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 maintaining 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.*