William Robson

A JIT compiler for OCaml bytecode

Computer Science Tripos Part II, 2021

Selwyn College

27 May 2021

Declaration of originality

I, William Robson of Selwyn College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. I am content for my dissertation to be made available to the students and staff of the University

Signed William Robson

Date 27 May 2021

Proforma

Candidate number:	**2352A**
Title	**A JIT compiler for OCaml bytecode**
Examination:	**Computer Science Tripos Part II, 2021**
Word Count:	**11,936**[1]
Code Line Count	**12,523**[2]
Project Originator:	**The author & Dr Timothy Jones**
Project Supervisor:	**Dr Timothy Jones**

Original Aims of the Project

The project aimed to implement a just-in-time compiler for OCaml bytecode to x86_64 machine code on Linux. The compiler would be integrated into the existing OCaml runtime and would support all functionality but debugging and introspection. The project would collect a suite of benchmark programs and build an automated method to use them to compare performance to the existing compiler. It was hoped that the compiler would lead to faster execution times than the existing OCaml bytecode interpreter.

Work Completed

All original project aims were completed. In addition to this, I built a significant extension: a second compiler, which focuses on performance of the compiled code. The two compilers are integrated with the second used for functions that are called often.

Special Difficulties

The author received a two week extension due to a hospitalisation close to the deadline.

[2] Computed using `texcount -1 -inc dissertation.tex | cut -d "+" -f 1`.

[2] Underestimate only counting Rust code. Computed by `cloc src/rust/ocaml-jit*` and taking the Rust number for code lines (excluding blank and comment lines).

Contents

1 Introduction

This project introduces two new just-in-time (JIT) compilers to x86_64 machine code for the OCaml programming language — one focusing on correctness and compile speed and the other on performance of the compiled code. When combined these compilers are capable of executing every OCaml program, including bootstrapping OCaml itself, and outperform the OCaml bytecode interpreter in the majority of cases.[1]

These compilers are integrated dynamically: the first compiler is used for all code and the second compiler is used for functions detected to be called often. In this document, I will refer to the faster but less intelligent compiler as the 'first' or 'initial' compiler and the slower but more optimising compiler as the 'second' or 'optimising' compiler.

1.1 Motivation

OCaml is a compiled functional programming language with a strong, static type system. The OCaml compiler has two main backends: a target-specific native-code compiler and a compiler to bytecode, which is interpreted by a program called `ocamlrun`.

The default bytecode interpreter in OCaml is significantly slower than the output of the native-code compiler. Although most users of OCaml use the native-code compiler, the bytecode compiler is still widely used.

It is the only target supported by the OCaml 'toplevel' read-evaluate-print-loop (REPL). JIT compilation techniques to decrease bytecode execution time are particularly useful here.

This project demonstrates how JIT compilation techniques can be applied to OCaml while retaining the bytecode format and semantics. As the 'toplevel' can only use bytecode, the JIT could lead to better performance in interactive OCaml environments.

Additionally, this project serves to demonstrate two contrasting approaches to writing a JIT compiler in Rust and shows how they can be integrated together for performance better than either individually.

1.2 Related work

This project has been attempted at least twice before: with OCamlJit in 2004 [13] and OCamlJit2 in 2010 [11]. On advice of my supervisor I decided not to closely read these papers until after implementation; my first compiler independently ended up with a very similar design to that of OCamlJit2 and my second was more sophisticated than either project.

This project fits into the space of JITs more generally as a method-based (as opposed to trace-based) compiler [3].

1.3 Work performed

The project achieved, on schedule, three core goals as set out in the proposal:

[1] Faster for 32/36 of the tested benchmarks: mean speedup = 1.46×, $\sigma = 0.40$.

1. There is a JIT compiler implemented into the existing OCaml source replacing the interpreter with all functionality but debugging and introspection.
2. There is a comprehensive and automated suite of benchmarks built comparing its performance to other alternatives.
3. It performs favourably compared to the original interpreter on benchmark programs.

This gave time for a signficant extension:

1. I have built a second compiler using more sophisticated analysis to generate faster machine code.
2. The first compiler dynamically decides to execute the second compiler at runtime.

The project is primarily written in the Rust and C programming languages. It replaces the interpreter component of the existing OCaml runtime.

1.4 Summary of results

The system is capable of correctly compiling and executing all OCaml bytecode instructions and every program I have tested. The features not supported are the debugger and backtraces (which are not used by default in the existing runtime). The entire OCaml compiler test suite passes, with the exception of tests of the debugger and backtraces. The JIT compiler can be used in the OCaml compiler's self-bootstrapping.

The new JIT-using runtime significantly increases performance for most of the tested programs. The mean speedup achieved across the benchmark suite of 36 different programs was 1.46×.

2 Preparation

The OCaml bytecode interpreter has no collected specification or documentation. Although aspects of the runtime system are explained in the OCaml documentation from the perspective of implementing a foreign-function interface (FFI) to C, most necessary implementation details can only be found through study of the compiler's source code. For this reason, I had performed a significant amount of work prior to the formal project start, investigating whether the project was even feasible. Investigation into OCaml's low-level details continued to be required throughout the project.

After this work was complete, I was in a position to write and then immediately start on the project plan given in the proposal. This plan set out the steps required to build a simpler version of the initial compiler. Work proceeded according to the optimistic schedule leaving time to build the significantly more sophisticated optimising compiler.

2.1 Relation to the Tripos

This project built on concepts from many areas of the Tripos. The most directly relevant courses are Part IB Compiler Construction and Part II Optimising Compilers, but the project required reading beyond the scope of either.

This project also used a large amount of low-level knowledge of the x86_64 architecture and Linux, as covered in Programming in C and C++, Computer Design and Operating Systems.

OCaml was not covered in the Tripos beyond its use as the implementation language for Compiler Construction but shares many similarities with Standard ML, as was covered in Foundations of Computer Science[1]. Although this project operated at a much lower level of abstraction, knowledge of the source language was crucial for understanding what a sequence of bytecode instructions was doing.

2.2 Technology choices

The Rust programming language is not covered in the Tripos but I was already very familiar with it from personal projects. The language's support for algebraic data types and pattern matching (with the compiler verifying all cases are matched) was particularly useful for the project.

x86_64 assembly was new to me but did not prove too challenging to learn given my experience with a variety of other architectures.

2.3 The OCaml bytecode interpreter

OCaml bytecode is interpreted by a stack-based abstract machine, optimised for patterns in functional programming.

[1] When I took the course — it now uses OCaml.

2.3.1 Data representation

OCaml has a uniform data representation for its *values*. Values are 64 bits long. Pointers to heap-allocated values are stored directly — but due to alignment they are guaranteed to have a 0 in the LSB. Integers are 63 bits long with the unused LSB storing a value of 1.

Every heap-allocated value has a 64-bit header containing the number of words stored (called the wosize) and a u8 tag. Most tag values correspond to a block, which can be thought of as a tagged tuple containing wosize fields. Each field is treated as an OCaml value by the garbage collector.

There are special tag values and cases in the garbage collector for elements like floating point numbers (which, due to the uniform representation, must be stored boxed on the heap), closures, objects and f64/u8 arrays.

2.3.2 Registers

The OCaml abstract machine uses five registers:

- sp is a stack pointer for the OCaml stack, which is used extensively in the bytecode.

- accu is an accumulator register used for the return values of functions and primitives as well as for most arithmetic operations.

- env holds a pointer to the current closure (an OCaml block), which is used for referencing closure variables (stored as fields in the block).

- extra_args is used to mark the number of arguments passed on the stack.

- pc contains the pointer to the bytecode instruction to be interpreted next.

In addition to these registers there is the Caml_state struct whose fields can be considered as another 30 or so registers. They are used by the interpreter mainly for supporting the garbage collector, exceptions and growing and reallocating the OCaml stack.

2.3.3 Function-calling model

The interpreter descends from the ZINC Abstract Machine (ZAM) [9] through various iterations of the Caml system. The most significant feature of these abstract machines for this project is the model used for calling functions and dealing with the functional-language concept of partial application.

Eval-apply vs push-enter

In a 2005 talk at the KAZAM workshop [8], the creator of OCaml, Xavier Leroy, describes the concept of a distinction between the *eval-apply* and *push-enter* model. These models were originally due to Simon Peyton Jones [6][10]. This distinction has to do with how functions deal with taking multiple arguments, and is particularly relevant to functions taking curried arguments.

In the eval-apply model — followed by most imperative programming languages and the OCaml native-code backend — a function has a set number of arguments it takes. If a caller provides fewer or more than the required arguments (partial application or calling a function returned by a function) the caller must contain the code to handle these cases.

By contrast, in the push-enter model the callee must support any number of arguments passed to it. This is the method used by the OCaml bytecode compiler and interpreter. The mechanism for doing this is somewhat intricate and becomes relevant in this project in Section 3.5, where it is explained in more detail.

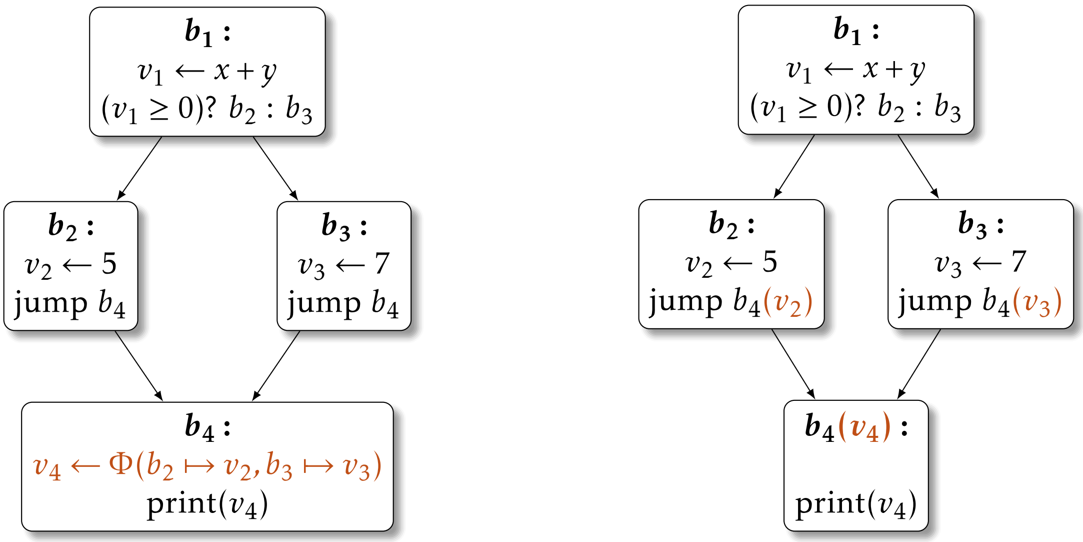
2.3.4 Garbage collection

OCaml is a garbage-collected language — memory is managed by the runtime and released once it is no longer reachable from any other live object by the garbage collector (GC). The OCaml garbage collector is precise — it needs to be able to identify exactly the set of values at any point in the program. This requirement was a significant source of complication, especially towards the end of the project.

2.4 Compiler concepts

A *basic block* represents a sequence of instructions with only one entry and exit — all instructions in the block will be executed in order.

These blocks are typically combined to make a data structure called a *control flow graph*. Each vertex represents a basic block and the directed edges represent the potential flow of control between basic blocks. Figure 2.1 shows example control flow graphs.



(a) Phi nodes

(b) Block parameters

Figure 2.1: Comparison of phi nodes and block parameters for SSA forms.

2.4.1 SSA form

A useful intermediate form used in compilers is static single assignment (SSA) form. If a program is in this form, every variable is assigned to only once and only binds a single immutable value.

This form is very useful to compiler writers as it can simplify the presentation and implementation of many optimisations.

In order to support conditional branching in the program, the form needs some way to mark choosing between values to use depending on the path taken through the program. The typical method[2] uses special phi node in the successor block to mark these cases.

This project uses an alternative method for this: *block parameters*. Here the blocks appear as if they are functions taking arguments and the values to use for the arguments are provided by the predecessor block in the branch instructions — see Figure 2.1 for an example. This is a newer and slightly cleaner formulation — but in typical uses the two representations are isomorphic.[3]

2.5 x86_64

Argument registers	rdi, rsi, rdx, rcx, r8, r9
Return registers	rax, rdx
Stack alignment	16-byte at call
Callee-saved registers	rax, rdi, rsi, rdx, rcx, r8, r9, r10, r11
Caller-saved registers	rbx, rsp, rbp, r12, r13, r14, r15

Table 2.1: Summary of the System V x86_64 calling convention

x86_64 is a large CISC (complex instruction-set computer) architecture descending from the Intel 8086 processor. Assembly listings in this project use Intel [`mov eax, 1`] rather than AT&T [`movl $1, %eax`] syntax.

This project targets Linux only, which uses the System V calling convention. A summary of the System V calling convention is given in Table 2.1 — the relevant details for this dissertation are that up to six 64-bit arguments can be passed to functions in registers, that up to two values may be returned by a function and that there are a larger large number of caller-saved registers.

2.6 Requirements analysis

The project plan set out ambitious success criteria.

1. There is a JIT compiler implemented into the existing OCaml source replacing the interpreter with all functionality but debugging and introspection.
2. There is a comprehensive and automated suite of benchmarks built, comparing its performance to other alternatives.
3. It performs favourably to the interpreter.

To avoid scope creep, I decided on some additional requirements:

1. The only target is x86_64 Linux; make no attempt at portability.

[2] As covered in the Optimising Compilers course.

[3] In fact, block parameters are slightly more general as different values can be passed from the same predecessor block — such as if both branches of a conditional branch entered the same block with a different value. This is rare in practice.

2. For the initial compiler, correctness > completeness > performance.

The last of these requirements was changed during the implemenation of the second compiler: performance was considered more important than completeness.

2.7 Dependencies and licences

As is common within the Rust ecosystem, I made use of third-party dependencies. There are many transitive dependencies that I have not explicitly chosen linked in to the final binary. My project integrates heavily with the OCaml runtime, so is licensed under LGPL 2.1 too.

Rust crate dependencies are under various licences. A full list is given in Appendix B; all licences are compatible with LGPL 2.1.

Some of my Rust dependencies needed to be tweaked slightly; their source is vendored in to the project tree. My tweaks there are licenced under the licences of the modified crates.

2.7.1 Dynasm-rs

The initial compiler uses `dynasm-rs` [4], which works as a Rust macro, turning assembly instructions into pre-assembled code, and calls into a simple runtime for relocation. Its design is based on that of `DynASM` used in the `LuaJIT` project [12].

2.7.2 Cranelift

The second compiler is designed around the cranelift [1], library which is a low-level retargetable code generator with an emphasis on use in JITs[4]. It is written in Rust and so benefits from an API designed for the language.

2.7.3 Sandmark

I based my benchmark suite on the excellent Sandmark project by OCaml Labs at Cambridge. The project consists of benchmark sources, build scripts (using `dune`), a benchmark runner, compiler definitions (using `opam`) and a complicated `Makefile` to tie it all together. I made some larger changes to the tool to support bytecode benchmarks (the project initially only benchmarked the native-code compiler) but was able to reuse most of the machinery.

Sandmark is licensed under CC0 (it is effectively public domain).

2.8 Starting point

The starting point of this project was the OCaml compiler 4.11.1, Sandmark from the commit hash starting `09862492`[5], and the source of any other vendored dependencies (identifiable by being included in under directories called `vendor`). The patches made to vendored dependencies are so small that it is best to consider them not my work for the purpose of this assessment.

Some prototyping performed was done before project start. The first aspect of this work was modifying the Makefiles of the OCaml compiler to link with a Rust-produced static library and ensure they could both call functions from the other. The other component

[4] It's largest use is for JIT-compiling webassembly as part of `wasmtime` and Firefox

[5] `https://github.com/ocaml-bench/sandmark/commit/09862492680c296fd659ceef9b34035ab97f7fe6`

was a simple disassembler for OCaml bytecode to gain familiarity with the format. Both aspects of this work were extended during the project and the current remnants of that work are minimal — limited to some of the changes to the OCaml Makefiles and the list of opcodes in `src/rust/ocaml-jit-shared/src/opcodes.rs`). The original disassembler was replaced with a more sophisticated one[6].

2.9 Development methodology

This project was developed in an iterative manner most closely aligned with the Agile methodology. Much of the Agile manifesto is based around customer and team collaboration which is not as relevant for this largely solitary project. However, many of the principles are still useful for single-developer projects. There are many practices which call themselves Agile. I mean:

- Prioritise delivering working software as frequently as possible.
- Work on small achievable tasks in a tight development loop encompassing all stages of the development lifecycle (planning, implementation and testing).
- Favour responding to change over following a plan; use tests and static type systems to make refactoring existing code an inexpensive and safe process.
- Reflect on progress and process at regular intervals and adjust behaviour accordingly.

The advantage of the methodology for my project is it allows for changing requirements. Changing requirements are usually something to be avoided but this is not always possible. Although the overall project goals did not change, the project interacts at a low level with many complicated aspects of the existing OCaml runtime. I did not have the knowledge to create a detailed plan for dealing with these systems. Agile embraces changing requirements making it an effective choice for this project.

Working detailed plans introduces a risk of falling behind without realising or wasting time on over-engineering. To mitigate this problem I used two strategies:

1. Set deadlines for each major component of the project with my supervisor, leading to accountability and external motivation.
2. Agree the goals and tasks for each week in a meeting with my project supervisor, allowing us to keep track of the progress towards the major deadlines.

Development followed an rapid iterative cycle of experimentation, short plans, implementation and testing, all directed to solving a clearly defined goal. I found this a particularly effective methodology for this project and my style of working.

[6] Not covered in this document due to space constraints but it can be found in the `src/rust/ocaml-jit-tools` directory.

2.9.1 Testing

Automated testing was essential to the development of this project. I used unit tests where it was appropriate for small functions or modules. However, by far the most useful class of test was automated trace comparison against the exact behaviour of the existing interpreter.

As OCaml bytecode has no specification outside the behaviour of the interpreter it can be hard to know exactly what the behaviour should be in all cases solely from the interpreter source code. To get around this I used automated comparison testing: the VM state was printed at every instruction and automated tooling would compare the JIT's trace against the interpreter's. More details are given in Section 3.4.5.

To develop larger components, like the optimising compiler, before enough code was written to allow running the entire compiler, I made use of 'expect tests'. These consist of tests that compare output of components against a reference string included in the source. The expect-test runner will show the diff on failure and has support for 'promoting' the new version, replacing the reference string with the new output. In addition to detecting regressions these tests serve as good self-contained examples of components and are the source of most of the output examples in this document. They are somewhat detached from the exact implementation details while testing components in isolation which makes them useful from refactoring.

The OCaml test suite was used towards the later stages of completeness and failures were added as cases to my test suite as trace comparisons. This helped isolate and fix rare bugs in uncommon code.

2.9.2 Tools

All project code was stored in a single Git repository (a 'monorepo' model). GitHub was used to host the repository online. Experimental ideas were developed on branches to allow evaluation of different strategies. I occasionally used self-reviewed pull requests for large changes.

The `cargo` build system was used for all Rust code. Linking in to the OCaml runtime was achieved by modifying OCaml's build system (autoconf and hand-written Makefiles). Smaller automation was achieved with bash scripts with more complicated tools written in Rust. All of these components are tied together with a toplevel Makefile.

The project can be used as a custom `opam` switch making it easy to test the system under different compile-time options with the entire ecosystem of OCaml dependencies — this is used in the implementation of the benchmark suite.

Data analysis is performed using the standard Python stack (pandas, matplotlib, jupyter, etc.). A frozen list of Python dependencies is at `benchmarks/requirements.txt`.

I used a combination of neovim, Visual Studio Code (with `rust-analyzer`) and CLion as text editors/IDEs for the project. Clippy was used for linting Rust code and all code was autoformatted using `rustfmt`.

2.9.3 Resources

I used my machine (Intel 10700K CPU, 32GB RAM on fast SSD) to develop the project. The project also works on my older and less powerful laptop but compilation is slower — but still usable as a backup system. The repository contains everything needed to bootstrap the project allowing for any Linux environment to be used in the case of disaster

3 Implementation

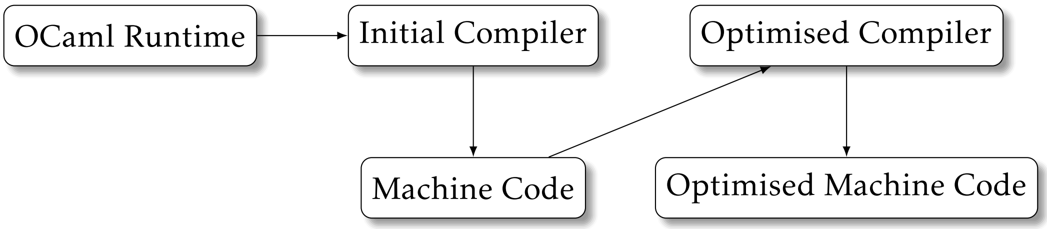


Figure 3.1: Control flow through the compiler

I have developed two new JIT compilers from OCaml bytecode to x86_64 assembly. The first compiler completely replaces OCaml's bytecode interpreter and is used for all programs and functions. The second is only used for functions that are called multiple times at runtime.

3.1 Overview

OCaml bytecode is executed using a program called `ocamlrun`. Both new compilers are implemented in a Rust static library that is linked with `ocamlrun`. `ocamlrun` hooks into this library when it loads bytecode and when it starts interpreting it.

If the JIT is enabled, either by setting an environment variable or compile-time setting, the bytecode load hook will trigger the **initial compiler**.

The initial compiler parses the bytecode into a stream of instructions. For each bytecode instruction it emits assembly with the same semantics to a buffer. After all code has been emitted (including headers and footers with shared routines) relocations are performed and the buffer is marked as executable.

When `ocamlrun` calls the hook to start interpretation, execution jumps to this assembly code. The operations are performed in the same order as the interpreter — except that every instruction has been inlined.

When an OCaml closure is applied[1], a closure execution count is incremented. Once this count passes a configurable threshold, the code instead branches to the **optimising compiler**.

The optimising compiler operates on the level of a single function. The code generated differs from the unoptimised machine code in that there is no concept of the OCaml stack and accumulator register; instead all values are stored in machine registers and the C stack. The function's code pointer is updated and any future calls (including the originally triggering call) will use the optimised implementation.

[1] All non-primitive function calls are translated to closure application.

3.2 Major milestones

In accordance with the development methodology (see Section 2.9) the focus throughout all aspects of the project was on working code that could be tested. However, it is worth noting the order of major milestones, where some degree of full-system completeness was achieved.

1. Execution of Hello World with the JIT.
2. Passing the OCaml test suite and bootstrapping the compiler.
3. Implementing the benchmark suite.
4. (Extension) Implementing the optimising compiler.

3.3 Repository overview

The project was developed using a 'monorepo' where everything for the project was contained in one Git repository. Some components are solely my work (marked **mine** below). Some components are 'vendored' (marked **vendor**) — where the source code is included to allow for small patches to be made. For OCaml and the sandmark benchmark suites, the patches are significant enough for me to consider it a fork containing both my work and the work of others (marked **fork**).

```
/
├── benchmarks
│   ├── sandmark ..................... fork: benchmark suite containing source & tooling
│   └── analysis .................. mine: Jupyter notebooks analysing benchmark results
├── docs ........................................... mine: source of this document
├── scripts ........................... mine: scripts to run tests and graph basic blocks
├── src
│   ├── rust
│   │   ├── ocaml-jit-shared ................ mine: shared code for the other two crates
│   │   ├── ocaml-jit-staticlib ............... mine: crate that is linked with ocamlrun
│   │   └── ocaml-jit-tools .................... mine: binary crate with standalone tools
│   ├── ocaml ................................ vendor: contains OCaml compiler 4.11.1
│   │   └── runtime ......................... fork: contains most large patches for the JIT
│   └── vendor .................. vendor: vendored rust dependencies with small patches
├── test-programs ..................... mine, vendor: OCaml source for test programs
└── vendor/no-aslr ......... vendor: A simple wrapper to run a program without ASLR
```

3.4 Initial compiler

The initial compiler was developed before any work on the optimising compiler was performed. In the process of implementing the initial compiler some modifications had to be made. These modifications will be covered later in Section 3.5 and this section will mainly cover the initial state.

3.4.1 Overview

The initial compiler consists of two major components — an instruction parser and an assembly emitter. The instruction parser converts bytecode into Rust data types and the assembly emitter translates the instructions into machine code.

Mapping of the OCaml abstract machine

Abstract-machine registers are mapped to x86_64 registers. A pointer to the `Caml_state` struct containing other global interpreter state is also stored in a register called `r_cs` for access to its fields with small code size. The mapping used is shown in Table 3.1.

OCaml register	x86_64 register
`r_env`	`r12`
`r_accu`	`r13`
`r_extra_args`	`r14`
`r_sp`	`r15`
`r_cs`	`rbx`

Table 3.1: Mapping of OCaml to x86_64 registers.

Note that all x86_64 registers mapped are callee-saved in the System V C calling convention. This means that no work needs to be performed spilling and restoring them from the C stack to call C/Rust primitives (either as part of `CCall*` instructions or supporting primitives written for the JIT).

Mapping of the PC

Bytecode pointers are mapped to native-code pointers and the interpreter's PC is replaced with the system's instruction pointer. This change is responsible for nearly all of the performance improvements of the JITed code. The benefits are

- Memory accesses are reduced — rather than loading operands and opcodes, they are baked in to the machine code.
- The CPU can predict execution and fill its pipeline further than it can with the use of native code pointers.
- Branch prediction in hot paths becomes more effective as each branch can be predicted differently.
- The CPU can schedule out-of-order execution more effectively across different instructions.

However, this approach does lead to more contention on the instruction cache as instruction implementation can no longer be shared, but this is offset by gains made elsewhere.

3.4.2 Hooks from OCaml

Code at `src/rust/ocaml-jit-staticlib/src/c_entrypoints.rs`

The existing OCaml runtime was modified to hook into the JIT at three points:

1. After a bytecode section is loaded.
2. Before a bytecode section is released.
3. When the interpreter is called.

In most programs there is only one section loaded (two if callbacks from C to OCaml are used). However, the interpreter is also used in the toplevel REPL where each section corresponds to a line of code entered by the user toplevel.

Compilation happens after the first hook and an entry is written to a global table containing the pointer to the buffer. The compiled code takes the form of a single large C taking no arguments, meaning branching into the code consists only of calling a function pointer.

3.4.3 Instruction parsing

The first stage in the pipeline is to parse the bytes into a stream of elements of the `Instruction` type.

The `Instruction` type

Code at `src/rust/ocaml-jit-shared/src/instructions/types.rs`

OCaml has 149 opcodes, each of which can take a different number of operands. Arguments and operands are all stored as `i32` values. Some opcodes are space-saving aliases for the composition of multiple simpler instructions. To simplify implementation I expand these aliases at parse time, so generating code only considers the simple primitives. There are about 60 such simplified instructions.

Rust has support for algebraic sum types called enums. I use this to store the `Instruction` type with one variant[2] per simple instruction.

Instructions such as integer comparisons and conditional branches have many variations varying only in the conditional used. The variants are extracted into a type allowing for a single instruction to represent all conditional branches.

Some instructions, like branches, have label arguments referencing the location of another opcode in memory. The `Instruction` type is polymorphic over the type of the label to allow for different label representations depending on use case.

Parser

Code at `src/rust/ocaml-jit-shared/src/instruction/parse.rs`

The parser is implemented using Rust iterators; the parser takes an iterator of `i32`s and produces an iterator producing values of type `Result<Instruction, ParseError>`.

This use of iterators makes the instruction parser consistently performant for large programs - consumers of the instruction stream take each instruction as they need rather than storing a vector of all instructions in memory.

In OCaml bytecode, jumps are encoded as offsets relative to the current PC. To simplify later uses, these are converted to absolute offsets relative to the start of the bytecode address.

[2] Also known as type constructor.

As single OCaml instructions may parse to more than one Instruction, the type contains a pseudo-instruction called LabelDef, which is emitted at the start of every OCaml instruction. This also allows a map to be be built from labels to locations in the translated code.

3.4.4 Code generation

Code at src/rust/ocaml-jit-staticlib/src/compiler/emit_code.rs

Code generation is the largest aspect of the compiler. It makes use of the dynasm-rs library. This library manages emitting machine code and relocation information, relocating and using mmap to mark the code as executable.

During this process dynasm-rs dynamic labels are used to set up relocations: these labels are defined before every bytecode instruction and can be referenced by any other instruction. dynasm-rs translates these at runtime into (machine PC)-relative jumps (as is usual on x86_64).

After all instructions are done, some shared code used by the instructions is emitted. dynasm-rs then performs relocations and uses mmap to mark the region of code as executable.

Simple example

The main code of the compiler itself is contained in a 2,000 line file (src/rust/ocaml-jit-staticlib/src/compiler/emit_code.rs). Most of this is taken up by a Rust large pattern match for each of the bytecode instructions. As a very simple example of what the code looks like, consider the implementation of the ADDINT opcode. It adds the value at the top of the OCaml stack to the accumulator and stores the result in the accumulator. In the original interpreter it is implemented:

```
case ADDINT:
    accu = (value)((intnat) accu + (intnat) *sp++ - 1); Next;
```

Note that the OCaml integer format means a decrement is required to keep a 1 is the LSB. In the compiler the relevant case is:

```
Instruction::ArithInt(ArithOp::Add) => {
    oc_dynasm!(self.ops
        ; add r_accu, [r_sp]    ; r_sp is alias for r13
        ; dec r_accu            ; r_accu is alias for r15
        ; add r_sp, BYTE 8
    );
}
```

This has exactly the same semantics as the interpreter source. At compile time, the macro component of dynasm-rs translates it to:

```
Instruction::ArithInt(ArithOp::Add) => {
    self.ops.extend(b"M\x03/I\xff\xcdI\x83\xc7");
    self.ops.push_i8(8);
}
```

In this way most translation from assembly to machine code is performed at compile time. The byte string above contains the machine code for those instructions. This strategy is very performant.

Branches

As an example of the label and relocation support, consider the `BranchCmp(Comp, i32, L)` instruction. It compares the current value of the accumulator to the `i32` constant using the condition (of type `enum Comp Lt, Gt, ...`). If the condition is true it branches to the label, otherwise it passes to the next instruction.

This is implemented:

```
Instruction::BranchCmp(cmp, i, l) => {
    let label = self.get_label(l);
    oc_dynasm!(self.ops
        ; mov eax, caml_i32_of_int(*i as i64)  // convert integer to ocaml
                                               // representation
        ; movsxd rcx, eax                      // sign extend to 64 bits
        ; cmp rcx, r_accu                      // compare with accumulator
    );
    match cmp {
        Comp::Eq => {
            oc_dynasm!(self.ops
                ; je =>label                   // jump if equal to label
            );
        }
        Comp::Ne => {
            oc_dynasm!(self.ops
                ; jne =>label                  // jump if not equal to label
            );
        }
        // .. other cases similar
    }
}
```

The macros translate it to:

```
Instruction::BranchCmp(cmp, i, l) => {
    let label = self.get_label(l);
    self.ops.extend(b"\xb8");
    self.ops.push_i32(caml_i32_of_int(*i as i64));
    self.ops.extend(b"Hc\xc8I;\xcd");

    match cmp {
        Comp::Eq => {
            self.ops.extend(b"\x0f\x84\x00\x00\x00\x00");
            // encodes PC-relative 4 byte dynamic relocation to `label`
            self.ops.dynamic_reloc(label, 0isize, 4u8, 0u8, (4u8, 0u8));
        }
        // ...
    }
}
```

This shows the assembler work still needing to be done at compile time — relocations.

Other cases

All other cases follow this pattern. Some more involved instructions call into C primitives I wrote, instead but most of them inline the operations directly in hand-written assembly. The main exception is function calling — this is covered in Section 3.5.

3.4.5 Implementation strategy

Implementation was incremental and highly test-driven. The initial focus was on building a system supporting only the opcodes required to run a hello world program. I then slowly expanded the complexity of programs, using them to drive the implementation of new instructions and fixing of bugs.

The implementation was remarkably efficient, proceeding according to my most optimistic plan. This is mainly due to the trace comparison tooling.

Trace comparison

There is no formal specification for the OCaml interpreter. The semantics of the interpreter are what `interp.c` and other files in the runtime say they are. Given this, I decided to build tooling to test the behaviour of my JIT-compiled code directly against the behaviour of the interpreter.

I added support for tracing after every instruction in both the existing interpreter and the JIT-compiled code. The log entry contains the instruction executed, state of all of the OCaml registers and the top five entries on the stack.

A wrapper program (in the `ocaml-jit-tools` crate) runs a specified program with tracing enabled twice simultaneously — one run uses the JIT and the other the existing interpreter. For every trace entry printed it compares the two lines. If there is a difference between the lines it shows a diff and then exits, as shown in Figure 3.2.

As many of the values are pointers there is a risk of non-determinism making this comparison fail. I used a small open-source wrapper program called `no-aslr` to disable ASLR[3]. In order to ensure that the Rust code doesn't cause them to become unaligned, I ran the compiler regardless of whether JITed code was enabled when tracing was enabled. These two things together worked well enough that all of the memory addresses were aligned and deterministic. This is unlikely to be true in general for all OS kernels and malloc implementations, but worked on Linux with glibc.

The only expected difference comes from the use of the machine PC rather than the bytecode PC – instruction pointers, like return addresses on the stack, could differ. This required a special case during the check.

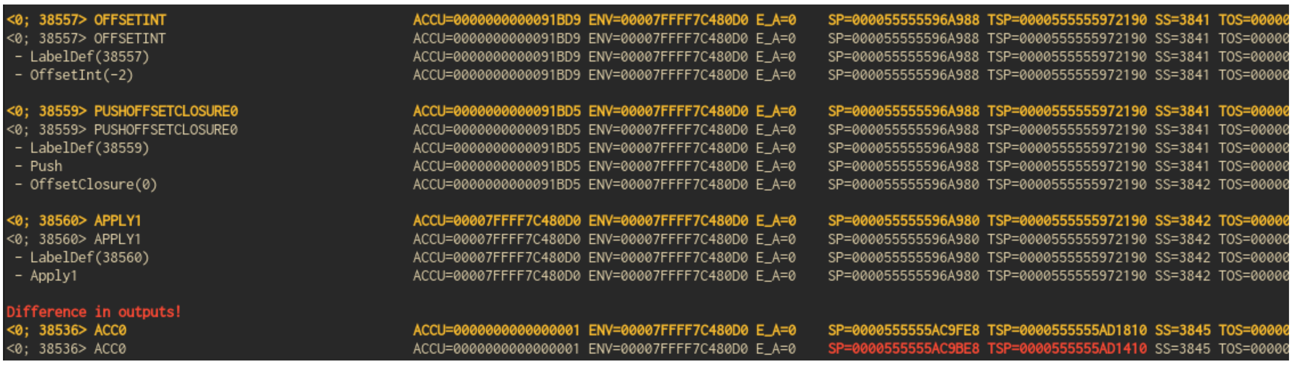
I wrote a script to run 11 test programs under trace comparison, failing if any of them failed. Running this frequently allowed me to test for regressions when making changes.

Final stages

Once I was happy that I had implemented every instruction, I started using the OCaml compiler's internal test suite. I discovered some subtle bugs and used it to add new test programs and fix them by trace comparison. One test heavily used callbacks from C to OCaml and I discovered my initial implementation was too slow.

I eventually managed to get nearly all tests in the test suite working — the only failures were testing the backtrace support and the debugger, which I had decided not to support

[3] GPLv2 licensed, `https://github.com/kccqzy/no-aslr`.



```
<0; 38557> OFFSETINT                ACCU=00000000000091BD9 ENV=00007FFFF7C480D0 E_A=0    SP=000055555596A988 TSP=0000555555972190 SS=3841 TOS=00000
<0; 38557> OFFSETINT                ACCU=00000000000091BD9 ENV=00007FFFF7C480D0 E_A=0    SP=000055555596A988 TSP=0000555555972190 SS=3841 TOS=00000
 - LabelDef(38557)                  ACCU=00000000000091BD9 ENV=00007FFFF7C480D0 E_A=0    SP=000055555596A988 TSP=0000555555972190 SS=3841 TOS=00000
 - OffsetInt(-2)                    ACCU=00000000000091BD9 ENV=00007FFFF7C480D0 E_A=0    SP=000055555596A988 TSP=0000555555972190 SS=3841 TOS=00000

<0; 38559> PUSHOFFSETCLOSURE0       ACCU=00000000000091BD5 ENV=00007FFFF7C480D0 E_A=0    SP=000055555596A988 TSP=0000555555972190 SS=3841 TOS=00000
<0; 38559> PUSHOFFSETCLOSURE0       ACCU=00000000000091BD5 ENV=00007FFFF7C480D0 E_A=0    SP=000055555596A988 TSP=0000555555972190 SS=3841 TOS=00000
 - LabelDef(38559)                  ACCU=00000000000091BD5 ENV=00007FFFF7C480D0 E_A=0    SP=000055555596A988 TSP=0000555555972190 SS=3841 TOS=00000
 - Push                             ACCU=00000000000091BD5 ENV=00007FFFF7C480D0 E_A=0    SP=000055555596A988 TSP=0000555555972190 SS=3841 TOS=00000
 - OffsetClosure(0)                 ACCU=00000000000091BD5 ENV=00007FFFF7C480D0 E_A=0    SP=000055555596A980 TSP=0000555555972190 SS=3842 TOS=00000

<0; 38560> APPLY1                   ACCU=00007FFFF7C480D0 ENV=00007FFFF7C480D0 E_A=0     SP=000055555596A980 TSP=0000555555972190 SS=3842 TOS=00000
<0; 38560> APPLY1                   ACCU=00007FFFF7C480D0 ENV=00007FFFF7C480D0 E_A=0     SP=000055555596A980 TSP=0000555555972190 SS=3842 TOS=00000
 - LabelDef(38560)                  ACCU=00007FFFF7C480D0 ENV=00007FFFF7C480D0 E_A=0     SP=000055555596A980 TSP=0000555555972190 SS=3842 TOS=00000
 - Apply1                           ACCU=00007FFFF7C480D0 ENV=00007FFFF7C480D0 E_A=0     SP=000055555596A980 TSP=0000555555972190 SS=3842 TOS=00000

Difference in outputs!
<0; 38536> ACC0                     ACCU=0000000000000001 ENV=00007FFFF7C480D0 E_A=0     SP=0000555555AC9FE8 TSP=0000555555AD1810 SS=3845 TOS=00000
<0; 38536> ACC0                     ACCU=0000000000000001 ENV=00007FFFF7C480D0 E_A=0     SP=0000555555AC9BE8 TSP=0000555555AD1410 SS=3845 TOS=00000
```

Figure 3.2: Output on trace comparison failure.

in my proposal. After this I successfully managed to bootstrap the compiler using the JIT which gave me a high level of confidence in the accuracy of the JIT-compiled code.

3.4.6 Omitted details

Although the basic structure as described holds, some details are omitted but appear in `emit_code.rs`.

- Callbacks from C to OCaml code require special handling.
- The compiler returns a pointer to the first instruction to support OCaml's metaprogramming `ocaml_reify_bytecode` function, used in the implementation of things like the toplevel `ocaml` program.
- Certain operations (like `ClosureRec(...)`) are involved enough that instead of inlining the definition in hand-written assembly, I push the registers to the C stack and call a C primitive to implement the operation (taking the registers as a struct).
- The compiler stores a persistent data structure of the sections to allow mapping bytecode addresses to machine-code addresses and allow for clean-up after a section is freed.

3.5 Dynamic recompilation

The initial compiler, although a large piece software, does nothing that could not inherently be done ahead-of-time with some more work on linking. The main benefit to operating as a JIT is it allows for OCaml toplevel sessions to be JIT-compiled. Adding dynamic recompilation allows for the system to discover hot paths in the code and optimise them more, which is something that requires information about runtime behaviour. This is a popular technique for many existing optimising JITs, like V8 for JavaScript. [14]

The structure of the bytecode means it is natural to make the optimised compiler operate at the granularity of a whole function, rather than a basic block or execution trace. Counting calls and only branching to the optimising compiler above a certain threshold is a natural method for determining hot functions and the method I decided to use.

However, allowing this required some changes to the initial compiler. In order to explain these changes I first give an overview of how the OCaml interpreter deals with efficient implementation of multiple arguments in the context of a functional language with partial application.

3.5.1 Existing OCaml calls

All functions in OCaml are implemented as closures — a heap-allocated tuple of a code pointer and a list of data. Function calls are implemented as:

```
check_stacks_and_signals(regs);
struct closure *c = (struct closure *) regs->env;
call_function(regs, f->code_ptr);
```

Note the actual call logic is performed in hand-written assembly and these structs do not exist anywhere - this and subsequent examples are just pseudo-code. However, it is easier to understand the logic at a higher level. regs is a pointer to a struct containing the OCaml registers — in the actual assembly these are machine registers.

Partial applications

OCaml uses curried arguments allowing for the use of partial application:

```
# let add a b = a + b;;
val add : int -> int -> int = <fun>
# add 2 3;;
- : int = 5
# let add3 = add 3;;
val add3: int -> int = <fun>
# add3 5;;
- : int = 8
```

In order to avoid making too many intermediate closures, OCaml has special support for this extensively used pattern. All arguments are passed in order on the stack and the extra_args register is used to mark how many arguments were used.

The number of arguments passed minus 1 is stored in the extra_args register. The compiler will insert a GRAB(req_extra_args) instruction at the start of any functions that take more than one argument. This instruction will cause an early return from the function if extra_args < passed_extra_args. The return value will be a closure itself — this represents partial application of curried functions. The data of this closure will be the arguments that were passed. The code pointer of this returned closure points to a RESTART opcode immediately preceding the grab, which moves the passed arguments off the closure's data section onto the stack, sliding them on top of any additional arguments passed. Otherwise, it will subtract required_extra_args from the extra_args register and continue executing.

This model is categorised as a push-enter model because it is the responsibility of the callee to deal with arity mismatches.

3.5.2 Function table

In order to support dynamic recompilation, it is necessary to store the call count somewhere. A naive method would be to store this in the closure as an extra data item, but this cannot support optimising cases where the same code is used with different closure environments.

Instead, I modified the code pointer to point to a structure in memory. This structure is shared between all closures using the same code. For now imagine it contains only one field: the code pointer. Calls become:

```
check_stacks_and_signals(regs);
struct closure *c = (struct closure *) regs->env;
struct function_metadata *f = c->code_ptr;
call_function(regs, f->code);
```

3.5.3 Moving the responsibility to the caller — eval/apply model

I do not wish for optimised functions to do any of the work described in Section 3.5.1. As is described later, most of the speed increases of the optimised compiler comes from completely eliminating use of the OCaml stack and the call method above makes heavy use of it.

Instead I shift the work to logically be performed by the caller. The way this is achieved is by adding a field storing `required_extra_args` to the function table and transforming the function call to look something like this:

```
check_stacks_and_signals(regs);
struct closure *c = (struct closure *) regs->env;
struct function_metadata *f = c->code_ptr;
if (regs->extra_args < f->required_extra_args) {
      // return partial application doing work of GRAB
} else {
      regs->extra_args -= f->required_extra_args;
      call_function(regs, f->code);
}
```

I then modified the assembly translation of GRAB to be a no-op. Identical work is performed but now it is the responsibility of the caller.

3.5.4 Call counts and dynamic recompilation

I added two new fields to the function-metadata struct: a reference to the original bytecode location of the function and a count/status variable. The count/status is initialised to 0. Calls become:

```
check_stacks_and_signals(regs);
struct closure *c = (struct closure *) regs->env;
struct function_metadata *f = c->code_ptr;
if (regs->extra_args < f->required_extra_args) {
      // return partial application doing work of GRAB
} else {
      regs->extra_args -= f->required_extra_args;
      if (f->count_status == -1) {
            // Call the optimised function
            call_opt(regs, f->code);
      } else if (f->count_status == THRESHOLD_CALLS) {
            // Run the optimising compiler then call the optimised function
            f->code = optimising_compiler(f->original_bytecode_location);
            f->count_status = -1;
            call_opt(regs, f->code);
      } else {
            // Increment count and call the unoptimised function
            f->count_status++;
            call_function(regs, f->code);
      }
}
```

3.5.5 Actual implementation

Even though most of the above snippets are pseudo-code, the function-metadata struct as listed with its four fields (code pointer, original bytecode location, required extra args, count/status) exists in the compilation. Before implementing the optimising compiler, the initial compiler was modified to generate, store and use it.

In order to generate this table, I added another pass to the initial compiler that discovers the bytecode locations of all closures. This is performed by iterating through all functions. When a `Closure(bytecode_location, num_vars)` bytecode operation is found (which creates a closure) the code pointer is inspected. By inspecting the first instruction found at the bytecode location, it is possible to find the value of the `required_extra_args` field.

After all the code is emitted, the function table is emitted. `dynasm-rs`'s relocations are used to load the initial code pointers into the function table and link to the struct in the closure table when compiling `Closure(...)` operations.

The net result of these changes is an extra pointer lookup being performed every time a function is called. However this is necessary to allow dynamic replacement of functions with optimised versions.

3.6 Optimising compiler

The previous section laid out a way to determine hot closures and dynamically call into an optimising compiler. This section describes the implementation of this optimising compiler and how calls are done.

The time left for the project meant building a complete x86_64 compiler backend (with register allocation, instruction scheduling, etc.) from scratch was infeasible. The usual toolkit used to avoid replicating this work is LLVM [7]. Although more typically known for its use in ahead-of-time (AOT) compilation, there is some support for use in JITs. However, its large size and complexity and AOT focus made it difficult to integrate.

Cranelift

Instead I used the cranelift library. Like LLVM it is a retargetable code generator with an IR. However, it has a number of different design decisions that made it much better suited for my planned uses:

- It is written in Rust so the API follows Rust idioms.
- It is designed primarily for JITs and focuses heavily on compilation performance.
- Although the support for garbage collection is not extensible, it fits well with the OCaml garbage collector.
- When I encountered any bugs or missing features I could easily submit and get merged patches to the project — I did this twice.

It didn't have all features I wanted but it is a good fit for my needs and was in a large part responsible for my being able to complete this ambitious extension.

3.6.1 Overview

At a high level the optimised compiler is a function from function bytecode to optimised code performing the same operations. The compiler consists of three major components in a pipeline.

1. Basic block conversion and stack-start analysis (Section 3.6.3).
2. IR generation from the basic blocks using the analysis (Section 3.6.5).
3. Compilation of IR to x86_64 assembly (performed by cranelift).

3.6.2 Optimisations performed

The largest change from the initial compiler to the optimised compiler is that it no longer generates code that uses the OCaml stack and accumulator registers. This is achieved by replacing these uses of the OCaml stack with machine registers that spill to the C stack when needed.

Arguments are passed to a function according to the System V C calling convention — the first argument is always the closure environment, `env`, and the remaining arguments are passed in registers (further details are given later in Section 3.6.4). The work done of Section 3.5 means the function will only be called with the complete number of arguments it is expecting.

The method used to remove the use of the stack is to convert the OCaml bytecode into an SSA form by keeping track of the contents of a virtual stack and accumulator as the closure is translated. Consider the simple OCaml function:

```
let add a b = a + b
```

The OCaml compiler produces this sequence of bytecode instructions:

```
Grab(1)          # Ensure 2 arguments passed
Acc(1)           # Load 2nd element from top of stack (arg a) to acc
Push             # Push acc onto top of stack
Acc(1)           # Load new 2nd element from top of stack (arg b) to acc
ArithInt(Add)    # Add the top of stack to the acc and pop it
Return(2)        # Pop 2 items, then return acc (now a + b)
```

Instead of translating all of these inefficient stack manipulations as the initial compiler does, the optimised compiler keeps track of the stack as it compiles. For an example of this, see Table 3.2.

3.6.3 Conversion to basic blocks

Code at `src/rust/ocaml-jit-shared/src/basic_blocks/`

The typical data structure used in optimising compilers is the basic block. Cranelift is no exception and the IR for a function consists of basic blocks containing instructions in SSA form with block parameters.

Initial state	Operation	Final state
`acc=?, stack=[b, a]`	`Acc(1)`	`acc=a, stack=[b, a]`
`acc=a, stack=[b, a]`	`Push`	`acc=a, stack=[a, b, a]`
`acc=a, stack=[a, b, a]`	`Acc(1)`	`acc=b, stack=[a, b, a]`
`acc=b, stack=[a, b, a]`	`ArithInt(Add)`	`acc=a+b, stack=[b, a]`
`acc=a+b, stack=[b, a]`	`Return(2)`	`acc=a+b, stack=[]`

Table 3.2: Translation of add.

However, the only information we have from the bytecode is a sequence of instructions with jumps. In order to move away from a heavy connection with the instructions the first step is to group them into basic blocks. These bytecode-instruction basic blocks are then translated into cranelift basic blocks.

Types

Code at `basic_blocks/types.rs`

A `BasicClosure` consists of a vector of `BasicBlock` along with some other data. A basic block has a vector of `BasicBlockInstruction` and a single `BasicBlockExit`.

Some `Instructions` map to a `BasicBlockInstruction` and others map to a `BasicBlockExit`. For example, `Instruction::Push` becomes `BasicBlockInstruction::Push` but `Instruction::BranchIf(label)` becomes `BasicBlockExit::BranchIf { then, else }`. The distinction encodes at the type level that each block only has one exit and entry point.

Stack starts

In addition to performing the conversion to basic blocks, the algorithm also keeps track of a concept I call the *stack start* of a basic block. Inspecting the bytecode compiler's source and validating with many programs finds an important invariant:

> For all paths leading from the entry block of a function to an instruction, the absolute stack size relative to the start of the function's stack frame is the same.

During the conversion the size of the stack on entry to each block stored. This is the *stack start* of the block.

Algorithm

Code at `basic_blocks/conversion.rs`

The algorithm consists of two iterations of depth-first search (DFS). The first finds all bytecode offsets that form the start of blocks. The second visits all these blocks working out stack sizes.

In order to describe this algorithm without too many OCaml-specific confusing details I will demonstrate it with a simplified model:

The first DFS pass is only necessary to deal with loop back edges. For simplicity let's assume this doesn't happen.

Let I be the set of all instructions. The program P is a list of instructions and can be indexed. P can be thought of as a function $\mathbb{N} \to I$. For simplicity, assume indices are consecutive[4].

Each instruction has four properties. With $\mathbb{B} = \{\textbf{true}, \textbf{false}\}$, define:

- $\delta(i) : I \to \mathbb{N}$ is the change to the stack pointer after executing the instruction. For example, `Push`, which pushes the acc to the stack has, $\delta(\texttt{Push}) = +1$.
- $\text{exit}(i) : I \to \mathbb{B}$ is **true** iff the instruction causes the block to end (jumps, returns).
- $\text{fallthrough}(i) : I \to \mathbb{B}$ is **true** iff $\text{exit}(i)$ and it can end up jumping to the following instruction. `BranchIf(branch_loc)` is fallthrough but `Branch(branch_loc)` is not.
- $\text{labels}(i) : I \to \mathcal{P}(\mathbb{N})$ is the set of all labels the instruction could end up jumping to (not including falling through). For example, $\text{labels}(\texttt{BranchIf(branch\_loc)}) = \{\text{branch\_loc}\}$. It is only relevant if $\text{exit}(i)$.

Under this model, the algorithm to find all the blocks and their starts is shown in Algorithm 1.

Additional operations performed during the search

The algorithm, as presented in Algorithm 1, shows only the basic logic of the search and storing of the stack starts for each block. The actual search also performs more work:

- When a block is visited, a vector is created to hold the parsed instructions.
- Blocks are numbered in reverse post-order (the natural order for basic blocks where, with the exception of loop back edges, blocks are visited in a way consistent with execution order).
- Every time an instruction is visited, any labels referencing other blocks are replaced with the block ids and a copy is added to the vector.
- At exit, all of the block metadata is stored in a `BasicBlock` struct and placed in a table of blocks referenced by the block number.
- The maximum stack size at any point in the function is computed. It is used later in the IR generation phase.

Appendix A shows an example of the output of the basic block conversion for a larger function with if statements.

3.6.4 Calling conventions

In the initial interpreter, all calling is performed using the OCaml interpreter's existing model. This passes all arguments on the OCaml stack and has the caller deal with partial application and tail calls.

We would like to avoid the stack in hot paths, so instead use a model where arguments are passed according to the System V calling convention. This is enabled by the changes made in Section 3.5.4.

[4] They're not, but it's fairly simple to handle.

Algorithm 1 DFS to find basic blocks and their starts

```
function FindBlocks($P, arity$)
    $seen \leftarrow \emptyset$
    $starts \leftarrow \{\}$
    function VisitBlock($initial\_index, initial\_stack\_size$)
        if $initial\_index \in seen$ then
            assert $starts[initial\_index] = initial\_stack\_size$
            return
        end if
        $starts[initial\_index] \leftarrow initial\_stack\_size$
        $i \leftarrow initial\_index$
        $s \leftarrow initial\_stack\_size$
        repeat
            $x \leftarrow P[i]$
            $s \leftarrow s + \delta(x)$
            if exit($x$) then
                for $j \in$ labels($i$) do
                    VisitBlock(j, s)
                end for
                if fallthrough($x$) then
                    VisitBlock(i + 1, s)
                end if
                return
            end if
            $i \leftarrow i + 1$
        until forever
    end function
    VisitBlock(0, arity)
    return $starts$
end function
```

Values stored in the closure's environment need to be accessible to the function, so the first argument is always a reference to the closure's environment. The remaining arguments correspond to the function's arguments when fully applied with the arguments it expects.

The function returns two values: the return value of the function and another value which corresponds to an `extra_args` offset used for doing tail calls.[5]

For example,

```
let add a b = a + b
```

is translated to a function with signature

```
(env: i64, a : i64, b : i64) -> (i64, i64)
```

The compiler can only process functions of up to five arguments, which corresponds to the maximum number of arguments that can be passed in registers in the System V calling convention. This simplifies the interface code and in practice nearly every function in the standard library can work in this way.

[5] The details of this are beyond the scope of this dissertation — it suffices to note this allows for tail calls even though cranelift doesn't support them.

- Calls from from optimised code to optimised code can be implemented with simple C function calls
- Calls from the stack-using output of the initial compiler require the caller to read the arguments from the stack and call the function with this convention
- Calls from optimised code to unoptimised code require writing the arguments back to the OCaml stack

All of these cases are implemented in hand-written assembly, optimising for the first case (optimised to optimised). Thus the hot path call sequence can avoid storing arguments in memory.

3.6.5 IR generation

Cranelift, much like LLVM, uses an IR with a binary, in-memory and textual representation. IR is stored in SSA form and each value has an associated type — such as I64 or F32.

IR generation starts by initialising the function header. Then every basic block is considered in turn, emitting cranelift IR.

Although SSA form is very useful, it can be somewhat difficult to translate from higher level forms where the same variable can hold multiple values. Cranelift provides some assistance with this during the process of building the IR. This is similar to LLVM's mem2reg pass but more direct, operating at the same time as IR is being built rather than as a separate optimisation pass.

This functionality is the key method for removing use of the OCaml stack and accumulator. Cranelift Variables are defined to correspond to each stack location used in the function and the accumulator. Each variable needs a unique integer id. This allows cranelift to perform dataflow analysis to track the loads and stores to the variable replacing them with SSA values.

```
let acc_var = Variable::new(0);
builder.declare_var(acc_var, types::R64);
let stack_vars: Vec<_> = (0..max_stack_size).map(|i| {
    let v = Variable::new(i + 1));
    builder.declare_var(acc, types::R64);
    v
}).collect();
```

When emitting IR, cranelift provides functions to get a reference to the current value of a variable or set a new value. The IR builder will track this against the control flow graph doing things like automatically inserting block parameters (see Section 2.4.1).

For example, PUSH (push current accumulator to stack) is handled in this way:

```
let current_acc = self.builder.use_var(self.acc_var);  // load acc
self.builder.def_var(                                  // store to sp[0]
      self.stack_vars[self.current_stack_size],
      current_acc
);
self.current_stack_size += 1;                          // increment current sp
```

Note the use of current_stack_size — as the function is compiled we keep track of the location of the stack pointer relative to the start of the frame. This is the purpose of computing the stack starts earlier; whenever we start emitting code for a basic block we can set the current stack size to what it would be on block entry.

There are utility methods for operations like pushing to and popping from the stack and modifying the accumulator to abstract the details of keeping the current stack size up to date

Example: add

Consider the ArithOp(Add) instruction from earlier. It adds the top of the stack to the current accumulator value (where adding integers in OCaml requires a decrement).

```
let a = self.get_acc_int();  // my utilities wrapping def_var and use_var
let b = self.pick_int(0)?;
self.pop(1)?;

// a is OCaml rep of x, b is OCaml rep of y
// a + b = (x * 2 + 1) + (y * 2 + 1) = (x + y) * 2 + 2
// result = a + b - 1 = (x + y) * 2 + 1
let added = self.builder.ins().iadd(a, b);
let result = self.builder.ins().iadd_imm(added, -1);
self.set_acc_int(result);
```

The code implementing this case is remarkably similar to the original interpreter — however, rather than directly performing the operation it is, instead making calls that trigger the emission of IR with the same semantics.

This is true also of the implementation of the other operations. Each instruction can be written in isolation using the concept of the stack pointer and accumulator, but rather than modify these at runtime we keep track of the state of these registers as we compile.

Complete example

To show this example in complete context consider the add function:

```
let add a b = a + b
```

```
Arity: 2 / Max stack size: 3
# Block 0 (stack_start = 2)
Acc(1)
Push
Acc(1)
ArithInt(Add)
Exit: Return(2)
```

As it is an OCaml function taking two arguments, it is compiled to a machine function taking 3 arguments — the closure environment (unused for this function) and the two original arguments, a and b.

It compiles to something like this[6]:

[6] The actual IR is larger as it uses R64 types and bitcasts for GC as described in Section 3.6.7 but the principle is the same.

```
function u0:0(i64, i64, i64) -> i64, i64 system_v {

block0(v0: i64, v1: i64, v2: i64):
    v3 = iconst.i64 0
    v4 = iadd v1, v2
    v5 = iconst.i64 -1
    v6 = iadd v4, v5
    jump block1

block1:
    return v6, v3
}
```

Despite the original function and the translation code making use of the stack and accumulator, there is no sign of this in the emitted IR. Values pass directly from the function parameters to nodes representing the arithmetic operations and to the return values. This is the result of our careful analysis of stack sizes combined with cranelift's mutable variable elimination

Larger example

Appendix A shows the translation for a function with multiple blocks, where variables could have different values due to if statements. Note that block 5 has a block parameter defined for the return value and jumps to it pass the specific values it could take. Inserting this block parameter was done automatically once cranelift determined it was needed.

Summary

The process described in this section is fairly intricate and it may not be immediately clear how everything fits together. The most important thing to take away from the section is that the accumulator and OCaml stack can efficiently be eliminated and replaced with SSA variables while keeping the same semantics and that the implementation of the cases is in very close correspondence with the OCaml interpreter's source.

3.6.6 Machine code translation

Cranelift compiles the IR to this machine code:

```
0000000000000000 <arith_add>:
   0: 55           push   rbp                  ; set up rbp chain
   1: 48 89 e5     mov    rbp,rsp              ;
   4: 48 01 d6     add    rsi,rdx              ; v4 = v1 + v2
   7: 48 83 c6 ff  add    rsi,0xffffffffffffffff ; v6 = v4 + (v5 = -1)
   b: 48 89 f0     mov    rax,rsi              ; 1st retval = v6
   e: 48 31 d2     xor    rdx,rdx              ; 2nd retval = v3 = 0
  11: 48 89 ec     mov    rsp,rbp              ; restore rbp
  14: 5d           pop    rbp                  ;
  15: c3           ret
```

Note in the System V calling convention, the first three arguments are passed in `rdi`, `rsi` and `rdx` respectively and return values are in `rax` and `rdx` respectively.

This code uses significantly fewer instructions than the output of the initial compiler would need due to the use of registers to hold intermediate values. No memory accesses are done at all outside of the entry and exits, compared to the multiple reads and writes to the OCaml stacks the interpreter and initial compiler output does.

The process is extensible to large functions with many basic blocks and complicated control flow patterns.

3.6.7 GC support

A key aspect of the OCaml runtime is its garbage collector. It is a precise, generational, stop-the-world tracing garbage collector. The relevant requirement for this project is that whenever the GC runs it must be able to exactly determine the set of all live values in memory. During garbage collection the values may change because the garbage collector will move things in the heap.

Safepoints

The GC will only run at certain locations — these are called safepoints. For OCaml, garbage collection occurs only on memory allocation if the minor heap (first generation) is full. This means every memory-allocation operation is a safepoint. Additionally, every call to another function, C primitive or signal handler could also trigger allocation, making them safepoints.

In the original interpreter and initial compiler's output, the `accu` and `env` registers need to be pushed on to the stack before the call, which is a safepoint. Afterwards the values are restored to account for the possibility garbage collection has caused them to change.

In the bytecode interpreter no particular effort is needed for this as every value is stored on the OCaml stack. However, for my compiled code, which does not use the OCaml stack, it is necessary to build something more involved:

Overview

1. During cranelift IR generation, I store everything that could contain a pointer to a GC-managed value (known as a *local root*) in a cranelift *reference type* (`R64`).
2. Cranelift performs live variable analysis, spilling and restoring of registers to the C stack and emission of *stack maps* at every GC safepoint.
3. My runtime stores the stack maps in a hash map keyed by the return address at time of the safepoint.
4. During GC, my runtime walks the entire frame pointer (`rbp`) chain. By looking at the return address and using the hash map to discover any stack maps, my runtime tells the OCaml GC about the location and value of any local roots that could contain pointers.

IR generation

Cranelift has support for precise garbage collection. It was originally created for webassembly reference types but luckily mapped well to the needs of OCaml. Precise (sometimes called *exact* or *accurate*) means that the GC is able to determine all roots and only the roots when tracing. The opposite is a *conservative* collector, which allows for an over-estimate.

In order to use it, my IR generation has to mark roots by storing them in values that have the R64 type. The R64 type means a 64-bit reference and is compiled identically to I64 — the purpose of this is to differentiate GC-managed pointers from other integers.

For this reason, anything that is an OCaml value and could be a pointer to the heap **must** be stored in a R64 type at every point the GC could trigger (a *safepoint*).

However, despite being almost equivalent in implementation to I64s, not all operations that can be done using I64 values can be performed using R64 values. This is only really a problem because of OCaml's mixed integer/pointer data representation. The integer add operation is not implemented for R64s but is needed to add two integers stored in OCaml's uniform representation.

I used cranelift's `raw_bitcast` (which does C-style type casting at the IR level) to temporarily convert R64 values to I64, perform any arithmetic, and then convert back.

The key invariant I had to hold to make this work was that these typecast R64-derived I64 were only alive for ranges shorter than the range between functions that could trigger garbage collection.

What cranelift does

At every safepoint (here function call), cranelift will ensure:

1. All live R64 values have a copy on the stack so the GC could find them.
2. No code or optimisation assumes that the old value of a R64 is still valid after the safepoint.

In order to determine the live reference types at every safepoint, cranelift performs a live variable analysis (LVA) pass. It then will spill and restore (push and pop) any reference type-containing machine registers to the C stack (other values will be on the stack already).

Stack maps

The output of this step is calls to a callback handler I provide with (native-code offset, stackmap) tuples.

The stack map is a set of offsets relative to the stack pointer at time of call where reference typed values are located. To store these entries efficiently, cranelift uses a bitset data structure. The interpretation of these offsets is shown in Figure 3.3 (where x is each member of the stack map's set):

Integration with OCaml

After compilation is finished, I translate the native-code offsets to absolute addresses by adding the pointer to the first instruction in the compiled function. I store all stackmaps in a hashmap indexed by the native code offset.

During garbage collection, the garbage collector has points where it scans all of the 'local roots'. It passes a callback function that takes two arguments: the address of the root in memory and the value at that address.

In order to use my return-address map, it is necessary to walk up the chain of stack frames in the same way as a debugger does. To do this I use the `rbp` register, which contains frame pointers. Supporting frame-pointer chains are optional in x86_64 calling conventions and usually omitted by optimising compilers. However, it is possible to tell the Rust compiler

```
        Stack
        +-------------------+
        | Frame 0           |
        |                   |
   |    |                   |
   |    +-------------------+ <--- Frame 0's SP
   |    | Frame 1           |
 Grows  |                   |
 down   |                   |
   |    | Live GC reference | --+--
   |    |                   |   |
   |    |                   |   |
   V    |                   |   x = offset of live GC ref
        |                   |   |
        |                   |   |
        +-------------------+ --+--  <--- Frame 1's SP
        | Frame 2           |
        | ...               |
```

This diagram is directly copied from Nick Fitzgerald's blog post [5] describing cranelift's GC. License: CC-BY-SA.

Figure 3.3: Interpretation of stack map entries

and GCC to not omit frame pointers. Cranelift emits frame pointers by default and I manually added using them to my hand-written assembly in the initial compiler. The way they are used means it is possible to walk the stack like a linked list by following the chain starting at the current rbp value:[7]

```
typedef void (*scanning_action) (value, value *);

CAMLexport void jit_support_scan_bp(scanning_action f) {
  uint64_t **bp;
  asm ("mov %%rbp, %0;" : "=r" (bp));

  while(bp != 0) {
    rust_jit_lookup_stack_maps(bp + 1, f);
    bp = (uint64_t **) *bp;
  }
}
```

The `rust_jit_lookup_stack_maps` function performs a search in the hashmap on the return address (bp + 1). If it finds a stack map, it will use the information to call `f` on each root found.

Summary

Garbage collection is incredibly difficult to get right. Luckily, cranelift had some support for it I could reuse. My use of it was non-standard but ultimately successful. Once I had this I

[7] Note inline gcc assembly is AT&T syntax (not the Intel syntax as for the rest of the document).

needed a way to walk through the C stack, which was done by enabling frame pointers and walking the stack using them.

I paid some performance penalty by using frame pointers and my hash map for looking up stack frames was unoptimised. However, once all components were in place the system worked remarkably well.

3.6.8 Exception handling

Exception handling was a particularly intricate detail to the project and explaining all the intricacies of how OCaml deals with exceptions would take many pages. In fact, the only bytecode instruction unsupported by the optimised compiler is the `PushTrap` operation, which is used in implementing `try-catch` operations.[8]

At a very high level, exception support in the existing interpreter is complicated by the fact the interpreter could call a C primitive that itself calls back into the interpreter in a different C frame. To support this, OCaml uses C `sigsetjmp` and `siglongjmp` functions. These functions work by saving and restoring the values of all machine registers and POSIX signal masks to a buffer, allowing for 'long' jumps up the stack.

Most OCaml exception raises do not require a longjmp as they do not need to pass through the callback stack. However, my optimised compiler emits most functions as C functions and, in general, exception raises do require a longjmp.

The buffer used to store the state to restore is rather large and in the naive implementation would need to be allocated by *every* function on the stack. This would have imposed an unacceptable performance penalty and thus needed to be addressed.

My solution was to note that I only needed two registers to completely restore the VM state state — `rbp` and the x86_64 instruction pointer. This is because the only places the longjmp could point to are at specific locations where all other registers can be seeded by other means. I ended up writing my own jump code to do this:

```
asm (
      "movq %0, %%rbp\n"
      "jmp *%1"
      :
      : "r" (buf->data.asm_saved.bp)
      , "r" (buf->data.asm_saved.pc)
);
```

[8] One particularly hard to debug problem involving exceptions led to a 2003 bug report in French where Xavier Leroy thanked the reporter for a particularly 'interesting' problem to fix. I had unknowingly re-introduced the conditions that led to that bug and unfortunately did not find it as interesting to fix.

4 Evaluation

There are three main criteria by which my compilers are evaluated: correctness of implementation, completeness of support for the OCaml language and runtime performance.

4.1 Correctness and completeness

Both correctness and completeness are verified by automatic testing. There are three main methods used for this:

- Trace comparison, as described in Section 3.4.5.
- The large existing OCaml compiler test suite.
- Automated Rust tests of components using `cargo test`.

Trace comparison guarantees the behaviour is observably identical to the existing interpreter. The OCaml test suite tests a large body of edge cases, ensuring correctness and completeness.

4.1.1 Initial compiler

The initial compiler fully covers the entire bytecode instruction set. The only features not supported are the debugger and backtraces. Implementing support for both of these features is feasible with some work but I did not consider them important enough to spend the time to get them working.

The compiler passes every test in the OCaml test suite except those that test backtrace functionality. The test suite is large and extensive, having accumulated many tests over OCaml's 25-year history.

The OCaml compiler is capable of bootstrapping itself from saved bytecode sources of a previous version of the compiler with my JIT enabled. The OCaml toplevel REPL works correctly allowing for interactive use of the JIT.

4.1.2 Optimising compiler

The optimising compiler has a slightly weaker completeness guarantee, due the limited time available to work on this extension; not all features are supported but those that are work correctly. The limitations are that compiler does not handle functions that take more than five arguments and the compiled functions cannot contain exception handlers (try-catch statements), although they can raise exceptions.

The compiler gracefully fails in these cases and falls back to using the existing machine code emitted by the initial compiler. For this reason, although the optimising compiler is not complete in isolation, the combination of the two compilers retains most of the completeness. The one additional restriction is that any dynamic libraries containing primitives using OCaml's C FFI must be compiled to use base pointers so that stack walking (as described in Section 3.6.7) works.

Correctness is tested both by the OCaml compiler test suite and using a variant of the trace-comparison system comparing function parameters, return values, and the parameters and results of calls to C primitives.

Correctness is mostly maintained from the original compiler with one exception: as the C/machine stack is used rather than the OCaml stack, highly recursive functions operating on large inputs can cause a stack overflow. This is because the OCaml stack will resize itself to larger sizes if it fills up and because the frames on the OCaml stack are smaller. However, this limitation does not apply to tail-recursive functions, which means that this problem is rare.

Nevertheless, this issue did cause a small number test cases to segfault on stack overflow and represents a small regression in the number of passing tests from those supported by the initial compiler.

4.2 The benchmark suite — Sandmark

As a large part of the motivation of the project was increased performance, a core goal was to create or integrate a suite of performance benchmarks. To do this, I adapted an existing suite called Sandmark, created by OCaml labs at Cambridge for their work on multicore OCaml. It uses the dune build tool and the OPAM package manager to compile and execute benchmarks. Although the nature of the multicore OCaml project means many benchmarks are parallel, the project also includes many sequential benchmarks with the purpose of ensuring the work on the multicore runtime does not create single-core performance regressions.

It took a some effort to adapt the suite to work for my purposes. The existing suite only supported native-code programs so it had to be adapted to work with bytecode instead. Now running all the benchmarks and collating the results only requires executing a single script.

4.2.1 Selection of programs

The suite includes hundreds of benchmarks, with support for tweaking benchmark parameters increasing this even more. In order to keep full execution time manageable, I settled on a suite of 36 programs taken from Sandmark's macro subcategory. These programs used many different language features with different workloads. Some were numeric (both integer and floating point), some tested asynchronous behaviour with Lwt and others tested use of OCaml libraries, like the yojson JSON parser.

4.3 Benchmark results

Four different configurations of the OCaml runtime are described in this section:

1. The stock OCaml 4.11.1 bytecode interpreter without any modifications.

2. The state of the JIT after creating the initial compiler (the 'old' initial compiler).

3. The current state of the JIT with the optimising compiler disabled, but retaining the modifications made to the initial compiler to support dynamic recompilation (the 'current' initial compiler).

4. The full current JIT including the optimising compiler with the hot function threshold set to 50 (the optimised compiler).

In order to allow for easier comparison between programs, I compared all JIT results relative to the baseline of the stock bytecode interpreter to obtain relative speedup values; these are calculated by dividing the stock interpreter time by the corresponding JIT time.

4.3.1 Overview

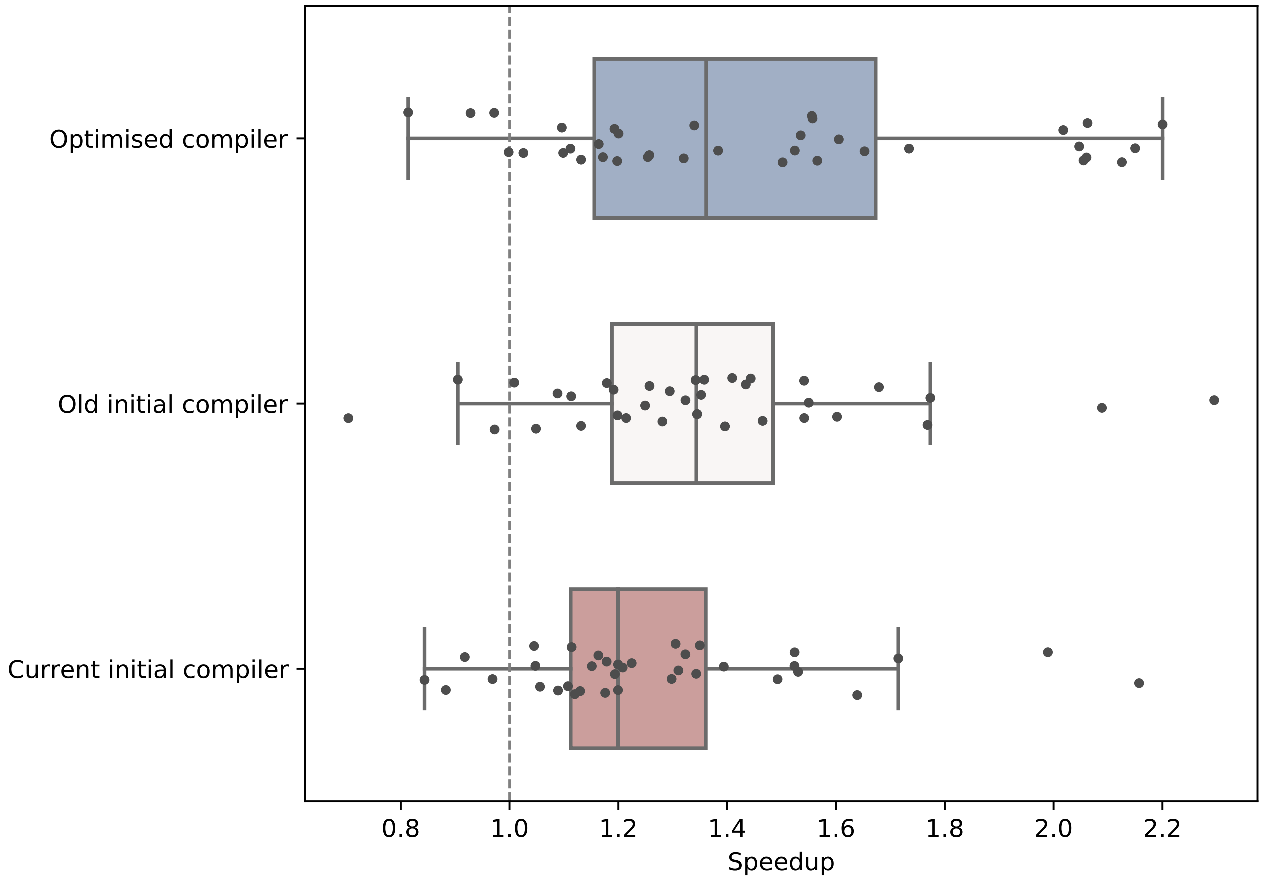


Figure 4.1: Distribution of speedup values for each version of the JIT

Figure 4.1 shows the high-level summary of results on execution speedup. The chart shows a standard box plot[1] with points for each result overlaid. The points are vertically offset randomly.

The general trend is of a severely decreased execution time for most programs executed using all three JITs. The initial compiler, as it existed before the changes made in Section 3.5, was faster than the current version.

The combination of the optimised compiler and current initial compiler version performed slightly worse on programs in the lower half of the speed distribution, which can be seen by the decreased lower quartile value. However it had a slightly increased median and much better performance on the top half of the distribution with a cluster of eight programs managing to halve the execution time relative to the stock interpreter.

[1] Lines for minimum, lower quartile, median, upper quartile and maximum. Outliers (outside 1.5 of the interquartile range) are not included in the whiskers.

Although most programs tested benefited from the changes, four programs did not and had decreased performance.

4.3.2 Analysis

In order to understand these trends and investigate anomalies it is useful to show the results for each individual benchmark. Figure 4.2 shows the relative speedup values for each benchmark program, ordered by decreasing optimising-compiler speedup. There are a few patterns:

- In some programs, the improvements were roughly equivalent for all types of program, such as `chameneos_redux_lwt`.
- In some the optimising compiler added significant performance gains, such as `revcomp2` and `pidigits5`.
- In nearly all cases, the new version of the initial compiler was slower than the initial version.
- In most cases, the optimising compiler increased performance above the initial compiler alone.

In order to explain these patterns it is necessary to consider the cost and potential benefits of each version of the compiler:

Initial version of initial compiler

The main change from the bytecode is replacing the interpreter with machine code performing the same operations, but with the operations inlined into their uses. The main cost of this is an increased pressure on the instruction cache. However, this also allows for the CPU to more effectively predict the sequence of execution, allowing for fewer pipeline hazards, better branch prediction and out-of-order execution across bytecode boundaries.

For most programs this trade-off is worth it. The programs where it isn't (`fft`, `nbody` and `durand-kerner-arberth`) all show heavy use of floating-point operations, meaning most of the work is performed using C primitives and boxed floats. Much of the execution time is spent allocating and freeing these boxed floats which cannot be sped up through the use of native code.

Current version of initial compiler

The main significant change in this version was the extra level of pointer indirection on closure code fields, requiring one more memory load to call a closure. In nearly all cases this represented a slowdown from the initial compiler. The differing amounts of slowdown reflects the variation in benchmark hot-path behaviour. A large drop reflects programs with a large number of calls to shorter functions and where it is small it indicates larger functions making fewer calls or spending much of their time in C primitives.

Optimising compiler

The optimising compiler has a more significant compilation cost. However, this is easily amortised over the long running time of the benchmark programs (see Section 4.4). The

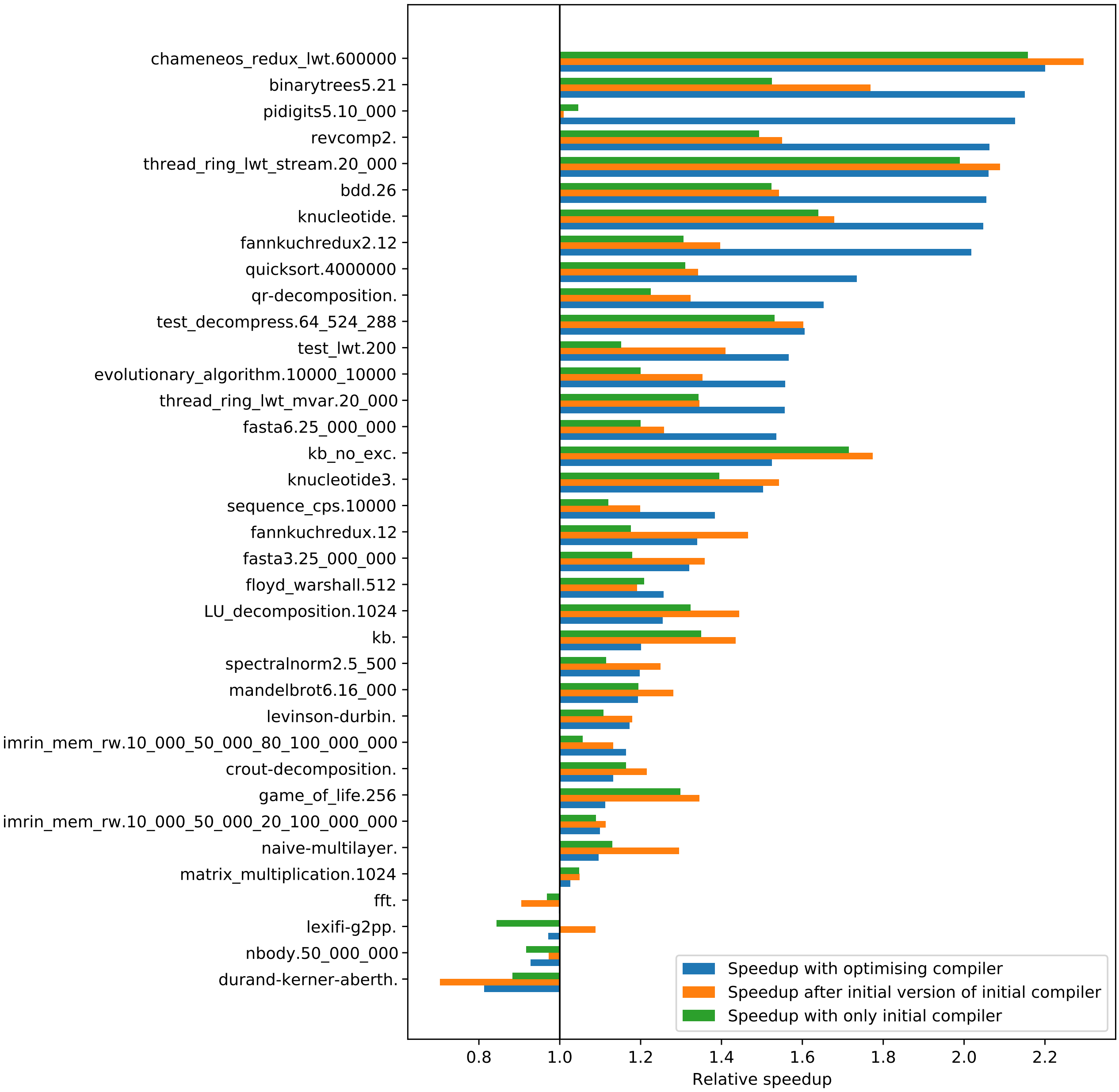


Figure 4.2: Speedup values for each benchmark.

major change from the other two methods and the interpreter is the use of registers rather than the OCaml stack, for storing intermediate values.

In most cases this led to better performance than current version of the initial compiler suggesting that the optimising compiler's code is generally faster. Where it isn't, it would suggest that the overhead of switching back and forth between using registers for arguments and the OCaml stack, as happens at the boundary between optimised and unoptimised functions, is overwhelming any performance benefits.

We would expect the most significant gains where a chain of multiple optimised functions can call each other in turn: this is the pattern shown by many of the cases where the optimising exceeds the performance of the old version of the initial compilers. A case where this is particularly obvious is in pidigits, where the optimising compiler was twice as fast as both initial-compiler-only runs (which themselves ran at about the same speed as the stock interpreter).

In half of the cases where the optimising compiler is faster than the current initial compiler, it is able to overcome the overhead imposed by the extra pointer lookup on function calls. In the other half of cases, performance for the optimising compiler is comparable but slightly worse than the new initial compiler.

One case causing this is demonstrated in `mandlebrot6` — here the main loop is implemented in imperative style at the root level of the module using a while loop rather than recursion. This means that the optimising compiler will never attempt to optimise it. It additionally makes use of exceptions, which are not well supported by the optimising compiler (see Section 3.6.8).

4.4 Bias

Although Sandmark is more representative of real OCaml workloads than many benchmark suites, there are some biases:

Execution time

Execution times ranged from 1 s to 500 s. Of the 36 benchmarks, 13 took below 10 s and 7 took above 50 s. In the context of this project, this means all programs classify as somewhat long-running and time spent on compilation was a negligible factor — when testing the OCaml compiler (a particularly large program in terms of linked bytecode), I could not show any statistically significant difference in execution times due to enabling compilation.

Work perormed

Nearly all of the benchmarks included a certain amount of work being performed many times in a hot loop. This is the best case for modern CPUs and dynamically optimising compilers.

Although the performance of programs is dominated by the hot-path loops, larger programs tend to also have more cold code executed once or only a few times. This is less well tested.

4.5 Summary

The initial compiler increased performance for most tested programs while retaining the semantics and the ability to run all OCaml bytecode programs.

Adding the optimised compiler led to significantly increased execution time on some tested programs while decreasing the time for others. Across the suite of benchmark programs it caused large improvement in half of programs and slightly decreased performance in the other half; overall it was an improvement.

All of this was achieved while retaining support for all OCaml features without compromising correctness.

5 Conclusions

This project demonstrates how the use of JIT compilation techniques can beat the performance of manually optimised interpreters. It contrasts two approaches to JIT compilation, showing how more sophisticated compilers can be applied only when needed for better performance.

Unlike many Part II compiler projects it covers the complete functionality of a large real-world programming language with a focus on correctness — beyond not supporting the debugger and backtraces, no simplifications are made to the language.

5.1 Lessons learned

I found the automated testing strategy very effective for this project — especially trace comparison. Starting with the building this meant I could continuously verify my work. When I had a bug, I could know exactly which instruction led to the mistake rather than debugging indirect effects later in execution.

I have gained a lot of OCaml domain knowledge over the course of the project and with hindsight there are a things I would have done differently. In many cases, I had to completely rewrite a component once I knew more about how it should work. This is difficult to avoid when working within a large existing system and can be a useful strategy for exploratory design.

However, I could have done a better job mitigating some of the time this took; towards the start of the project, there were points where I started with a complicated design only to find I didn't need the component or had to redesign it. By the end of the project, I always started with the simpler approach.

5.2 Possible extensions

More sophisticated analysis of types in the optimising compiler

Performance could be greatly improved by adding OCaml-specific type information to the optimising compiler. For example, once a value is statically known to be a floating-point value the compiler can avoid boxing intermediate values and inline the operations, rather than call out to C primitives. This would greatly help with some of the worst cases for the system which were all heavy uses of floating-point numbers. Implementing this would require adding an additional intermediate representation and dataflow analysis passes in between the basic-block stage and cranelift IR, but would be fairly simple optimisations once the machinery for this was built.

Tail-call recursion optimisation within functions

The current compiler could be extended with a moderate amount of work to lower tail-recursive functions to loops rather than function calls. This is something the native-code compiler does, since tail-recursion is a common pattern in OCaml. This does not require any support from cranelift — only an analysis pass to detect this pattern.

Extending cranelift to handle tail calls directly

Cranelift is a new project and its major user, WASM, has no support for emitting efficient machine code for tail calls. For this reason, there is no support within the library for avoiding pushing another stack frame on tail calls. I had to implement tail calls by use of a wrapper, which hurt performance.

Bibliography

[1] Bytecode Alliance. *cranelift*. URL: https://github.com/bytecodealliance/wasmtime/tree/main/cranelift.

[2] John Aycock. "A Brief History of Just-in-Time". In: *ACM Comput. Surv.* 35.2 (June 2003), pp. 97–113. ISSN: 0360-0300. DOI: 10.1145/857076.857077. URL: https://doi.org/10.1145/857076.857077.

[3] Spenser Bauman et al. "Pycket: A Tracing JIT for a Functional Language". In: *SIGPLAN Not.* 50.9 (Aug. 2015), pp. 22–34. ISSN: 0362-1340. DOI: 10.1145/2858949.2784740. URL: https://doi.org/10.1145/2858949.2784740.

[4] CensoredUsername. *dynasm-rs*. URL: https://github.com/CensoredUsername/dynasm-rs.

[5] Nick Fitzgerald. *WebAssembly Reference Types in Wasmtime (blog post)*. 2020. URL: https://fitzgeraldnick.com/2020/08/27/reference-types-in-wasmtime.html.

[6] Simon L. Peyton Jones. "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine". In: *Journal of Functional Programming* 2.2 (1992), pp. 127–202. DOI: 10.1017/S0956796800000319.

[7] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation". In: San Jose, CA, USA, Mar. 2004, pp. 75–88.

[8] Xavier Leroy. *From Krivine's machine to the Caml implementations*. Talk at the KAZAM workshop which took place at the Edgbaston campus of the University of Birmingham, UK on 17 May 2005. 2005. URL: https://xavierleroy.org/talks/zam-kazam05.pdf.

[9] Xavier Leroy. *The ZINC experiment : an economical implementation of the ML language*. Tech. rep. RT-0117. INRIA, Feb. 1990, p. 100. URL: https://hal.inria.fr/inria-00070049.

[10] Simon Marlow and Simon Peyton Jones. "Making a fast curry: push/enter vs. eval/apply for higher-order languages". In: *Journal of Functional Programming* 16.4-5 (2006), pp. 415–449. DOI: 10.1017/S0956796806005995.

[11] Benedikt Meurer. *OCamlJIT 2.0 — Faster Objective Caml*. 2011. arXiv: 1011.1783 [cs.PL].

[12] Mike Pall. *DynASM*. URL: http://luajit.org/dynasm.html.

[13] Basile Starynkevitch. *OCAMLJIT a faster Just-In-Time Ocaml Implementation*. 2004. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.140.9297&rep=rep1&type=pdf.

[14] *V8 Javascript Engine*. URL: https://v8.dev/.

A Extended optimised compiler example

We will consider the clamp function which calculates

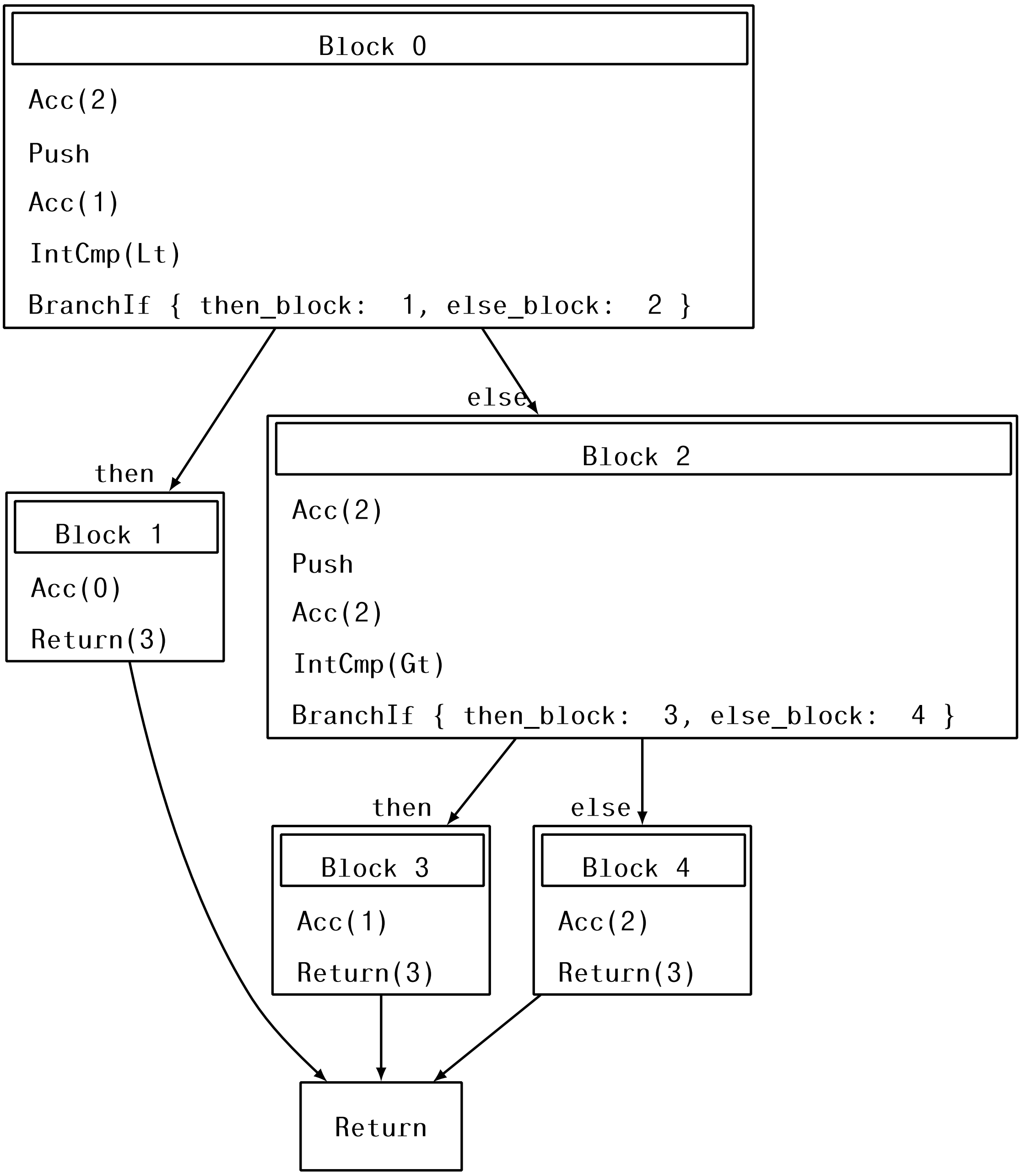
$$\text{clamp}(h,l,x) = \begin{cases} h & \text{if h < x} \\ l & \text{if l > x} \\ x & \text{otherwise} \end{cases}$$

In OCaml it can be defined

```
(% clamp : int -> int -> int %)
let clamp h l (x : int) =
  if h < x then
    h
  else if l > x then
    l
  else
    x
```

A.1 Basic blocks

It is compiled to bytecode which basic block conversions parses to:



A.2 Cranelift IR

My cranelift IR generation produces this:

```
function u0:0(r64, r64, r64, r64) -> r64, i64 system_v {
    gv0 = symbol u1:0

block0(v0: r64, v1: r64, v2: r64, v3: r64):
    v4 = null.r64
    v5 = iconst.i64 0
    v26 -> v5
    v6 = symbol_value.i64 gv0
    v7 = raw_bitcast.i64 v1
    v8 = raw_bitcast.i64 v3
    v9 = ifcmp v7, v8
    v10 = iconst.i64 3
    v11 = iconst.i64 1
    v12 = selectif.i64 slt v9, v10, v11
    v13 = raw_bitcast.r64 v12
    v14 = raw_bitcast.i64 v13
    v15 = icmp_imm eq v14, 1
    brz v15, block1
    jump block2

block1:
    jump block5(v1)

block2:
    v16 = raw_bitcast.i64 v2
    v17 = raw_bitcast.i64 v3
    v18 = ifcmp v16, v17
    v19 = iconst.i64 3
    v20 = iconst.i64 1
    v21 = selectif.i64 sgt v18, v19, v20
    v22 = raw_bitcast.r64 v21
    v23 = raw_bitcast.i64 v22
    v24 = icmp_imm eq v23, 1
    brz v24, block3
    jump block4

block3:
    jump block5(v2)

block4:
    jump block5(v3)

block5(v25: r64):
    return v25, v5
}
```

A.3 Machine code

The IR is compiled to

```
0000000000000000 <clamp>:
   0: 55                      push   rbp
   1: 48 89 e5                mov    rbp,rsp
   4: 48 89 f0                mov    rax,rsi
   7: 49 89 c9                mov    r9,rcx
   a: 41 b8 03 00 00 00       mov    r8d,0x3
  10: bf 01 00 00 00          mov    edi,0x1
  15: 4c 39 c8                cmp    rax,r9
  18: 48 89 f8                mov    rax,rdi
  1b: 49 0f 4c c0             cmovl  rax,r8
```

```
1f: bf 01 00 00 00         mov    edi,0x1
24: 48 39 f8               cmp    rax,rdi
27: 0f 85 2a 00 00 00      jne    57 <clamp+0x57>
2d: 48 89 d0               mov    rax,rdx
30: be 03 00 00 00         mov    esi,0x3
35: bf 01 00 00 00         mov    edi,0x1
3a: 4c 39 c8               cmp    rax,r9
3d: 48 0f 4f fe            cmovg  rdi,rsi
41: be 01 00 00 00         mov    esi,0x1
46: 48 39 f7               cmp    rdi,rsi
49: 0f 84 0b 00 00 00      je     5a <clamp+0x5a>
4f: 48 89 d1               mov    rcx,rdx
52: e9 03 00 00 00         jmp    5a <clamp+0x5a>
57: 48 89 f1               mov    rcx,rsi
5a: 48 31 d2               xor    rdx,rdx
5d: 48 89 c8               mov    rax,rcx
60: 48 89 ec               mov    rsp,rbp
63: 5d                     pop    rbp
64: c3                     ret
```

Cranelift's simple optimisation passes have performed jump coalescing, live variable analysis and register allocation to produce fairly well optimised assembly. My code enabled this by translating uses of the OCaml stack to its primitive values.

B Rust dependencies and their licences

Names are according to the SPDX License List. Most dependencies are transitive dependencies of projects I directly depend on.

Apache-2.0 OR Apache-2.0 WITH LLVM-exception OR MIT (2)
wasi, wasi

Apache-2.0 OR BSL-1.0 (1)
ryu

Apache-2.0 OR MIT (75) anyhow, arrayvec, autocfg, base64, bitflags, bstr, cc, cfg-if, cfg-if, crc32fast, crossbeam-utils, dirs, dissimilar, either, encode_unicode, env_logger, errno, expect-test, gcc, getrandom, getrandom, gimli, hashbrown, heck, hermit-abi, humantime, indexmap, itertools, itoa, lazy_static, libc, log, memmap2, object, once_cell, ppv-lite86, proc-macro-error, proc-macro-error-attr, proc-macro2, proc-macro2, quote, quote, rand, rand_chacha, rand_core, rand_hc, regex, regex-syntax, remove_dir_all, rust-argon2, rustc-hash, serde, serde_derive, serde_json, shell-words, smallvec, structopt, structopt-derive, syn, syn, tempfile, term, terminal_size, thiserror, thiserror-impl, thread_local, unicode-segmentation, unicode-width, unicode-xid, unicode-xid, vec_map, version_check, winapi, winapi-i686-pc-windows-gnu, winapi-x86_64-pc-windows-gnu

Apache-2.0 WITH LLVM-exception (13)
cranelift, cranelift-bforest, cranelift-codegen, cranelift-codegen-meta, cranelift-codegen-shared, cranelift-entity, cranelift-frontend, cranelift-jit, cranelift-module, cranelift-native, cranelift-object, regalloc, target-lexicon

BSD-2-Clause (2)
arrayref, mach

BSD-3-Clause (1)
prettytable-rs

CC0-1.0
constant_time_eq

MIT (20)
ansi_term, atty, blake2b_simd, clap, console, default-env, errno-dragonfly, indicatif, nix, number_prefix, os_pipe, pretty-hex, redox_syscall, redox_syscall, redox_users, region, strsim, strum, strum_macros, textwrap

MIT OR Unlicence (8)
aho-corasick, byteorder, csv, csv-core, memchr, regex-automata, termcolor, winapi-util

MPL-2.0 (3)
colored, dynasm, dynasmrt

A JIT compiler for OCaml bytecode

William Robson (wnr21), Selwyn College

23 October 2020

Special Resources:	*None required*
Project Supervisor:	Timothy Jones
Director of Studies:	Richard Watts
Project Overseers:	Neel Krishnaswami & Sean Holden

Introduction

The OCaml compiler has two main targets - to platform-specific machine code and an interpreted bytecode.

Although the bytecode interpreter is fairly well optimised for interpreting a functional language it still has some degree of overhead over what even a simple JIT compiler could achieve. This is because it cannot benefit from modern CPU pipelining and branch prediction as an abstract machine PC is used rather than the CPU's. This project aims to experiment with replacing the core interpreter loop with JIT compiling the bytecode and to quantify how much speedup could be achieved.

Beyond this basic core the project aims to use all of the remaining time to discover and implement performance optimisations in the compiler and its emitted machine code.

To allow the use of higher level features the compiler will have as much of it as possible written in the Rust programming language. Some of it will be written in the C language to interface with the existing OCaml runtime.

Technical Decisions and Scope

The project will only target x86_64 Linux[1] and no attempts will be made to be portable to other architectures or calling conventions.

[1] And macOS if it proves simple

When implementing a JIT there is a choice of how often to run the compilation process and whether to fall back to the interpreter. To allow for larger optimisations across basic blocks the project will compile all code as it is loaded by the OCaml abstract machine. This means that rather than co-ordinating jumping to and from the interpreter everything can use the compiled version at the expense of startup time.

Work that has to be done

Starting Point

I have decent working knowledge of Rust, C and OCaml. I have experience with writing virtual machines and emulators. I have spent a significant amount of time over the summer holidays and start of Michaelmas term investigating the OCaml compiler source and format of the bytecode.

For this project I have researched x86_64 assembly and understand the basics enough to build the initial version of the JIT compiler (below). Part of this project forks from the OCaml compiler source version 4.11.1.

The runtime and bytecode interpreter internals are fairly poorly documented outside the code. Through my prior investigation I have become familiar with the structure of the bytecode, calling convention (including how closures, partial application and the tail-call optimisation works) and the interface to a large part of the OCaml runtime system. I have not yet investigated the garbage collector, debugger and toplevel in as much detail; however they do not matter as much for the project (at least initially).

In preparation for this project prior to formal start I have already:

- Modified the Makefiles and autoconf scripts to support linking to a static library component written in Rust.
- Added a Rust hook functions when bytecode gets loaded and unloaded and the interpreter is called.
- Added Rust type definitions for the structure of the bytecode instructions and a parser and disassembler for them.
- Used a library to emit machine code for the subset of the bytecode needed to run a hello world program and branch to it

To avoid confusion about what work was completed before the project start submission of the code will have 3 components:

- The final state of the code

- A full git history [2]
- A snapshot of the state of the code at time of formal project start for comparisons

This issue will also be covered explicitly in the final dissertation to avoid misleading the examiners about how much was achieved in the time allotted. This is as although the quantity of code is not very high it will form the framework from which the rest of the project will grow.

Success criteria

I will consider the project successful if

1. There is a JIT compiler implemented into the existing OCaml source replacing the interpreter with all functionality but debugging and introspection.
2. There is a comprehensive and automated suite of benchmarks built comparing its performance to other alternatives.
3. It performs favourably to the interpreter when discounting initial compile time for all programs and overall for longer running programs.

In addition to the basic criteria I have left significant time in the plan for performance and functionality related extensions. I would strongly hope that the project will implement some of these ideas, or other extensions of similar scale:

- Avoid storing values on the heap when they are used and consumed within the same basic block.
- Support running the OCaml toplevel with the JIT compiler.
- Support the debugging instructions.
- Use registers and the system stack rather than the OCaml stack for values local to a given basic block or function.
- Replace calls from OCaml to C primitives with inlined versions - especially floating point operations.
- Delay compilation of a function until it is first called.
- Add microbenchmarking support at the instruction or function level to inform future optimisation efforts.

Rather than commit to these now the project plans to build the necessary benchmarking to aid in the decision of whether to persue each of these extensions and the correctness tests to ensure the optimisations do not break under edge cases before selecting the optimisations to use.

[2] As a git-archive file

Components

The core functionality breaks down into these major components:

1. Integrate with an external library for emitting and relocating x86_64 assembly at runtime from Rust. [3]
2. Build a compiler that emits assembly with the same semantics as bytecode instructions while using the system PC.
3. Add support functions written in C or Rust called by the assembly to link into the OCaml runtime and garbage collector.
4. Build a script for comparing trace outputs of the existing compiler and interpreter to spot errors.
5. Build scripts for automatically running benchmarks comparing the performance of the JIT compiler and existing interpreter.
6. Collect and write a set of benchmark OCaml programs.

Evaluation

The evaluation section of proposal will consist of both qualitative and quantitative analysis. The qualitative section will investigate the limitations of the JIT compiler compared to the current interpreter (if any) as well as the decisions made when deciding which optimisations and extensions to include.

The key quantitative metric used to evaluate this project is benchmark performance. When writing a JIT there is an inherent tradeoff between compilation time and execution speedup. For this reason I will measure both time to startup and execution time. I will compare both of these values against the native code compiler and existing interpreter on a variety of test programs of different sizes and types of computation being done to work out where the tipping point lies before the JIT becomes much better.

Difficulties to Overcome

The following main learning tasks will have to be undertaken as part of the project.

- Learn about the OCaml compiler internals.
- Learn enough about x86_64 assembly to be able to know the right instructions to emit to replace the existing C.

[3] `https://crates.io/crates/dynasm`

The first of these learning tasks at first seems to be made more difficult by much of the OCaml implementation details being undocumented. However I have already managed to learn enough to JIT compile a hello world program (which actually covers about half of the opcode space) and I am becoming much more confident navigating the compiler and runtime source. I am confident that the approach of using C primitives for complicated operations will make this easier.

The assembly issue is made less challenging due to the library mentioned above. Rather than reinventing the wheel I will use the library to handle emitting instructions and relocation. It works by using a compile-time macro to compile mnemonics to calls to a runtime support library that handles relocation. This means that rather than having to write an assembler myself I only need to learn x86_64 assembly which is very well documented. So far I have had no problems understanding and finding resources for it.

The other potential risk is the risk of hard-to-debug issues slowing progress. My plan to build an automated suite of test programs comparing behaviour at the instruction trace level will greatly aid managing this risk as I can use the existing interpreter as a gold standard to compare against.

Resources

This project only requires a modern Unix environment with the build requirements for OCaml and the Rust compiler installed. So far I have tested this on Linux, Windows Subsystem for Linux and macOS.

I plan to use my own computers. I have a desktop that I very recently built with a 10700K CPU and 32GB of RAM. It runs triple-boot Windows and Arch Linux. Both OSes use seperate SSDs.

I also have a 2018 laptop running Arch Linux which is my secondary development system and can serve as backup against main system failure.

I am using Git for version control of the dissertation source and source code and the repository is frequently pushed to GitHub (at least daily). This is my primary line of defence against system failure.

I additionally have already set up automated synchronisation between all of my computers and a server I have. This server runs daily encrypted incremental backups of all everything in my synchronised folders to OneDrive.

Work Plan

Overview

Rather than take a waterfall-style approach with high risk of incompatible components and incorrect assumptions [4] this project aims to follow a much more agile-inspired development methodology. However, there are firmer milestones to limit scope creep and ensure adequate time is given to things like evaluation.

There are three major implementation milestones across all of the components:

1. Implement the JIT so it runs programs correctly.
2. Implement the benchmarking and evaluation tooling.
3. Use the benchmarks to guide the implementation of optimisations while mainting correctness.

As covered earlier, my starting point is a JIT system that can run hello world correctly.

The first task is to implement the instruction tracing capability to compare the state of abstract machine stack and registers and spot any differences.

Then I will follow an iterative loop of:

1. Write or find a new test program.
2. Add compiler code for the operations needed to execute the program.
3. Fix bugs that the new program uncovers by comparing with the trace of the interpreter.

Being able to run nearly all programs and having a good set of benchmark programs is the first milestone at the end of Michaelmas term. Once this is done the benchmark and evaluation tooling is created over the Christmas holiday. Optimisations will be done over Lent leading to code completion at the start of the Easter holiday.

Timetable

Michaelmas Term

This plan targets finishing the first major milestone by the end of Michaelmas and the second by the start of Lent, both of which are listed in this section:

[4] This is as testing the correctness of the components in the absence of a formal specification is best guaranteed by running the entire system together and comparing against the gold standard.

Dates	Work to be completed
16/10 - 28/10	• Implement printing of detailed instruction traces and machine state for the interpter. • Add tracing to the compiler. • Add a script to automatically run both in parallel and compare the output.
Milestone:	*Script to automatically test the compiler for corectness against the interpreter working for hello world.*
29/10 - 11/11	• Follow iteration procedure above aiming to test support for simple integer numeric programs with recursion. • Create and iterate as described above test programs for all numeric operations focussing on edge cases.
Milestone:	*Integer arithmatic is correct and fast.*
29/10 - 11/11	• Continue to iterate and test with larger open source OCaml programs to gain coverage of most features such as the benchmark programs in the OCaml compiler source
Milestone:	*Most programs now run with some degree of JIT compilation.*
12/11 - 25/11	• Collect a benchmark suite of at 10 programs and ensure they all run - some test programs can be reused. • Start work on supporting more ambitious programs such as the OCaml compiler and toplevel. • Attempt to support the interpreter.
Milestone:	*Benchmark programs collected and running correctly with the JIT.*
26/11 - 26/12	• Create tooling to run benchmarks repeatedly and average the results. • Implement microbenchmarking at the C function and JIT bytecode level to show potential optimisation opportunities.
Milestone:	*Benchmarks working - project core success criteria done.*

Lent Term

Lent term is deliberately underspecified currently and is where the project will start to get more open-ended and interesting. The progress report will be used to update on the exact optimisations and extensions being attempted and a more detailed plan will be agreed with my supervisor around that time.

Code completion is set at the end of this term and work will cover only the dissertation.

27/12 - 03/02	• Start work on basic versions of a few optimisations and evaluate their relative impact. • Create the progress report detailing which extensions are to be attempted.
Milestone:	*Progress report completed. Plan in place for any remaining extensions.*
4/02 - 17/03	• Implement the planned optimisations. • Start collecting together notes and references for the dissertation updating the bib file.
Milestone:	*Code completion of project.*

Easter Term

Easter holidays and easter term is completely dedicated to the dissertation. The open-ended nature of the project's extensions make it crucial that this deadline stuck to.

18/03 - 31/03	• Write the first draft of the introduction, preparation and implementation sections. • Send it for feedback.
Milestone:	*First 3 sections completed.*
01/04 - 14/04	• Write the remaining evaluation and conlusions sections. • Write a reference guide to OCaml bytecode as an appendix. • Send and gather feedback on the first draft.
Milestone:	*First complete draft created.*
15/04 - 28/04	• Respond to feedback and continue sending drafts. • Ensure bibliography is complete and correct and that the work has all the needed citations. • Proofread the project and get approval of the final draft.
Milestone:	*Dissertation complete.*
29/04 - 14/05	• Slack time for writing the dissertation. • Submit early.
Milestone:	*Project complete.*