Making Interpreters Fast (at Fastly)

Michelle Thalakottur, Chris Fallin
2023-08-25

Introduction

Interpreters are the black sheep of the language runtime family. Compilers are what you write when you want performant runtimes and when completely ahead-of-time compilation is not possible, as in the case of dynamically typed languages like JavaScript and Python, the community has turned to Just-In-Time compilers to make our runtimes performant. However, a few truths remain. Interpreters are fairly easy to implement, or easier to implement than compilers. They're also portable - you can write your interpreter in Rust and rely on the Rust compiler to compile to different targets instead of handling different targets yourself, as is the case with compilers. You also have a quick edit-compile-run cycle that is useful for language implementers in the throes of implementing a new programming language. Yet, for all these advantages, the fact remains that they aren't used in major runtimes. They're simply too slow and in a fast paced world where our websites need to load quickly, interpreters fall behind.

However in some cases, interpretation is the only option. A target like WebAssembly does not allow for code generation at runtime, but dynamic languages need runtime information like type information to be performant. An interpreter for a dynamic language like JavaScript to WebAssembly might be able to use this type information to be performant. One might suggest a different approach than interpretation, when targeting WebAssembly from JavaScript - bless bytecode as a new function in a running Wasm instance. ~~However, this is unsafe since we do not know the source of the bytecode and might pose a security risk.~~[*]

Instead, we're interested in making interpreters fast, particularly for a dynamically typed language. This foray leads us to study interpreter patterns, look at the performance of interpreters on different hardware, as well as, propose an optimization that improves microbenchmarks by an average of 40% and is expected to lead to meaningful improvements on real interpreters.

Studying Interpreter Performance

In order to study performance of an interpreter for dynamically typed languages, we first create a toy untyped language, called `Inst`. Our goal is to have a language that could express behavior that we were interested in, like property accesses, operations on untyped data and interesting control flow, like loops, while still being simple enough that we can try out different optimizations and inspect disassembly. `Inst` operates on a stack machine and has the following datatypes: u32 Numbers, String, Objects with properties and Local Variables. You can operate on the data types using arithmetic and comparison operators. Labels and Goto's are

[*] However, when you want short-lived low-latency instances, you don't want to be JITing in each instance independently. Additionally, this poses a security risk. As a security measure, we want to disallow the whole OS process that runs the guest code from making writable memory executable. These constraints can be faced by any number of server or embedded environments trying to target WebAssembly from JavaScript.

used to create programs with interesting control flow. Shown below is a translation of pseudo-code to an `Inst` program.

Pseudocode	Inst program
`let y = new()`	`NewObject` `LocalSet(y)`

`y.b = 100000`	`ConstI32(100000)`
	`LocalGet(y)`
	`SetProp(b)`

`lab:`	`lab:`

`y.b = y.b - 1`	`ConstI32(1)`
	`LocalGet(y)`
	`GetProp(b)`
	`Sub`
	`LocalGet(y)`
	`SetProp(b)`

`if y.b != 0 : goto lab`	`LocalGet(y)`
	`GetProp(b)`
	`GotoIf(lab)`

`y.b`	`LocalGet(y)` `GetProp(b)`

Fig.1. *Inst program example*

The interpreter for `Inst` is similar to most interpreters for simple languages. We have a main loop over the `Inst` Opcodes of the program we are executing. In the loop is a switch over the different types of Opcodes, of which there are 17. However, since we want to explore how to build an *efficient* interpreter, we consider several interpreter source-level optimizations that we can employ to improve performance.

Inline Caches

Inline Caches have been used extensively in JavaScript Just-In-Time (JIT) runtimes like Chrome's V8 and FireFox's Spidermonkey to improve runtime performance. JITs profile code that is executed a lot (also called "hot" code) to determine various properties of the code, such as, type information. They then create a small snippet of compiled code, called an IC Stub, that is specialized to a particular property of the hot code.

For example, consider that you have an Add instruction that is polymorphic and you have hot code that always adds two i32 numbers together. A JIT will recognize this and create a compiled IC Stub. This stub has guards that check that the property the IC Stub is specializing for holds in a particular execution. In our example, an IC Stub would have guards that check that the two

things being added together are i32 numbers. This newly created IC Stub is linked to from this hot code. When the hot code is executed again when trying to add two i32 numbers, the guards will pass and the IC Stub will execute and return their result. In contrast, a JIT that does not employ Inline Caching will check what combination of types you are attempting to add together, one by one, and then perform the right addition. The latter is often more expensive, especially if most of your code only adds two i32 numbers together. ICs are also used to speed up object property accesses where the guard is on the object shape.

Inline Caches aren't usually interpreted. However, it's possible that here too they provide enough performance speedup to be useful. Particularly, each instruction that you are interpreting can be associated with a list of IC Bodies that record the execution path that was previously taken. If this instruction is executed in a loop, when executed, the guards on each IC Body are checked. If the sequence of guards passes, the IC Body is interpreted and the interpreter execution might get faster. If all the IC Bodies fail, you can interpret the instruction in the general case and add an IC Body to record this specific execution path.

We test out this hypothesis in the `Inst` interpreter. The interpreter has a flag that can be set to enable the inline cache optimization which employs ICs for certain opcodes that deal with object property accesses and operations on data types. We compared the performance of the interpreter with IC's enabled and switched-off on various tests, as reported in Table 1. We run each test a 100 times and report the average execution time. The interpreter source code can be found here, along with the tests.

We found that Inline Caches do help Interpreter performance, particular in the case of programs that have a lot of object accesses (`loops_bench.inst` and `prop_accesses.inst`) as well as when we have polymorphic code (`dynamic_types.inst`).

Tests	With IC's (aarch64-M1)	Without IC's (aarch64-M1)
compare.inst	10.76 ms	8.31 ms
dynamic_types.inst	5.99 µs	5.80 µs
loops.inst	18.33 ms	18.50 ms
loops_bench.inst	5.45 ms	6.44 ms
prop_accesses.inst	28.78 µs	32.63 µs
sample.inst	3.15 µs	2.75 µs

Table 1. *Interpreter performance when compiled to aarch64 on M1 with ICs enabled vs switched off.*

We also ran the Interpreter by compiling the Rust source code to WebAssembly and running the interpreter using the WebAssembly standalone runtime, `wasmtime`, the results of which are shown in Table 2. While ICs still help performance, we see that interpreting via Wasm rather than compiling directly to aarch64 on M1 introduces a significant overhead. It's only in

`sample.inst`, a relatively boring test with no loops or interesting datatype operations, that the execution times are similar.

Tests	With IC's (wasm-M1)	Without IC's (wasm-M1)
compare.inst	63.95 ms	56.67 ms
dynamic_types.inst	9.83 µs	8.51 µs
loops.inst	113.32 ms	113.13 ms
loops_bench.inst	31.05 ms	19.75 ms
prop_accesses.inst	40.72 µs	51.30 µs
sample.inst	2.62 µs	2.76 µs

Table 2. *Interpreter performance when compiled to WebAssembly on M1 with ICs enabled vs switched off.*

We also wanted to test interpreter performance on different hardware and so ran the tests on an Intel-i7 machine as well. Table 3 reports the performance when compiling to x86 and you see that the performance of the interpreter is a little better than in aarch64 (Table 1). Other than that ICs improve interpreter performance only slightly. The difference between interpreting with ICs enabled vs switched off is not as large as in aarch64.

Tests	With IC's (x86-intel)	Without IC's (x86-intel)
compare.inst	17.66 ms	10.23 ms
dynamic_types.inst	4.50 µs	4.75 µs
loops.inst	23.44 ms	23.47 ms
loops_bench.inst	8.97 ms	8.77 ms
prop_accesses.inst	26.85 µs	27.72 µs
sample.inst	1.86 µs	1.79 µs

Table 3. *Interpreter performance when compiled to x86 on Intel with ICs enabled vs switched off.*

We also compile the interpreter to WebAssembly and report on execution times in Table 4. Here, the difference between execution time when running the interpreter compiled to x86 vs WebAssembly is not as great as on the M1. WebAssembly execution is still slower in comparison, but not as slow as on the M1.

Tests	With IC's (wasm-intel)	Without IC's (wasm-intel)
compare.inst	19.72 ms	14.47 ms
dynamic_types.inst	6.69 µs	6.27 µs
loops.inst	11.69 ms	10.17 ms
loops_bench.inst	30.44 ms	30.42 ms
prop_accesses.inst	48.64 µs	47.94 µs
sample.inst	2.87 µs	2.98 µs

Table 4. *Interpreter performance when compiled to WebAssembly on Intel with ICs enabled vs switched off.*

From these investigations we were left with two main questions: Why was interpreter performance better on x86 than on aarch64? Why was interpreter performance worse on wasm compiled to aarch64 than on x86?

Profiling

In order to answer the above questions, we decided to profile the interpreter execution on different hardware. On the Intel-i7 machine, we ran the interpreter through `perf` to profile its execution on the `prop_accesses.inst` test. This would allow us to see hot paths in the code which we could then optimize to improve performance. We also performed profiling on the M1 using a sampling profiler called `samply` to profile code. Our profiling investigations led to the following realization. ***The main hot path in the interpreter was the switch over Inst opcodes in the main interpreter loop body, as well as the switch over IC Opcodes when interpreting IC Bodies***. For the rest of this report, we refer to this pattern as the "*loop-switch pattern"*, referring to the fact that in both hot areas, there exists a loop whose body contains a switch case.

This realization, in addition to the fact that Interpreter performance seemed to be tied to the hardware it ran on, made us come up with the following hypothesis. Branch prediction on Intel-i7 might be better than on M1, which could lead to more branch hits when choosing which switch case to execute next in out-of-order execution. In fact, since the hot paths are in the loop-switch itself, this might cause the inline caching optimization to give less of a performance boost on Intel rather than on the M1.

Optimization: Direct Dispatch

Branch predictors work on context. The more context the branch predictor has on what it has seen before, the better job it can do predicting what will happen next. For example, consider the illustration of the CFG of a simplified loop-switch pattern in Figure 2. A branch predictor is posed with the question of "what case do I interpret next" when it is at Block B, ie, the head of the

switch case itself. When at Block B, it might forget that it had just executed Block C and that in all previous executions, it executed Block E after Block C. That is, in this loop-switch pattern, there is a single dispatch point, which is at the switch statement itself. At this single dispatch point, the branch predictor has less context than at the end the execution of a case, and so might do a worse job of branch prediction.

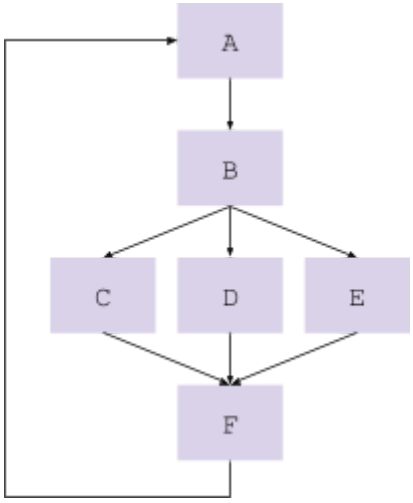

Fig.2. *Control Flow Graph (CFG) of a simplified Loop-Switch pattern*

A way to improve branch prediction would be to give the branch predictor more context. If, say, the branch predictor was making a prediction on which case to execute next at the end of a case it knew succeeded successfully, it might do a better job of the prediction. Specifically, if the loop-switch had multiple dispatch points, one for every case in the switch case, the branch predictor might make better predictions. This would involve code duplication, as seen in Figure 3, to give us multiple dispatch points at the end of each case. We refer to this optimization as **direct-dispatch** in the rest of the report.

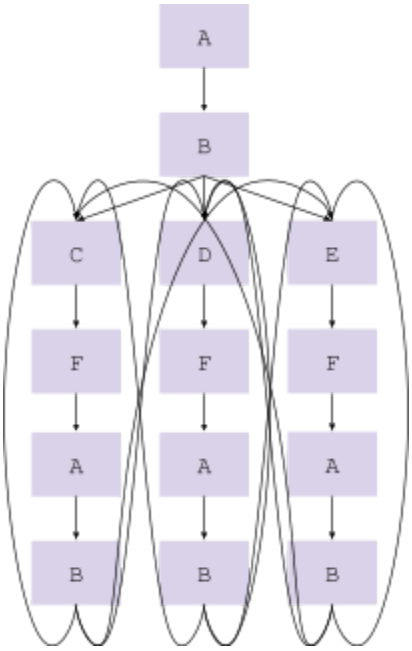

Fig.3. *Control Flow Graph (CFG) of a simplified Loop-Switch pattern with direct-dispatch*

Direct Dispatch in Cranelift

There are two compiler pipelines in the experiments we were running. The interpreter source (written in Rust) to native and the interpreter source to WebAssembly, which was then run using wasmtime. Wasmtime is the Bytecode Alliance standalone runtime for WebAssembly and it uses Cranelift, a compiler from WebAssembly to four different targets, including x86 and aarch64, to compile WebAssembly down to native code.

We decided to implement direct-dispatch in Cranelift to see if it improved our toy interpreters performance when running the interpreter compiled to WebAssembly. We decided to implement this as part of the CLIF (Cranelifts internal representation) optimization pass. CLIF is an SSA based internal representation, using block parameters instead of phi nodes. Doing the optimization at the CLIF level would mean that we would not need to specialize it for each target and that it would be a general optimization. The implementation would require recognizing the loop-switch pattern in a given CLIF program and then duplicating basic blocks from the end of a switch case to the loop header and switch instruction, to the ends of each case in a switch case.

We stumbled very early on. A loop-switch pattern isn't as easy to recognize as it seems, as there can be arbitrary computation at the start and end of the loop that makes the pattern hard to find. Additionally, data dependence between blocks in CLIF make it so that you can not simply duplicate blocks. Specifically, in SSA, a basic block can use variables defined in blocks that it is dominated by. When this block is duplicated to a different part of the program, so that it is dominated by a different block, we have to make this dependence explicit by passing in

values as block parameters. We have to then also change every branch to the block so that it passes in these additional values.

We then tried a different approach. We thought to do the optimization after register allocation so that we do not need to worry about data dependencies. This is called the “block emission stage” in Cranelift. However, this stage assumes that blocks are only emitted once and when duplicating blocks, things get complicated very quickly.

Before investing more engineering time into this problem, we decided to write a small CLIF program to check that the direct-dispatch optimization yielded a performance gain. Figure 4 shows the CFG of the small loop-switch pattern test. We iterate through a loop a hundred million times while switching between three cases that add different numbers to a count that is returned by the function. We switch on a variable that is incremented in each loop iteration and updated to be `variable%3`. We also wrote a version of this CLIF program with direct-dispatch implemented, and timed the execution of each program to see which was faster.

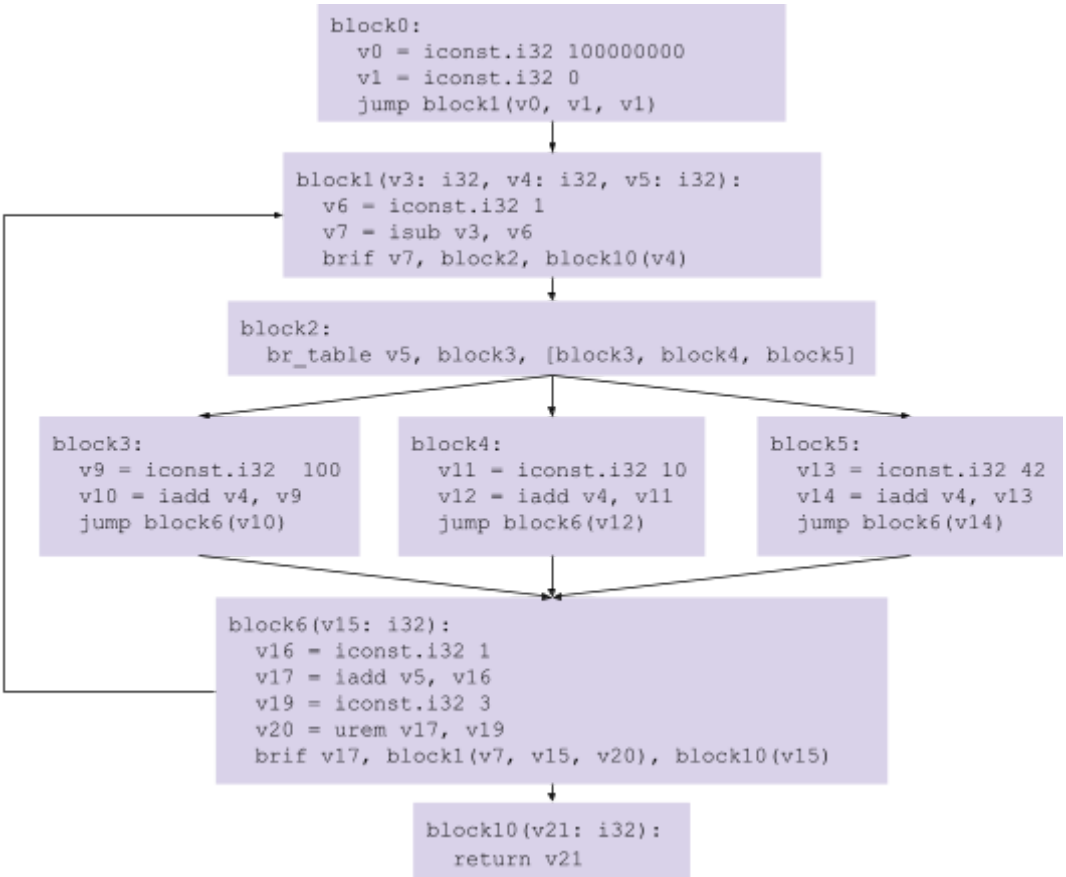

Fig.4. *Loop-switch pattern in CLIF*

Unfortunately, both turned out to have roughly the same execution time. In fact, the execution time of the direct-dispatch version was slower on both the M1 and on Intel.

Test	aarch64	x86
loopswitch.clif	11.06s	13.37s
loopswitch-direct-dispatch.clif	12.03s	13.40s

Table 5. *Execution time of loop-switch with and without direct-dispatch on the M1 and Intel*

This was surprising. We wondered if we needed to go even lower to understand what was going on in the different hardware. We needed to stare the branch predictor and the loop-switch pattern in the eye to understand the difference in execution times on different hardware and understand if there was any optimization that could be done to improve interpreter performance.

Studying Loop-Switch at the Assembly Level

We implemented the loop-switch pattern in assembly. We wanted to isolate the branch predictor as much as possible and so made several changes to the pattern.

1. We decided to increase the number of cases in the switch statement to be much higher than three, so that this simulated more realistic workloads. The assembly tests we wrote had 256 cases.
2. In Figure 4 you can see that we use a `urem` instruction to get the remainder of the switch match value, when divided by three. Generally, division instructions require quite a few clock cycles. We decided to replace this by an `and` instruction since `value&255` gives us the low eight bits of the register, which is equivalent to `value%256` , while being much faster.
3. We also decided to compare different access patterns to the switch case. In Figure 3, you see that we access each case one at a time, in order, in each loop iteration. That pattern of accessing switch cases might be easier to predict for the branch predictor and is not realistic of real world program behavior that we want to optimize. We decided to use a Linear-feedback shift register to generate a pseudo-random number in each iteration, of which we then use the lower eight bits as our match value for the switch. We call accessing the cases one-by-one in order “in-order” and pseudo random access “lfsr” in our results in Table 6, 7, 8 and 9.

The way that Cranelift implements a switch case (the `br_table` instruction from WebAssembly) is by generating labels to every case in the switch and creating a table of 32-bit PC-relative offsets which are the cases in the switch. To jump to the right switch case, it loads the offset of the case from the table, while sign extending it, and adds this to the base address of the table. It then jumps to this address. You could instead have your table contain the addresses of the cases directly, or branches to the cases. Since the direct-dispatch optimization had not yielded a performance boost in Cranelift, we decided to test these different table accesses out to see if one was more performant than the others.

We ran these experiments on different hardware - aarch64 on M1, aarch64 on Ampere Altra with Neoverse N1 cores and x86 on Intel-i7-7700. The Ampere Altra and Intel machines had

support for profiling programs using `perf` so we also compared different program execution statistics using `perf stat`. The different execution times can be found in Table 6 and execution statistics can be found in Table 7.

Machine Type	Case Access Type	Table of branches to cases	Table of 32-bit PC-rel offsets	Table of case addresses
M1	in-order	4.39s	1.90s	1.90s
	lfsr	8.17s	8.82s	8.41s
Apere Altra	in-order	6.83s	6.53s	6.74s
	lfsr	8.79s	8.79s	8.57s
Intel	in-order	2.07s	1.51s	1.51s
	lfsr	6.93s	7.33s	7.84s

Table 6. *Execution time for different switch case access types with different table access methods, across different hardware.*

As is evident from both the execution time and statistics measurements, tables with branches to cases performed worse than both, tables with 32-bit PC-relative offsets and tables with case addresses. Additionally, both aarch64 machines performed worse than x86, behavior that we saw with the toy interpreter execution. We also see that the execution time of `lfsr` is worse than accessing cases in-order, which supports our hypothesis that the branch predictor has a harder time predicting the next case when we use pseudo random access. This can also be seen in the branch-miss measure in Table 7. Since Cranelift has 32-bit PC-relative offsets, in its table, there does not seem to be an optimization rooted in table access that we can implement to improve interpreter performance.

Machine Type	Case Access Type	perf stat measure	Table of branches to cases	Table of 32-bit PC-rel offsets	Table of case addresses
Ampere Altra	cycle	cycles ($x10^9$)	19.03	18.23	18.84
		instructions ($x10^9$)	13.01	12.02	11.01
		branches ($x10^9$)		Not Available	
		branch-miss ($x10^3$)	637014	759142	757249
	lfsr	cycles ($x10^9$)	24.56	24.59	23.98
		instructions ($x10^9$)	20.01	19.01	18.01
		branches ($x10^9$)		Not Available	

		branch-miss ($x10^3$)	774215	894453	893484
Intel	cycle	cycles ($x10^9$)	8.46	6.3	6.3
		instructions ($x10^9$)	12.00	13.00	13.00
		branches ($x10^9$)	5.00	4.00	4.00
		branch-miss ($x10^3$)	2420	104	332
	lfsr	cycles ($x10^9$)	29.12	30.6	32.77
		instructions ($x10^9$)	27.00	28.00	27.00
		branches ($x10^9$)	5.00	4.00	4.00
		branch-miss ($x10^3$)	712073	750795	744848

Table 7. *Execution statistics for different switch case access types with different table access methods, across different hardware.*

With these more realistic test cases, we wanted to see if the direct-dispatch optimization would fare better than it had on the toy program we had written in CLIF. We compared a loop-switch pattern with 32-bit PC-relative offset tables with no direct-dispatch optimization vs with the optimization enabled. We recorded execution time on different hardware in Table 8 and different execution statistics in Table 9.

Machine Type	Case Access Type	No direct-dispatch	Direct-dispatch enabled
M1	in-order	1.90s	0.71s
	lfsr	8.82s	6.23s
Ampere Altra	in-order	6.53s	1.09s
	lfsr	8.79s	6.08s
Intel	in-order	1.51s	0.86s
	lfsr	7.33s	5.67s

Table 8. *Execution time for loop-switch with direct-dispatch enabled vs switched off, across different hardware.*

Direct-dispatch gives us a performance speedup! In fact, on average, across different hardware, it gives us a 40% speedup! The speedup is greater for aarch64 machines rather than Intel which supports our hypothesis that the branch predictor for aarch64 machines does a worse job than on Intel. Surprisingly, the speedup exists for both programs where the switch cases are accessed in order versus a pseudo-random access pattern (`lfsr`).

You might wonder why the CLIF loop-switch example with direct-dispatch did not show a performance speedup, when it does show up when the optimization is implemented in assembly. Especially since in-order case access showed us speed-ups too. The immediate differences between the assembly program and CLIF, in the in-order case access case, is the number of cases as well as the absence of the division operator to get `value%256`. It might be that direct-dispatch is a valid optimization only in the case of non-toy workloads. We haven't tested this out by constructing a more realistic CLIF program with more cases and no div instruction mainly because generating the direct-dispatch version of the program for 256 cases is tedious and we are fairly confident in the optimizations ability to give us improved interpreter performance since the speedup is consistent over different hardware and in multiple runs.

Machine Type	Case Access Type	perf stat measure	No direct-dispatch	Direct-dispatch enabled
Ampere Altra	cycle	cycles ($x10^9$)	18.23	3.00
		instructions ($x10^9$)	12.02	10.00
		branches ($x10^9$)	Not Available	
		branch-miss ($x10^3$)	759142	20
	lfsr	cycles ($x10^9$)	24.59	17.00
		instructions ($x10^9$)	19.01	17.01
		branches ($x10^9$)	Not Available	
		branch-miss ($x10^3$)	894453	537140
Intel	cycle	cycles ($x10^9$)	6.3	3.7
		instructions ($x10^9$)	13.00	11.00
		branches ($x10^9$)	4.00	2.00
		branch-miss ($x10^3$)	104	2748
	lfsr	cycles ($x10^9$)	30.6	23.72
		instructions ($x10^9$)	28.00	26.00
		branches ($x10^9$)	4.00	2.00
		branch-miss ($x10^3$)	750795	513484

Table 9. *Execution statistics for loop-switch with direct-dispatch enabled vs switched off, across different hardware.*

Implementation in Cranelift

While implementing the optimization is a stretch goal of this project, we describe an algorithm to implement direct-dispatch in Cranelift. It is our hope that we are able to report on the effect of this optimization on real workloads once this has been implemented.

1. *Recognize the loop-switch pattern in Cranelift.* Cranelift has a loop analysis that can be used to recognize loops in a function body. For a loop header block, traverse the CFG from the block, while checking each block's last instruction. If the last instruction is a `br_table`, this indicates the presence of a switch statement in the block.
2. *Convert blocks in loop-switch to Maximal-SSA.* We have to make the data dependence between blocks in the loop-switch pattern explicit. This can be done by converting from SSA form to Maximal-SSA form. If a block uses a variable that has not been defined in the block, the variable has to be added as a block parameter. All branches to the block have to also be modified to pass in the right value.
3. *Duplicate necessary blocks.* In the loop-switch pattern, start duplicating blocks at the start of each switch case, following the CFG, till you reach a block that has its last instruction as the `br_table` (switch) instruction. This is done for each case of the switch and so multiple copies of the blocks between the end of the case, through to the loop header and switch itself are made.

Relevant Literature

Most literature in improving interpreter performance focuses on changing interpreter source code to be more performant. Shipilëv et al explore improving interpreter performance in the face of dynamic dispatch in Java and suggest techniques like inlining. They do however discuss the importance of having less branch mispredictions. A study of building Efficient Interpreters by Ertl and Gregg studies the effect of branch predictors on several different types of interpreters. Their findings are clear - branch prediction and the context problem, referring to the context a branch predictor has when making a prediction, is the biggest bottleneck for interpreter implementers. They suggest creating multiple points of dispatch in an interpreter rather than a single point of dispatch, just as we do, however, they do not propose an algorithm for this optimization or evaluate it on real-world workloads. Neither do they study its points of failure. Our hunch is that direct-dispatch is an optimization that is helpful in certain kinds of workloads.

The other direction of interpreter research seems to be to augment interpreters with Just-In-Time information in order to improve performance. Work by Zaleski et al. on the Yeti Interpreter falls into this category and so does work by Gal et al. Polito et al explore using special "interpreter variables", that are critical to the efficient execution of the interpreter loop with a technique called auto-localization, that shows performance improvements for x86 benchmarks. Zhang et al look at the difference between a register based vs stack based interpreter for Python.

Conclusion

While interpreter performance is largely looked down on in the face of large improvements in JIT runtimes and compilers, the fact remains that interpreters are easier to write for language runtime implementers and require a significantly less amount of engineering effort. We hope that direct-dispatch is able to improve performance for interpreters shouldering real workloads and improve the state-of-the-art in interpreter performance.

A future direction for research might be looking at interpreters for dynamic languages as part of a delayed JIT loop. Namely, you profile realistic workloads and use that profile information to specialize interpreter performance. The specialized interpreter can then be deployed on real workloads and profiled again, thus improving performance slowly. This would specialize an interpreter to a certain kind of workload but this is fair game in the world on runtimes for dynamic languages. JavaScript developers are already taught to write monomorphic code for the sake of runtime performance, a delayed interpreter loop would hardly pose more cognitive overhead.