

BPF CO-RE support in Rust

Michał Rostecki (vad), anza.xyz

Simple Rust type

```
pub struct Foo {  
    a: i32,  
    b: u32,  
}
```

LLVM Debug Info

```
!87 = !DIBasicType(name: "u32", size: 32,  
    encoding: DW_ATE_unsigned)  
[...]  
!155 = !DIBasicType(name: "i32", size: 32,  
    encoding: DW_ATE_signed)  
[...]
```

```
!152 = !DICompositeType(tag: DW_TAG_structure_type,  
    name: "Foo", scope: !9,  
    file: !14, size: 64, align: 32,  
    flags: DIFlagPublic,  
    elements: !153,  
    templateParams: !74,  
    identifier:"bce6452370c655d63a49cfb6bdce4bc7")  
!153 = !{!154, !156}
```

LLVM Debug Info

```
!154 = !DIDerivedType(tag: DW_TAG_member, name: "a",  
    scope: !152, file: !14,  
    baseType: !155, size: 32, align: 32,  
    flags: DIFlagPrivate)  
[...]  
!156 = !DIDerivedType(tag: DW_TAG_member, name: "b",  
    scope: !152, file: !14,  
    baseType: !87, size: 32, align: 32, offset: 32,  
    flags: DIFlagPrivate)
```

```
#2: <STRUCT> 'Foo' sz:8 n:2
    #00 'a' off:0 --> [3]
    #01 'b' off:32 --> [4]
#3: <INT> 'i32' bits:32 off:0 enc:signed
#4: <INT> 'u32' bits:32 off:0
```

Program's BTF

Lives in the .BTF section of the program binary.

```
#351: <struct> 'task_struct'  
[...]
```

```
    #96 'pid' off:21632 --> [1553]
```

pid has offset of 21632 bits (2704 bytes).

Program's bytecode

```
r0 = *(u32 *) (r2 + 2704)
```

Access to the `task_struct` pointer with an offset of 2704 bytes.

BTF relocation

Lives in the `.BTF.ext` section of the program binary.

```
RELO: FIELD_OFFSET
    type_id = task_struct
    access = pid
    insn_offset = 5
```

```
#351: <STRUCT> 'task_struct'  
[...]
```

```
    #96 'pid' off:21664 --> [1553]
```

Offset of 21664 bits (2708 bytes).

Patched bytecode

```
r0 = *(u32 *) (r2 + 2708)
```

How CO-RE works in C?

There are several options:

- Wrapping field access with the `__builtin_preserve_access_index` intrinsic.
- Annotating the type with `__attribute__((preserve_access_index))`, then all field projections emit relocations.
- `__builtin_preserve_field_info` intrinsic:
 - Checked access.
 - Offset.

Wrapping field access

```
struct task_struct {  
    pid_t pid;  
    pid_t tgid;  
};  
  
__attribute__((always_inline))  
pid_t get_pid(struct task_struct *task) {  
    return __builtin_preserve_access_index(task->pid);  
}
```

This is what BPF_CORE_READ macro does underneath.

Type annotation

```
struct task_struct {  
    pid_t pid;  
    pid_t tgid;  
}__attribute__((preserve_access_index));  
  
__attribute__((always_inline))  
pid_t get_pid(struct task_struct *task) {  
    return task->pid;  
}
```

Checked access

```
struct task_struct {
    pid_t pid;
    pid_t tgid;
    // Recently introduced field.
    struct rseq_data rseq;
};

__attribute__((always_inline))
bool rseq_exists(struct task_struct *task) {
    return __builtin_preserve_field_info(
        task->rseq,
        BPF_FIELD_EXISTS);
}
```

Checked access

```
__attribute__((always_inline))
struct rseq_data* rseq(struct task_struct *task) {
    if rseq_exists(task) {
        return &__builtin_preserve_access_index(
            task->rseq_data);
    } else {
        return NULL;
    }
}
```

```
struct task_struct {  
    pid_t pid;  
    pid_t tgid;  
};  
  
__attribute__((always_inline))  
size_t pid_offset(struct task_struct *task) {  
    return __builtin_preserve_field_info(  
        task->pid,  
        BPF_FIELD_INFO_OFFSET);  
}
```

How LLVM emits CO-RE relocations?

LLVM lowers CO-RE field accesses to a set of intrinsics:

- `@llvm.bpf.preserve.field.info` — queries a field's offset/size with BTF metadata
- `@llvm.preserve.struct.access.index /`
`@llvm.preserve.union.access.index /`
`@llvm.preserve.array.access.index` — internal index chains built during lowering to track the path through nested struct/union/array accesses

BTF support in Rust and Aya

- Rust is able to produce correct BTF for all types.
- **BTF relocations in rustc are under review.**
 - RFC: <https://github.com/rust-lang/rfcs/pull/3966>
 - PR: <https://github.com/rust-lang/rust/pull/161107>
- Aya is able to load BTF from the program and perform the relocations against kernel's BTF.
- Support for BTF-compatible map definitions is coming in the next release.

How does the PR work?

It consists of the following parts:

- User-facing API (macros).
- Builtin macros.
- MIR representation.
- Codegen backends support.

- `#[btf_relocatable]` on structs/unions (BPF only). Direct field access on such a type is a compile error.
- User facing macros:
 - `core::arch::bpf::field_byte_offset!(Container, field.path) -> Option<usize>`
 - `core::arch::bpf::field_byte_size!(Container, field.path) -> Option<usize>`
- Gated by `#![feature(btf_relocations)]`.

- Internal building macros (not exposed to the user):
 - `btf_field_byte_offset(Type, field.path)`
 - `btf_field_byte_size(Type, field.path)`
 - `btf_field_exists(Type, field.path)`
- `btf_field_exists` has no user-facing counterpart. Existence is already folded into the `Option` return of the `offset/size` macros.

MIR representation

The user-facing API is agreed. What's more controversial is how the compiler carries the query from the frontend to the backend.

- Frontend knows about field paths (`field.path`).
- Backends (LLVM, GCC) operate on field indices from DI.
- We need to convert field paths to indices.

The open question: what MIR construct should represent this conversion?

```
Rvalue::BtfFieldInfo { kind, path: Box<[(Ty,  
VariantIdx, FieldIdx)]> } variant.
```

Pros:

- Never exposed to users.

Cons:

- Puts a BPF-specific construct in backend-agnostic MIR.

Intrinsics (with runtime args)

```
pub fn btf_preserve_access_index<T: PointeeSized>(
    base: *const (),
    variant: u32,
    field: u32,
) -> *const ();

pub fn btf_preserve_field_byte_offset(field: *const ())
    -> usize;

pub fn btf_preserve_field_byte_size(field: *const ())
    -> usize;

pub fn btf_preserve_field_exists(field: *const ())
    -> bool;
```

Intrinsics (with const generics)

```
pub fn btf_preserve_access_index<
    T: PointeeSized,
    const VARIANT: u32,
    const FIELD: u32
>(base: *const ()) -> *const ();
pub fn btf_preserve_field_byte_offset(field: *const ())
    -> usize;
pub fn btf_preserve_field_byte_size(field: *const ())
    -> usize;
pub fn btf_preserve_field_exists(field: *const ())
    -> bool;
```

Pros:

- Structurally identical to `offset_of`.

Cons:

- Intrinsics are usually exposed to the users, these ones would not be.
- Intrinsics with `const` arguments are not welcome by Rust maintainers.
- BTF relocations operate on path indices, but these intrinsics suggest as if they were about pointer arithmetic.

Any other ideas?

Any less invasive and easily maintainable MIR entity would be great!

The BuilderMethods trait gains four methods:

- `btf_preserve_access_index`
- `btf_preserve_field_byte_offset`
- `btf_preserve_field_byte_size`
- `btf_preserve_field_exists`

- `btf_preserve_access_index` ->
`llvm.preserve.{struct,union}.access.index`
- `btf_preserve_field_byte_offset` ->
`llvm.bpf.preserve.field.info(0, ...)`
- `btf_preserve_field_byte_size` ->
`llvm.bpf.preserve.field.info(1, ...)`
- `btf_preserve_field_exist` ->
`llvm.bpf.preserve.field.info(2, ...)`

Code example

```
#![feature(btfs_relocations)]

#[btfs_relocatable]
#[repr(C)]
pub struct task_struct {
    pub pid: i32,
    pub tgid: i32,
}
```

Code example

```
#[inline]
pub fn maybe_pid(&self) -> Option<&i32> {
    btf_field_byte_offset!(
        task_struct, pid
    ).map(|offset| {
        let ptr = self as *const task_struct
            as *const u8;
        unsafe {
            &*(ptr.add(offset) as *const i32)
        }
    })
}
```

Code example

```
#[inline]
pub fn pid(&self) -> Option<&i32> {
    let offset = btf_field_byte_offset!(
        task_struct, pid
    ).unwrap_unchecked();
    let ptr = self as *const task_struct
        as *const u8;
    unsafe {
        &*(ptr.add(offset) as *const i32)
    }
}
```

Nested field support

```
#[btf_relocatable]
pub struct load_weight {
    pub weight: usize,
}
#[btf_relocatable]
pub struct sched_entity {
    pub load: load_weight,
    pub vruntime: u64,
}
#[btf_relocatable]
pub struct task_struct {
    pub se: sched_entity,
}
```

Nested field support

```
impl task_struct {  
    pub fn maybe_sched_vruntime(&self) -> Option<&u64> {  
        btf_field_byte_offset!(  
            task_struct, se.vruntime  
        ).map(|offset| {  
            let ptr = self as *const task_struct  
                as *const u8;  
            unsafe {  
                &*(ptr.add(offset) as *const u64)  
            }  
        })  
    }  
}
```

Nested field support

```
pub fn maybe_sched_load_weight(&self) ->
    Option<&usize> {
    btf_field_byte_offset!(
        task_struct, se.load.weight
    ).map(|offset| {
        let ptr = self as *const task_struct
            as *const u8;
        unsafe {
            &*(ptr.add(offset) as *const usize)
        }
    })
}
```

That looked scary and not safe at all!

Don't worry, that's not the code that the end users are going to write.

Aya is going to provide safe abstraction and macros, and users won't be able to do anything unsound.

#[btf] macro

```
#[btf_relocatable]
pub struct load_weight {
    pub weight: usize,
}

#[btf_relocatable]
pub struct sched_entity {
    pub load: load_weight,
    pub vruntime: u64,
}

#[aya_btf::btf]
pub struct task_struct {
    pub pid: i32,
    pub tgid: i32,
    pub se: sched_entity,
}
```

#[btf] macro

```
fn process_task(task: &task_struct) -> Option<()> {  
    // These methods are generated by the macro.  
    let pid = task.maybe_pid()?;  
    let tgid = task.tgid();  
    let vruntime = task.maybe_se_vruntime()?;  
    let weight = task.se_load_weight();  
    [...]  
}
```

But can we do CO-RE field projection in Rust?

Can't we allow the following?

```
fn process_task(task: &task_struct) {  
    let pid = task.pid;  
    let tgid = task.tgid;  
    let vruntime = task.se.vruntime;  
    let weight = task.se.load.weight;  
    [...]  
}
```

That was the first thing we proposed, but...

- Currently field projection requires types to be Sized (size known on compile time).
- Rust team is still figuring out a proposal of distinguishing between `const` Sized and dynamically sized on runtime.
- Requires an intrusive change in Rust's MIR field projection, and it's specific to one target.

Thank you

Any questions?