati and Adve demonstrated that the overhead of backwards-compatible array bounds checks in C could be drastically reduced by exploiting a fine-grain partitioning of memory called Automatic Pool Allocation [10].

7 Future work

The relationship between Rust and Morello seems clear, both aim to provide a programmer with guarantees that memory safety violations cannot happen in their programs. Rust achieves this through linear types, and runtime bounds checks, and Morello achieves this

through hardware pointer provenance and hardware bounds checking. It would be interesting to show a formal relationship between the semantics of Rust and the guarantees provided by Morello.

With the Morello prototype running in Hybrid mode it is possible to mix the use of capability instructions, where the hardware enforces bounds checking at run time, with normal load/store instructions. Through static analysis it might be possible to find the pointers in Rust which require runtime bounds checks and to promote those to capabilities on a per-pointer basis: thereby pushing bounds checks into hardware, rather than by emitting additional code. This could also be used to martial code in/out of unsafe components or through the FFI. Using capabilities to compartmentalise each `unsafe` block and each FFI call could bring new safety guarantees to Rust code which interacts with components that could introduce memory safety violations.

If ARM later commits to incorporating CHERI extensions into the ARM ISA at large, then incorporating the changes from our prototype Rust compiler into upstream Rust would be a natural extension of this work.

8 Conclusions

Morello provides a costly, but not impractical, means for achieving always-on memory safety. Like Rust, Morello has been built with the benefit of hindsight: memory safety is the most significant problem for building bug-free code. We have found that the overhead for Morello in this first prototype is around 39%, and elimination of Rust's bounds checks might yield a 1% speed up.

There is a two-way benefit for a programmer using Rust on Morello. When using safe Rust, a programmer knows that for all their safe code they cannot get a CHERI protection error which would cause their code to fault at runtime – unlike if they were to program in C. When using `unsafe` Rust, the programmer knows that if things do go terribly wrong, the Morello platform will protect them from memory errors which could cause security vulnerabilities.

The Rust for Morello compiler presented in this paper is fully featured. We have demonstrated the compiler works for a significant chunk of wild Rust code, without modification. In-all, the code we've compiled and run for Morello is around 108k lines of Rust, all from the Rust Crates repository, and all without modification to the Rust code.

This compiler forms the groundwork for future research into safe systems programming on hardware designed from the ground up to provide memory safety.

References

- 1 Arm. *Arm® Architecture Reference Manual Supplement Morello for A-profile Architecture*. Arm, 2020.
- 2 ARM. Morello project – release notes, January 2022. last accessed: July 25, 2022. URL: <https://git.morello-project.org/morello/docs/-/blob/morello/mainline/release-notes.rst>.
- 3 ARM and contributors. The android/morello release, April 2022. last accessed: September 28, 2022. URL: <https://git.morello-project.org/morello/docs/-/blob/morello/mainline/android-readme.rst>.
- 4 ARM and contributors. Morello project – linux, August 2022. last accessed: September 28, 2022. URL: <https://git.morello-project.org/morello/kernel/linux>.

- 5 Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 290–301, New York, NY, USA, 1994. Association for Computing Machinery. doi:10.1145/178243.178446.
- 6 Silviu Baranga. Don't replace a 96-bit memcpy with a capability load/store, September 2022. URL: https://git.morello-project.org/morello/llvm-project/-/merge_requests/205.
- 7 Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. Abcd: Eliminating array bounds checks on demand. *SIGPLAN Not.*, 35(5):321–333, May 2000. doi:10.1145/358438.349342.
- 8 Common Weakness Enumeration. 2022 CWE Top 25 Most Dangerous Software Weaknesses. Technical report, MITRE, August 2022. URL: https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html.
- 9 CTSRD CHERI. cheribuild, 2022. last accessed: July 25, 2022. URL: <https://github.com/CTSRD-CHERI/cheribuild>.
- 10 Dinakar Dhurjati and Vikram Adve. Backwards-Compatible Array Bounds Checking for C with Very Low Overhead. In *Proceedings of the 2006 International Conference on Software Engineering (ICSE'06)*, Shanghai, China, May 2006. URL: <http://llvm.org/pubs/2006-05-24-SAFECODE-BoundsCheck.html>.
- 11 Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 282–293, New York, NY, USA, 2002. Association for Computing Machinery. doi:10.1145/512529.512563.
- 12 Rajiv Gupta. Optimizing array bound checks using flow analysis. *ACM Lett. Program. Lang. Syst.*, 2(1–4):135–150, March 1993. doi:10.1145/176454.176507.
- 13 Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *In Proc. of the Winter 1992 USENIX Conference*, pages 125–138, 1991.
- 14 Michael Hicks, Greg Morrisett, Dan Grossman, and Trevor Jim. Experience with safe manual memory-management in cyclone. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 73–84, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/1029873.1029883.
- 15 Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Proceedings of the Third International Workshop on Automated Debugging, AADEBUG 1997*, 1997.
- 16 Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), 2017. doi:10.1145/3158154.
- 17 Ben Kimock. Remove ptr-int transmute in std::sync::mpsc, April 2022. URL: <https://github.com/rust-lang/rust/commit/dec73f5>.
- 18 Steve Klabnik, Carol Nichols, et al. *The Rust Programming Language*. The Rust Project Developers, 2021. URL: <https://doc.rust-lang.org/1.55.0/book/>.
- 19 Kevin Knapp. cargo-count, November 2017. URL: <https://github.com/kbknapp/cargo-count>.
- 20 LLVM Project and CTSRD CHERI. CTSRD llvm-project, 2022. last accessed: July 25, 2022. URL: <https://github.com/CTSRD-CHERI/llvm-project>.
- 21 Nicholas D. Matsakis and Felix S. Klock. The rust language. *Ada Lett.*, 34(3):103–104, October 2014. doi:10.1145/2692956.2663188.
- 22 Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. Exploring c semantics and pointer provenance. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290380.
- 23 Matt Miller. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. last accessed: July 25, 2022. URL: https://github.com/microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf.

- 24 Miguel Ojeda. [PATCH 00/13] [RFC] Rust support, April 2021. URL: <https://lore.kernel.org/lkml/20210414184604.23473-1-ojeda@kernel.org/>.
- 25 Rust project contributors. [Pre-RFC] usize is not size_t, September 2021. URL: <https://internals.rust-lang.org/t/pre-rfc-usize-is-not-size-t/15369>.
- 26 Rust project contributors. The Rust Standard Library - Primitive Type usize, 2021. URL: <https://doc.rust-lang.org/1.55.0/std/primitive.usize.html>.
- 27 Rust project developers. Rust 0.1, 2012. URL: <https://github.com/rust-lang/rust/releases/tag/0.1>.
- 28 Rust project developers. *The Rust Reference*. The Rust Project Developers, 2021. URL: <https://doc.rust-lang.org/1.55.0/reference/>.
- 29 Rust project developers. *usize - Rust*. The Rust Project Developers, 2021. URL: <https://doc.rust-lang.org/std/primitive.usize.html>.
- 30 Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *In Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pages 159–169, 2004.
- 31 Nicholas Sim. Support index size != pointer width. <https://github.com/rust-lang/rust/issues/65473>, October 2019. last accessed: November 28, 2022.
- 32 Nicholas Sim. Strengthening memory safety in Rust: exploring CHERI capabilities for a safe language. Master’s thesis, University of Cambridge, August 2020. URL: <https://nw0.github.io/cheri-rust.pdf>.
- 33 The Chromium Projects. Memory Safety. <https://www.chromium.org/Home/chromium-security/memory-safety/>. last accessed: July 25, 2022. URL: <https://www.chromium.org/Home/chromium-security/memory-safety/>.
- 34 The FreeBSD Project and CTSRD CHERI. CTSRD-CHERI cheribsd, 2022. last accessed: March 4, 2021. URL: <https://github.com/CTSRD-CHERI/cheribsd>.
- 35 Aaron Turon. Rust blog: Abstraction without overhead: Traits in rust, May 2015. URL: <https://blog.rust-lang.org/2015/05/11/traits.html>.
- 36 Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Peter G. Neumann. An Introduction to CHERI. Technical Report UCAM-CL-TR-941, University of Cambridge, Computer Laboratory, September 2019. doi:10.48456/tr-941.
- 37 Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Marketos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8). Technical Report UCAM-CL-TR-951, University of Cambridge, Computer Laboratory, October 2020. doi:10.48456/tr-951.
- 38 Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. Oxide: The essence of Rust. *CoRR*, 2019. arXiv:1903.00982.
- 39 Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Marketos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. Cornucopia: Temporal Safety for CHERI Heaps. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 608–625, May 2020. doi:10.1109/SP40000.2020.00098.
- 40 Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation, PLDI ’98*, pages 249–257, New York, NY, USA, 1998. Association for Computing Machinery. doi:10.1145/277650.277732.

- 41 Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. Cherivoke: Characterising pointer revocation using cheri capabilities for temporal memory safety. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO '22*, pages 545–557, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3352460.3358288.
- 42 Yuchen Zhang, Yunhang Zhang, Georgios Portokalidis, and Jun Xu. Towards understanding the runtime performance of rust. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, ASE '22*, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3551349.3559494.

A Test suite

Benchmark	Version	Description	SLOC	Unsafe
<code>arrayvec</code>	0.7.2	A vector with fixed capacity, backed by an array.	2,059	26
<code>block-ciphers/aes</code>	0.7.2	Pure Rust implementation of the Advanced Encryption Standard.	4,218	25
<code>fixedbitset</code>	0.3.1	A simple bitset collection.	1,431	4
<code>hashbrown[†]</code>	0.11.2	Google's SwissTable hash map.	8,454	182
<code>hashes/sha2</code>	0.10.2	Pure Rust implementation of the SHA-2 hash function family.	1,086	2
<code>hashes/sha3</code>	0.10.1	SHA-3 (Keccak) hash function	320	0
<code>indexmap[†]</code>	1.0.0	A hash table with consistent order and fast iteration.	6,092	3
<code>itoa</code>	1.0.3	Fast integer primitive to string conversion.	324	5
<code>lebe</code>	0.5.0	Tiny, dead simple, high performance endianness conversions with a generic API.	527	40
<code>matrixmultiply</code>	0.3.2	General matrix multiplication for f32 and f64 matrices.	3,898	22
<code>ndarray</code>	0.15.6	An n-dimensional array for general elements and for numerics.	25,508	340
<code>num-bigint</code>	0.4.3	Big integer implementation for Rust.	12,541	0
<code>petgraph[†]</code>	0.6.0	Graph data structure library.	19,559	5
<code>priority-queue[†]</code>	1.3.1	A Priority Queue implemented as a heap with a function to efficiently change the priority of an item.	3472	68
<code>rust-decimal</code>	1.23.1	Decimal number implementation written in pure Rust suitable for financial and fixed-precision calculations.	11,469	0
<code>ryu</code>	1.0.12	Fast floating point to string conversion.	2,930	317
<code>smawk</code>	0.2.0	Functions for finding row-minima in a totally monotone matrix.	740	0
<code>strsim</code>	0.10.0	Implementations of string similarity metrics.	837	0
<code>uuid-rs</code>	1.3.0	A library to generate and parse UUIDs.	3,505	2
Total			108,970	1,041

Tests marked with [†]required patches to their Config.toml to pin compatible versions of dependencies.