Universiteit Leiden
The Netherlands

Opleiding Informatica

Exception handling support for the Cranelift code generator

Björn Roy Baron

Supervisors:
Kristian Rietveld

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

dd/mm/yyyy

Abstract

Compilers that wish to have the compiled code interoperate with native C++ code and those that wish to securely sandbox untrusted code have conflicting requirements on their implementation of exceptions. The former has to use the native unwinder for compatibility, while the latter can not use it as the native unwinder is not secure against untrusted inputs. This thesis designs and implements a method to handle exceptions in the Cranelift code generator which is flexible enough to fit with the requirements of both compilers interoperating with native code and compilers which want to sandbox their input. The design has been validated with rustc_codegen_cranelift to interoperate with native C++ exceptions and could possibly be used in the future with Wasmtime for implementing WebAssembly exceptions while maintaining the robustness of Wasmtime's sandbox. Outside of microbenchmarks the performance penalty is no larger than 3%.

Contents

1 Introduction

Many compilers and compiler frameworks make use of an intermediate representation (IR). The role of an intermediate representation is to simplify the compiler by removing details present in the input language that are irrelevant to later parts of the compiler.

Cranelift is a compiler framework specifically designed for correctness. It has a fairly straight forward IR and makes use of features like ISLE and a regalloc checker to help with verifying the correctness of its various components. Currently several language frontends for Cranelift exist amongst which are a Rust and a WebAssembly frontend. Many programming languages require support for exceptions. For example Rust has exceptions in the form of "panics", which can unwind into C++. For WebAssembly a proposal exists to add exception handling support (was23a). However, as Cranelift currently does not support exceptions, it is not possible to generate target code for Rust panics and not efficiently for WebAssembly exceptions.

In this thesis we investigate the design and implementation of exception handling support for Cranelift. This is a non-trivial problem as there are conflicting requirements. Exception handling for Rust requires compatibility with native C++ code, whereas WebAssembly requires safety against malicious WebAssembly code. Native C++ exception handling is not an option for WebAssembly because the unwinder used for native C++ exceptions is not designed to handle untrusted input. It blindly follows the instructions it is given. Therefore, we propose a generic way of expressing exceptions in Cranelift IR which allows target code to be generated in different ways adapting to the requirements of varying contexts.

As a qualitative validation we implement the proposed design for Rust and the x86_64 and ARM backends of Cranelift to demonstrate the viability of the design. In the quantitative experimental evaluation we show that the runtime performance cost is no larger than 3% outside of microbenchmarks.

The implementation developed as part of this thesis is released as open source. For links to source code repositories please refer to Appendix A.

The remainder of this thesis is organized as follows. In Chapter 2 we will be describing background information about various technologies and projects mentioned in the rest of the thesis. Chapter 3 describes the design for implementing exceptions in Cranelift that was chosen as well as a rationale why this particular design was chosen. In Chapter 4 the changes that were made to Cranelift and rustc_codegen_cranelift are described. After that in Chapter 5 the implementation will be evaluated. And finally in Chapter 6 we will give a conclusion.

2 Background

In this Chapter we will concisely describe the background knowledge required to understand the remainder of this thesis. This in particular comprises the Cranelift compilation flow. We will introduce Cranelift by example. Therefore we first give a brief introduction of WebAssembly. Next we describe Cranelift and how WebAssembly is lowered to Cranelift IR. Finally we introduce the Cranelift based codegen backend for Rust and discuss various ways of implementing exception handling.

2.1 WebAssembly

WebAssembly (HRS+17) (Web) is a bytecode language designed for portable and secure execution of untrusted code. It was originally designed for compiling native programs to run in the browser. It is now getting used on cloud platforms like Cloudflare Workers (Var18) and Fastly (Hic19) for execution of programs provided by untrusted clients of their CDN networks. Traditionally isolation of different clients has been done using containers or virtual machines. This is a fair bit heavier than using the sandboxing capabilities provided by WebAssembly (GFD22). WebAssembly does not require a full OS environment, but executes in a sandboxed process VM. As such using WebAssembly reduces costs for CDN networks.

```
(module
  (func $square (param i32) (result i32)
    local.get 0
    local.get 0
    i32.mul)
  (func $call_square (result i32)
    i32.const 2
    call $square)
  (export "do_call_square" (func $call_square)))
```
Listing 1: Example WebAssembly module

A small example of a WebAssembly module is shown in Listing 1. This module defines two functions. `square` accepts a signed integer, multiplies with it self and returns the result. It does this by doing `local.get 0` twice to push the first argument onto the stack twice and then `i32.mul` to multiply the top two values on the stack. The function implicitly returns the single remaining value on the stack. `call_square` calls `square` with 2 as argument and returns the result. The final line exports `call_square` from the module as `do_call_square`.
Several WebAssembly implementations exist like wasm3, WebAssembly Micro Runtime and Wasmer. In this thesis we only consider Wasmtime because it is co-developed with Cranelift. Wasmtime (Was23b) is a WebAssembly runtime with support for both ahead-of-time and just-in-time compilation of WebAssembly modules. It has a high focus on security as it is used and developed by Fastly for running untrusted code on their CDN network.

2.2 Cranelift

The Cranelift code generator is used as backend of Wasmtime and, as described in Section 2.3, rustc_codegen_cranelift. Cranelift is also focused on security to enable the security goals of Wasmtime (Fit22). To this end the combination of Wasmtime and Cranelift undergoes thorough fuzzing of every stable feature (basically feeding random wasm binaries into Wasmtime with the aim to either crash it or find a difference in results with the official interpreter of the WebAssembly specification) as well as working together with academia to integrate state of the art techniques for verification of security. For example it has a register allocator checker which checks that the output of the register allocator is valid, in addition a research group has been formally verifying the lowering rules from Cranelift's intermediate representation (CLIF IR) to the VCode IR which has an almost one-to-one correspondence with real CPU instructions (VPF+23). Most VCode instructions direcly correspond to a single real CPU instruction, but some VCode instructions expand to a sequence of real CPU instructions. The latter case is mostly used for instruction sequences where the register allocator is not allowed to insert any moves, spills or other instructions in between as it could do if multiple VCode instructions were used. A high level overview of the Cranelift compilation pipeline is shown in Figure 1.

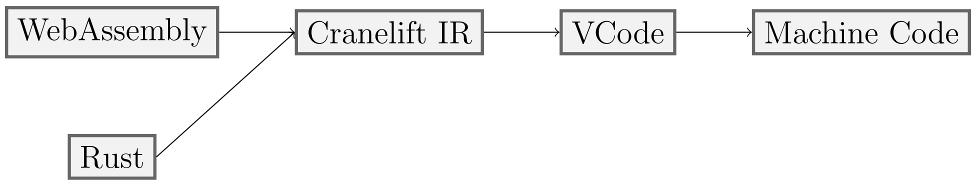

Figure 1: Cranelift compilation pipeline

```
function u0:0(i32, i64 vmctx) -> i32 fast {
block0(v0: i32, v1: i64):
    v3 = imul v0, v0
    v2 -> v3
    jump block1

block1:
    return v2
}

; Exported as "do_call_square"
function u0:1(i64 vmctx) -> i32 fast {
    sig0 = (i32, i64 vmctx) -> i32 fast
    fn0 = u0:0 sig0

block0(v0: i64):
    v2 = iconst.i32 2
    v3 = call fn0(v2, v0)  ; v2 = 2
    v1 -> v3
    jump block1

block1:
    return v1
}
```

Listing 2: Cranelift IR for the example WebAssembly module

Cranelift IR is a control flow graph in SSA form (RWZ88). As an example consider Listing 2, which is the Cranelift IR produced for code shown in Listing 1. `function u0:0` here is `square` (Cranelift IR does not contain human readable names). The client has to map any names into numerical identifiers. The function consists of the basic blocks. The initial block that is being executed is `block0` and it gets the function arguments as block arguments. Cranelift IR uses block arguments in the place of phi nodes like commonly used with SSA form IR's including LLVM IR. There is no strong reason to choose block arguments. It was picked very early on in the lifetime of Cranelift and there has not been any good reason to change it. In the body of `block0`, `v3 = imul v0, v0` multiplies the argument with itself and names the result `v3`. `v2 -> v3` aliases `v2` to `v3` such that any reference to `v2` gets turned into a reference to `v3`. `jump block1` then jumps to `block1`. If `block1` had a block argument, it would have been specified when jumping like `jump block1(v0)`. In `block1` `return v2` then returns `v2`. In `function u0:1` (`call_square`) the `v3 = call fn0(v2, v0)` calls `function u0:0`.

ISLE (Fal23) ("instruction selection/lowering expressions") is a pattern matching domain specific language used by Cranelift to define the lowering from Cranelift IR to the backend specific VCode. It was introduced when it became clear that hand writing pattern matching is error prone. Using ISLE allows running checks for various classes of bugs that were common before the introduction of ISLE. It also enables automatically generating optimal pattern matching trees. A simplified excerpt of the lowering rules for the `iadd` instruction is given in Listing 3. As can be seen in the example, an ISLE rule consists of a priority, a pattern matcher for one or more Cranelift instructions and VCode that should be produced in case of a match.

```
(rule -5
    (lower (has_type (ty_32_or_64 ty) (iadd x y)))
    (x64_lea ty (to_amode_add (mem_flags_trusted) x y (zero_offset)))
)

(rule -4
    (lower (has_type (fits_in_64 ty) (iadd x (sinkable_load y))))
    (x64_add ty x y)
)
```

Listing 3: ISLE rule for iadd

Semantically speaking for each instruction that needs to be lowered, each rule is processed from the highest priority (`-5` and `-4` in this example) and the first rule which matches the Cranelift instruction(s) and satisfies all additional conditions will be picked and the resulting VCode instructions will be emitted and any applicable helper functions will be called as necessary to emit further VCode instructions. An example where a helper function is used rather than directly emitting the VCode instructions is in the `call` instruction lowering, shown in Listing 4. Here `gen_call` is a helper function which performs all necessary steps to handle the calling convention like putting arguments in the right registers and stack slots.

```
(rule
    (lower (call (func_ref_data sig_ref extname dist) inputs))
    (gen_call sig_ref extname dist inputs)
)
```

Listing 4: ISLE rule for call

The actual ISLE implementation is smarter than the above description and for example produces a pattern matching tree with equivalent behavior which only looks at potentially applicable rules.
As the last step, each VCode instruction is translated to a sequence of bytes representing generally one, but sometimes more, machine code instructions.

2.3 The rustc_codegen_cranelift project

Rustc_codegen_cranelift (cg_clif) (all23) is a project by the author of this thesis to use the Cranelift code generator as backend for the Rust compiler in the place of the default LLVM based backend. While Wasmtime's reason for using Cranelift is security, cg_clif's reason for using Cranelift is that it is faster at producing unoptimized code than LLVM. The Rust compiler (rustc) is often critizized for substandard performance. In the 2023 rust annual survey 45% of the respondents considers compile time performance high priority (Tea24). Because of considerable work has been done on optimizing rustc and the introduction of several features like check-only builds and incremental compilation that reduce the amount of work that needs to be done by the compiler during development. Furthermore, work is currently on the way to parallelize the frontend of rustc in addition to the already parallelized backend. Still there is a lot of room for further improvement. Cg_clif was created to help with improving performance of rustc during development by accelerating the generation of unoptimized machinecode. It is not meant to be used in production as Cranelift lacks many optimizations that LLVM does support for optimized builds. Cg_clif has been adopted by the Rust project and has recently started shipping as optional part of nightly releases (development builds released every night) (Bar23).

2.4 Exceptions

Exceptions are a programming construct for handling errors. It allows a function to throw an exception and then from the point of the throw the exception bubbles up the call stack until a function catches the exception, at which point execution continues from the point the exception was caught. Each function on the call stack that the exception bubbles past can run arbitrary code to for example deallocate resources or restore invariants. While not every programming language supports exceptions (C does not), many mainstream languages do like C++, C#, Java and Python.
To allow languages that support exceptions to be compiled to WebAssembly, there is a pre-existing proposal to add exception handling support to WebAssembly (was23a). While it is possible to support exceptions without this proposal, doing so comes at the cost of having to check if an exception happened on every function call, which adds a non-trivial amount of overhead. To be able to implement this proposal in Wasmtime, support for exceptions in Cranelift is necessary.
Rust also has exceptions in the form of panics. Unlike most languages with exception support, in Rust panics are expected to happen very rarely and almost always be the result of a bug somewhere, instead for regular error handling the Result type is expected to be used. Because exceptions are

rare in Rust, the lack of exception support in Cranelift has not been much of an issue for the cg_clif project. Still some programs like rust-analyzer regularly emit panics as a means of unwinding the stack. In addition Rust RFC 2945 (all19) added support for interfacing with exceptions thrown by C++. As such cg_clif would benefit from exception support in Cranelift too.
For C++ exceptions on native platforms there are a couple of different implementations. In most cases it consists of two parts. Firstly, a language independent part which defines how to unwind from the current call frame to the caller's call frame. This includes which registers it needs to load from which stack locations or move from other registers, and how the cleanup/catch code is invoked. Secondly, a language dependent part which determines how to determine if a specific exception should be caught by the current function and where the code to cleanup or catch the exception is located. LLVM only supports the C++ version of this language dependent part and as such rustc uses it too. GCC supports the language dependent part for a couple of languages including C++, Go and Ada. The language independent part is implemented by the so called unwinder which is generally shared by all libraries in the program, while the language dependent part is implemented in a so called personality function provided by the language implementation.
For the language independent part there are three common basic variants:

- Landingpad based: When an exception occurs the unwinder will set everything up such that it looks like the function that threw an exception returned normally with the exception data passed as return value, except that execution continues at a landingpad within the caller rather than immediately after the call instruction. Effectively a call can return to multiple locations, either the regular return location, or a dedicated exception handling location. The landingpad can continue execution like normal, in which case the exception is caught, or it can after cleanup call the unwinder again to unwind another call frame. Itanium unwinding which used on most UNIX systems including Linux follows this pattern. It is called this because it was originally introduced by Intel for use on the Itanium architecture. While the Itanium architecture failed to meet its promises, use of Itanium unwinding has spread to most CPU architectures due to its performance advantage over SjLj based unwinding that was previously used.

- Funclet based: When an exception occurs the unwinder will call a so-called funclet like a regular call, except that it passes the stack pointer of the call frame for which the funclet does cleanup as argument. Catching the exception and resuming regular execution requires explicit interaction with the unwinder. While unwinding, the original call stack is preserved all the way until the point the exception is caught. SEH unwinding which is used on Windows follows this pattern.

- SjLj based: Setjmp/longjmp unwinding is what used to be common on UNIX until it was replaced with landingpad-based unwinding due to the significantly better performance for regular execution without exceptions getting thrown. This method is no longer in common use.

With the exception of SjLj based unwinding all aforementioned variants of unwinding store the information necessary for unwinding in side tables like `.eh_frame` and `.gcc_except_table` for Itanium unwinding. In most cases for space efficiency it contains byte code which needs to be evaluated to find the locations of all stored registers, but (BKZN19) evaluated compiling these instructions down to machine instructions for faster execution.

2.5 Related work

LLVM (LA04) is a compiler framework which is in some sense similar to Cranelift but favors runtime performance over compilation speed and does not have the same focus on correctness as Cranelift. It has support for exception handling. The Cranelift IR extension proposed in this paper is inspired by the way LLVM IR represents exceptions. Unlike the mechanism proposed in this paper, LLVM is only able to generate a couple of different kinds of exception handling tables.
In (VPF+23) the ISLE rules of Cranelift are verified. It is an example of the focus on security development of Cranelift has.
In (Din00) the landingpad based unwinding mechanism that C++ uses on Itanium is introduced. This work has since been adapted by most UNIX systems to work on all CPU architectures they support. The mechanism we propose can be used to implement the unwinding mechanism of (Din00).
(DFP+23) shows how the method by which exceptions are implemented in C++ can break the control-flow integrity (CFI) exploit mitigation. An exception handling implementation for a WebAssembly runtime should ideally be designed to mitigate the attacks described by this paper.
Rustc represents the control flow for panics in it's MIR IR (a control flow graph based IR which is the input to all codegen backends of rustc (Mat16)) in two separate ways: For running destructors when unwinding, each call basic block terminator has two edges/targets. One for regular returns and one for unwinding (the cleanup edge). The one for unwinding points to a basic block marked as cleanup which calls all necessary destructors and finally a resume terminator continues unwinding. The exception value itself is implicitly forwarded by the codegen backend from the cleanup edge to the instruction where unwinding is resumed. For catching exceptions a `catch_unwind` compiler intrinsic is used which calls a function and if an exception unwinds through `catch_unwind` a second function is called with the exception value as argument. The return value of `catch_unwind` indicates if an exception was caught or not.

3 Design

Both Wasmtime and cg_clif constrain the chosen design for exception handling in Cranelift. In the case of Wasmtime, the constraint is security. In the case of cg_clif the constraint is the ability to use the same unwinding mechanism as C++ to be able to handle C++ exceptions as well as to throw rust panics across C++ code. The use case of cg_clif would be satisfied by only supporting the same unwinding mechanism as C++. This is what both the LLVM and GCC backend of rustc currently do. This would however make it entirely unusable for Wasmtime. The unwinding mechanism used by C++ is very flexible to support other languages, but also comes with a lot of complexity due to this like support for distinguishing exception types, for separate handling of cleanup, catching exceptions and aborting when an exception happens. In addition it is not designed for untrusted input unlike Wasmtime. This means that there is a risk that using it for Wasmtime would enable a sandbox escape in case a bug in the unwinder or unwind table generation of Cranelift allows overwriting memory outside of the sandbox. In the past Wasmtime used the system unwinder for generating backtraces, but moved away from this to using frame pointer based unwinding for security.

To satisfy both constraints the design we chose does not dictate a specific format for the unwinder metadata, instead opting to support any landingpad style unwinding mechanism and leaving it up to the user of Cranelift to produce the unwinding metadata from the side tables produced by Cranelift during compilation. For time constraint reasons we did choose to reuse the existing code of Cranelift which emits tables to unwind the current call frame. This code can only emit this information in the `.eh_frame` and Windows SEH formats. Adding support to allow the user to emit this information in arbitrary formats is possible without significant changes to Cranelift. In addition we chose not to support the funclet style unwinding mechanism used by Windows SEH. Supporting this requires significant changes to the way Cranelift IR as well as the backend is structured as there is currently no way for multiple functions to share a stack frame, nor for there to be a difference between the stack frame of the funclet itself and the stack frame of the associated regular function. For Wasmtime supporting this is also not required as it would implement its own unwinder and thus would be able to choose a landingpad style unwinder on all platforms.

Summarizing, this design aims to add a generic way of expressing landing-pad style unwinding in Cranelift IR, capable of handling both the needs of WebAssembly and Rust. From the output of Cranelift after compilation, the user can generate the desired unwinding code and metadata, such as what is necessary for C++ exceptions in the case of Rust.

3.1 Extensions of Cranelift IR

New `invoke` and `invoke_indirect` instructions will be added to CLIF IR. These act like `call` and `call_indirect` except that they are terminators and have an additional `BlockCall` list argument with all possible landingpads that can be reached when unwinding from the called function. More than one landingpad may be given if the personality function wants to choose between them depending on for example the exception type. The `call` and `call_indirect` instructions can not be reused as these are not basic block terminators. As such they can not have a successor basic block that is jumped to when an exception is thrown by the callee. Execution will always continue at the instruction directly after the call. By having the new instructions be terminators, they can have multiple successors. One for regular return and any number of successors for exceptions.

Each landingpad has zero or more additional parameters compared to what is listed in the `BlockCalls`. Each of these extra parameters can be set by the personality function before jumping to the landingpad by writing to one of the registers dedicated by the Itanium unwinding ABI for the target platform for landingpad arguments.

3.2 Cranelift changes

Cranelift will be changed to ensure that each landingpad follows ABI conventions like which registers are preserved at entry. In addition it will generate the correct metadata for unwinding the stack in a way that restores all registers that need to be restored.
After compilation Cranelift will return in the `call_sites` field of `MachBufferFinalized` (the type containing all output produced by Cranelift after compilation) a list of the code offsets immediately after the call instruction of each invoke combined with a list of code offsets to the start of each landingpad for the respective invoke instruction. This list can be used to construct an LSDA (the data read by the personality function, for example `.gcc_except_table` for C++) to be read by the personality function in the case of cg_clif and whatever other simpler format is chosen in the case of Wasmtime.
As an example the code in Listing 5 will call `fn1` at line 6 and if `fn1` unwinds, execution will jump to the landingpad at `block2`, which immediately calls `_Unwind_Resume` to continue unwinding.

```
 1 function %f(i32, f32) -> i32 system_v {
 2     fn0 = %func0(i32) -> i32 system_v
 3     fn1 = %func1(i64, i64) system_v ; _Unwind_Resume
 4
 5 block0(v0: i32, v1: f32):
 6     invoke fn0(v0), block1, [block2(v1)]
 7
 8 block1(v2: i32):
 9     return v2
10
11 block2(v3: f32, v4: i64, v5: i64):
12     call fn1(v4, v5)
13     trap unreachable
14 }
```

Listing 5: Cranelift IR example for the invoke instruction

4 Implementation

In this chapter we describe the necessary changes to Cranelift and cg_clif for adding unwinding support.

4.1 Cranelift IR

The first step was extending the Cranelift IR to add the `invoke` and `invoke_indirect` instructions. This was done in `cranelift/codegen/meta/src/shared/instructions.rs` and required adding support for new instruction formats to fit these instructions in to the IR and the parser and printer for Cranelift IR. Once this was done the IR verifier needed to be adopted to account for the extra arguments passed by the `invoke` and `invoke_indirect` instructions to the target blocks that were not mentioned in the `BlockCalls`. And finally a couple of helper functions and optimization passes needed to be adopted to account for the new instructions.

4.2 Use invoke in cg_clif

The next step was to extended cg_clif to emit `invoke` instructions and cleanup blocks for panics where the MIR IR says these need to be emitted. To keep the initial implementation simpler, the exception data passed to the landingpad is immediately stored on the stack rather than kept in an SSA value. Changing this would likely yield a very small performance improvement for panics but is otherwise harmless. In addition Rust dictates that exceptions unwinding into Rust code which has unwinding support disabled should abort the process. This has not yet been implemented and will need to be implemented before upstreaming all changes made for this thesis.

4.3 Interpreter support

After that the Cranelift IR interpreter was extended to support exceptions. This allowed testing of all parts before machinecode generation without having to worry about miscompilations in the backend. This required fairly intrusive changes as the interpreter did not support some of the functionality required by cg_clif such as libcalls (Function calls with hardcoded knowledge in Cranelift. Examples include memcpy and ceilf.) and thread local storage. It also required adding support for serializing entire Cranelift IR modules and merging them together to the cranelift-module interface for the interpreter to be able to get Cranelift IR for the entire program even when multiple parts are compiled separately. In the end it did uncover an optimization pass which needed to be adapted for the new instructions as well as a couple of mistakes in the IR produced by cg_clif for exceptions. In particular the `unreachable_code` optimization pass needed to be adapted to consider the tables used in `invoke` and `invoke_indirect` instructions such that it doesn't remove them due to thinking they are unused. In addition the `remove_constant_phis` optimization pass needed to stop removing the return values and exception values passed to the `invoke` and `invoke_indirect` target blocks.

4.4 Implementation in the Cranelift backends

As the next step support for the new instructions in the Cranelift backends for the x86_64 and arm64 architectures was added as well as support for returning the necessary information used for emitting the unwind tables to the user of Cranelift.
Most of the backend changes necessary are in the ABI handling code. Cranelift reuses a fair amount of code for the ABI handling between the various backends leaving only the actually architecture specific details in the individual backends. The most important shared types are:

- `SigData` which contains the calling convention used as well as the computed locations of all arguments and return values at the point of the call instruction. And after the changes done for implementing exception support also the locations of the landingpad arguments.
- `Caller` which contains all state used for lowering a `call(_indirect)` or `invoke(_indirect)` CLIF instruction to VCode that moves all arguments to the right locations and loads the return values. It also contains methods to emit the right machine instructions.
- `Callee` which contains all state used for implementing the calling convention on the callee's side reading all arguments from the right locations and storing the return values correctly. It also contains methods to emit the right machine instructions.
- `ABIMachineSpec` is an interface implemented by each backend which contains the code for creating the `SigData` such as assigning arguments to the right registers and call stack locations as well as functions to create the machine instructions which `Caller` and `Callee` emit.

The first step was modifying `SigData::from_func_sig` to compute the locations of the landingpad arguments. To make the prototype easier to implement, this code currently has hard coded that two 64-bit integer arguments are used. This works as the ABI on x86_64 and arm64 only allows integer arguments and does not allow more than two arguments. It also does not come at a runtime performance cost as the registers which are passed are caller-saved. `Caller::emit_call` was then modified to compute the set of clobbered registers by directly subtracting the list of registers "defined" by the call instruction from the list of caller-saved registers as opposed to computing it at a higher level by subtracting the list of returned values defined in `SigData`. The latter would miss the landingpad arguments as definitions and Cranelift requires that every register is marked either as definition or as clobber, but not both. Next a `Caller::gen_landingpad_argval` method was added for defining the return values that will be used as landingpad arguments by the individual backends. Then a new `gen_invoke_common` function was added, which is like the existing `gen_call_common` function used for lowering CLIF IR calls to VCode, except that it also handles adding the landingpad arguments as return values for the call instruction using `gen_landingpad_argval` and forwarding those as arguments to the landingpad blocks as block arguments.
The next step was wiring up support in the arm64 and x86_64 backends. Only the steps taken for the arm64 backend are listed so as not to take up too much space. The same steps apply to the x86_64 backend with a couple of minor differences. For the arm64 backend wiring up exception support required adding some code to `AArch64MachineDeps::compute_arg_locs` (implementation of `ABIMachineSpec`) to compute the register locations of the landingpad arguments. Next up was adding ISLE lowering rules for the `invoke` and `invoke_indirect` instructions. This involved calling

the `gen_invoke_common` function that was added earlier to emit the actual call followed by emitting a jump instruction to the regular return destination to terminate the block.
The final step was modifying `MachCallSite` to store the locations of all landingpads for consumption by the user of Cranelift. This also required adding a new `MachCallSiteFinalized` struct which is like `MachCallSite` except that it stores the final `CodeOffset`s rather than a symbolic `MachLabel`s that do not have a fixed location yet. With this the changes to the arm64 backend were in place.

4.5 Write unwind tables

The final step was adding support for writing the required unwind tables to cg_clif. cg_clif already has support for writing the `.eh_frame` section with the information necessary to produce backtraces. For this the gimli DWARF debuginfo reader/writer library is used (PC24). The only necessary changes here were adding a reference to the personality function and LSDA to `.eh_frame` and generating the LSDA. For the personality function, the existing `rust_eh_personality` made for the LLVM backend was used. As a consequence the LSDA is required to be in the same format as emitted by LLVM. This is stored in the `.gcc_except_table` section. (LLVM uses the same format as GCC for compatibility with GCC's implementation of the C++ standard library.) After validating with a simple personality function that exceptions can be successfully triggered, a writer for the `.gcc_except_table` section was written based on the description of the format found in the LLVM source code. Care was taken to produce an identical section as LLVM to reduce the risk of introducing bugs. Finally the writer was wired up to cg_clif.

4.6 Results

With all this in place cleaning up works, catching exceptions works and the majority of the panic tests in the rustc test suite pass. The only failing tests are couple of tests for aborting when C++ exceptions pass through places that they are not allowed to pass through. The fix for this is a couple of relatively simple changes to cg_clif. As this is not required for the evaluation of the final product, implementing the fixes is left for once the changes are submitted to be upstreamed to Cranelift and cg_clif.

5 Experiments

In this chapter we will quantify the performance of the implemented exception handling support on Rust code on the x86_64 and arm64 architectures. To this end a number of experiments have been performed on two different platforms:

- An AWS EC3 virtual machine. This was done on a c6g.8xlarge instance, which uses an AWS Graviton2 arm64 processor with 32 vCPU and 64GB RAM.
- An x86_64 laptop with an Intel Core i3-7130U CPU running at 2.7GHz.

The virtual machine is shared with multiple people, some of whom have been using the virtual machine concurrently with the executions of the benchmarks, and in general when running in the cloud, other VM's on the same machine can cause performance to change over time. All performance benchmarks are relatively quick, so most likely would not have been affected too much by this.
The raytracer and CSS parser benchmarks have been performed using hyperfine (`https://github.com/sharkdp/hyperfine`). This benchmarking tool invokes a command a specified amount of times (in this case 10 times) and measures wall time of each execution, reporting the average.
Measurements for LLVM without exception support are omitted as at the time of benchmarking the cargo build system had a bug that made it impossible to recompile the Rust standard library with LLVM with exception support disabled on targets that support exceptions.

5.1 Raytracer

The first benchmark is comparing the time to render an image using a simple raytracer implementation in Rust compiled with rustc_codegen_cranelift both with and without exception support enabled. This will give an indication of the overhead of introduced by exception support if any. The results are shown in Figure 2 and 3. The difference is within the margin of error on x86_64 and arm64 both with and without optimizations enabled.
Furthermore the binary size overhead has been compared. This shows between 4.7% and 6.0% overhead of supporting exceptions over not supporting them as can be seen in Table 1.

	No exceptions	Exceptions
arm64 unopt	16.4 MiB	17.3 MiB (+5.1%)
arm64 opt	13.0 MiB	13.8 MiB (+6.0%)
x86_64 unopt	17.3 MiB	18.1 MiB (+4.7%)
x86_64 opt	14.0 MiB	14.8 MiB (+5.5%)

Table 1: Size of the raytracer executable

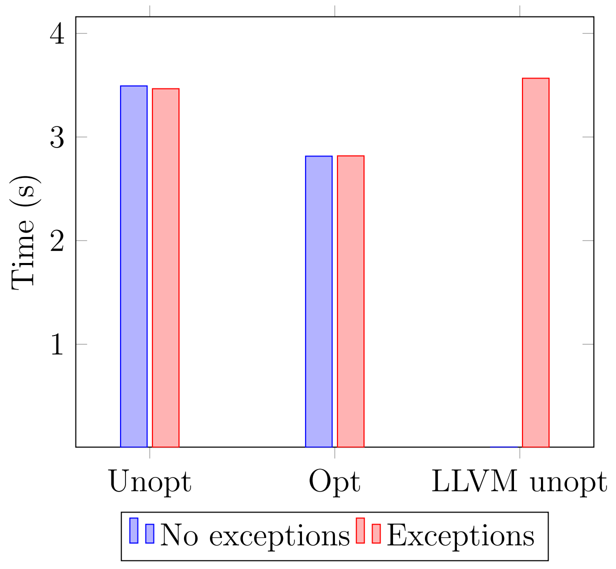

Figure 2: Raytracer execution time on x86_64

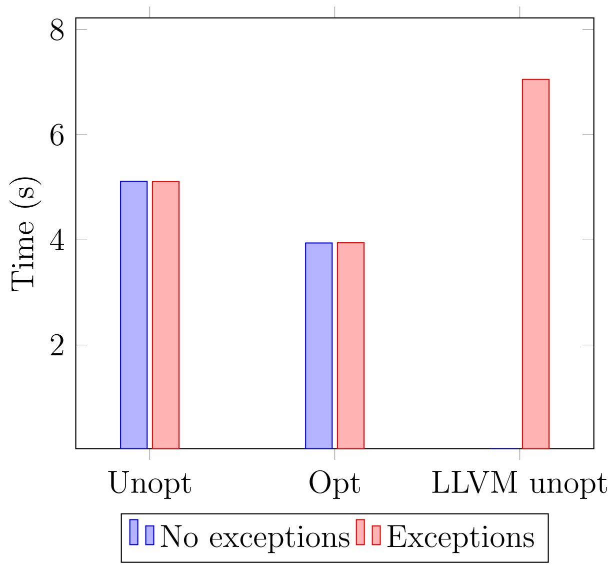

Figure 3: Raytracer execution time on arm64

5.2 CSS parser

Next up a benchmark adapted from the rustc-perf benchmarking tool used by the Rust project has been tested. This benchmark makes use of the `lightningcss` CSS parser to parse a stylesheet. The stylesheet originated from facebook.com. The results are shown in Figure 4 and Figure 5. Again the difference in execution time between an executable compiled with and one compiled without exception support has been measured. Unlike with the raytracer this did show a difference in execution time. On the arm64 machine with optimizations disabled this showed a 2.7% slowdown. With optimizations enabled the difference was within the margin of error. On the x86_64 machine however with optimizations disabled the difference was within the margin of error while with optimizations enabled a 3.1% slowdown was observed.

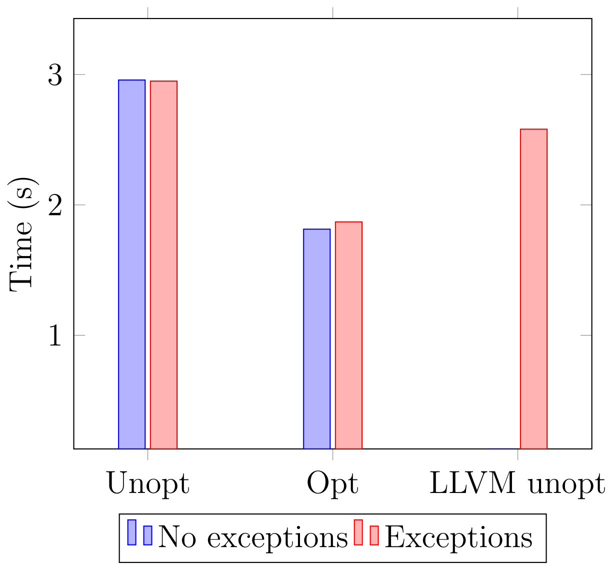

Figure 4: CSS parser execution time on x86_64

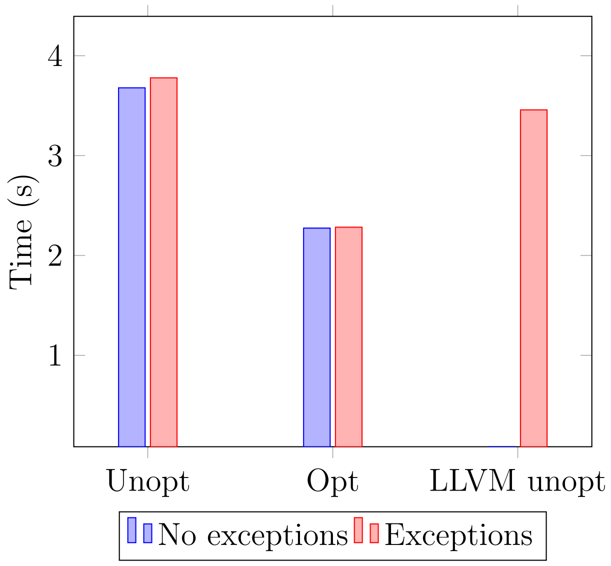

Figure 5: CSS parser execution time on arm64

5.3 Unwinder experiments

To compare the performance of exceptions between different implementations of an unwinding runtime, the demo example of Cranelift has been adapted to support throwing and catching exceptions. Unlike with rustc_codegen_cranelift this is not tied to a single unwinding runtime and personality function. Four different versions were implemented:

- The original version is used as baseline.
- The system unwinder on Linux with the same C++ personality function as rustc_codegen_cranelift.
- The system unwinder on Linux with a specialized personality function (henceforth referred to as the fast personality) only usable for this example program and not for C++.
- A custom unwinder which does the minimum necessary to get working on the example program and will crash when needing to unwind multiple call frames. This unwinder is only to give an indication of the lower bound on the overhead unwinding can give.

The results are shown in Figure 6. Any unwinding mechanism at all results in a 20% performance hit on making a lot of calls in a loop. The fast personality function gives about a 10% performance improvement when unwinding past a lot of frames, or less when only unwinding a single frame. The custom unwinder gives about a 30x improvement over the system unwinder for unwinding a single frame. A decent amount of the speedup is likely because registers are not correctly restored while unwinding. Exactly measuring the possible performance improvement for a correctly implemented custom unwinder is beyond the scope of this thesis.

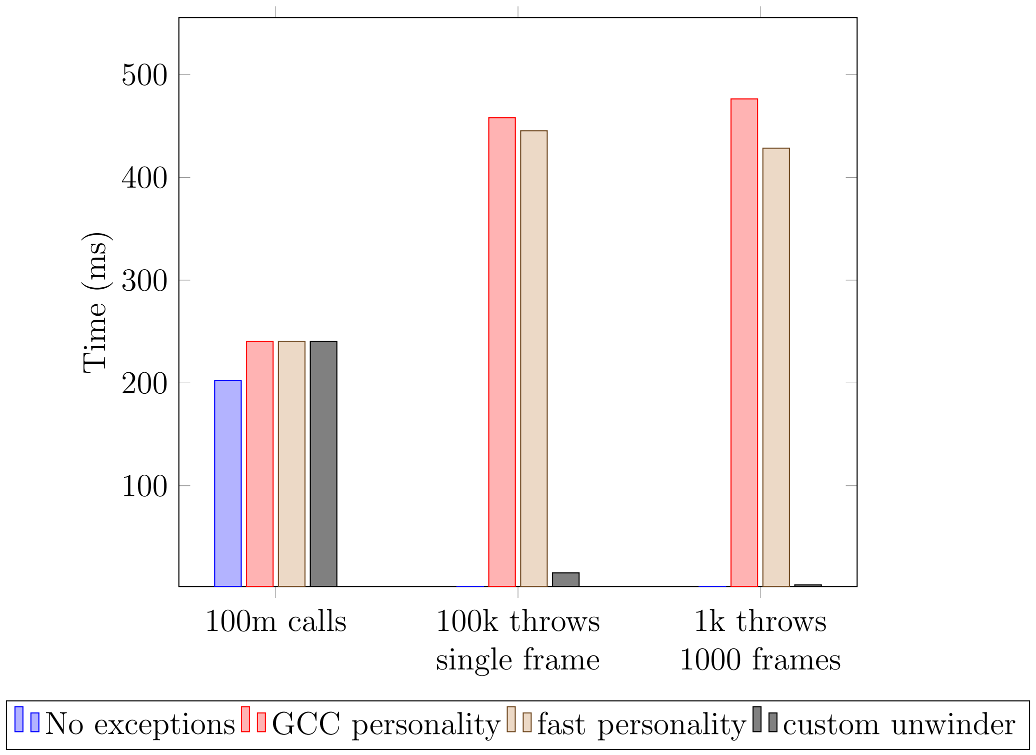

Figure 6: Unwinder experiments execution time on arm64

6 Conclusion

In this thesis we have designed and implemented a mechanism to implement exception support in the Cranelift code generator in a generic way suitable for both interoperability with native C++ exceptions and for use in WebAssembly runtimes. We have shown that it is a viable implementation of unwinding in rustc_codegen_cranelift with only a minor performance cost of up to 3% outside of microbenchmarks.
All code implemented as part of this thesis are released as open source and will be upstreamed to Cranelift and the main branch of rustc_codegen_cranelift. As future work, exception handling support will be implemented in Wasmtime.

References

[all19] ALL, Kyle J S.: *2945-c-unwind-abi - The Rust RFC Book*. https://rust-lang.github.io/rfcs/2945-c-unwind-abi.html. Version: 2019. – Accessed: 2023-12-21

[all23] ALL, Björn Roy B.: *Cranelift codegen backend for rust*. https://github.com/rust-lang/rustc_codegen_cranelift. Version: 2023. – Accessed: 2023-12-21

[Bar23] BARON, Björn R.: *Progress report on rustc_codegen_cranelift (Oct 2023)*. https://bjorn3.github.io/2023/10/31/progress-report-oct-2023.html. Version: 2023. – Accessed: 2023-12-21

[BKZN19] BASTIAN, Théophile ; KELL, Stephen ; ZAPPA NARDELLI, Francesco: Reliable and Fast DWARF-Based Stack Unwinding. In: *Proc. ACM Program. Lang.* 3 (2019), oct, Nr. OOPSLA. http://dx.doi.org/10.1145/3360572. – DOI 10.1145/3360572

[DFP+23] DUTA, Victor ; FREYER, Fabian ; PAGANI, Fabio ; MUENCH, Marius ; GIUFFRIDA, Cristiano: Let Me Unwind That For You: Exceptions to Backward-Edge Protection. In: *NDSS*, 2023

[Din00] DINECHIN, Christophe de: C++ Exception Handling for {IA64}. In: *First Workshop on Industrial Experiences with Systems Software (WIESS 2000)*, 2000

[Fal23] FALLIN, Chris: *Cranelift's Instruction Selector DSL, ISLE: Term-Rewriting Made Practical*. https://cfallin.org/blog/2023/01/20/cranelift-isle/. Version: 2023. – Accessed: 2024-03-28

[Fit22] FITZGERALD, Nick: *Security and Correctness in Wasmtime*. https://bytecodealliance.org/articles/security-and-correctness-in-wasmtime. Version: 2022. – Accessed: 2023-12-21

[GFD22] GACKSTATTER, Philipp ; FRANGOUDIS, Pantelis A. ; DUSTDAR, Schahram: Pushing Serverless to the Edge with WebAssembly Runtimes. In: *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022, S. 140–149

[Hic19] HICKEY, Pat: *Lucet Takes WebAssembly Beyond the Browser — Fastly*. https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime. Version: 2019. – Accessed: 2023-12-21

[HRS+17] HAAS, Andreas ; ROSSBERG, Andreas ; SCHUFF, Derek L. ; TITZER, Ben L. ; HOLMAN, Michael ; GOHMAN, Dan ; WAGNER, Luke ; ZAKAI, Alon ; BASTIEN, JF: Bringing the Web up to Speed with WebAssembly. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA : Association for Computing Machinery, 2017 (PLDI 2017). – ISBN 9781450349888, 185–200

[LA04] LATTNER, Chris ; ADVE, Vikram: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California, Mar 2004

[Mat16] Matsakis, Niko: *Introducing MIR*. https://blog.rust-lang.org/2016/04/19/MIR.html. Version: 2016. – Accessed: 2024-03-28

[PC24] Philip Craig, Nick Fitzgerald et a.: *A library for reading and writing the DWARF debugging format*. https://github.com/gimli-rs/gimli/. Version: 2024. – Accessed: 2024-07-02

[RWZ88] Rosen, B. K. ; Wegman, M. N. ; Zadeck, F. K.: Global value numbers and redundant computations. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '88* (1988). http://dx.doi.org/10.1145/73560.73562. – DOI 10.1145/73560.73562

[Tea24] Team, The Rust S.: *2023 Annual Rust Survey Results*. https://blog.rust-lang.org/2024/02/19/2023-Rust-Annual-Survey-2023-results.html. Version: 2024. – Accessed: 2024-03-28

[Var18] Varda, Kenton: *WebAssembly on Cloudflare Workers*. https://blog.cloudflare.com/webassembly-on-cloudflare-workers/. Version: 2018. – Accessed: 2023-12-21

[VPF+23] VanHattum, Alexa ; Pardeshi, Monica ; Fallin, Chris ; Sampson, Adrian ; Brown, Fraser: Lightweight, Modular Verification for WebAssembly-to-Native Instruction Selection, 2023. – Accessed: 2023-12-21

[was23a] *Exception Handling Proposal for WebAssembly*. https://github.com/WebAssembly/exception-handling/. Version: 2023. – Accessed: 2023-12-21

[Was23b] *Wasmtime: A fast and secure runtime for WebAssembly*. https://wasmtime.dev. Version: 2023. – Accessed: 2023-12-21

[Web] Rossberg, Andreas (Hrsg.): *WebAssembly Core Specification*. https://www.w3.org/TR/wasm-core-1/. – Accessed: 2023-12-21

A Source code

All source code associated with this thesis is available online at the following locations:

- Cranelift unwinding implementation: `https://github.com/bjorn3/wasmtime/tree/bsc-unwinding-final`
- rustc_codegen_cranelift interpreter support: `https://github.com/bjorn3/rustc_codegen_cranelift/tree/bsc-unwinding-interp`
- rustc_codegen_cranelift unwinding implementation: `https://github.com/bjorn3/rustc_codegen_cranelift/tree/bsc-unwinding-final`
- Writer for the GCC LSDA format: `https://github.com/bjorn3/eh_frame_experiments`
- Unwinder experiments: `https://github.com/bjorn3/cranelift-jit-demo/blob/bsc-unwinding-simple-invoke`
- Raytracer benchmark: `https://github.com/ebobby/simple-raytracer/tree/804a7a21b`
- CSS parser benchmark: `https://github.com/bjorn3/rustc_codegen_cranelift/tree/bsc-unwinding-final/rustc_perf_bench_css`