

Porting Rust to Morello

A safe software layer for a safe hardware layer

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

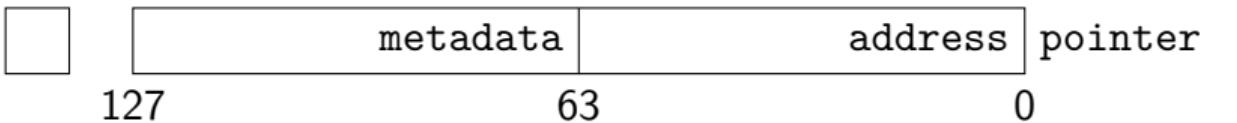


March 2023



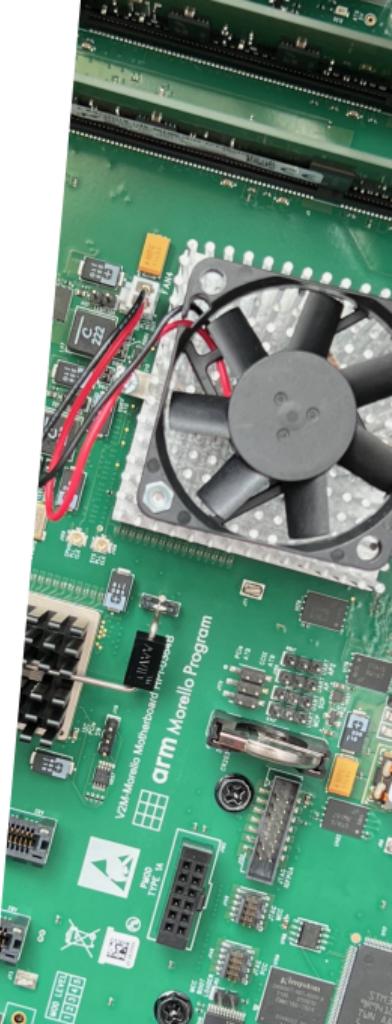
Capability machine

- ▶ 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.

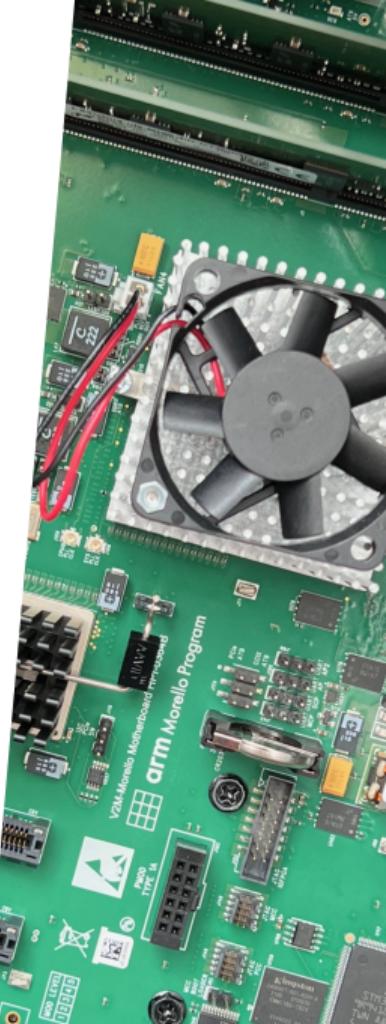


Morello prototype

- ▶ 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.

```

    &mut self) -> InterpResult<'tcx>
self.step()?;
}

ns.`true` as long as there are more

is used by [priroda](https://github.com/priroda/rustc)

is marked `#inline(always)` to work
(always)
step(&mut self) -> InterpResult<'tcx>
elf.stack().is_empty().{
return Ok(false);

loc = match self.frame().loc {
Ok(loc) -> loc,
Err(_) -> []
// We are unwinding and this fn is
// Just go on unwinding.
trace!("unwinding skipping frame");
self.pop_stack_frame(/* unwinding */);
return Ok(true);] Mark Roussin
}

basic_block = &self.body().basic_block;
old_frames = self.frame_idx();
let Some(stmt) = basic_block.statements
assert_eq!(old_frames, self.frame_idx());
self.statement(stmt)?;
return Ok(true);

::before_terminator(self)?;
let terminator = basic_block.terminator();
assert_eq!(old_frames, self.frame_idx());
self.terminator(terminator)?;
Ok(true)

runs the interpretation logic for the given
statement counter. This also moves the state
info(`{}`), stmt);
use rustc_middle::mir::sema;
// sema

```

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;  
}
```

```
&mut self) -> InterpResult<'tcx,  
self.step()? -&{  
  
ns `true` as long as there are more  
is used by [priroda](https://github.com/prirod  
e-is-marked `#inline(always)` to work  
e(always)]  
step(&mut self) -> InterpResult<'tcx,  
self.stack().is_empty(){  
    return Ok(false);  
  
let loc = match self.frame().loc {  
    Ok(loc) => loc,  
    Err(_) => []  
    // We are unwinding and this fn is  
    // Just go on unwinding.  
    trace!(unwinding: skipping frame  
    self.pop_stack_frame(/* unwinding */);  
    return Ok(true);  
};  
Mark Rousskin  
let basic_block = &self.body().basic_blocks[  
let old_frames = self.frames_idx();  
if let Some(stmt) = basic_block.statements.get(  
    assert_eq!(old_frames, self.frames_idx());  
    self.statement(stmt)?;  
    return Ok(true);  
  
M::before_terminator(self)?;  
let terminator = basic_block.terminator();  
assert_eq!(old_frames, self.frames_idx());  
self.terminator(terminator)?;  
Ok(true)  
}  
/// Runs the interpretation logic for the given  
/// statement counter. This also moves the state  
create_fn(statement(&mut self, stmt: &mir::StatementInfo<{>}, stat  
use rustc_middle::mir::...  
// Some  
// Some
```

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  
|
```

```
&mut self) -> InterpResult<'tcx>  
self.step()?.-{  
  
ns `true` as long as there are more  
is used by [priroda](https://github.com/prirod...)  
is marked `#inline(always)` to work  
(always)]  
step(&mut self) -> InterpResult<'tcx>  
self.stack().is_empty(){  
    return Ok(false);  
  
let loc = match self.frame().loc{  
    Ok(loc) => loc,  
    Err(_) => []  
    // We are unwinding and this fn is  
    // Just go on unwinding.  
    trace!(unwinding::skipping_frame);  
    self.pop_stack_frame(/* unwinding */);  
    return Ok(true);  
};  
Mark Rousskov  
let basic_block = &self.body().basic_blocks[loc];  
let old_frames = self.frames_idx();  
if let Some(stmt) = basic_block.statements[loc]  
    assert_eq!(old_frames, self.frames_idx());  
    self.state(statement(stmt)?);  
    return Ok(true);  
}  
M::before_terminator(self)?;  
let terminator = basic_block.terminator();  
assert_eq!(old_frames, self.frames_idx());  
self.terminator(terminator)?;  
Ok(true)  
}  
/// Runs the interpretation logic for the given  
/// statement counter. This also moves the state  
create fn statement(&mut self, stat: &mir::Statement);  
info!("{}:", stat);  
use rustc_middle::mir::...  
// Some code here
```

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.

```
    &mut self) -> InterpResult<'tcx,
    self.step()?-{}}

    ns `true` as long as there are more
    is used by [priroda](https://github.com/prirodar)
    .is_marked(`#inline(always)` to work
    (always))
    step(&mut self) -> InterpResult<'tcx,
    self.stack().is_empty()?-{}}
    .return Ok(false);

let loc = match self.frame().loc {
    Ok(loc) => loc,
    Err(_) => []
    // We are unwinding and this fn is
    // Just go on unwinding.
    trace!(unwinding: skipping frame
    self.pop_stack_frame(/* unwinding */));
    return Ok(true); Mark Rousskin
};

let basic_block = &self.body().basic_blocks[0];
let old_frames = self.frames_idx();
if let Some(stmt) = basic_block.statements[0] {
    assert_eq!(old_frames, self.frames_idx());
    self.state(statement(stmt)?);
    return Ok(true);
}

M::before_terminator(self)?;
let terminator = basic_block.terminator();
assert_eq!(old_frames, self.frames_idx());
self.terminator(terminator)?;
Ok(true)

/// Runs the interpretation logic for the given
/// statement counter. This also moves the state
create_fn(statement(&mut self, stmt: &mir::Statement,
info!({}), stmt);
use rustc_middle::mir::interpret;
// Some code here
// Some code here
```

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;  
    }  
}
```

```
    &mut self) -> InterpResult<'tcx,  
    self.step()? -&gt;  
  
ns `true` as long as there are more  
is used by [priroda](https://github.com/prirod  
e-is-marked `#inline(always)` to work  
e(always)]  
step(&mut self) -> InterpResult<'tcx,  
self.stack().is_empty(){  
    return Ok(false);  
  
let loc = match self.frame().loc {  
    Ok(loc) => loc,  
    Err(_) => []  
    // We are unwinding and this fn is  
    // Just go on unwinding.  
    trace!(unwinding: skipping frame  
    self.pop_stack_frame(/* unwinding */);  
    return Ok(true);  
};  
Mark Rousskin  
let basic_block = &self.body().basic_blocks[  
let old_frames = self.frames_idx();  
if let Some(stmt) = basic_block.statements[  
    assert_eq!(old_frames, self.frames_idx());  
    self.statement(stmt)?;  
} else {  
    return Ok(true);  
}  
M::before_terminator(self)?;  
let terminator = basic_block.terminator();  
assert_eq!(old_frames, self.frames_idx());  
self.terminator(terminator)?;  
Ok(true)  
}  
/// Runs the interpretation logic for the given  
/// statement counter. This also moves the state  
create_fn(statement: &mut self, stmt: &mir::StatementIndex, terminator: &mir::Terminator);  
use rustc_middle::mir::...  
// ...  
// ...
```

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
```

```
    &mut self) -> InterpResult<'tcx,  
    self.step()? .  
  
ns `true` as long as there are more  
is used by [priroda](https://github.com/prirod  
e-is-marked `#inline(always)` to work  
e(always)]  
step(&mut self) -> InterpResult<'tcx,  
self.stack().is_empty().  
    return Ok(false);  
  
let loc = match self.frame().loc {  
    Ok(loc) => loc,  
    Err(_) => []  
    // We are unwinding and this fn is  
    // Just go on unwinding.  
    trace!(unwinding: skipping frame  
    self.pop_stack_frame(/* unwinding */);  
    return Ok(true);  
};  
Mark Rousskin  
let basic_block = &self.body().basic_blocks[  
let old_frames = self.frames_idx();  
if let Some(stmt) = basic_block.statements[  
    assert_eq!(old_frames, self.frames_idx());  
    self.statement(stmt)?;  
} else {  
    return Ok(true);  
}  
M::before_terminator(self)?;  
let terminator = basic_block.terminator();  
assert_eq!(old_frames, self.frames_idx());  
self.terminator(terminator)?;  
Ok(true)  
}  
/// Runs the interpretation logic for the given  
/// statement counter. This also moves the state  
create_fn(statement(&mut self, stat: &mir::Statement);  
info!("[{}]", stat);  
use rustc_middle::mir::...  
// ...  
// ...
```

Why port Rust to Morello?

- The guarantees of capabilities complement the guarantees of Rust

```
(&mut self) -> InterpResult<'tcx>
self.step()?- {}

ns `true` as long as there are more frames
is used by [priroda](https://github.com/prirodan)
- is marked `#inline(always)` to work around [issue #10000]
step(&mut self) -> InterpResult<'tcx>
self.stack().is_empty() -> Ok(false);

let loc = match self.frame().loc {
    Ok(loc) => loc,
    Err(_) => []
    // We are unwinding and this fn is not unwind-safe
    // Just go on unwinding.
    trace!(unwinding: skipping frame);
    self.pop_stack_frame(/* unwinding */);
    return Ok(true); // Mark Rousskov
};

let basic_block = &self.body().basic_blocks[old_frames];
if let Some(stmt) = basic_block.statements[old_frames..self.frame_idx()];
    assert_eq!(old_frames, self.frame_idx());
    self.statement(stmt)?;
    return Ok(true);

M::before_terminator(self)?;
let terminator = basic_block.terminator();
assert_eq!(old_frames, self.frame_idx());
self.terminator(terminator)?;
Ok(true)

/// Runs the interpretation logic for the given
/// statement counter. This also moves the state
create_fn(statement(&mut self, stmt: &mir::StatementInfo<{<...>}>, statm
use rustc_middle::mir::StatementInfo;
// Some code here
```

Why port Rust to Morello?

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

```

    &mut self) -> InterpResult<'tcx>
self.step()?;
}

ns.`true` as long as there are more
is used by [priroda](https://github.com/priroda/rustc)
is marked `#inline(always)` to work
(always)]
step(&mut self) -> InterpResult<'tcx>
self.stack().is_empty() {
    return Ok(false);
}

loc = match self.frame().loc {
Ok(loc) => loc,
Err(_) => {
    // We are unwinding and this fn has
    // just gone on unwinding.
    trace!{"unwinding::skipping frame
    self.pop_stack_frame(/* unwinding */);
    return Ok(true);"} Mark Roussin
}

basic_block = &self.body().basic_block
old_frames = self.frame_idx();
let Some(stmt) = basic_block.statements
assert_eq!(old_frames, self.frame_idx());
self.statement(stmt)?;
return Ok(true);

::before_terminator(self)?;
let terminator = basic_block.terminator();
assert_eq!(old_frames, self.frame_idx());
self.terminator(terminator)?;
Ok(true)

Runs the interpretation logic for the given
statement counter. This also moves the state
info(`{:#?}`), stmt);
use rustc_middle::mir::StatementKind;
// So

```

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

```
    &mut self) -> InterpResult<'tcx>
    self.step()?-{}}

ns `true` as long as there are more frames to execute.

is used by [priroda](https://github.com/prirodan/morello-rust)

- is marked `#inline(always)` to work around compiler bugs
e(always)]
step(&mut self) -> InterpResult<'tcx>
self.stack().is_empty()?-{}}
return Ok(false);

let loc = match self.frame().loc {
    Ok(loc) => loc,
    Err(_) => []
    // We are unwinding and this fn is not unwind-safe
    // Just go on unwinding.
    trace!(unwinding: skipping frame);
    self.pop_stack_frame(/* unwinding */)
    return Ok(true); Mark Rousskov
};

let basic_block = &self.body().basic_blocks[old_frames];
let old_frames = self.frame_idx();
if let Some(stmt) = basic_block.statements[old_frames];
assert_eq!(old_frames, self.frame_idx());
self.statement(stmt)?;
return Ok(true);

M::before_terminator(self)?;
let terminator = basic_block.terminator();
assert_eq!(old_frames, self.frame_idx());
self.terminator(terminator)?;
Ok(true)

/// Runs the interpretation logic for the given
/// statement counter. This also moves the state
create_fn(statement(&mut self, stmt: &mir::Statement));
info!("{}: {}", stmt);
use rustc_middle::mir::interpretation::InterpResult;
// Some code here
// Some code here
```

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

```
    &mut self) -> InterpResult<'tcx,
    self.step()?- {}
}

ns `true` as long as there are more frames to unwind.

is used by [priroda](https://github.com/priroda/morello)

    .is_marked(`#inline(always)`).to_writable(always)]
step(&mut self) -> InterpResult<'tcx,
self.stack().is_empty()?- {
    return Ok(false);
}

let loc = match self.frame().loc {
    Ok(loc) => loc,
    Err(_) => []
    // We are unwinding and this fn is not unwind-safe.
    // Just go on unwinding.
    trace!(unwinding: skipping frame {loc});
    self.pop_stack_frame(/* unwinding */);
    return Ok(true); Mark Rousskov
};

let basic_block = &self.body().basic_blocks[0];
let old_frames = self.frames[0..frame_idx];
if let Some(stmt) = basic_block.statements[frame_idx];
    assert_eq!(old_frames, self.frames[0..frame_idx]);
    self.state(statement(stmt)?);
    return Ok(true);

M::before_terminator(self)?;
let terminator = basic_block.terminator();
assert_eq!(old_frames, self.frames[0..frame_idx]);
self.terminator(terminator)?;
Ok(true)

/// Runs the interpretation logic for the given
/// statement counter. This also moves the state
create_fn(statement(&mut self, stat: &mir::Statement,
info: "{:?}", stat);
use rustc_middle::mir::interpretation::InterpResult;
// Some code here
// Some code here
```

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

```
    &mut self) -> InterpResult<'tcx,
    self.step()?-{}}

ns `true` as long as there are more frames to unwind.

is used by [priroda](https://github.com/priroda/mir-interpreter)

    .is_marked(`#inline(always)`).to_writable(always)]
step(&mut self) -> InterpResult<'tcx,
self.stack().is_empty()?-{}}
    return Ok(false);

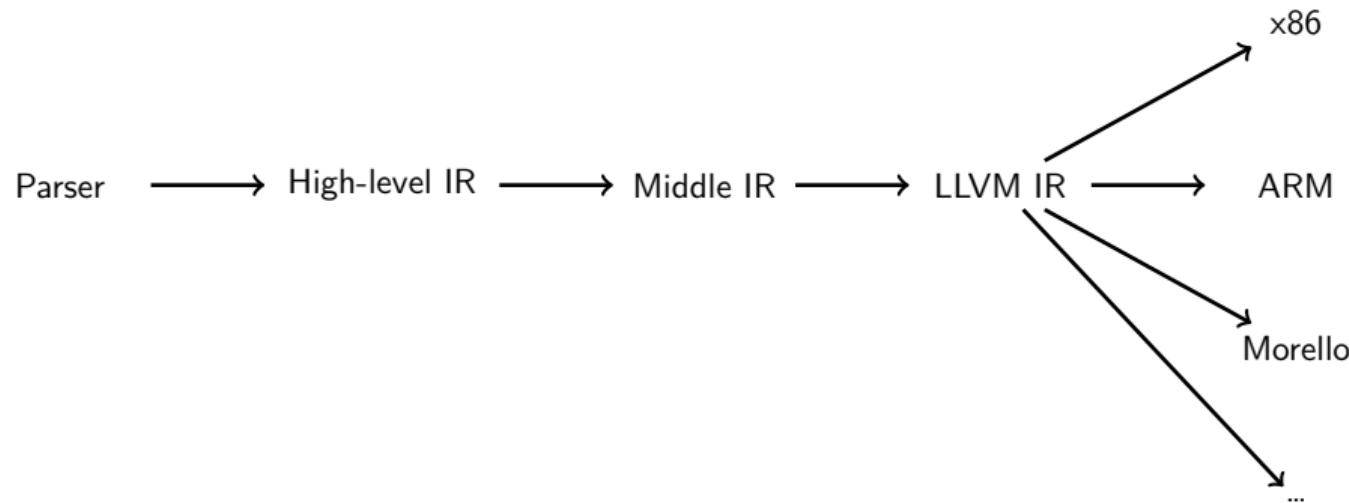
let loc = match self.frame().loc {
    Ok(loc) => loc,
    Err(_) => [] // We are unwinding and this fn is not unwindable.
    // Just go on unwinding.
    trace!(unwinding: skipping frame);
    self.pop_stack_frame(/* unwinding */);
    return Ok(true); // Mark Rousskov
};

let basic_block = &self.body().basic_blocks[0];
let old_frames = self.frames_idx();
if let Some(stmt) = basic_block.statements.get(0) {
    assert_eq!(old_frames, self.frames_idx());
    self.state(statement(stmt)?);
    return Ok(true);
}

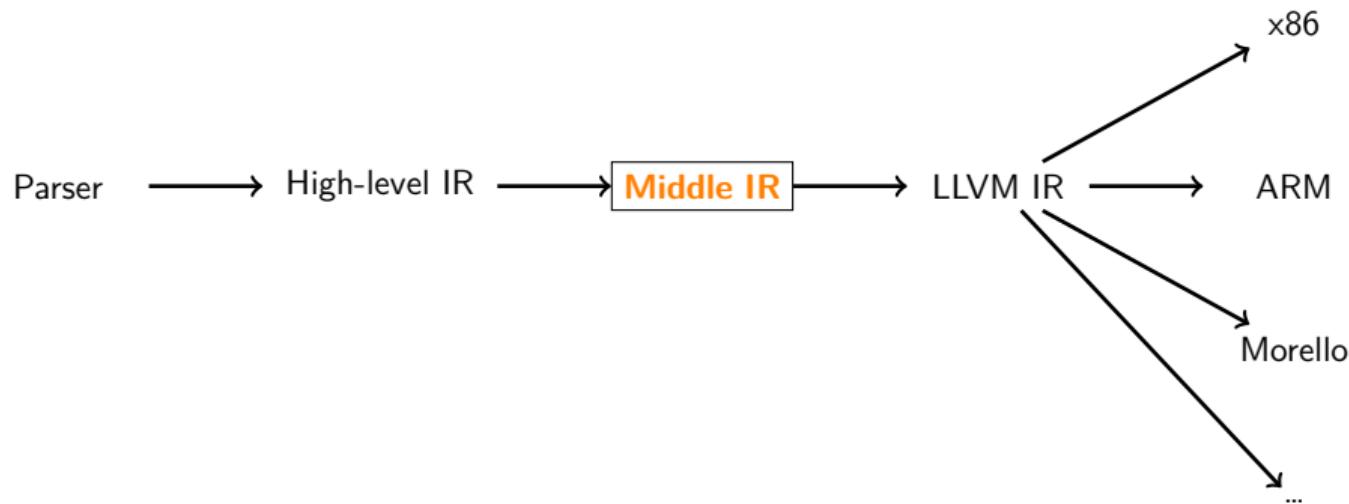
M::before_terminator(self)?;
let terminator = basic_block.terminator();
assert_eq!(old_frames, self.frames_idx());
self.terminator(terminator)?;
Ok(true)

/// Runs the interpretation logic for the given
/// statement counter. This also moves the state
create_fn(statement(&mut self, stat: &mir::Statement,
info: "{?}" stat));
use rustc_middle::mir::interpretation::InterpResult;
// Some code here
// Some code here
```

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...

```
    &mut self) -> InterpResult<'tcx,
    self.step()?-{}}

    ns `true` as long as there are more frames to unwind.
    is used by [priroda](https://github.com/priroda/rust-morello)
    .is_marked(`#inline(always)` to work around the bug in LLVM's
    step(&mut self) -> InterpResult<'tcx,
    self.stack().is_empty()?-{}}
    .return Ok(false);

    let loc = match self.frame().loc {
        Ok(loc) => loc,
        Err(_) => []
    };
    // We are unwinding and this fn is not tail recursive.
    // Just go on unwinding.
    trace!(unwinding: skipping frame {loc});
    self.pop_stack_frame(/* unwinding */);
    return Ok(true); // Mark Rousskov
};

let basic_block = &self.body().basic_blocks[old_frames..];
let old_frames = self.frame_idx();
if let Some(stmt) = basic_block.statements.get(old_frames) {
    assert_eq!(old_frames, self.frame_idx());
    self.statement(stmt)?;
}
return Ok(true);

M::before_terminator(self)?;
let terminator = basic_block.terminator();
assert_eq!(old_frames, self.frame_idx());
self.terminator(terminator)?;
Ok(true)

/// Runs the interpretation logic for the given
/// statement counter. This also moves the state
/// create_fn(statement(&mut self, stat: &mir::Statement));
info!("[{?}]", stat);
use rustc_middle::mir::interpretation::InterpCx;
// Some code here
// use rustc_middle::mir::interpretation::InterpCx;
```

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.

```
    &mut self) -> InterpResult<'tcx,
    self.step()?-{}}

ns `true` as long as there are more frames to unwind.

is-used-by [priroda](https://github.com/priroda/rustc-mir)
    .is-marked(`#inline(always)`)-> work_on_frame(always)]
step(&mut self) -> InterpResult<'tcx, ()>
self.stack().is_empty() -> return Ok(false);

let loc = match self.frame().loc {
    Ok(loc) => loc,
    Err(_) => []
    // We are unwinding and this fn is not unwind-safe
    // Just go on unwinding.
    trace!(unwinding: skipping frame {loc});
    self.pop_stack_frame(/* unwinding */);
    return Ok(true); Mark Rousskov
};

let basic_block = &self.body().basic_blocks[old_frames..];
let old_frames = self.frame_idx();
if let Some(stmt) = basic_block.statements.get(old_frames) {
    assert_eq!(old_frames, self.frame_idx());
    self.statement(stmt)?;
}
return Ok(true);

M::before_terminator(self)?;
let terminator = basic_block.terminator();
assert_eq!(old_frames, self.frame_idx());
self.terminator(terminator)?;
Ok(true)

/// Runs the interpretation logic for the given
/// statement counter. This also moves the state
create_fn(statement(&mut self, stat: &mir::Statement));
info!("{}: {}", stat, self.frame_idx());
use rustc_middle::mir::interpret::InterpResult;
// Some code here
```

Compiler changes — object layout

```
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
            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.

```
    &mut self) -> InterpResult<'tcx>
    self.step()?- {}

ns `true` as long as there are more frames to step through.

is used by [priroda](https://github.com/prirodan/rustc-mir)

    .is_marked(`#inline(always)`).to_writable(always)]
step(&mut self) -> InterpResult<'tcx>
self.stack().is_empty(){
    return Ok(false);

at loc == match self.frame().loc {
    Ok(loc) => loc,
    Err(_) => []
        // We are unwinding and this fn is not unwind-safe.
        // Just go on unwinding.
        trace!(unwinding: skipping frame);
        self.pop_stack_frame(/* unwinding */);
        return Ok(true);    Mark Rousskov
    };
let basic_block = &self.body().basic_blocks[0];
let old_frames = self.frames[0..frame_idx];
if let Some(stmt) = basic_block.statements[frame_idx] {
    assert_eq!(old_frames, self.frames[0..frame_idx]);
    self.state(statement(stmt)?);
    return Ok(true);
}

M::before_terminator(self)?;
let terminator = basic_block.terminator();
assert_eq!(old_frames, self.frames[0..frame_idx]);
self.terminator(terminator)?;
Ok(true)

/// Runs the interpretation logic for the given
/// statement counter. This also moves the state
create_fn(statement(&mut self, stat: &mir::Statement));
info!("{}: ", stat);
use rustc_middle::mir::interpretation::InterpResult;
// Some code here
```

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());
}
```

```
    &mut self) -> InterpResult<'tcx,
    self.step()?-{}
}

ns `true` as long as there are more frames to unwind.

is used by [priroda](https://github.com/prirodan/rustc-mir)

    .is_marked(`#inline(always)` to work around #39111)
step(&mut self) -> InterpResult<'tcx, ()> {
    self.stack().is_empty()?;
    return Ok(false);
}

at_loc == match self.frame().loc {
    Ok(loc) => loc,
    Err(_) => [] // We are unwinding and this fn has no location.
    // Just go on unwinding.
    trace!("unwinding: skipping frame");
    self.pop_stack_frame(/* unwinding */);
    return Ok(true); // Mark Roussel
};

let basic_block = &self.body().basic_blocks[0];
let old_frames = self.frames();
if let Some(stmt) = basic_block.statements.get(0) {
    assert_eq!(old_frames, self.frames());
    self.state(statement(stmt)?);
    return Ok(true);
}

M::before_terminator(self)?;
let terminator = basic_block.terminator();
assert_eq!(old_frames, self.frames());
self.terminator(terminator)?;
Ok(true)

/// Runs the interpretation logic for the given statement counter. This also moves the state
create_fn(statement(&mut self, stmt: &mir::Statement));
info!("{}: {}", self, stmt);
use rustc_middle::mir::interpretation::InterpResult;
// Some platform-specific code here
}

    &mut self) -> InterpResult<'tcx,
    self.step()?-{}
}

ns `true` as long as there are more frames to unwind.

is used by [priroda](https://github.com/prirodan/rustc-mir)

    .is_marked(`#inline(always)` to work around #39111)
step(&mut self) -> InterpResult<'tcx, ()> {
    self.stack().is_empty()?;
    return Ok(false);
}

at_loc == match self.frame().loc {
    Ok(loc) => loc,
    Err(_) => [] // We are unwinding and this fn has no location.
    // Just go on unwinding.
    trace!("unwinding: skipping frame");
    self.pop_stack_frame(/* unwinding */);
    return Ok(true); // Mark Roussel
};

let basic_block = &self.body().basic_blocks[0];
let old_frames = self.frames();
if let Some(stmt) = basic_block.statements.get(0) {
    assert_eq!(old_frames, self.frames());
    self.state(statement(stmt)?);
    return Ok(true);
}

M::before_terminator(self)?;
let terminator = basic_block.terminator();
assert_eq!(old_frames, self.frames());
self.terminator(terminator)?;
Ok(true)

/// Runs the interpretation logic for the given statement counter. This also moves the state
create_fn(statement(&mut self, stmt: &mir::Statement));
info!("{}: {}", self, stmt);
use rustc_middle::mir::interpretation::InterpResult;
// Some platform-specific code here
}
```

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) }  
}
```

```
&mut self) -> InterpResult<'tcx>  
self.step()?-{}  
  
ns `true` as long as there are more  
is used by [priroda](https://github.com/prirodar)  
is marked `#inline(always)` to work  
(always)]  
step(&mut self) -> InterpResult<'tcx>  
self.stack().is_empty() -> Ok(false);  
  
let loc = match self.frame().loc {  
    Ok(loc) => loc,  
    Err(_) => []  
};  
// We are unwinding and this fn is  
// Just go on unwinding.  
trace!(unwinding: skipping frame  
self.pop_stack_frame(/* unwinding */);  
return Ok(true);  
Mark Rousskin  
};  
let basic_block = &self.body().basic_blocks[  
let old_frames = self.frames_idx();  
if let Some(stmt) = basic_block.statements.get(0)? {  
    assert_eq!(old_frames, self.frames_idx());  
    self.state(statement(stmt)?);  
    return Ok(true);  
}  
M::before_terminator(self)?;  
let terminator = basic_block.terminator();  
assert_eq!(old_frames, self.frames_idx());  
self.terminator(terminator)?;  
Ok(true)  
}  
/// Runs the interpretation logic for the given  
/// statement counter. This also moves the state  
create fn statement(&mut self, stat: &mir::Statement);  
info! "{?}" stat;  
use rustc_middle::mir::...  
// Some code here  
// Some code here
```

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	±	Time/iter	±
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:

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

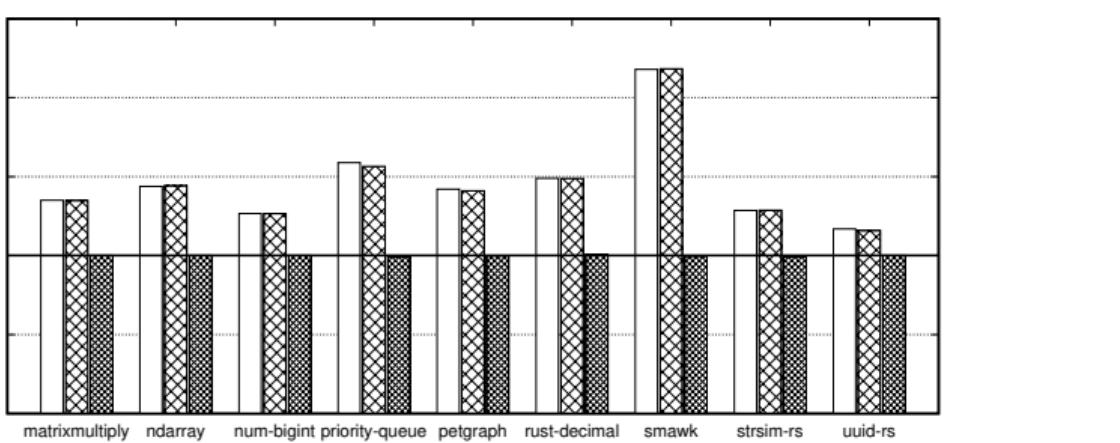
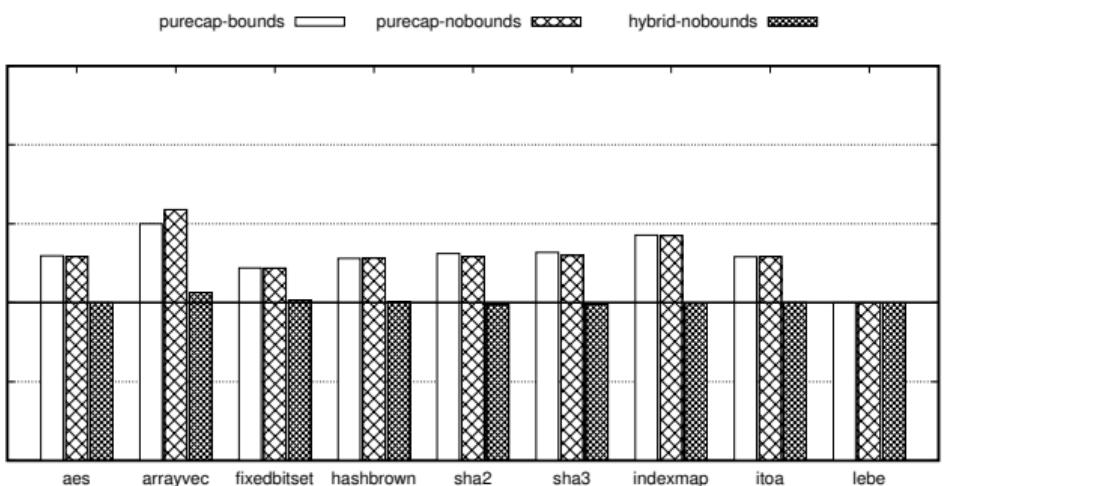
Relative error	Count
0 – 1%	3300
1 – 5%	142
$\geq 5\%$	38

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



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!

