Progress Report: 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

This report is longer than might normally be expected for a typical progress report. The summary in section 1 is the main part that should be read by my overseers.

The other sections are included as a more detailed report for my Director of Studies. I do not expect them to be read by my overseers.

Contents

1 Summary

1.1 Work done

The project is on track according to the planned timeline for Michaelmas and the Christmas holidays and has met its core 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 when discounting initial compile time for all programs and overall for longer running programs.

The JIT compiler is capable of running *every* OCaml bytecode program including:

- the full OCaml compiler test suite[1],
- the compiler's bootstrapping compilation of itself,
- and the OCaml toplevel interpreter.

There is no support for the debugger or printing backtraces on exception which was a limitation initially planned in the project proposal.

The compiler works in the way initially proposed:

- Mapping OCaml abstract-machine registers to callee-saved x86_64 registers
- Replacing the stream of bytecode instructions with a stream of machine-code instructions with the same semantics
- Relocating bytecode-relative instruction pointers to machine-code instruction pointers

I forked the Sandmark benchmark suite used by the multicore OCaml project and used its suite of varied benchmarks. Much work has gone into creating and curating

[1] Apart from tests of the debugger and backtraces

this suite, which it makes sense not to needlessly duplicate, although it took about three full days of work to adapt it to the project.

Performance is better than the existing interpreter in nearly all of the benchmarks tested. The compiler is well-optimised and the impact on startup time is negligible. For more details see section 2.3.

For a more detailed overview of work done see section 2.

1.2 Extensions

In the initial plan the work to do in Lent was deliberately underspecified to allow flexibility in the choice of extensions with an expectation that this would be more concrete by the time of the progress report.

Much of my work in recent weeks has been identifying these extensions and building a plan.

The current state of that work is to attempt the following three extensions in priority order:

1. Inline C primitives into the emitted bytecode

2. Create an alternative compiler translating bytecode to a more register-oriented IR to emit faster machine code

3. Count closure invocations to dynamically invoke the compiler from extension 2 only for "hot" functions

I expect 1 and 3 to take at most one Cambridge week each and 2 to take four weeks, leading to the initial planned code-completion deadline of the 17th of March. Any additional time will be spent on iterative optimisations.

See section 3 for more details of the work involved in each of these optimisations.

2 Details of work done so far

2.1 Compiler

Implementation

I built on top of my initial work which supports linking a Rust static library into the OCaml runtime static library and a simple bytecode-instruction parser/disassembler.

I have built a compiler that runs when a bytecode block is loaded. It translates the sequence of bytecode instructions to assembly code with equivalent semantics. At runtime the only difference in semantics is that code pointers point to machine code rather than the initial bytecode and the machine's PC is used rather than an abstract-machine register.

Throughout the initial implementation I prioritised correctness, an approach which means it now correctly executes every program I have tested it with, including the full compiler test suite.

The compiler was developed incrementally. I added tracing support to both the existing interpreter and the new JIT as I developed it. After every bytecode instruction a trace is printed of the state of all abstract-machine registers and the top of the stack. I found that after disabling ASLR any addresses in these registers or on the stack would nearly always be identical between the bytecode interpreter and the JIT, and five retries was sufficient to cover the small number of cases where they differed[2].

I then developed a harness that launched the two processes and compared the two traces. If any discrepancy was found a diff was printed. This tool allowed a very fast iteration time and was invaluable while debugging errors.

I started with a simple hello world program. Then I extended the suite of test programs used to incrementally gain coverage over all the possible bytecode instructions.

After this process was completed, I moved on to tackling the full OCaml compiler test suite. Some of these tests used hooks into GC internals, which made extensive use of callbacks from C to OCaml. I found that my initial implementation of these callbacks was highly inefficient and required special casing to avoid taking hours to run tests. After doing this these tests executed in seconds.

[2] I'd imagine this is due to nondeterminism from syscalls and largely unavoidable

All tests passed except for a few expected failures due to the few scope limitations.

Limitations

There is no support for the OCaml debugger which hooks heavily into the interpreter source and would complicate code generation.

There is no support for printing a backtrace on exceptions as the machinery for that uses the bytecode-relative PCs stored on the stack to unwind the call frames.

I am happy with both of these limitations to keep the scope of the project reasonable. Both are possible to implement in my architecture but would require taking time away from more interesting project extensions detailed below.

2.2 Benchmark suite

I found and extended the excellent Sandmark[3] suite developed by OCaml Labs here at Cambridge for their multicore OCaml project.

This required a hard fork and significant changes; I had to make changes to the way that compilers were built to support compiler source in-tree and making the Dune files and targets for benchmarks also generate bytecode versions of the executables (upstream Sandmark is only concerned with native-code compiler performance).

However, despite the multicore OCaml project focusing on multicore programs a key requirement is that there is no significant performance degradation in single-core programs. This meant the Sandmark project has collected hundreds of single-core benchmarks covering many different workflows. This is why I decided to extend Sandmark rather than build my own benchmarking infrastructure.

2.3 Current performance

Despite no significant effort to perform optimisations (floating-point operations still call out to C primitives rather than use the relevant machine instructions) initial performance results are highly promising even taking into account the additional time taken to do the compilation and relocation passes.

[3] `https://github.com/ocaml-bench/sandmark`

Compared to the OCaml 4.11.1 stock bytecode interpreter, across a representative sample of 39 benchmarks, speedups ranged from 0.7-2.3 with all but four benchmarks experiencing some speedup. The mean and median speedup were both about 1.35. The full results are listed in section 4.

3 Details of extensions

3.1 More efficient handling of C primitives

Some calls to built-in C primitives can be replaced and inlined, treating them as extensions of the bytecode instruction set rather than black-box calls.

The most obvious place to do this comes with floating-point values. All floating-point operations are implemented with calls to primitives to support maximum portability. This is a clear source of performance gains that can be exploited and should improve performance on benchmarks which make heavy use of floating-point operations.

3.2 Creating and decompiling to an IR to optimise assembly output

The bytecode generated for a given Lambda tree (OCaml IR) is highly syntax-directed and the translation to the stack machine is very naive in places. Taking the function as the atomic level (and considering basic blocks), with it there is some limited scope to extract and simplify the bytecode into a more register-oriented IR, which could then drive a more sophisticated (but faster) machine-code generator backend.

Initial work along these lines has been promising but two OCaml-specific difficulties have been uncovered: the need to cooperate with the existing OCaml garbage collector and the implications of the OCaml memory model on this kind of optimisation.

Garbage collector

The OCaml minor garbage collector can potentially run at every allocation and inspects the stack to discover which GC-managed values are still in scope. This limits the extent to which values can be stored in registers and means that cooperation with the OCaml stack will still be required.

However, as garbage collection is rare and it is possible to know at runtime whether it will be triggered we can emit two code paths for each case and benefit from the CPU branch predictor to learn the common path.

Uniform representation

OCaml takes the "uniform representation" approach to dealing with generics[4] which means things like a single double value having to take up multiple words in the heap. This makes some initially clear optimisation opportunities more difficult to apply.

The solution for this project with the time remaining is not to try and instead work on locally using more efficient representations but keeping the calling convention at inter-procedure boundaries.

Approach

Given the limited time left to work on the project, the approach I will take is to build the IR and take the simplest opportunities for optimisations directed from this IR — focussing more on proof of concepts of the approach, perhaps targeted towards helping behaviour in specific benchmark programs. The optimisations will be peephole optimisations but will be able to be done at a level of abstraction much closer to the assembly.

Even if the IR cannot be used as fully as it might be possible to do given more time the correct implementation of this kind of translation will exceed the sophistication of earlier OCaml JIT projects[5].

Even a proof of concept of a compiler operating along these lines would represent a novel contribution to JIT compiling OCaml bytecode.

3.3 Using the more optimised compiler for hot functions

Although the current JIT compiler is very fast and causes a near-negligible startup penalty on most longer running programs the more sophisticated compiler is likely to

[4] using the terminology of `https://thume.ca/2019/07/14/a-tour-of-metaprogramming-models-for-generics/`

[5] I chose not read the papers until very recently to avoid too much influence by prior attempts before in my design but I independently came up with a surprisingly similar design to OCAMLJIT (B. Starynkevitch, 2004) and OCamlJit 2.0 (Meurer, 2010).

be slower — I am unlikely to have time to optimise it heavily and will likely perform a few passes over the bytecode (in my current proof of concept I have four so far).

Every function in the OCaml runtime is expressed as a closure with a code pointer and some data. In order to avoid paying the cost of the slower compiler a count can be stored within the closure. Every time a function is called it increments the count. Once a threshold is reached, we call into the slower compiler. Updating the code pointer stored in this closure will then easily allow replacing the function with the optimised version.

4 Appendix: Current benchmark scores

	Benchmark name and parameters	Interpreter time (s)	JIT time (s)	Speedup
0	chameneos_redux_lwt.600000	7.115391	3.099997	2.295290
1	thread_ring_lwt_stream.20_000	28.747959	13.762132	2.088918
2	kb_no_exc.	6.334980	3.572056	1.773483
3	binarytrees5.21	19.891920	11.249499	1.768249
4	knucleotide.	119.142926	70.959650	1.679024
5	yojson_ydump.sample.json	0.915832	0.569438	1.608309
6	test_decompress.64_524_288	23.514494	14.678336	1.601986
7	revcomp2.	5.228612	3.373420	1.549944
8	bdd.26	19.562112	12.689203	1.541634
9	knucleotide3.	114.241900	74.114721	1.541420
10	fannkuchredux.12	556.014558	379.488397	1.465169
11	LU_decomposition.1024	13.788923	9.553678	1.443310
12	kb.	6.909526	4.817259	1.434327
13	test_lwt.200	28.455001	20.191331	1.409268
14	fannkuchredux2.12	458.026770	328.105117	1.395976
15	fasta3.25_000_000	15.434487	11.366368	1.357908
16	evolutionary_algorithm.10000_10000	196.210916	145.109579	1.352157
17	game_of_life.256	32.058093	23.836251	1.344930
18	thread_ring_lwt_mvar.20_000	7.184721	5.343317	1.344618
19	quicksort.4000000	3.108646	2.316454	1.341985
20	qr-decomposition.	6.485996	4.901057	1.323387
21	naive-multilayer.	13.764337	10.632680	1.294531
22	mandelbrot6.16_000	266.007555	207.662356	1.280962
23	fasta6.25_000_000	14.758505	11.738926	1.257228
24	spectralnorm2.5_500	42.068011	33.674721	1.249246
25	crout-decomposition.	3.991149	3.286668	1.214345
26	sequence_cps.10000	3.651221	3.046677	1.198427
27	floyd_warshall.512	6.825925	5.730140	1.191232

	Benchmark name and parameters	Interpreter time (s)	JIT time (s)	Speedup
28	levinson-durbin.	11.342873	9.622498	1.178787
29	imrin_mem_rw.10_ . . . _80_. . .	26.834964	23.716842	1.131473
30	imrin_mem_rw.10_ . . . _20_. . .	22.039981	19.795780	1.113368
31	lexifi-g2pp.	36.014223	33.095634	1.088187
32	regexredux2.	9.821115	9.353625	1.049980
33	matrix_multiplication.1024	30.672572	29.249789	1.048643
34	pidigits5.10_000	2.566873	2.544595	1.008755
35	nbody.50_000_000	85.799349	88.224394	0.972513
36	zarith_pi.5000	0.611246	0.634778	0.962929
37	fft.	6.244920	6.901376	0.904880
38	durand-kerner-aberth.	0.917878	1.304301	0.703732

To make the table fit, native-code results are omitted. In all cases the native-code compiler is significantly faster, ranging from twice as fast to 38 times as fast as the JIT.