Whose Baseline Compiler Is It Anyway?

Ben L. Titzer

Software and Societal Systems Department
Carnegie Mellon University
Pittsburgh, USA
btitzer@andrew.cmu.edu

2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) | 979-8-3503-9509-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/CGO57630.2024.10444855

***Abstract*—Compilers face an intrinsic tradeoff between compilation speed and code quality. The tradeoff is particularly stark in a dynamic setting where JIT compilation time contributes to application runtime. Many systems now employ multiple compilation *tiers*, where one tier offers fast compile speed while another has much slower compile speed but produces higher quality code. With proper heuristics on when to use each, the overall performance is better than using either compiler in isolation. At the introduction of WebAssembly into the Web platform in 2017, most engines employed optimizing compilers and pre-compiled entire modules before execution. Yet since that time, all Web engines have introduced new "baseline" compiler tiers for Wasm to improve startup time. Further, many new non-web engines have appeared, some of which also employ simple compilers. In this paper, we demystify single-pass compilers for Wasm, explaining their internal algorithms and tradeoffs, as well as providing a detailed empirical study of those employed in production. We show the design of a new single-pass compiler for a research Wasm engine that integrates with an in-place interpreter and host garbage collector using value tags, while also supporting flexible instrumentation. In experiments, we measure the effectiveness of optimizations targeting value tags and find, somewhat surprisingly, that the runtime overhead can be reduced to near zero. We also assess the relative compile speed and execution time of six baseline compilers and place these baseline compilers in a two-dimensional tradeoff space with other execution tiers for Wasm.**

***Index Terms*—compilers, JITs, single-pass, baseline, compilation time, tradeoff, instrumentation WebAssembly**

I. Introduction

Software virtual machines (VMs) provide a way to execute a *guest* programming language, instruction-set architecture, or bytecode format on a different *host* machine. VMs employ a variety of execution strategies that balance memory consumption, startup time, and peak performance. In settings where loading or generating code at runtime is possible, new code can "appear from nowhere", and purely ahead-of-time translation is not possible. This leaves such virtual machines with the option to employ an interpreter or a dynamic compiler.

A. WebAssembly

First appearing in major Web Browsers in 2017, WebAssembly [1] (or *Wasm*) is a bytecode format designed to offer portable native-level performance and software fault isolation via efficient in-process sandboxing. Wasm is a low-level, machine-independent compilation target that can be executed on modern CPUs with very low overhead. It has allowed an explosion of new, powerful Web applications and capabilities, such as desktop applications like AutoCAD [2] and Photoshop [3], video conference acceleration [4], real-time audio processing for echo reduction [5], and many others. Many of these are made possible by recompiling (potentially millions of lines of) legacy C/C++ code using standard toolchains that now support Wasm as a target.

WebAssembly is the first example of a major language that has employed formal specification and verification from design inception. With a fully-formalized specification [6] and machine-checked proof of type safety [7], it offers the most rigorously-specified compilation target to date, making it the most robust option for strongly isolating untrusted code on the Web or in other intra-process scenarios. In the literature, Wasm has inspired a number of exciting directions in Web research [8] [9], verification research [10] [11], systems research [12], cloud and edge computing [13] [14] [15], and PL research [16].

B. Execution Strategies for Dynamic Code

Interpreters. An interpreter executes a program by examining *data* that represents guest code. Strict interpreters can execute any given input program without generating new machine code[1]. Interpreters have the advantage that little or no up-front processing of the program is required and can often execute the code directly from the disk or wire format, saving both startup time and memory. Interpreters also excel at debugging and introspecting execution states, as they often directly implement the state abstractions of their respective code format, such as an operand stack. However, dispatch overhead means interpreters can never match the performance of compiled code in the long run.

Baseline JIT compilers. VMs have deployed *dynamic translation* to machine code as far back as LISP in 1960. Often called just-in-time (JIT) compilation, a dynamic compiler generates new machine code at runtime that behaves equivalently to the interpreter's semantics, but is much faster. A *baseline* compiler is designed to generate machine code as fast as possible, forgoing the use of an intermediate representation (IR). The very first dynamic translators were baseline compilers, stamping out templates of the interpreter's logic for each

[1]Some interpreters may generate machine code stubs or, e.g. per-signature helper routines, but don't translate guest code directly to machine code.

guest instruction or AST node, one after another, thus neatly eliminating the interpreter dispatch loop. Despite the simplicity of baseline compilers, execution time improvements of 3× to 10× are common.

Optimizing JIT compilers. JITs in today's virtual machines are powerful, integrating many ideas from static compilers, employing state-of-the-art IRs and sophisticated optimization passes. For example, TurboFan [18], the optimizing compiler in V8, employs a program dependence graph (PDG) representation called the "sea of nodes" [19], with two different but overlapping optimization pipelines, one for JavaScript, and one for WebAssembly. Key optimizations employed by most modern optimizing JITs are inlining, load elimination, strength reduction, branch folding, loop peeling and unrolling, global code motion, instruction selection, and register allocation.

C. Overview and Contributions

This paper is about maximizing compile speed for WebAssembly. It presents a new single-pass compiler design for a research engine and compares and contrasts it with other single-pass compilers and other tiers for Wasm execution. This paper's contributions are:

- **A new baseline compiler**, **`Wizard-SPC`**, designed for interoperability with in-place interpretation of Wasm in the Wizard Research Engine, supporting full-fidelity instrumentation and debugging.
- **Distillation** of the key designs for five other Wasm baseline compilers that all share the same foundational abstract interpretation approach, yet are discussed nowhere in the literature.
- **Novel value tag optimizations** that reduce their runtime cost nearly to zero, greatly simplifying runtime systems.
- **Novel instrumentation optimizations** that support flexible instrumentation for dynamic analysis.
- **Empirical evaluation** of baseline compiler optimizations in **`Wizard`**, including novel value tag optimizations.
- **Multi-tier performance comparison** among interpreters, baseline compilers, and optimizing compilers for Wasm.

As description of fast Wasm baseline compilers do not yet appear in the literature, this paper first clarifies these designs by describing the basic abstract interpretation algorithm which they all share. We then report on **`Wizard-SPC`**, a new, state-of-the-art single-pass compiler for a research Wasm engine. A novel design problem is integrating with an in-place interpreter to support full-fidelity debugging and instrumentation of Wasm code. For evaluation, we compare six baseline compilers found in industry across a wide variety of benchmarks and place them in context with interpreters and optimizing compilers.

II. Executing Wasm

Wasm bytecode is organized into modules, with top-level functions containing instructions for a stack machine. Wasm bytecode is unusual in that it has structured control-flow constructs like **`block`**, **`if`**, and **`loop`**. Such constructs improve the compactness of the code format and the efficiency of the code validation algorithm. A key property is that branches that target a **`block`** or **`loop`** must be nested inside the construct. This leads to a natural notion of a "control stack" that allows the validator algorithm to immediately reuse any internal metadata for control constructs as soon as the construct is exited[2]. Another intentional design property is that all control-flow predecessors of a label (except **`loop`**) precede the label, enabling highly efficient single-pass forward data flow analysis via abstract interpretation.

Wasm now exhibits execution tiers of all three basic designs. Interestingly, these appeared in the exact opposite order to most VMs. Optimizing compilers for Wasm appeared first in Web engines, made possible by the engineering effort put into making JavaScript fast. Later, Web Engines added baseline compiler tiers, as startup time became an issue for large Wasm modules. Concurrently, non-Web compilers and interpreters started appearing. Initially, interpreters employed rewriting of Wasm code to another representation, but recent work [20] showed an in-place interpreter can be on par with rewriting interpreters.

III. Single-pass Compilation of Wasm

Single-pass compilers for Wasm are designed for compile speed and simplicity. A single pass affords no time to build an intermediate representation of the code. Instead, such compilers are limited to generating code for one (or a small number) of instructions at a time based on limited context accumulated from prior instructions.

In our study of Wasm compilers, we found that all single-pass compilers are simply variations on a basic abstract-interpretation approach that is similar to Wasm code validation[3]. Thus, by understanding this common approach, we can compare and contrast the variations and more easily understand the innovation represented by **`Wizard-SPC`**.

Figure 1 gives an example compilation of Wasm code using the common abstract-interpretation approach. The abstract state consists of an *abstract value stack* (shown), an *abstract control stack* (not shown), and register allocation state (not shown). Each local variable slot and operand stack slot in the abstract value stack has an *abstract value* that can contain information such as:

- **stored** - tracks whether the slot has been stored into memory, and where,
- **register** - the register, if any, which holds the value, and
- **const** - the concrete value, if a constant.

Local variables representing parameters are initialized from the signature of the function and the calling convention. In the example, argument values arrive from the caller in an explicit value stack stored in memory. Declared local variables (not shown) are by Wasm semantics initialized to the zero value of their respective type. Not shown, the algorithm maintains an abstract control stack that tracks the nesting of control constructs such as **`block`** and **`loop`**. Each construct has a

[2]It is believed, but has not been yet shown, that this representation is optimally efficient.

[3]In fact, some baseline compilers in this study, like Liftoff, reuse parts of their validation algorithms to drive compilation.

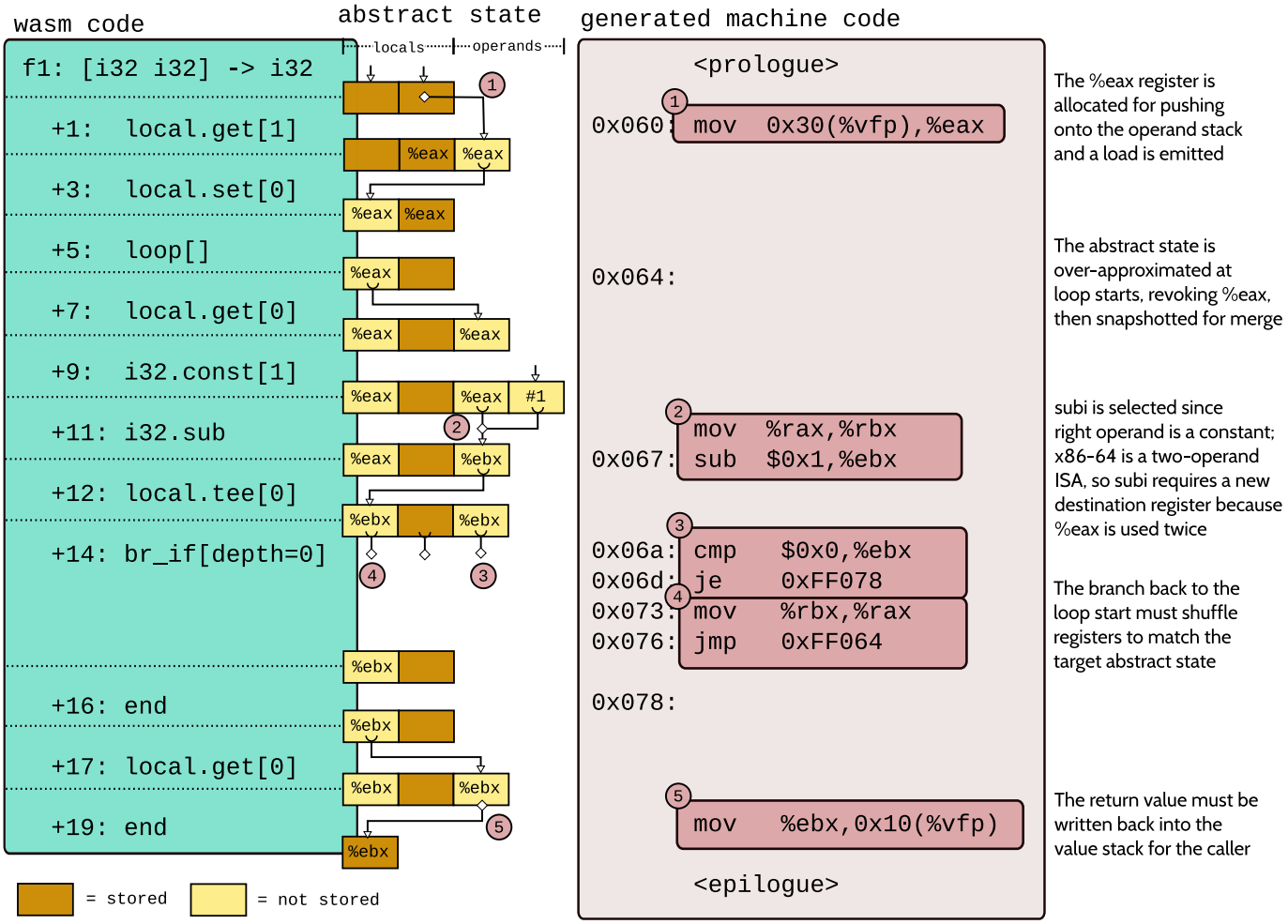

Fig. 1. Illustration of single-pass compilation using abstract interpretation. (Actual code emitted by the Wizard single-pass compiler.)

label which represents the place in the machine code where branches targeting it will jump.

After emitting a few machine instructions for the prologue, compilation procedes by examining each instruction in sequence. Instructions that access locals (**`local.get`**, **`local.set`** and **`local.tee`**) manipulate the abstract state. Depending on whether the local or top-of-stack is allocated to a register, the compiler may emit a load or store instruction, but often emits no code at all. We'll see in the next section that variations in the abstract state of the compilers we examined impact the amount of moves generated. For constant-generating instructions like **`i32.const`**, abstract values can model concrete values and avoid generating any code at all. Of the six compilers we study, all but one model constants.

Control flow requires the compiler to manage *snapshots* of the abstract state that represent the contents of registers and the stack at labels, i.e. merges in control flow. All constructs except **`loop`** have their label at the end, which means that all branches to the label will be seen before the label itself. For **`loop`**, absent any knowledge of the code in the loop body, compilers must over-approximate the abstract state before compiling the loop body, e.g. by assuming all slots could be modified on backedges.

A key design consideration in making a fast compiler is efficiently snapshotting the abstract state and merging states coming from multiple branches, since the abstract state can have tens of thousands of slots for large functions. Different compilers we studied have different strategies, either making copy extremely cheap (i.e. `memcpy`), keeping a delta index, or tracking only a subset of slots and spilling the rest. A nice benefit of Wasm's structured control flow is that the snapshot for a merge point can be deallocated as soon as a control construct is exited. These considerations help avoid JIT bombs, which are small programs that exploit a non-linearity in the algorithmic complexity of a compiler as a form of denial-of-service attack [21].

The deceptively simple compilation approach is quite tricky to implement correctly and efficiently, but nevertheless yields surprisingly good code, as can be seen in the example. Wasm's control flow design helps here; labels (other than **`loop`**) will have had all their predecessors visited before they are reached, allowing abstract interpretation to propagate constants

to merge points in a single forward pass. All-in-all, a single-pass compiler can peform:

- **register allocation** - if abstract values track register occupancy for each slot, codegen can elide code for most local accesses, often just updating the abstract state,
- **constant-folding** - if abstract values track constants, codegen can compile-time-evaluate side-effect free instructions, producing more constants,
- **branch-folding** - if abstract values track constants, then branches whose input condition is a constant can be removed or compiled to unconditional jumps,
- **strength-reduction** - if abstract values track constants, then some simple patterns such as `(i32.add x (i32.const 0))` can be reduced or eliminated,
- **instruction selection** - if abstract values track register occupancy, codegen can select memory or register addressing modes, and if abstract values track constants, it can emit immediate-mode instructions such as `addi`,
- **avoid redundant spills** - if the abstract values track spill state, codegen can avoid repeated spills to the stack in subsequent instructions, and
- **peephole optimization** - if codegen can peek one or more instructions ahead, it can combine multiple instructions, such as a compare and a branch.

From our study of baseline Wasm compilers, code generation for most opcodes (over 440 in Wasm today) is tedious and formulaic but not intrinsically difficult. The crux of good single-pass compilation is two subtle things that require careful data structure design. First, **managing the abstract state**, whose size is proportional to the locals and operand stack, must be done carefully and efficiently at all control flow points (branches, loops, and merges) to avoid (or at least mitigate) quadratic compilation time. And second, abstract values should model constants *and* registers, allowing **efficient forward-pass register allocation** so that most Wasm instructions use machine registers and avoid spills. The two are intertwined; the abstract state of all compilers contains register assignments and must be checkpointed at control flow split points and merged at control-flow join points.

IV. Baseline Compiler Integration

A JIT compiler in any virtual machine must integrate with other execution tiers and services such as debugging, instrumentation, and garbage collection. Thus, in a mature system, a JIT compiler becomes invisible, and users experience better performance with no loss of functionality. This becomes progressively more complicated with more execution tiers, as handoff between different types of code efficiently can involve very delicate machine code tricks. In this section we cover aspects of integrating **`Wizard-SPC`** that motivated and are in turn constrained by **`Wizard`**'s prior design decisions.

A. In-place Interpreter Integration

Prior to this work, the Wizard Research Engine was an interpreter-based system with debuggability and introspection as the main priorities. The in-place interpreter [20] (hereafter referred to as **`Wizard-INT`**) executes Wasm code without rewriting, allowing tracing, profiling, and debugging in terms of the original bytecodes and offsets.

Relevant points on the **`Wizard-INT`** design are:

- Interpreter performance is competitive with production interpreters for Wasm.
- The value stack is explicitly emulated at runtime, including locals and operand stack values.
- Stack walking uses *value tags* to precisely find GC roots (**`externref`** and Wasm GC objects).
- Users can insert *probes* into bytecode locations which call back to instrumentation and implement tracing, debugging, and profiling in an extensible way.

The last three points were addressed in **`Wizard-SPC`** by 1) using an identical value stack and nearly-identical execution stack layout as **`Wizard-INT`**, 2) emitting (and optimizing) value tag stores in JITed code, and 3) emitting efficient callbacks to user code and 4) intrinsifying key probe kinds.

B. Value Stack and Execution Frame Layouts

As we've seen, all single-pass compilers for Wasm use abstract interpretation to statically compute the operand stack height and approximate stack contents at every instruction in a function. Some of the baseline compilers we studied *reallocate* the storage of operand stack slots and locals to machine stack slots and registers, i.e. they scramble the stackframe layout. Scrambling the stack creates a mapping problem for debugging and instrumentation: where are original values stored on the machine stack? This metadata imposes a space cost, and is remarkably complex, tricky and error-prone. In fact, of the five previous compilers we studied, only two support introspection in their baseline compilers; the others just *do not support debugging at all*.

`Wizard`'s baseline compiler is meant to integrate with **`Wizard-INT`** that has an exact model of the value stack. It does not scramble the stack, and moreover, uses a nearly identical execution frame layout between the interpreter and JITed code. In Figure 2, we see the layout of execution frames in **`Wizard`** for the interpreter and JIT code. Both use the same value stack representation for storing Wasm values and only differ in their native execution (machine stack) frames. In particular, interpreter frames contain bytecode-level pointers (**`IP`**), a sidetable pointer (**`STP`**), and additional metadata. When executing in the interpreter, more registers are needed to store these additional pointers, whereas in JIT code, only the value frame pointer (**`VFP`**), instance (**`inst`**), and memory base (not shown) are needed. That leaves more registers to be allocated to compiled code. While values are in registers, the value stack in memory may not be up-to-date. At observable points like outcalls, JITed code simply writes values into the value stack in memory. For stacktraces, instrumentation, and debugging, the current program counter (i.e. bytecode offset) can be recomputed from the machine code instruction pointer or explicitly saved into the execution stack.

The compatibility between the two frame layouts allows **`Wizard`** to *tier-up* from **`Wizard-INT`** to baseline-compiled

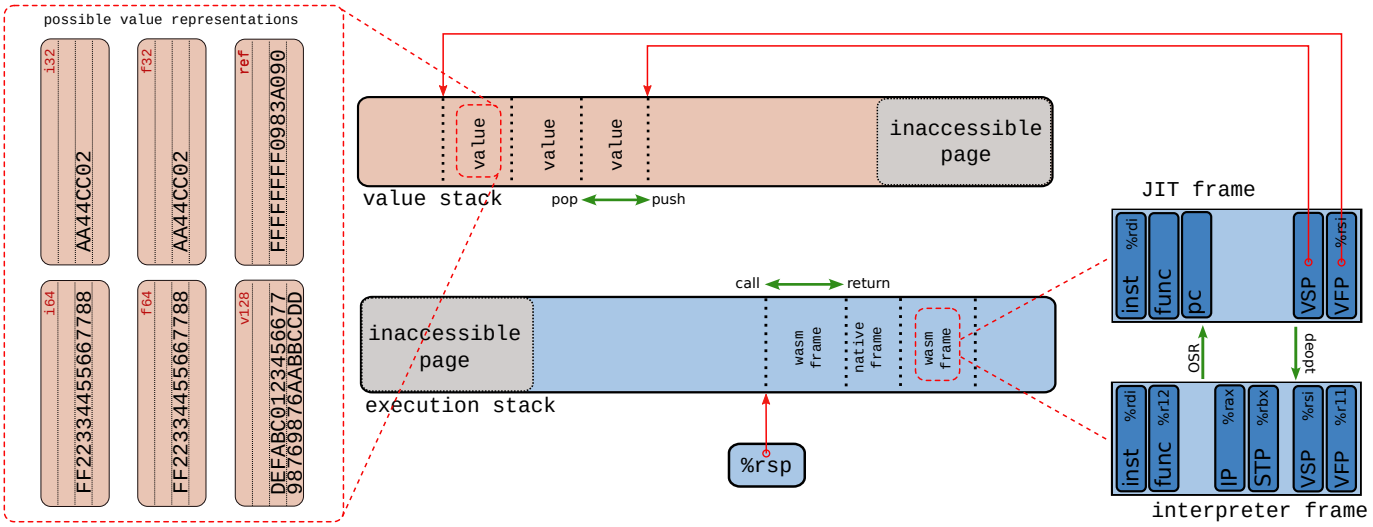

Fig. 2. Execution frame and value stack layout for **`Wizard-INT`** and **`Wizard-SPC`**. Both kinds of execution frames are the same number of machine words, allowing quick tier-up (OSR) and tier-down (deopt) by rewriting execution frames in place. The interpreter directly manipulates the value stack in memory, while JIT code only spills to the value stack when registers are exhausted and across calls.

code (e.g. when a function is detected as *hot*) by changing only the execution frame and jumping into JITed machine code. Conversely, **`Wizard`** can *tier-down* (for debugging or to support user instrumentation) by simply reconstructing **`IP`** and **`STP`** and jumping back into **`Wizard-INT`**.

C. Value Tags Versus Stackmaps for GC

Wasm code can contain references to host objects as values of type **`externref`**. In a host environment with precise garbage collection, the VM must find all roots, including those that may be in the Wasm value stack. There are two basic strategies that allow the VM to distinguish references from non-references: *stackmaps* or *value tags*. The primary difference between the two is that stackmaps are basically static and value tags are basically dynamic.

Stackmaps. For JIT-compiled code, compilers often emit metadata called *stackmaps* attached to the code which encodes how to find references in stack frames of JITed code. Such metadata usually adds space proportional the size of JITed code, so it is often very compactly stored. It is also notoriously hard to get right, as bugs in stack walking logic or errors in compressed metadata result in VM-level crashes that are insanely tedious to debug[4]. Despite the added complexity and potential robustness problems of stackmaps, they have less dynamic cost, normally only used during GC.

Value Tags. Value tags are an entirely dynamic strategy where values themselves contain the metadata that distinguishes references from non-references. This metadata can be encoded in various ways, such as a tag bit, an indirection, a value range restriction, or often an additional byte or word, such as a tag byte or dynamic type information. The possibilities for encodings varies with the kinds of values that are used to implement the guest language. Value tags allow the VM to easily inspect a value anywhere in memory (such as GC scanning stacks for references) making it vastly simpler and more robust. Another important advantage is that a JIT compiler may save compile time, space, and complexity by skipping stackmaps altogether. A disadvantage is the dynamic cost, since tags require additional space and may introduce dynamic checks.

Of the Wasm engines in the wild, including the ones containing the six baseline compilers, none use value tags except **`Wizard`**. These systems either do no precise garbage collection at all, needing no stackmaps, or they reuse the battle-tested stackmap logic of their host system, as is the case in all Web engines. Since **`Wizard`** makes unusual choices here, we evaluate some of the tradeoffs specific to that design in the experimental section.

Optimizing Value Tags. The dynamic cost of value tags can be reduced with compiler optimization. While an optimizing compiler can use a sophisticated global register allocator to only store tags on spills, a baseline compiler cannot afford an IR. Instead, we outline three optimizations for reducing the dynamic cost of value tags in a single-pass compiler.

- ***lazy tagging*** of locals. Since Wasm is a typed bytecode, local variables have static types that do not change during the execution of a function. Thus the types of locals can determined from the first bytes of a function body. Instead of writing value tags at runtime, the stackwalker computes them on-the-fly by decoding local declarations in the original bytecode, needing no additional metadata.
- ***lazy tagging*** of operand stack. While the types of local variables of a Wasm function don't change during execution, the types of operand stack slots certainly can. With this optimization, the compiler omits tag stores for operand stack slots. Like lazy tagging for locals, types are reconstructed at stackwalking time, but this is more

[4]Generally the least welcome type of GC+JIT bug.

Name	Language	Year	Features	Description
wizeng-spc	Virgil	2023	MR K KF ISEL TAG MV	The Wizard Research Engine's single-pass compiler.
wazero	Go	2022	R	An open-source engine written in Go [22].
wasm-now	C++	2022	MR K ISEL	A research project using Copy&Patch [23] code generation.
wasmer-base	Rust	2020	R K MV	The **--singlepass** option of **wasmer** [24].
v8-liftoff	C++	2018	MR K ISEL MAP MV	The baseline Wasm compiler in **V8** [25].
sm-base	C++	2018	MR K ISEL MAP MV	The baseline Wasm compiler in **Spidermonkey** [26].

Fig. 3. WebAssembly baseline compilers used in this study. MR = multiple register allocation, R = register allocation, K = constant tracking, KF = constant-folding, ISEL = instruction selection, TAG = value tags, MAP = stackmaps, MV = multi-value, a Wasm feature where blocks can return (multiple) values.

complicated than for locals, because the types could be different at each bytecode. That means storing additional metadata (basically a stackmap), or reconstructing them from the bytecode by revalidating the code.

- ***on-demand tagging*** using abstract interpretation. The default for **Wizard-SPC**, value tag stores are only emitted by the compiler across possible observations (calls, traps, and instrumentation) and the abstract state tracks whether each slot has an up-to-date tag. Parameters are assumed to have their tags stored by the caller.

We evaluate these alternatives in the experiments section by comparing with the worst-case overhead (an implementation that always stores value tags at each instruction, exactly as an interpreter would do) and the best-case alternative of simply disabling value tags. As we will see, on-demand tagging mostly eliminates tag overhead.

D. Supporting and Optimizing Instrumentation

Like **Wizard-INT**, **Wizard-SPC** supports flexible instrumentation via local *probes*, which are user callbacks that fire before a given instruction is executed. **Wizard** exposes an API that allows user code called a *monitor* to instrument Wasm modules as they are loaded. Probes are written in the implementation language of engine and when fired can access both engine APIs and the state of the Wasm program, including the values in execution frames, memory, tables, etc. In the interpreter, a probed instruction triggers a call to **Wizard**'s runtime system which looks up attached probes and fires them, passing an opaque, lazily-allocated *accessor object* that exposes methods to access to the frame's internal state. Probes are supported transparently, and more efficiently, in **Wizard-SPC**. **Wizard-SPC** inserts direct calls at probed instructions, saving indirection through the runtime by statically determining which probes to fire for which instruction. **Wizard-SPC** further optimizes certain types of probes by emitting specialized machine code, such as a direct increment of a counter's value or directly passing the top-of-value-stack to a probe, eliding the accessor object.

V. Baseline Compiler Comparison

We studied the implementation of six single-pass compilers for WebAssembly that employ the basic abstract-interpretation algorithm. The table in Figure 3 compares their designs in terms of features. In particular, we find that both Web engine compilers (**v8-liftoff** and **sm-base**) implement GC with stackmaps, using the same metadata format as their optimizing compilers. As discussed, **Wizard-SPC** uses value tags, and the three remaining compilers *do no GC*, because their host environment is not garbage-collected. A key feature is *multiple register allocation*, where the abstract state allows a register to be used for more than one slot. This is more complex to track and merge efficiently, but experimental results show it significantly improves code quality. All compilers except **wazero** track constants. Experiments also show that tracking constants measurably improves code quality, as it allows some local instruction selection. Of the six, only **Wizard-SPC** performs constant-folding and branch-folding, though our experiments show that it has marginal benefit for the benchmarks studied.

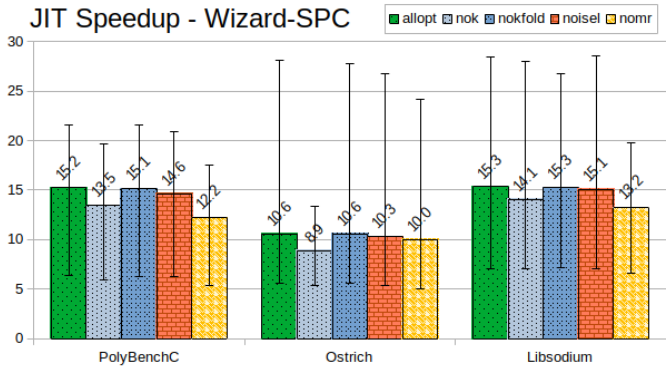

Fig. 4. Execution time speedup of **Wizard-SPC** over **Wizard-INT** (1× = same speed, 10 = 10× faster, *up* is better).

VI. Experiments

This section details a number of experiments we conducted to evaluate **Wizard-SPC**'s optimizations and design choices, compare it against other baseline compilers, and place baseline compilers in context with other tiers.

Benchmark Suites. We use three different benchmark suites: PolyBenchC [27], an often-used suite of numerical kernels, Libsodium [28] a suite of cryptographic primitive benchmarks, and Ostrich [29]. Each of these suites consists of a number of *line-items* comprised of different programs (28 for PolyBenchC, 39 for Libsodium, and 11 for Ostrich), each compiled into a separate Wasm module.

A. Speedup over Interpreter

Our first experiment evaluates speedup over **Wizard**'s existing configuration with its in-place interpreter (**Wizard-INT**). Here we focus on code quality by

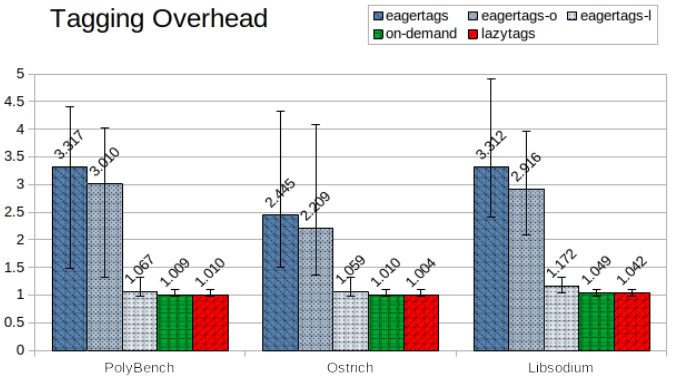

Fig. 5. Execution time of **`Wizard-SPC`** tagging configurations relative to a no-tagging configuration (1.0 = same speed as **`notags`**, *lower* is better).

measuring the *main execution time*, the time from the start of the program's main function until program exit. This intentionally factors out VM startup and compilation time, pitting the interpreter speed against the speed of compiled code directly. We study startup and compilation time in following experiments.

We evaluate five different optimization settings of **`Wizard-SPC`** to assess the impact of each optimization.

- **allopt** - (default) all optimizations turned on.
- **nok** - abstract values do not track constants, thus no constant-folding or instruction selection.
- **nokfold** - no constant-folding or branch-folding.
- **noisel** - no instruction selection, e.g. immediate modes.
- **nomr** - no "multi-register" support; a register can cache at most one slot at a time.

Figure 4 summarizes speedups across the three benchmark suites. For each configuration, we run each benchmark line item 25 times, each time in a separate VM instance (9750 data points). The height of each bar corresponds to the average speedup across line items in that suite. Note the error bars are not measurement variance[5], but variance amongst line items in that suite, i.e. the minimum and maximum average speedup for any line item.

From these results we can see that the compiled code runs between 5× and 28× faster than the interpreter for all line items, while suites averages are 10× to 15×. From the **nok** configuration, we can see that disabling constant tracking in abstract interpretation has the most dramatic effect on code quality. Disabling multiple register allocation (**nomr**) is significant, in some cases larger than disabling constant tracking. Finally, disabling constant-folding (**nokfold**) and instruction selection (**noisel**) are small but measurable effects.

B. Optimizations for Value Tags

Our second experiment in Figure 5 compares design alternatives for **`Wizard-SPC`**'s support for value tags. Using the same measurement methodology as the previous experiment, we measure relative main execution time of various tagging configurations. Here, the baseline in the figure is no longer **`Wizard-INT`** but **notags**, where we disabled value tags altogether, including removing their space from the value stack. The configurations tested here are:

- **eagertags** - "eagerly" store modified tags at every instruction.
- **eagertags-o** - "eagerly" store tags for operand slots only.
- **eagertags-l** - "eagerly" store tags for locals only.
- **on-demand** - (default) store tags on-demand by tracking their state in abstract interpretation.
- **lazytags** - store tags on-demand, but leave tagging of locals to the stack walker.

Figure 5 shows the average relative main execution time over the line-items in each benchmark suite. As before, error bars represent the minimum and maximum of any line item in the respective suite. We see that the eager-tagging imposes a 2.4×- 3.3× overhead on execution time. By measuring eager-tagging of locals separately from the operand stack, we can attribute that overhead mostly to tagging of the operand stack[6]. We also see that the default **on-demand** tagging strategy almost completely eliminates the cost of value tags, within 0.9 - 4.9% of the ideal **notags** configuration. We can also see that **lazytags** can further reduce the tagging overhead of **on-demand**, statistically measurable, but the improvement is marginal, to 0.4 - 4.2% on average. Given that **lazytags** would imply design complexity to perform tagging in the stack walker, it was not productionized.

C. Instrumentation Optimizations

Our next experiment evaluates the effectiveness of **`Wizard-SPC`**'s optimizations targeting instrumentation. In Figure 6, we report measurements with the *branch monitor*, a standard **`Wizard`** tool that profiles the targets of all conditional branches using a local probe that reads the top-of-value-stack. We show the increase in main execution time normalized to each line item's execution time on the interpreter, grouped by benchmark suite, and with error bars as before. In interpreted mode (**int**), this monitor imposes a moderate 20-49% average slowdown per suite. Without optimization (**jit**), **`Wizard-SPC`** simply emits calls to probe code which produces similar but slightly lower overhead. It's similar because the overhead consists of runtime calls, the allocation of the accessor object, and accesses of the value stack through it, which are all in engine code. In the (**optjit**) configuration however, **`Wizard-SPC`** emits direct calls to the probe passing the top-of-stack value, skipping the runtime and accessor object allocation, which reduces overhead by approximately 10×. Of course JIT code runs 10-30× faster than the interpreter, so renormalizing the data in Figure 6 to the JIT baseline, without optimizations the branch monitor slowdown is 5.4-9×, which reduces to 42-77% with optimization.

[5]While there is significant variance amongst line items, measurements for a single line item are stable within a small variance.

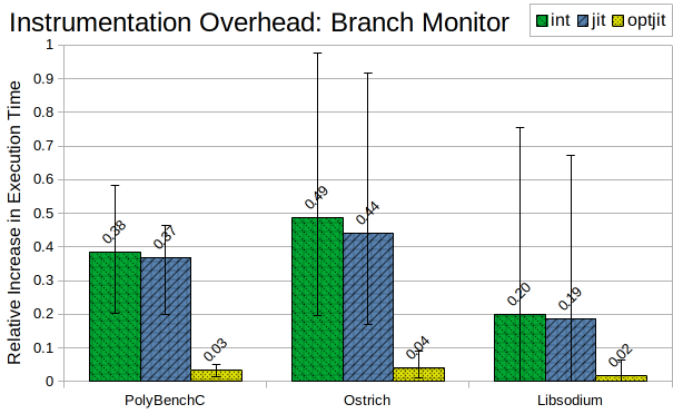

Fig. 6. Probe overhead in **`Wizard-INT`** and **`Wizard-SPC`**, reported as the increase in execution time relative to the interpreter. (0.0 = same speed, 1.0 = increase of 1× the interpreter execution time; *lower* is better).

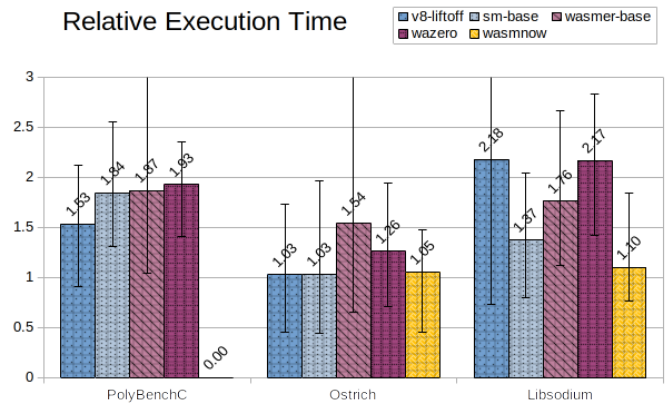

Fig. 7. Relative execution time over **`Wizard-SPC`** for other baseline compilers. (1.0 = same speed, 2.0 = 2× as long; *lower* is better).

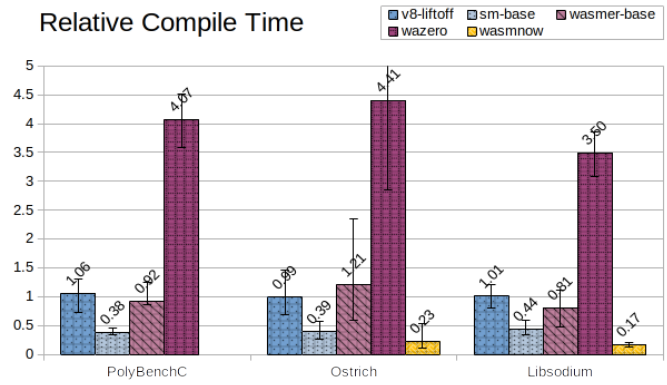

Fig. 8. Relative compilation time over **`Wizard-SPC`** for other engines in their baseline compiler configurations. (1.0 = same speed, 2.0 = 2× as long; *lower* is better).

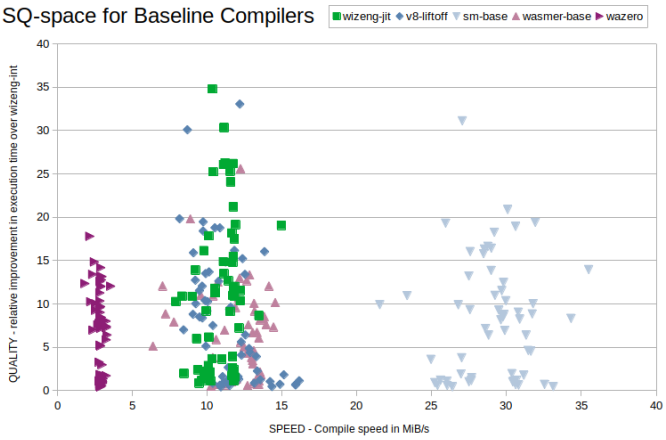

Fig. 9. SQ-space comparison for baseline compilers. Quality is measured by the relative improvement in execution time over **`Wizard-INT`**. (1.0 = same speed, 2.0 = 2× as fast; *up* and *right* are better).

D. Baseline Shootout

Our next experiment compares the compile speed and code quality of baseline compilers listed in Figure 3. To gather compile times, we instrumented each engine to measure and report the time taken to compile each module, as well the number of input Wasm code bytes. We compute the compile time as the time taken *per byte* of input code, which naturally normalizes across different function and benchmark sizes[7]. We normalize the results relative to **`Wizard-SPC`** for each line item.

Figure 8 displays the results of measuring compile time. The height of each bar represents the compile time per byte of input code normalized to **`Wizard-SPC`**, averaged over the line items in each suite. The error bars represent the minimum and maximum of line items within each suite. We were not able to run **`wasmnow`** on all benchmarks; it is clearly fastest on Libsodium and Ostrich. Besides WasmNow, **`sm-base`** is the fastest compiler; nearly 3× faster than the others, and **`wazero`** is 3× to 4× slower than the others. **`Wizard-SPC`** is roughly on par with **`v8-liftoff`** in compile speed, varying between 0.6× the speed to 1.5× the speed over different line items.

To measure code quality of compilers, we compare the execution time of benchmarks relative to **`Wizard-SPC`**. For this experiment, we use a more comprehensive measurement methodology that factors in VM startup and compilation. If necessary we configure their respective engine to use *only* a specific tier, and disable on-disk caching of compiled code. Figure 7 displays the results of our measurements.

With this data we can approximate each compiler's *SQ-region* (speed-quality region), the general area in the tradeoff space for the runtime of the compiler versus the runtime of the generated code, which is characteristic of the specific compiler. Figure 9 displays the SQ-space for baseline compilers using

[6]Which is to be expected, as the operand stack is where the action is!

[7]and also controls for lazy compilation, though engines in this study were configured to eagerly compile modules when possible.

the same data as Figures 7 and 8. It uses a scatter-plot with all benchmark line items to illustrate the variance in both compilation time and execution time across items. Since many short-running benchmark line items are included, clusters towards the bottom of the graph (lower speedups) indicate where VM startup time is more significant.

Our last experiment puts baseline compilers in context with other execution tiers. We compare baseline compilers to other tiers (interpreters, optimizing JIT compilers, and ahead-of-time translations) in two dimensions: *setup time (S)* and execution speed, or *quickness*. This makes a larger *SQ-space* that is similar in nature but more general than the compiler *SQ-space* because it includes other setup costs than compiling. We define *setup time* as the time a VM takes from starting the load of a program to executing its first instruction. This therefore will characterize the per-module processing time before execution, such as loading and verifying code, building program IR, and compiling. Since most of these costs are a function of module size, it's reasonable to define their ratio as the *setup speed* and measure it in megabytes per second (*MiB/s*).

In this experiment, we measure an even larger set of Wasm execution tiers that includes several interpreters and optimizing compilers, drawn from a larger set of engines. All new compiler tiers are IR-builders, and all interpreter tiers rewrite the bytecode, with the exception of **`Wizard-INT`**. Most, but surprisingly not *all*[8], verify the bytecode. Thus every engine has some measurable per-module parsing, verification, translation, or compilation cost. Measuring setup time can be done by instrumenting engines, but requires intrusive modifications. Instead, we use a simpler strategy to empirically bound setup time without missing hidden costs. We chose this strategy because it avoids engine modifications and thus can be applied to any engine[9].

We define $T_E(m)$ as the time to execute a module m on engine configuration E. First, we measure VM startup time by executing the smallest possible Wasm module M_{nop}, which has only one function that simply returns (total module size is 104 bytes). We run this hundreds of times to get a statistically significant characterization of startup time. Next, we approximate the processing cost of each benchmark line-item by inserting an early return in its **`_start`** function, resulting in module m_0. The new module will undergo loading and processing (often compilation) in each engine, but execution time is near zero.

With measurements $T_E(M_{\text{nop}})$, $T_E(m_0)$, and $T_E(m)$:

- $T_E(m_0) - T_E(M_{\text{nop}})$ approximates[10] the *upper bound* of pre-processing time by removing VM startup time,
- $\tilde{T}_E(m) = T_E(m) - T_E(m_0)$ defines the *adjusted execution time* which is the program's execution time without VM startup or module setup time, and
- $\tilde{S}_{E,B}(m) = \frac{\tilde{T}_B(m)}{\tilde{T}_E(m)}$ defines the *adjusted speedup* of configuration E over a baseline config B.

This strategy is inherently noiser than intrusive modifications because of the variance in the (somewhat large) engine startup time $T_E(M_{\text{nop}})$.

[8] **`wasm3`** does not, in fact, verify the bytecode!

[9] A longer term goal, outside the scope of this work, is to track engine performance over time, and intrusive patches become a maintenance problem.

[10] In fact, all of these quantities are all subject to sampling error and thus form individual distributions. The resulting "crude" approximation is just another distribution that approximates processing time.

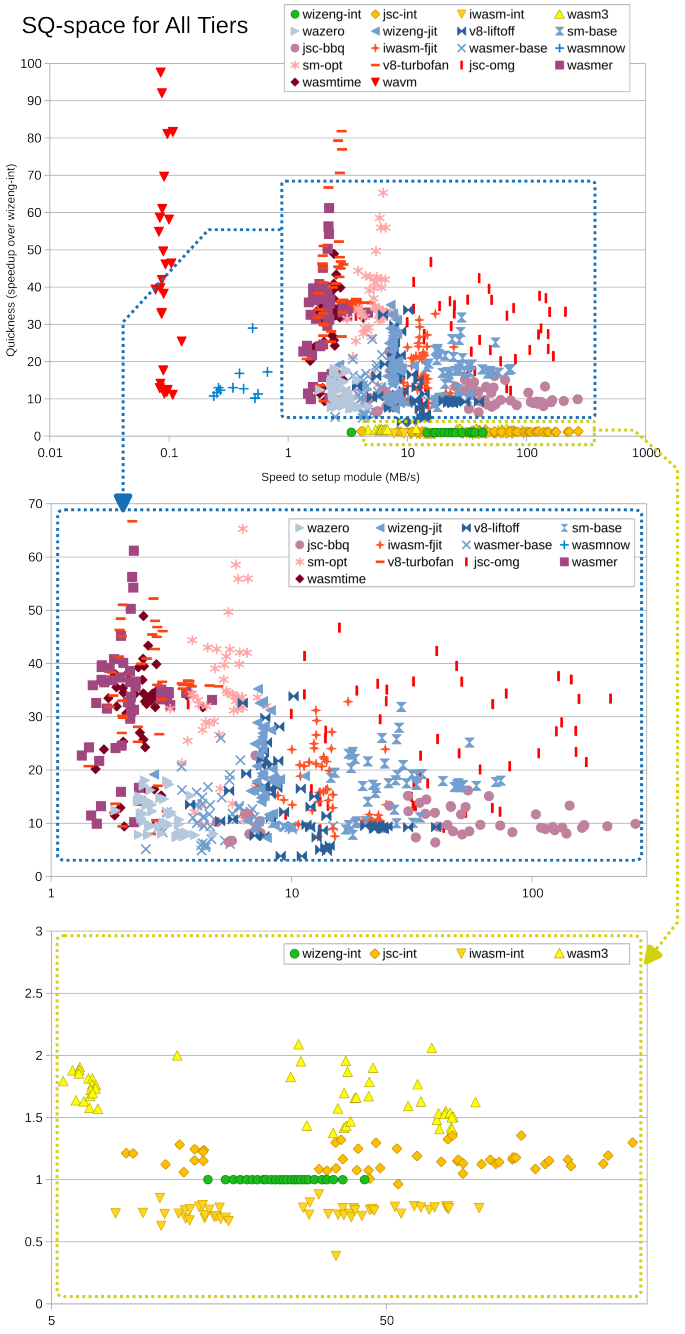

Fig. 10. The SQ-space for 18 different Wasm execution strategies.

E. Mapping the Larger SQ-space

Figure 10 presents averages of 25 runs of each of the 78 benchmark line items on 18 different engines (3 data points each = 106550 data points). The vertical axis is $\tilde{S}_{E,\texttt{wizeng-int}}(m)$ (i.e. adjusted speedup over **`Wizard-INT`**)

and the horizontal axis represents *setup speed*, (the speed of loading, verifying, and translating). New tiers are:

- **`jsc-int`**, **`jsc-bbq`**, **`jsc-omg`**, the interpreter, less optimizing, and more optimizing compiler tiers of JavaScriptCore [30].
- **`wasmtime`** [31] and **`wasmer`**, two different Wasm runtimes written in Rust which both use the Cranelift [32] optimizing compiler.
- **`wavm`** [33], a primarily ahead-of-time Wasm engine that uses LLVM.
- **`iwasm-int`** and **`iwasm-fjit`**, the interpreter and fast JIT of the WebAssembly MicroRuntime [34].
- **`wasm3`** [35], a fast rewriting interpreter for embedded systems.

In the top plot of Figure 10, we see all tiers compared. The primarily ahead-of-time **`wavm`** engine uses LLVM to compile up-front; a slow compiler, this is clearly the slowest at setting up due to a large compile time. Apparent in the zoomed-in middle plot, baseline compilers (blue colors) all cluster together in the middle; they all have very similar speedups, and though they vary by an order of magnitude in setup speed, are clearly distinguishable from optimizing compilers (red and purple colors) which definitely produce bigger speedups, about $2\times$-$3\times$ faster than baseline compilers, though at an order-of-magnitude slower compile speed than baseline. When we zoom in on interpreters in the bottom plot, it is clear they have a clear performance ceiling; they are all fairly close to each other, within $2\times$ of **`Wizard-INT`**. Interpreter setup time varies the most; we attribute this to the fact that 1) some don't verify bytecode, and 2) all the **`jsc-*`** (JavaScriptCore) tiers use lazy translation, which we could not control.

In general, laziness (i.e. translating a function upon first invocation) is a confounding factor in these measurements, as lazy compile time is not measured in setup time, but attributed to run time, and therefore the adjusted speedup is lower. As can be seen in the figure, this might be factor for the **`jsc-*`** compiler tiers, whose speedups appear lower than other optimizing compilers and setup speeds appear faster. Another confounding factor is parallelism in compilation. Some engines have fully parallel compilation pipelines and others do not[11]. We chose to leave default threading settings for all engines. Benchmark modules used in this study are fairly small, so parallel speedup may not be as big of a factor. A third confounding factor is caching of compiled code. After noticing anomalies in initial experiments[12] **`wasmtime`** and **`wasmer`**, we disabled caching in both of these.

Overall, we can see a great diversity of execution characteristics for Wasm engines, as each tier tends to occupy its own region in this space. Precision of the plot could probably be improved with metrics measured directly within engines, rather than empirically bounding them as we've done. Nevertheless, we believe the SQ-space analysis provides insight into tradeoffs in a new way and can further inform the design of tomorrow's virtual machines.

[11]Parallel speedup for multiple compiler threads may be greater for optimizing compilers due to longer work units.

[12]Optimizing tiers with instant startup? Too good to be true.

VII. Related Work

The first disk format for intermediate code was invented as early as 1968, in the first BCPL compiler's O-Code [36]. Prioritizing compiler simplicity and speed above code quality is an old idea that has roots at least as far back as the design of the first Pascal compiler [37] in 1970. Pascal compilers gave rise to the first widely-used intermediate code format, P-code [38], in the mid 1970s, which was still in use as late as 1990 [39]. P-code was certainly not the last portable low-level code, with others such as TIMI [40], LLVM bitcode [41], PNaCl [42] (itself a variant of LLVM bitcode). Fast P-code translators might be considered the first baseline compilers.

Dynamic Compilation. Over the years, many virtual machines and bytecode formats have been developed, from Smalltalk [43], to Java [44], to the Common Language Runtime (CLR). The first dynamic compilers were simple, fast, and performed little optimization. They were often instruction-by-instruction translators, with extremely simple, or even no, register allocation. They were essentially baseline compilers, but some had IR, e.g. to harness type feedback [45]. Later, runtime profiling led to more complex compilers that build and optimize IRs.

Copy & Patch Code Generation Recent work [23] on fast compilation uses code templates with data patching. The key idea is to use an offline compiler (e.g., LLVM) to generate machine code snippets under various register assignments and with "holes" for constants. When compiling Wasm, an assembler isn't needed; instead, a cache supplies the appropriate snippet for the register assignment at each step of abstract interpretation, patched with appropriate constants. Our paper evaluated the artifacts of that work, but on a subset of the benchmarks, which did confirm their blistering compile speed, but execution time was not better than other baseline compilers. Correspondence with the authors helped us understand its SQ-region in Figure 10, which is explained by the template generation occuring during VM startup. One issue with a template-based approach is that the number of templates is combinatoric in the possible abstract values. **`Wizard-SPC`** tracks value tags in its abstract state, which could potentially double or quadruple the number of templates needed.

Synthesizing and Verifying JITs. Simple compilers are easier to build, specify, verify, and even synthesize. Recent work [46] has advanced the generation of *correct* JIT compilers from a specification, which demonstrated a instruction-by-instruction compiler for eBPF running in-kernel with correctness guarantees. Another approach is to verify the output of the compiler for sandboxing properties, and has been employed for Wasm in [11].

Fast compilers in other domains. Many other domains than VMs employ dynamic code generation. Generating machine code without an intermediate representation has been repeatedly shown to dramatically improve compile speed. For

example, the VCode [47] research system improved on its predecessor, DCG [48] by 35×. Simple AST-walking compilers have been deployed in database systems and programmable networks. Regexes are often implemented with JIT compilers today. For example, all Web engines use JITs in their regex implementations [49], as well as popular libraries [50].

Fast compilers cooperate with other tiers. Today, many production virtual machines employ multiple compiler tiers. OpenJDK [52] employs an interpreter and two (tierable) JIT compilers; C2, a highly-optimizing sea-of-nodes compiler, and C1, a faster, SSA-based optimizing compiler. Web engines continue to evolve, and all employ multiple tiers for both JavaScript and WebAssembly. The V8 JavaScript engine [53] became multi-tier in 2010 when its first optimizing compiler "Crankshaft" [54] joined its fast AST-walking code generator named "full codegen" [55]. In 2018 V8 replaced both tiers with an interpreter and a new TurboFan [18] optimizing compiler, and in 2021 added a baseline compiler "Sparkplug" for JavaScript [56]. In 2023, V8 introduced a fourth compiler, Maglev [57], which sits between Sparkplug and TurboFan. The JavaScriptCore [30] virtual machine in Safari employs three different compiler designs, even briefly using LLVM as a top-tier optimizing compiler.

VIII. Conclusion

This paper captured the core design ideas of baseline compilers for Wasm and documented six implementations, which have appeared nowhere in the literature to date. As this paper documents, efficient forward-pass register allocation via abstract interpretation is widespread in single-pass Wasm compilers. Examples in this paper illustrate and experiments show that single-pass compilers for Wasm can generate good code very quickly. This paper also presented the design of a new, state-of-the-art single-pass compiler, **`Wizard-SPC`** with the unique design choice of value tags, which simplifies integration with an in-place interpreter for Wasm and the host garbage collector. Measurements show that the overhead of **`Wizard-SPC`**'s value tag approach is mostly eliminated by optimizations and that the resultant performance is on par with production single-pass compilers. Discussion compared and contrasted the six designs and experiments evaluated them on benchmarks, showing that single-pass compilers vary in code quality, primarily due to the differences in modeling constants and register allocation. Additional benchmarking data allows us to place all single-pass compilers in a two-dimensional speed-quality tradeoff space (SQ-space) with other available execution tiers for Wasm, including rewriting interpreters and optimizing compilers. We find these developments extremely exciting; the explosion of execution strategies for WebAssembly holds great promise to shed new light on long-standing tradeoffs in VM design by studying many diverse engines that all accept a common, well-specified code format and can run the same benchmark programs.

Acknowledgments

This work is supported in part by NSF Grant Award #2148301, as well as funding and support from the WebAssembly Research Center. Thanks to Hannes Payer, Toon Verwaest and Clemens Backes on the V8 team for JIT compiler and tiering discussions. Thanks to Lars Hansen (formerly Mozilla) for questions on the Spidermonkey baseline compiler design. Thanks to students Bradley Teo, Yash Anand, and Kazuyuki Takayama, and Elizabeth Gilbert for work on the instrumentation framework in **`Wizard`**. Thanks to Heather Miller, Josh Sunshine, Jonathan Aldrich, and Anthony Rowe at CMU. Thanks to Ulan Degenbaev at DFinity.

IX. Data Availability Statement

The data that support the findings of this study are openly available in the artifact at https://doi.org/10.5281/zenodo.10205323 [58].

Appendix

A. Abstract

The artifact submitted with this paper contains everything necessary to reproduce the experimental data and figures included in this paper, including a snapshot of the source code of the Wizard engine (with additional configuration options), the source code and build configurations for other Wasm engines compared in the paper, build instructions for all of them, scripts to run experiments that generate experimental data, and spreadsheets used to generate the figures. For convenience, it also includes pre-built binaries so that artifact reviewers do not have to rebuild large pieces of software, those these are not guaranteed to work on every system. The archive includes the actual experimental data recorded from experiments conducted that generated the actual figures in the paper and the spreadsheet is pre-populated with this data.

B. Artifact Check-list

The linked artifact https://doi.org/10.5281/zenodo.10205323 contains all data and scripts including:

- Source for **`wazero`**, **`WAVM`**, **`wasmer`**, **`wasmtime`**, **`WasmNow`** and **`WAMR`**.
- Binaries for all Wasm engines used in the study
- Source for Wizard, including various configuration options for experiments
- Data collected to generate figures
- Spreadsheet generating figures

Keywords. WebAssembly, virtual machines, JavaScript engines, Web engines, compiler optimizations, benchmarking.

- **Algorithm: WebAssembly compilation and execution**
- **Program: Wizard Research Engine and various other WebAssembly engines**
- **Compilation: Virgil Compiler III-7.1632**
- **Binaries: included in artifact, but can be compiled on target platform**
- **Data set: PolyBenchC, libsodium, and Ostrich open-source benchmark suites**
- **Run-time environment: Ubuntu Linux 22.04**
- **Hardware: AMD Ryzen 9 5950X 16-Core Processor 64GiB RAM**
- **Execution: benchmark shell scripts for several experiments**
- **Metrics: Execution time: relative to various baselines compilation time: comparative**

startup time: in bytes/sec
setup time: in bytes/sec

- **Output: benchmark outputs**
- **Experiments: We conduct a value tagging performance comparison, Wizeng-SPC optimization setting comparison, baseline JIT comparison, and all-tiers comparison across the metrics.**
- **How much disk space required (approximately)?: 20GiB**
- **How much time is needed to prepare workflow (approximately)?: 1 hour**
- **How much time is needed to complete experiments (approximately)?: 3 hours**
- **Publicly available?: Yes**
- **Code licenses (if publicly available)?: Apache license for Wizard, various for others**
- **Workflow framework used?: experiment scripts included**
- **Archived:** https://doi.org/10.5281/zenodo.10205323

C. Description

1) How Delivered: Source for Wasm engines was cloned from public repositories.

2) Hardware Dependencies: While the Wizard Research Engine can run on any target supported by the Virgil compiler, **`Wizard-INT`** and **`Wizard-SPC`** only support x86-64-linux platforms. Several of the comparison Wasm engines support other platforms.

3) Software Dependencies:

4) Data Sets: We use the PolyBenchC, libsodium, and Ostrich benchmark suites, binaries of which are included in the artifact.

D. Installation

Unpack the archive given by the DOI.

E. Experiment Workflow

The source archive contains engines pre-built in a (hopefully) runnable state. All of the Wizard configurations tested are included as binaries. These engines are intended to be run natively on a target machine *without* Docker or another VM to more reliably measure extremely short-running processes, like what is needed to estimate setup and engine startup time in our experiments. See the `README.md` in the archive if engines need to be rebuilt from source. Of those included in the archive, the only one that has caused trouble is JavaScriptCore, provided the requisite library dependencies are met.

Figure 4 was generated by running **`Wizard-INT`** and **`Wizard-SPC`** with different optimization configurations using the `run.bash speedup` experiment. These configurations are not standard in Wizard, but are included in the source archive and are available in a branch in the main Wizard GitHub repository. The summarized data was used to generate the plot via the `charts.ods`. The `summarize.bash <experiment>` script generates a readable tabular output. Similarly, Figure 5 was generated by `run.bash tagging`, and Figure 6 by `run.bash probe`. Two comparative experiments were performed, one against (instrumented) baseline JITs, which generated data for Figures 7, 8 and 9, and one against all (uninstrumented) tiers, which generated Figure 10. This data is generated by `run.bash translation` and `execution`. See the README in the artifact for details on how to run these scripts and the specific settings that were used to get the data in the exact format to reproduce the figures from the spreadsheet. Because of the higher measurement error in the setup time methodology (estimated via averages), some datapoints for very short-running programs were unusable; thus all negative *adjusted-execution-time* values were excluded from the scatter plot as can be seen in the pre-populated spreadsheet.

F. Evaluation and Expected Result

Running the experiments according to the instructions in the `README.md` should produce data in the output directory, and running the summarization scripts should produce tables of numbers that track what is reported in the figures. To double-check, figures can be regenerated by pasting the tabular output from summarize scripts into the appropriate places in the spreadsheet. As we've tested on at least 3 different microarchitectures, the broad results hold, though standard caution of comparing across CPUs is warranted.

G. Experiment Customization

This scripts to run experiments and summarize the results are intended to be reusable[13]. The experiments and summarize scripts have several customization dimensions. These scripts make use of a number of environment variables that allow selecting alternative benchmarking suites or line items, configuring the number of runs, as well as the engines and configurations tested. Adding new engines and engine configurations (i.e. command-line options to an engine) is done via adding symlinks in the `exp/engines/` directory according to detailed instructions in the README. The instrumentation (i.e. source modifications) added to baseline compilers gathers detailed translation (i.e. compilation time) metrics, prints the sum of compilation time gathered from the beginning of compilation to the end. For modified engines, patches are part of the source control (i.e. the last commit in the git history). Modifications to engines not required to evaluate *setup time*, as discussed in the paper; m_0 modules that immediately exit, which are included, are used to estimate startup time on unmodified engines.

References

[1] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 185–200. [Online]. Available: https://doi.org/10.1145/3062341.3062363

[2] L. Friedman, "The AutoCAD Web App at Google I/O 2018," https://blogs.autodesk.com/autocad/autocad-web-app-google-io-2018/, 2018, (Accessed 2023-5-7). [Online]. Available: https://blogs.autodesk.com/autocad/autocad-web-app-google-io-2018/

[3] T. Nattestad and N. Al-Shamma, "Photoshop's journey to the Web," https://web.dev/ps-on-the-web/, 2021, (Accessed 2023-5-7). [Online]. Available: https://web.dev/ps-on-the-web/

[4] T. Hou and T. Mullen, "Background features in Google Meet, powered by Web ML," https://ai.googleblog.com/2020/10/background-features-in-google-meet.html, Oct 2020. [Online]. Available: https://ai.googleblog.com/2020/10/background-features-in-google-meet.html

[5] Amazon, "The amazon chime sdk now offers enhanced echo reduction," https://aws.amazon.com/about-aws/whats-new/2021/11/amazon-chime-sdk-echo-reduction/, 2021, (Accessed 2023-5-7). [Online]. Available: https://aws.amazon.com/about-aws/whats-new/2021/11/amazon-chime-sdk-echo-reduction/

[13] The spreadsheet, not so much, unfortunately.

[6] W. C. Group, "WebAssembly Specification Draft 2.0," https://webassembly.github.io/spec/core/, Jan 2023. [Online]. Available: https://webassembly.github.io/spec/core/

[7] C. Watt, "Mechanising and verifying the WebAssembly specification," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 53–65. [Online]. Available: https://doi.org/10.1145/3167082

[8] A. Kohn, D. Moritz, M. Raasveldt, H. Mühleisen, and T. Neumann, "Duckdb-wasm: Fast analytical processing for the web," *Proc. VLDB Endow.*, vol. 15, no. 12, p. 3574–3577, aug 2022. [Online]. Available: https://doi.org/10.14778/3554821.3554847

[9] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan, "Ct-wasm: Type-driven secure cryptography for the web ecosystem," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, jan 2019. [Online]. Available: https://doi.org/10.1145/3290390

[10] A. E. Michael, A. Gollamudi, J. Bosamiya, E. Johnson, A. Denlinger, C. Disselkoen, C. Watt, B. Parno, M. Patrignani, M. Vassena, and D. Stefan, "MSWasm: Soundly enforcing memory-safe execution of unsafe code," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, January 2023.

[11] J. Bosamiya, W. S. Lim, and B. Parno, "Provably-safe multilingual software sandboxing using WebAssembly," in *Proceedings of the USENIX Security Symposium*, August 2022.

[12] S. Narayan, T. Garfinkel, M. Taram, J. Rudek, D. Moghimi, E. Johnson, C. Fallin, A. Vahldiek-Oberwagner, M. LeMay, R. Sahita, D. Tullsen, and D. Stefan, "Going beyond the limits of sfi: Flexible and secure hardware-assisted in-process isolation with hfi," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 266–281. [Online]. Available: https://doi.org/10.1145/3582016.3582023

[13] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in *Proceedings of the International Conference on Internet of Things Design and Implementation*, ser. IoTDI '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 225–236. [Online]. Available: https://doi.org/10.1145/3302505.3310084

[14] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer, "Sledge: A serverless-first, light-weight wasm runtime for the edge," in *Proceedings of the 21st International Middleware Conference*, ser. Middleware '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 265–279. [Online]. Available: https://doi.org/10.1145/3423211.3425680

[15] O. Nakakaze, I. Koren, F. Brillowski, and R. Klamma, "Retrofitting industrial machines with webassembly on the edge," in *Web Information Systems Engineering – WISE 2022*, R. Chbeir, H. Huang, F. Silvestri, Y. Manolopoulos, and Y. Zhang, Eds. Cham: Springer International Publishing, 2022, pp. 241–256.

[16] D. Pinckney, A. Guha, and Y. Brun, "Wasm/k: Delimited continuations for webassembly," in *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*, ser. DLS 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 16–28. [Online]. Available: https://doi.org/10.1145/3426422.3426978

[17] M. Paul, "Cve-2020-8835: Linux kernel privilege escalation via improper eBPF program verification," https://www.zerodayinitiative.com/blog/2020/4/8/cve-2020-8835-linux-kernel-privilege-escalation-via-improper-ebpf-program-verification, 2020, (Accessed 2023-5-7). [Online]. Available: https://www.zerodayinitiative.com/blog/2020/4/8/cve-2020-8835-linux-kernel-privilege-escalation-via-improper-ebpf-program-verification

[18] "Turbofan: V8's optimizing compiler," https://v8.dev/docs/turbofan, 2018, (Accessed 2021-07-29). [Online]. Available: https://v8.dev/docs/turbofan

[19] C. Click and M. Paleczny, "A simple graph-based intermediate representation," *SIGPLAN Not.*, vol. 30, no. 3, p. 35–49, mar 1995. [Online]. Available: https://doi.org/10.1145/202530.202534

[20] B. L. Titzer, "A fast in-place interpreter for webassembly," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, oct 2022. [Online]. Available: https://doi.org/10.1145/3563311

[21] A. Gal, C. W. Probst, and M. Franz, "A denial of service attack on the java bytecode verifier," University of California, Irvine, Tech. Rep., November 2003.

[22] Tetrate.io, "WAZERO: The zero-dependency WebAsssembly runtime for Go developers." https://wazero.io/, 2022, (Accessed 2023-5-7). [Online]. Available: https://wazero.io/

[23] H. Xu and F. Kjolstad, "Copy-and-patch compilation: A fast compilation algorithm for high-level languages and bytecode," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi.org/10.1145/3485513

[24] "Wasmer: A Fast and Secure Webassembly Runtime," https://github.com/wasmerio/wasmer, 2021, (Accessed 2021-07-06). [Online]. Available: https://github.com/wasmerio/wasmer

[25] C. Backes, https://v8.dev/blog/liftoff, 2018, (Accessed 2022-4-07). [Online]. Available: https://v8.dev/blog/liftoff

[26] "SpiderMonkey: Mozilla's JavaScript and WebAssembly engine," https://spidermonkey.dev, 2021, (Accessed 2021-07-29). [Online]. Available: https://spidermonkey.dev

[27] M. J. Reisinger, "Polybenchc," https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1, 2016, (Accessed 2023-5-7). [Online]. Available: https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1

[28] F. Denis, "Libsodium WebAssembly benchmarks," https://github.com/jedisct1/webassembly-benchmarks, 2021, (Accessed 2023-5-7). [Online]. Available: https://github.com/jedisct1/webassembly-benchmarks

[29] D. Herrera, H. Chen, E. Lavoie, and L. Hendren, "Numerical computing on the web: Benchmarking for the future," *SIGPLAN Not.*, vol. 53, no. 8, p. 88–100, apr 2020. [Online]. Available: https://doi.org/10.1145/3393673.3276968

[30] "JavaScriptCore, the built-in JavaScript engine for WebKit," https://trac.webkit.org/wiki/JavaScriptCore, 2021, (Accessed 2021-07-29). [Online]. Available: https://trac.webkit.org/wiki/JavaScriptCore

[31] "Wasmtime: a standalone runtime for WebAssembly," https://github.com/bytecodealliance/wasmtime, 2021, (Accessed 2021-08-11). [Online]. Available: https://github.com/bytecodealliance/wasmtime

[32] C. authors, "Cranelift code generator," https://github.com/bytecodealliance/wasmtime/tree/main/cranelift, 2018, (Accessed 2023-5-7). [Online]. Available: https://github.com/bytecodealliance/wasmtime/tree/main/cranelift

[33] "WAVM: a non-browser WebAssembly virtual machine," https://github.com/WAVM/WAVM, 2018, (Accessed 2022-1-10). [Online]. Available: https://github.com/WAVM/WAVM

[34] "WebAssembly Micro Runtime (WAMR)," https://github.com/bytecodealliance/wasm-micro-runtime, 2022, (Accessed 2022-04-11). [Online]. Available: https://github.com/bytecodealliance/wasm-micro-runtime

[35] "Wasm3: The fastest WebAssembly interpreter, and the most universal runtime." https://github.com/wasm3/wasm3, 2020, (Accessed 2021-08-11). [Online]. Available: https://github.com/wasm3/wasm3

[36] M. Richards, "The BCPL Cintsys and Cintpos User Guide," University of Cambridge, 2023. [Online]. Available: https://www.cl.cam.ac.uk/~mr10/bcplman.pdf

[37] H. Mössenböck, "Compiler construction - the art of niklaus wirth." 01 2000, pp. 55–68.

[38] E. Gilbert, "P-Code intermediate assembler language," USA, Tech. Rep. 148, Mar 1978.

[39] K. Louden, "P-code and compiler portability: Experience with a Modula-2 optimizing compiler," *SIGPLAN Not.*, vol. 25, no. 5, p. 53–59, may 1990. [Online]. Available: https://doi.org/10.1145/382080.382632

[40] T. Van Looy, "The IBM AS/400: A technical introduction," https://www.scss.tcd.ie/SCSSTreasuresCatalog/hardware/TCD-SCSS-T.20121208.068/IBM-AS400-technical-introduction.pdf, Jan 2009, (Accessed 2023-5-7). [Online]. Available: https://www.scss.tcd.ie/SCSSTreasuresCatalog/hardware/TCD-SCSS-T.20121208.068/IBM-AS400-technical-introduction.pdf

[41] "LLVM bitcode file format," https://llvm.org/docs/BitCodeFormat.html, 2010, (Accessed 2023-5-7). [Online]. Available: https://llvm.org/docs/BitCodeFormat.html

[42] A. Donovan, R. Muth, B. Chen, and D. Sehr, "PNaCl : Portable Native Client executables," 2010, (Accessed 2023-5-7).

[43] L. P. Deutsch and A. M. Schiffman, "Efficient implementation of the smalltalk-80 system," in *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '84. New York, NY, USA: Association for Computing Machinery, 1984, p. 297–302. [Online]. Available: https://doi.org/10.1145/800017.800542

[44] T. Lindholm and F. Yellin, *Java Virtual Machine Specification*, 2nd ed. USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[45] U. Hölzle and D. Ungar, "Optimizing dynamically-dispatched calls with run-time type feedback," *SIGPLAN Not.*, vol. 29, no. 6, p. 326–336, jun 1994. [Online]. Available: https://doi.org/10.1145/773473.178478

[46] J. Van Geffen, L. Nelson, I. Dillig, X. Wang, and E. Torlak, "Synthesizing jit compilers for in-kernel dsls," in *Computer Aided Verification*, S. K. Lahiri and C. Wang, Eds. Cham: Springer International Publishing, 2020, pp. 564–586.

[47] D. R. Engler, "Vcode: A retargetable, extensible, very fast dynamic code generation system," in *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, ser. PLDI '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 160–170. [Online]. Available: https://doi.org/10.1145/231379.231411

[48] D. R. Engler, W. C. Hsieh, and M. F. Kaashoek, "C: A language for high-level, efficient, and machine-independent dynamic code generation," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 131–144. [Online]. Available: https://doi.org/10.1145/237721.237765

[49] E. Corry, C. Plesner Hansen, and L. R. H. Nielsen, "Irregexp, google chrome's new regexp implementation," https://blog.chromium.org/2009/02/irregexp-google-chromes-new-regexp.html, 2009, (Accessed 2023-5-7). [Online]. Available: https://blog.chromium.org/2009/02/irregexp-google-chromes-new-regexp.html

[50] Z. Herczeg, "Extending the pcre library with static backtracking based just-in-time compilation support," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 306–315. [Online]. Available: https://doi.org/10.1145/2581122.2544146

[51] J. Fransham, "Introducing Lightbeam: An optimizing streaming WebAssembly compiler," https://www.parity.io/blog/lightbeam-webassembly-compiler/, 2019, (Accessed 2023-5-7). [Online]. Available: https://www.parity.io/blog/lightbeam-webassembly-compiler/

[52] "OpenJDK: Open Java Development Kit," https://openjdk.org, 2007, (Accessed 2023-5-7). [Online]. Available: https://openjdk.org

[53] "V8 development site," https://v8.dev, 2021, (Accessed 2021-07-29). [Online]. Available: https://v8.dev

[54] K. Millikin, "A New CrankShaft for V8," https://blog.chromium.org/2010/12/new-crankshaft-for-v8.html, 2010, (Accessed 2023-05-7). [Online]. Available: https://blog.chromium.org/2010/12/new-crankshaft-for-v8.html

[55] A. Wingo, "Inside full-codegen, v8's baseline compiler," https://wingolog.org/archives/2013/04/18/inside-full-codegen-v8s-baseline-compiler, 2013, (Accessed 2023-05-7). [Online]. Available: https://wingolog.org/archives/2013/04/18/inside-full-codegen-v8s-baseline-compiler

[56] L. Swirski, "Sparkplug - a non-optimizing JavaScript compiler," https://v8.dev/blog/sparkplug, 2021, (Accessed 2023-5-7). [Online]. Available: https://v8.dev/blog/sparkplug

[57] V. Team, "Maglev - v8's fastest optimizing jit," https://v8.dev/blog/maglev, 2023, (Accessed 2023-12-18). [Online]. Available: https://v8.dev/blog/maglev

[58] B. L. Titzer, "Artifact for whose baseline compiler is it anyway?" https://doi.org/10.5281/zenodo.10205323, 2023, (Accessed 2023-11-24). [Online]. Available: https://doi.org/10.5281/zenodo.10205323