Porting Rust to Morello

A safe software layer for a safe hardware layer

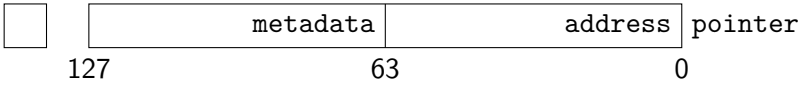
Sarah Harris, **Simon Cooksey**, Michael Vollmer, Mark Batty

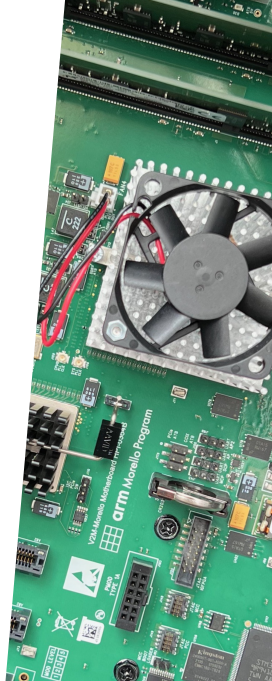
University of Kent

March 2023

Capability machine

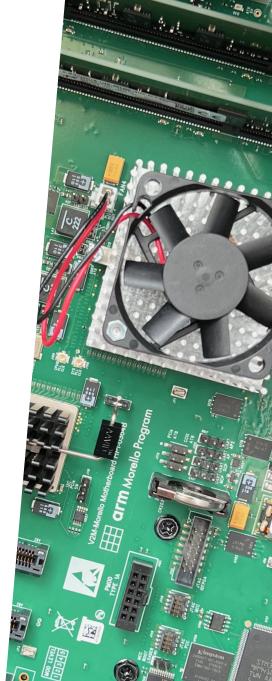
- Morello has hardware pointer provenance
- Pointers have an address (as usual), but also some metadata including a bounds and permissions flags.
- There is a transparent hardware managed validity bit which prevents pointer spoofing.



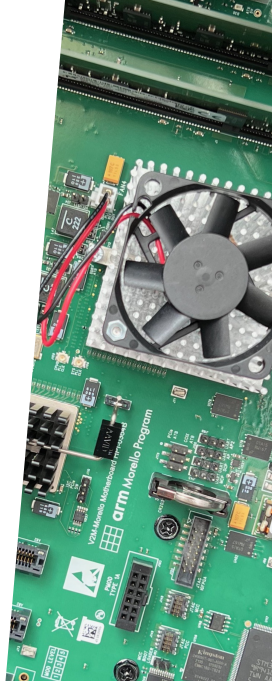


Morello prototype

- ARMv8a Neoverse N1 modified with CHERI extensions
- 2.4GHz quad core
- 16GiB of DDR4 memory, PCIe gen 3, USB, Ethernet, looks like a computer!

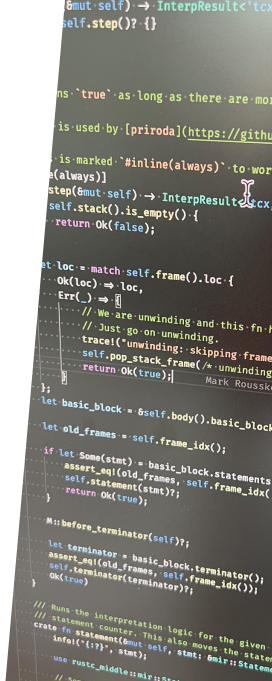


Morello prototype



Rust

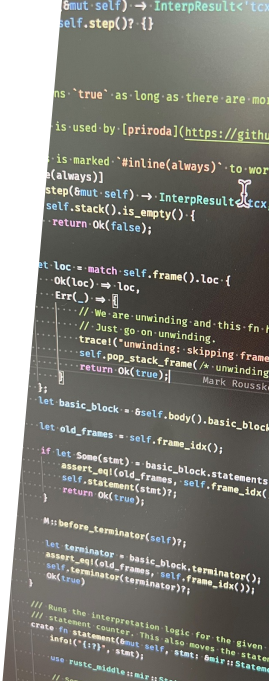
- Rust is designed to be a safe systems programming language.
- The compiler statically verifies that memory safety issues like use-after-free, and buffer overruns cannot happen.



Rust

- Rust is designed to be a safe systems programming language.
- The compiler statically verifies that memory safety issues like use-after-free, and buffer overruns cannot happen.

```
fn main() {
  let mut x : [u8; 8] = [0; 8];
  x[9] = 1;
}
```

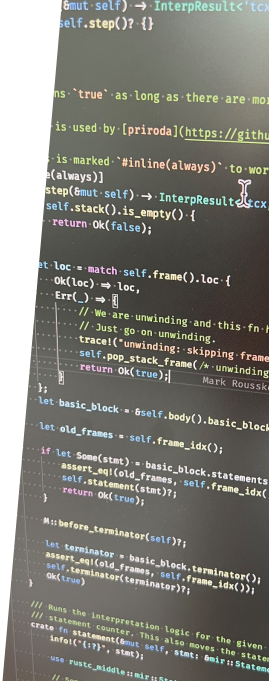


Rust

- Rust is designed to be a safe systems programming language.
- The compiler statically verifies that memory safety issues like use-after-free, and buffer overruns cannot happen.

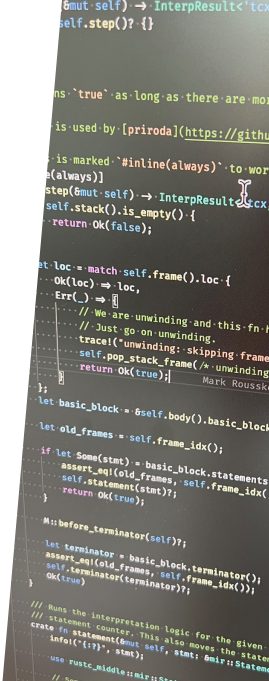
```
fn main() {
  let mut x : [u8; 8] = [0; 8];
  x[9] = 1;
}
```

```
$ rustc ./main.rs -o oob-compile
error: this operation will panic at runtime
--> src/main.rs:3:5
  |
3 |     x[9] = 1;
  |     ^^^^ index out of bounds: the length is 8 but the index is 9
  |
```



Rust

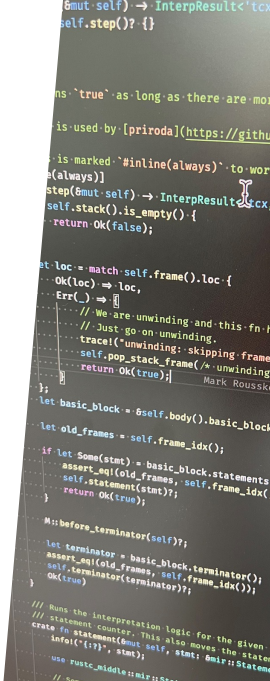
- Rust is designed to be a safe systems programming language.
- The compiler statically verifies that memory safety issues like use-after-free, and buffer overruns cannot happen.
- Rust is designed to be used in places where C/C++ is used.



Rust

- Rust is designed to be a safe systems programming language.
- The compiler statically verifies that memory safety issues like use-after-free, and buffer overruns cannot happen.
- Rust is designed to be used in places where C/C++ is used.
- Rust has an escape keyword **unsafe**.

```
fn main() {
    let mut x : [u8; 8] = [0; 8];
    unsafe {
        *x.get_unchecked_mut(7) = 1;
    }
}
```

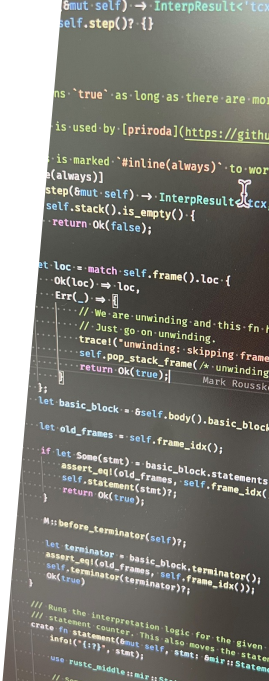


Rust

- Rust is designed to be a safe systems programming language.
- The compiler statically verifies that memory safety issues like use-after-free, and buffer overruns cannot happen.
- Rust is designed to be used in places where C/C++ is used.
- Rust has an escape keyword **unsafe**.

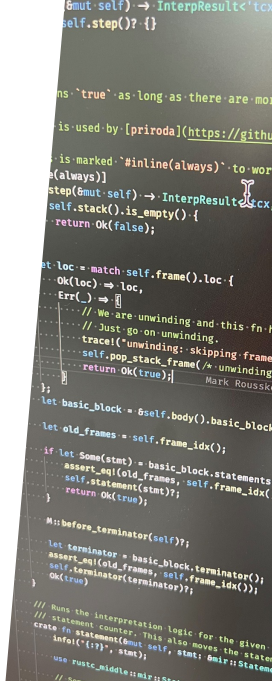
```
fn main() {
    let mut x : [u8; 8] = [0; 8];
    unsafe {
        *x.get_unchecked_mut(7) = 1;
    }
}
```

```
$ ./oob-runtime
Segmentation fault
```



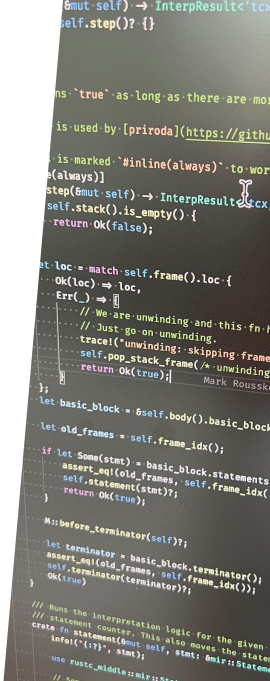
Why port Rust to Morello?

- The guarantees of capabilities complement the guarantees of Rust



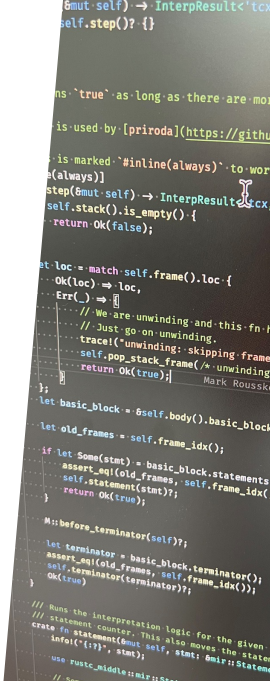
Why port Rust to Morello?

- The guarantees of capabilities complement the guarantees of Rust
- Rust provides compile-time guarantees for `safe` code



Why port Rust to Morello?

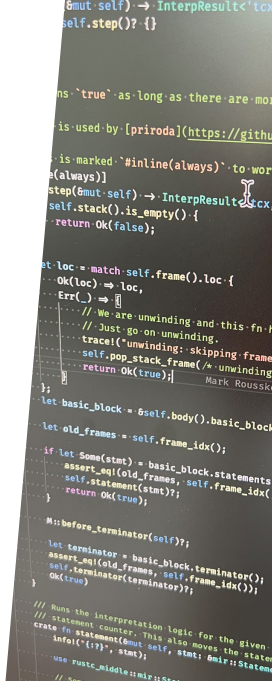
- The guarantees of capabilities complement the guarantees of Rust
- Rust provides compile-time guarantees for `safe` code
- Capabilities provide run-time guarantees for **`unsafe`** code



Why port Rust to Morello?

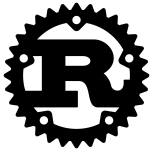
- The guarantees of capabilities complement the guarantees of Rust
- Rust provides compile-time guarantees for `safe` code
- Capabilities provide run-time guarantees for **`unsafe`** code

Digital Security by Design

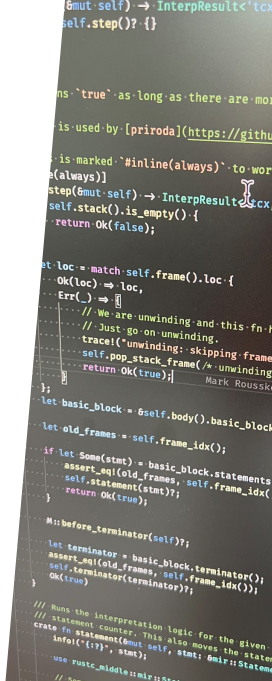


Why port Rust to Morello?

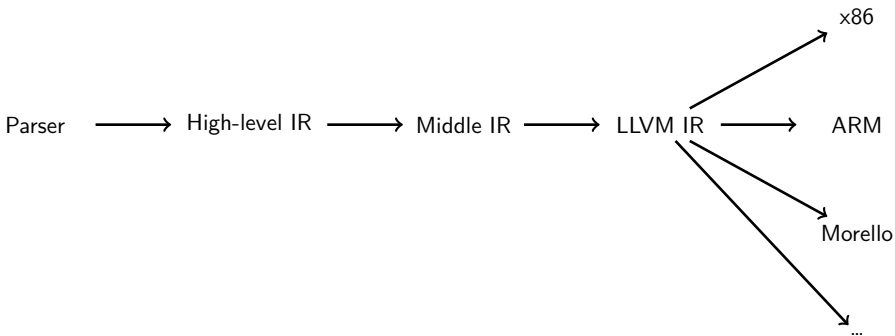
- The guarantees of capabilities complement the guarantees of Rust
- Rust provides compile-time guarantees for `safe` code
- Capabilities provide run-time guarantees for **`unsafe`** code



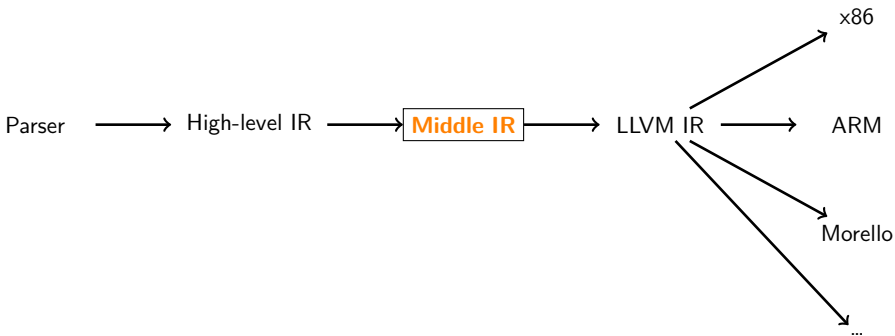
Digital Security by Design



The Rust Compiler



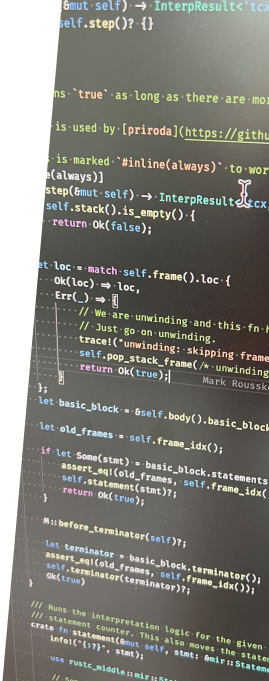
The Rust Compiler



Compiler changes — plumbing

The first task is hooking Rust up with Morello LLVM.

- We added a target, and set the appropriate options
- We hooked up Morello clang as the linker for the Rust compiler
- We extended the Rust target options to allow us to describe object layout differences...

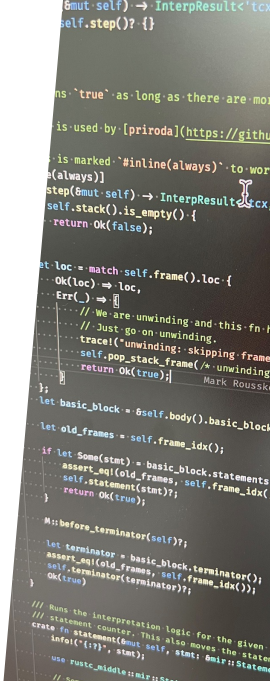


Compiler changes — object layout

Object layout differences, you say?

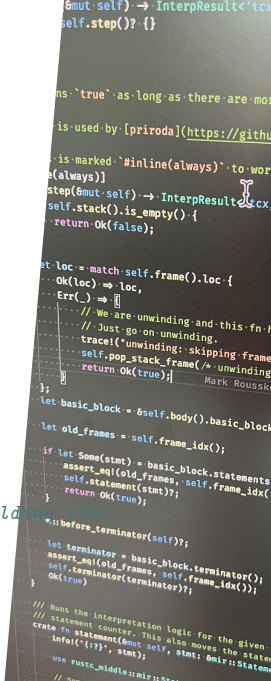
- **`usize`** is a type which must represent the whole range of addresses a pointer can dereference.
- It is used for array indexing, and array bounds.
- We don't want **`usize`** to be 128 bits, memory isn't 128 bit on Morello[†].
- So, we needed to change the layout of a pointer instead.

[†]This approach was explored by Nicholas Sim in his Masters Thesis.



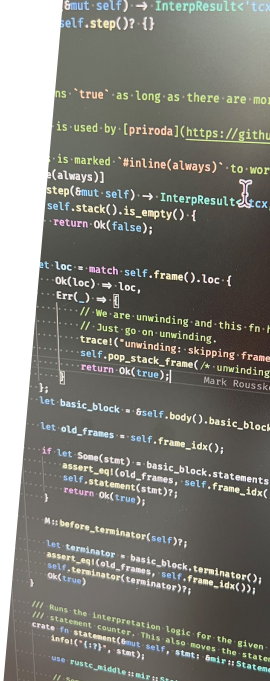
Compiler changes — object layout

```rust
pub fn target() -> Target {
    Target {
        llvm_target: "aarch64-unknown-freebsd".to_string(),
        pointer_range: 64,
        pointer_width: 128,
        data_layout: /* ... */,
        arch: "aarch64".to_string(),
        options: TargetOptions {
            features: "+morello,+c64".to_string(),
            llvm_abiname: "purecap".to_string(),
            max_atomic_width: Some(128),
            // Atomic pointers are supported and converting to integers
            // invalidates capabilities so we *must* use atomic pointers.
            atomic_pointers_via_integers: false,
            // TODO: figure out why this optimisation causes crashes when building libc
            merge_functions: MergeFunctions::Disabled,
            ..super::freebsd_base::opts()
        },
    }
}
```



Compiler changes — constant evaluation

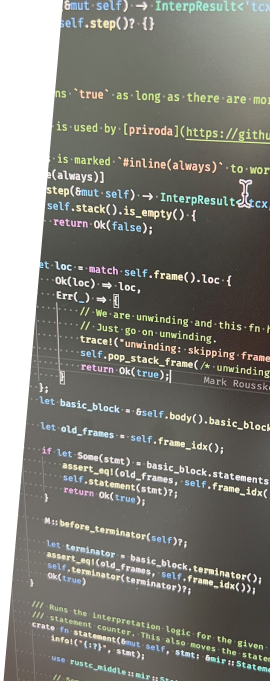
- Rust's IR is interpreted within the compiler to do constant evaluation.
- If it attempts to read uninitialised data that's considered an error.
- We cannot initialise the metadata of these pointers at compile time, so we had to patch up that divide.



Compiler changes — code generation

There are some baked in assumptions in the Rust compiler about valid operations on pointers, for example…

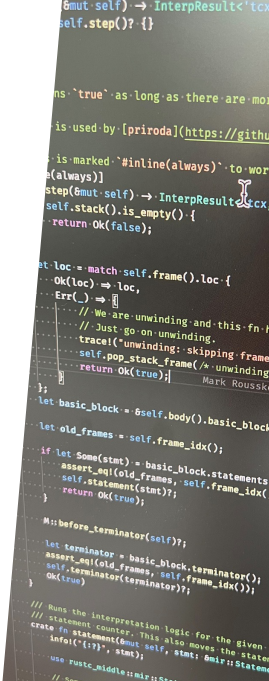
```
if ty.is_unsafe_ptr() {
  // Some platforms do not support atomic operations on pointers,
  // so we cast to integer first.
  let ptr_llty = bx.type_ptr_to(bx.type_isize());
  ptr = bx.pointercast(ptr, ptr_llty);
  val = bx.ptrtoint(val, bx.type_isize());
}
```



Standard library changes

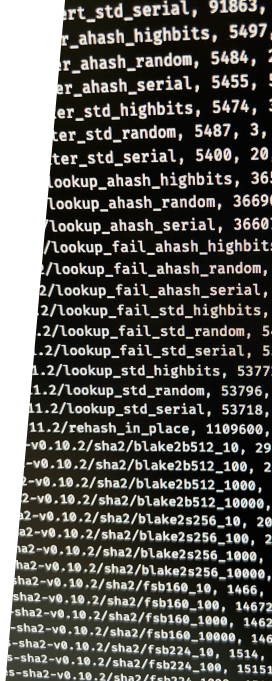
- The worst so far has been in a concurrency library which casts pointers to/from integers to tag them with metadata in the lower bits.
- Some bits of the FFI needed some tweaks, integer types being replaced with pointer types.

```
pub unsafe fn cast_from_usize(signal_ptr: usize) -> SignalToken {
    SignalToken { inner: mem::transmute(signal_ptr) }
}
```



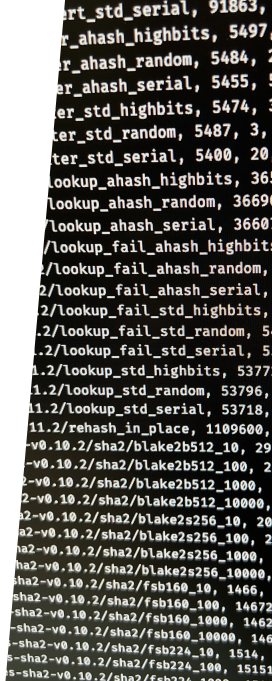
Measuring the performance of bounds checking

- It is interesting to understand what cost there is to Rust's dynamic bounds checking and how it relates to the always-on bounds checking in Morello.
- We have implemented a flag on the Rust compiler, `-C drop_bounds_checks`, which prevents the compiler from emitting software bounds checks. We call this version of the language Rust_{DBC}.



Measuring the performance of bounds checking

- We have picked 19 suites from the `crates.io` repository, which in total contain 872 benchmarks.
- These crates have 108k lines of Rust, of which 1k is `unsafe`. This does not include the dependencies.
- The benchmarks are run many times using the standard `cargo bench` command, and results are aggregated.

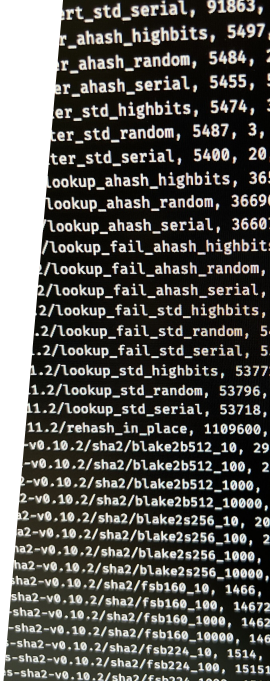


Measuring the performance of bounds checking

- We have picked 19 suites from the `crates.io` repository, which in total contain 872 benchmarks.
- These crates have 108k lines of Rust, of which 1k is **`unsafe`**. This does not include the dependencies.
- The benchmarks are run many times using the standard `cargo bench` command, and results are aggregated.

From `cargo bench` we extract time per iteration of the benchmark, and the run-to-run variance, for each of the four modes under test:

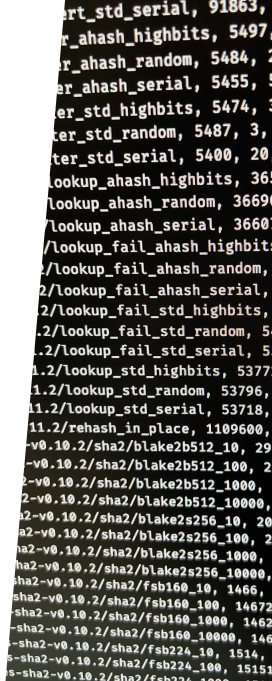
`hashbrown-0.11.2/clone_from_large`				
	Rust		Rust_{DBC}	
	Time/iter	$\pm$	Time/iter	$\pm$
Purecap	15,779	8	15,818	59
Hybrid	15,557	53	15,601	16



Performance results

To gain some confidence in our benchmarks, we have quantified Relative Error (R_E) for each benchmark we ran:

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



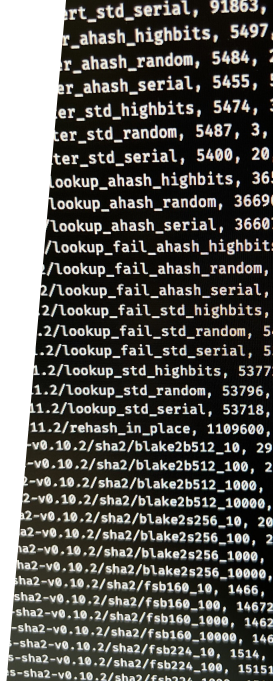
Performance results

To gain some confidence in our benchmarks, we have quantified Relative Error (R_E) for each benchmark we ran:

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

Relative error	Count
0 – 1%	3300
1 – 5%	142
$\geq$ 5%	38

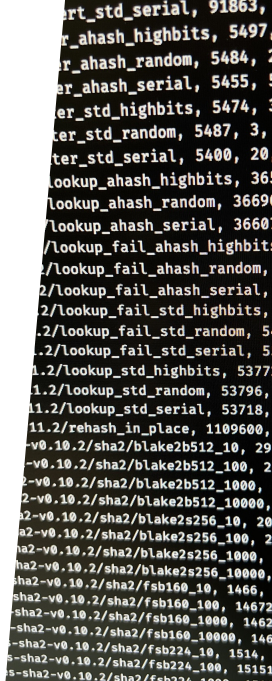
The vast majority of the benchmarks showed very small error margins.

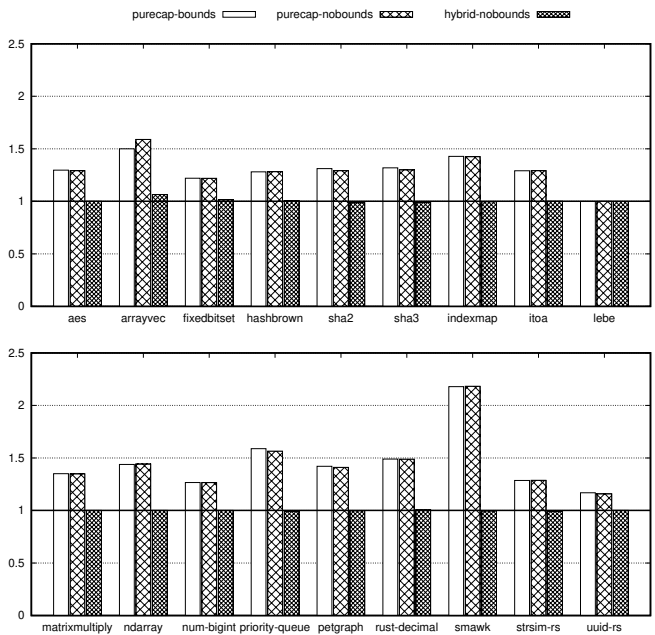


Performance results

Overall (by geometric mean) the slow-down on Purecap Morello is about 39%. We found the cost of Rust's dynamic bounds checking to be extremely low.

	Rust	Rust_{DBC}
Purecap	1.39	1.38
Hybrid	1.00	0.99




purecap-bounds
purecap-nobounds
hybrid-nobounds
2.5
2
1.5
1
0.5
0
aes
arrayvec
fixedbitset
hashbrown
sha2
sha3
indexmap
itoa
lebe
2.5
2
1.5
1
0.5
0
matrixmultiply
ndarray
num-bigint
priority-queue
petgraph
rust-decimal
smawk
strsim-rs
uuid-rs

Conclusion

- Port of the Rust compiler to the Morello platform.
- Significant Rust code running on Morello, and a performance analysis of that code.
- Demonstrating a 39% performance cost for always-on hardware bounds checks in Rust.

Conclusion

- Port of the Rust compiler to the Morello platform.
- Significant Rust code running on Morello, and a performance analysis of that code.
- Demonstrating a 39% performance cost for always-on hardware bounds checks in Rust.

Any questions?

P.S. We're looking for two RAs!