Partial Evaluation, Whole-Program Compilation

CHRIS FALLIN[*], F5, USA
MAXWELL BERNSTEIN, Recurse Center, USA

There is a tension in dynamic language runtime design between speed and correctness. State-of-the-art JIT compilation, the result of enormous industrial investment and significant research, achieves heroic speedups at the cost of complexity. This complexity leads to subtle and sometimes catastrophic correctness bugs. Much of this complexity comes from the existence of multiple tiers and the need to maintain correspondence between these separate definitions of the language's semantics; it also comes from the indirect nature of the semantics implicitly encoded in a compiler backend. One way to address this complexity is to automatically derive, as much as possible, the compiled code from a single source-of-truth, such as the interpreter tier. In this work, we introduce a partial evaluator that can compile a whole guest-language function ahead-of-time, without tracing or profiling, "for free." This transform unrolls an interpreter function expressed in a standard compiler intermediate representation (static single assignment or SSA) and uses partial evaluation of the interpreter function and its regular control flow to drive the guest-language compilation. The effect of this is that the transform is applicable to almost unmodified existing interpreters in systems languages such as C or C++, producing ahead-of-time guest-language compilers. We show the effectiveness of this new tool by applying it to the interpreter tier of an existing industrial JavaScript engine, SpiderMonkey, yielding 2.17× speedups, and the PUC-Rio Lua interpreter, yielding 1.84× speedups. Finally, we outline an approach to carry this work further, deriving more of the capabilities of a JIT backend from first principles while retaining correctness.

CCS Concepts: • **Software and its engineering → Compilers**; **Interpreters**; **Translator writing systems and compiler generators**.

Additional Key Words and Phrases: partial evaluation, ahead-of-time compilation, WebAssembly

ACM Reference Format:
Chris Fallin and Maxwell Bernstein. 2025. Partial Evaluation, Whole-Program Compilation. *Proc. ACM Program. Lang.* 9, PLDI, Article 160 (June 2025), 24 pages. https://doi.org/10.1145/3729259

1 Introduction

Most dynamic language runtimes start as interpreters, for their numerous initial advantages: interpreters are easier to develop and extend than compilers; they are likewise easier to debug; and they are usually more portable, relying less on platform- or ISA-specific details to generate and execute code. Over time, dynamic language runtimes tend to build run-time type profiling and code specialization features, and, going further with more engineering investment, some develop just-in-time (JIT) compiler backends to remove interpretation overhead. A JIT is effective but not free: it is a second implementation of language semantics that may diverge in hard-to-debug ways from the interpreter, it generates specialized code that may depend on invariants that can be invalidated at run-time, and this generated code is ephemeral and thus harder to audit or debug. It also requires warmup: the engine must observe execution before producing specialized code.

[*]Work done while at Fastly.

Authors' Contact Information: Chris Fallin, F5, San Jose, California, USA, chris@cfallin.org; Maxwell Bernstein, Recurse Center, Boston, Massachusetts, USA, acm@bernsteinbear.com.

ACM 2475-1421/2025/6-ART160
https://doi.org/10.1145/3729259

This complexity can result in serious issues. The three major JavaScript engines (V8, SpiderMonkey and JSC) regularly have CVEs resulting from subtle JIT bugs (e.g. [3] as one recent example in V8). Security-conscious platforms often eschew run-time code generation for this reason. For example, iOS does not permit it outside the built-in web engine (and turns it off in this engine too in lockdown mode), and the Edge browser has a secure mode that disables JIT [36]. Even coarse-grained sandboxing, such as V8's "ubercage" [26], does not protect against correctness bugs that can violate isolation when multiple server-side tenants share one VM via isolates [10], or when multiple requests with different users' data are processed by one engine. The continued prevalence of JS engines' security bugs indicates that full language-semantics correctness is difficult to maintain in a runtime-optimized system with many subtle invariants. Additionally, even if a bug does not yield a sandbox escape, miscompilations can alter application logic in catastrophic ways.

We thus see a tension between the ever-increasing need for efficient execution of dynamic languages—manifested in the enormous engineering investments in JIT compilers and language runtime optimization—and the need for security and correctness. This is especially true as these languages are used to implement the underpinnings of modern infrastructure.

To achieve better performance without sacrificing correctness, we would first ideally *derive a compiler backend* from the language semantics expressed in an interpreter: that is, a single source of truth, written in a direct way. Second, to best address the security concerns that lead us away from JIT-based runtimes, we may wish to compile a dynamic language *ahead-of-time*. This permits stronger isolation by decoupling compilation from warmup (possibly even in a separate environment). The lack of warmup may allow separate requests or inputs to be processed by new program executions. Finally, this approach permits disallowing run-time code loading completely.

Past work has explored many techniques to automatically derive compilers from interpreters. Most prominently, *partial evaluation*, as envisioned by Futamura [21, 22] and implemented practically in Graal/Truffle [47], and *metatracing*, as implemented in PyPy [11], permit one definition of a language's semantics to provide both interpretation and compilation.

The first downside to these systems is that it they require the interpreter to be *expressed in terms of the framework*. Though this avoids re-expressing language semantics across interpreter and compiler tiers, the approach fails the *single source of truth* implementation principle in a different way: *rewriting* an interpreter (especially in a mature language implementation) is also error-prone, and introduces compatibility and migration concerns for a mature language and library ecosystem.

The second downside is that, for a mixture of fundamental and pragmatic reasons, these existing approaches work either *only as* or *best as* a JIT. Metatracing must identify program paths to compile, and so cannot compile ahead-of-time at all. Truffle's design has been optimized for JIT compilation with warmup, and though it has some provision for ahead-of-time compilation [5], its usual dynamic-language use-cases require run-time optimization, which we hope to avoid.

In this work, we propose a new partial evaluation algorithm, the `weval` transform, that addresses both of these needs. First, `weval` applies to an interpreter with its existing bytecode input, requiring only a lightweight annotations to guide the specialization. This allows its application to mature industrial interpreters, as we evaluate. Second, `weval` supports ahead-of-time compilation of whole function bodies, with no need for run-time feedback or profiling.

`weval` is a partial-evaluation transform that works on a mostly unmodified interpreter body, at the IR level: it transforms an arbitrary control-flow graph of basic blocks, in SSA form, unrolling an interpreter loop as a side-effect that falls out of a general "context specialization" mechanism (§3). Unlike metatracing, the resulting IR has a control-flow graph that replicates the bytecode's control flow, including reconvergence points and loops. Thus, `weval` permits ahead-of-time whole-function compilation. And, unlike partial evaluation techniques that are explicitly built to unroll interpreter loops, our mechanism is general, allowing this "specialization" to nest, or to be used for other kinds of static code unrolling or specialization, and to adapt to different program representations.

Our tool is an open-source [2], industrial-strength compiler that, in its initial form, processes interpreters compiled to WebAssembly [27] (Wasm); however, the algorithm could be applied to any IR that uses basic blocks of SSA, such as LLVM [30], with some minimal requirements (§3.6).

This approach provides several benefits. First, it allows easy, rapid provisioning of a compiler-based backend for a language runtime, as we show in our case study on Lua (§7) where we managed to achieve a speedup of 1.84× with a minimal code diff. Many language implementations have only interpreters and could benefit from this technique. Even established language runtimes can benefit on new platforms not supported by their existing JIT backends: for example, the SpiderMonkey JS engine (§6) has no JIT support when running on a server-side Wasm-based platform, whereas `weval` allows us to attain a 2.17× speedup on average "for free"—deriving the result from exactly the same interpreter source. Second, it provides a realistic pathway toward single-source-of-truth definitions of language semantics. We describe a future path for carrying this approach forward to include profile feedback in a *semantics-preserving* way, which could lead to a competitive JIT derived from language semantics in an interpreter.

2 Futamura Projections and Partial Evaluation

In this work, we observe that we can *automatically* produce compiled code from an interpreter body and its interpreted program input. In order to understand this further, we first need to understand how to automatically produce compiled code from an interpreter: the *Futamura projection.*

2.1 The Futamura Projection

Futamura [21, 22] introduced the concept of partial evaluation in the context of compilation: by *partially evaluating* an interpreter with its interpreted program, we obtain a compiled program. Consider an interpreted program execution as a function invocation, where the interpreter receives two inputs, the interpreted program and the input to that program: $Interp(Prog, Input)$. The key idea of the first Futamura projection is to substitute in a constant C for the $Prog$ argument, yielding a new function that we can consider a compiled form of the user program. Then we have $Compiled(Input) = Subst_{Prog=C}(Interp(Prog, Input))$. What we have described so far is the *first* Futamura projection: it is the partial evaluation of the interpreter with an interpreted program, yielding a compiled program. Futamura also defines the second and third projections: the second projection enhances compilation speed, and the third produces a compiler-compiler tool, but both are more difficult than the first projection, and we will not describe them further in this section.

2.2 Optimizing Compilation: Interpreted Program to Specialized Code

One could achieve a basic kind of compilation by joining an interpreter with a snapshot of its input (bytecode), perhaps by linking the interpreter with an additional data section and some startup code. This fits the definition of a Futamura projection in a trivial sense. However, practically speaking, this "compilation" lacks many of the properties one usually expects from compiled code. Mainly this relates to performance: the combined module retains the performance characteristics of the interpreter, because the interpreter body is unchanged. Let us now state a definition that sets a minimum bar for a compilation with the desired performance:

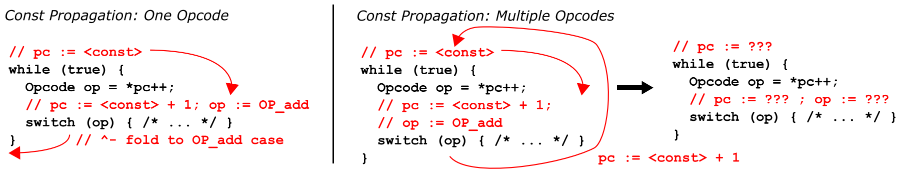
Definition 1. A partial evaluator performs **bytecode-erased compilation** on an interpreter if the resulting code does not contain the original interpreter loop or opcode-driven dynamic dispatch and incorporates all bytecode parameters as constants: if a constant value from bytecode determines a conditional branch or switch, the branch is folded, and if it is used as data, the data is a constant.

```c
void interpret(bytecode_t* pc,
               Value* stack) {
 while (true) {
  switch (*pc++) {
   case OP_add:
    Value v1 = *stack++;
    Value v2 = *stack++;
    *--stack = value_add(v1, v2);
    break;
   /* ... other opcodes ... */
  } } }
```

Fig. 1. A sketch of an interpreter loop in C.

```c
void interpret_specialized_func0(
  bytecode_t* _pc, Value* stack) {
 Value v1 = stack[0];
 Value v2 = stack[1];
 stack[1] = value_add(v1, v2);
 return;
}
```

Fig. 2. Compiled code resulting from constant propagation of `interpret` from Fig. 1 on one opcode. Branch folding retains only the relevant switch-case based on the opcode known at specialization time.


Const Propagation: One Opcode
Const Propagation: Multiple Opcodes
pc := <const> + 1

```
// pc := <const>
while (true) {
  Opcode op = *pc++;
  // pc := <const> + 1; op := OP_add
  switch (op) { /* ... */ }
} // ^- fold to OP_add case
```

```
// pc := <const>
while (true) {
  Opcode op = *pc++;
  // pc := <const> + 1;
  // op := OP_add
  switch (op) { /* ... */ }
}
```

```
// pc := ???
while (true) {
  Opcode op = *pc++;
  // pc := ??? ; op := ???
  switch (op) { /* ... */ }
}
```

Fig. 3. An illustration of constant propagation over an interpreter loop: with one iteration, we can deduce constant values, but multiple iterations cause the analysis to degrade to "unknown" because all iterations are considered together.

Because this definition requires that the interpreter's dynamic dispatch must be optimized away, the compiled code's control-flow graph (CFG) must correspond to the bytecode's CFG instead.[1]

This elimination of dynamic dispatch is both itself a speedup (in our observations, often 1.5-2×) and a substrate for further optimizations: because each instance of an opcode becomes its own static code, we can optimize it both separately and together with the opcodes around it.

2.3 Optimizing an Interpreter with its Input

The key question is: how can we practically *expand bytecode to specialized code* by partially evaluating an interpreter loop? As we will see in the rest of this section, there are various design points, requiring various compromises in the way that the interpreter is expressed.

Above we introduced an algebraic analogy to partial evaluation, namely, substituting a variable for a constant value and simplifying (optimizing). What happens if we apply the analogous compiler analysis and transform, namely constant propagation and folding?

Consider the body of the interpreter loop in Fig. 1. If we take a function `func0` of a single opcode, say `OP_add`, and we take a constant initial stack pointer offset, we might imagine taking the body of the interpreter and producing code similar to Fig. 2.

This code results because constant propagation can convert the fetch of the opcode to its constant value `OP_add`. This in turn works because we are processing a partial evaluator invocation in which the user has promised that this memory is constant ("specialize this function when this pointer points to this data"). The specializer can then branch-fold the `switch` to the one case actually taken due to the constant selector, and constant-fold the offsets from `stack`.

[1]This concept is very similar to Jones-optimality [28], which specifies that a partial evaluator should "remove all computational overhead caused by interpretation."

However, as soon as we advance to a program of *two* opcodes—before even considering control flow within the interpreted program—we run into issues with constant propagation. In fact, we glossed over the issue in the single-opcode example: how do we handle the interpreter loop backedge? A classical iterative dataflow analysis, such as constant propagation, computes a Meet-Over-All-Paths solution [6], meaning that it produces one analysis conclusion per *static program point*, merging together all paths that could reach that point. At the top of the interpreter loop (one program point), what is `pc`? When we merge all iterations together, we only reach the conclusion that it is not constant, because we are analyzing all opcodes at once. The rest of the interpreter then fails to specialize to the bytecode: `pc` is not constant, so neither is `*pc`, so we cannot branch-fold the `switch`, so the result of specialization is only a copy of the original interpreter, with nothing changed. This situation is illustrated in Fig. 3.

The heart of the issue is that in order to compile the bytecode to target code, we need to somehow iterate over the bytecode operators and emit code for each one, and ensure this iteration happens *at compile-time*. A constant-folding pass that retains the original CFG, only substituting in constants where known, will not lead to this output. Can we build an analysis that somehow knows how to *unroll arbitrary interpreter loops over the bytecode*?

One possible approach is to *unroll all loops* by analyzing the interpreter along a *trace*: in other words, discarding the Meet-Over-All-Paths principle. This approach is appealing in its simplicity. However, it can result in unbounded work: thus, it must be limited by trace size or some other metric, and it can have surprising worst-case cost.

A more targeted approach, taken by PyPy [11], is to detect *hot loops* that result from loops in the interpreted program, and trace the interpreted execution of these loops (together with annotations that identify the interpreter loop itself). PyPy then instantiates and specializes the interpreter loop once for each opcode in the loop.

This approach resolves the above limits but, as a *profile-driven* approach, it requires execution of the interpreted program before compilation can commence. In some settings, we may desire fully ahead-of-time compilation, or we may not have adequate or representative test inputs for the interpreted program (or may not be able to run it at all in the compiler's execution environment, if it has other dependencies). Additionally, it can suffer from brittle *performance cliffs*: if the control-flow path during run-time diverges from that seen during compilation, execution must revert to the interpreter (possibly ameliorated by attaching "side traces" over time). This behavior of "falling off the trace" was a well-known failure mode in the TraceMonkey JavaScript JIT [23].

The downside of both of the above options is that, in attempting to specialize an interpreter loop fully automatically, they rely on heuristics that can fail fairly easily. One alternative is to allow—or indeed, require—the interpreter author to *explicitly denote the interpreter loop* and how it should be specialized. The high-level idea would be to devolve control of the main interpretation loop to a *framework* that the partial evaluator is somehow aware of. This framework would understand the format of the interpreted program (e.g., bytecode or AST nodes), and would take care of dispatching to implementations of opcode semantics provided by the interpreter author. In this way, the partial evaluator could directly translate the bytecode or other interpreted program representation to compiled form by copying over and concatenating the implementations of each opcode.

While this ought to work robustly, because the partial evaluator is co-designed and developed with the interpreter framework, it has the major disadvantage that it requires the interpreter to be written in a way specific to this tool. An existing interpreter is likely to be difficult to port to this framework. Instead, we design a generic partial evaluator that can apply to existing interpreters.

3 The weval Transform: User-Context-Controlled Constant Propagation

We have argued that to produce a *bytecode-erased compilation*, we need to somehow *unroll the interpreter loop* during partial evaluation, analyzing the loop body separately for each interpreted-program operator. Furthermore, we wish to do this without rewriting the interpreter to conform to a framework that understands the structure of the interpreted program. Rather, we want to support an existing interpreter, with minimal modifications, using its own logic to "parse" the bytecode as we translate it opcode by opcode. In this section, we will introduce a transform that does exactly this. We call this the `weval` transform, short for "WebAssembly [partial] evaluator."

The transform operates on a function body represented as a control-flow graph (CFG) of basic blocks in static single assignment (SSA) form. Due to the problem-space that we built this tool to address (see §6), we build and use a framework that allows for SSA CFG-based Wasm-to-Wasm compilation. However, without loss of generality, this transform can apply to any IR that is a CFG of basic blocks, such as LLVM [30] (§3.6). This transform's implementation is relatively small for its power, measuring at 5 KLoC of Rust. We introduce this transform with three key ideas.

3.1 Key Idea #1: User Context

Recall that we began our discussion of the Futamura projection by noting how constant propagation addresses the problem fully in the single-opcode case, but fails as soon as more than one opcode exists in the interpreted program (Fig. 3). Specifically, when the constant-propagation analysis follows the interpreter backedge, the "next" value of the interpreter program counter conflicts with the previous value, and we conclude that nothing is constant at all.

To address this, we allow the interpreter to *selectively introduce context specialization via intrinsics* to separate the analysis of each loop iteration of the interpreter (or other loop in the original program). The intrinsic invocation appears like `update_context(pc)` at some point before the loop backedge, as shown in Fig. 4; when performing an iterated dataflow analysis for constant propagation, this causes analysis to flow to successor blocks in a *new context*. In other words, the set of program-point locations analyzed by the iterated dataflow analysis is dynamic and expandable. This will be illustrated by an example in Fig. 6, described below.

This annotation is lightweight and minimal, yet it unlocks an entire specialization pipeline: it drives code duplication only where needed to replicate the interpreter body according to the overall schema of the interpreted bytecode, and the rest of the specialization falls out. By avoiding the "meet-over-all-paths" trap that we described in §2.3, we achieve a *bytecode-erasing compilation* that produces an output CFG that follows the bytecode rather than the interpreter.

Note that context may be nested: we include intrinsics to push and pop context values. This allows value-specialization or loop unrolling to occur inside of a single opcode case in the interpreter.

The context value (`pc` here) must be a known *constant* at specialization time. This will be the case for bytecode-driven control flow with a fixed CFG, but, e.g., an opcode that computes an arbitrary bytecode destination would not be compatible with this specialization. (What CFG should result in the compiled code? Will there be an edge to every block?) To support interpreted control flow with a known but variable number of destinations, such as switch opcodes, we allow for value specialization on the selector value (§3.3).

Finally, observe that the context intrinsics are not load-bearing for correctness: they split constant-propagation context, but the `weval` transform is sound regardless of whether separate contexts are used to analyze duplicates of code. The worst that happens is that the analysis cannot derive any constant values and the transform produces the original interpreter body. Separately, we provide an intrinsic that *asserts compile-time constantness* to ensure the intrinsics are working as intended.

```
void interpret(bytecode_t* pc) {
 while (true) {
  // Loaded data from `pc` is a
  // constant, per specialization.
  switch (*pc++) { /* ... */ }
  // Update analysis context:
  // backedge reaches loop header
  // in a new context, maintaining
  // constantness of `pc`.
  update_context(pc);
 } }
```

Fig. 4. Annotations to context-specialize analysis of an interpreter function (§3.1).

```
switch (pc++) {
 case OP_CONDITIONAL_BRANCH: {
  pc = targets[specialized_value(
   stack.pop(), 0, 2)];
  break;
 }
 case OP_SWITCH: {
  pc = targets[specialized_value(
   stack.pop(), 0, num_targets);
  break;
 } }
```

Fig. 5. Annotations to value-specialize the inputs to interpreted control flow (§3.3).

3.2 Key Idea #2: Context-Specialized Code Duplication

Given a *generic* function to be *specialized* with a set of constant parameters, we now define the worklist-driven algorithm that produces the specialized function body, shown in Algorithm 1.

The algorithm operates over the generic (input) function in an SSA-based IR containing basic blocks, and takes a list of abstract states for arguments: “run-time” (do not specialize), or a constant value, or a pointer to constant data in memory. As output, it produces IR for another function that is equivalent to the input whenever the actual arguments are consistent with the constant values claimed at specialization time.

The algorithm clones both basic blocks and SSA values separately for each context they appear in, and keeps maps to track the correspondence (lines 10 and 12). It performs constant propagation as it runs, tracking state per cloned SSA value (line 14) over a standard constant-propagation lattice type (line 5). As noted above, contexts for cloning are nested sequences of user-specified values/loop counters (line 6). In practice these are interned to short integer identifiers for efficiency.

The algorithm is driven by a worklist of blocks to specialize (line 8). It does not clone everything into every context: it only visits reachable code, and visits blocks in new contexts as analysis sees “update context” intrinsics with context values provided by the co-running constant propagation.

The main worklist loop (line 25) invokes a per-block specializer (line 31) that either creates a new block or re-specializes a block if it has already been visited in a context. A (*context*, *block*) pair may be processed more than once if the abstract state at its input is updated, as with ordinary fixpoint analyses; this may happen, for example, with loops. The specialization maps ensure that we use consistent block and value numbers for a specialization in a given context. As such, when we re-specialize a block, we clear out and recreate its instructions. (For space reasons, we omit the details related to reusing value numbers, which essentially consists of map lookups.)

The block specializer transcribes instructions from the block in the input (“generic”) function to the specialized block, and performs constant propagation. This is the heart of the partial evaluation: constant values known during specialization (e.g., opcodes and parameters from bytecode) may be incorporated into the specialization output.

The specializer processes context-update intrinsics during this pass over the block: in effect, the “current context” is flow-sensitive state, known at the start of the block from the workqueue entry and possibly updated by the end of the block.

When the pass reaches the end of a block, it processes the terminator (branch instruction) in light of this context and the constant-propagation state. First, if the input to a two-way conditional branch or a switch opcode is a known constant, the branch can be folded—simplified to only the target that will always be taken. This is essential for partial evaluation of an interpreter loop: the main switch over opcode should always be folded to only one case.

The algorithm then follows the targets to continue its context-sensitive cloning traversal: given the (possibly updated) context, it looks up each block in the block-specialization map, enqueuing this context and block on the worklist if not yet present. In any case, as with ordinary iterative dataflow analysis, the block parameters (in our formulation of SSA; equivalently, inputs to ϕ-nodes) receive the abstract states from the arguments, meeting the lattice values into their existing states. If the block had already been specialized but this abstract state changes, it is re-enqueued as well.

Note that this slightly simplified algorithm as presented requires a restricted form of SSA that passes all live values through block parameters at every edge. In §3.4 we address this limitation.

In Fig. 6 we show an example of a specialization of a simple interpreter (supporting ADD, SUB and GOTO) for a bytecode program that performs ADD and SUB operations in an infinite loop. The interpreter is annotated with context updates, and the `weval` transform is invoked with a specialization argument for the "program counter" argument that indicates the memory it references is constant (i.e., *ConstantMemory*(..)). Note, however, that no other knowledge of *interpreters*, per-se, is needed: this is a general transform for duplicating and constant-specializing code.

The analysis is worklist-driven and runs until fixpoint, but (as seen in this example) in practice in most cases, makes one pass over the bytecode, emitting the portion of the interpreter-switch corresponding to each opcode. That is, the overall scheme of a single-pass template compiler *falls out automatically*, without us having to adapt the interpreter or bytecode into a framework that understands this flow. (Note, however, that unlike a classical template compiler, `weval` is able to constant-propagate across opcodes, and then produces IR that can be further optimized.)

The resulting compiled code in Fig. 6 contains a control-flow graph that corresponds to the *interpreted program*, with its loop (the JMP backedge), rather than the interpreter. This results in a *bytecode-erased compilation* per Definition 1: because loads from the interpreter's program representation (the bytecode pointer) are constant-propagated by the partial evaluation, and the interpreter-switch body is cloned for each separate opcode (PC location) and then branch-folded, all information from the bytecode is propagated into the resulting IR, and the bytecode is no longer referenced or needed. This is an instance of a first Futamura projection.

3.3 Key Idea #3: Directed Value-Specialization

Basic block specialization requires *compile-time constant* context values: otherwise, we cannot resolve branch targets to blocks in the specialized function statically.

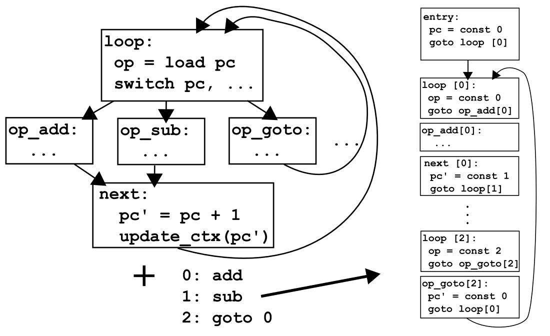
However, an interpreted program will naturally have run-time-data-dependent control flow in the form of conditional branches. An interpreter will implement these branch opcodes either with its own branch, conditionally updating its "next PC" value, or with a branchless conditional-select operator (e.g., `condition ? targetPC : fallthroughPC`). The issue with both of these is that control flow reconverges to a single backedge to the next interpreter loop iteration. At the `update_context` intrinsic call, what will constant-propagation know about `pc`? In fact, it will not have a known constant value.

One possible solution to this dilemma is to write the interpreter control-flow with *two* backedges, one for the taken- and one for not-taken case. This way, the next PC is always constant at any given static program point, and the interpreter's conditional branch becomes the conditional branch in the compiled code. However, this approach falls short: it is vulnerable to tail-merging optimizations when the interpreter itself is compiled[2], and it does not scale to opcodes with a dynamic number of

[2]We considered investigating intrinsics or other optimization directives in the interpreter source to prevent this optimization from breaking the `weval` transform, but in the end, we decided this was a philosophical dead-end: it is better for the transform

Algorithm 1 An abbreviated form of the `weval` transform, as described in §3.2. Some details regarding value specialization (§3.3) and SSA (§3.4) are omitted for space.

```
1: ▷ Input: A function to specialize (e.g., an interpreter), generic, and special-
   ization arguments $specarg_i$ : CPropLattice. ◁
2: ▷ Output: A function specialized such that $specialized(arg_0, arg_1, \ldots) =$
   $generic(arg_0, arg_1, \ldots)$ if $arg_i$ is consistent with $specarg_i$. ◁
3: ▷ Omitted IR manipulation routines: NewBlock, ClearBlock, AppendInst,
   CloneInst, AddArg and context-list truncation helper Truncate. ◁
4: ▷ Blocks are sets of instructions and have .params (SSA block parameters)
   and .terminator properties. Instructions have a .args property. ◁
5: type CPropLattice =
     |Top
     |Constant(value)
     |ConstantMemory(bytes, offset)
     |RunTime
6: type Context =
     |Root
     |Context ∗ Value(value)
     |Context ∗ Spec(value)
7: ▷ Worklist of (Context, block) tuples to specialize. ◁
8: worklist ← []
9: ▷ Map from (Context, block) to specialized basic block labels. ◁
10: blockmap ← {}
11: ▷ Map from (Context, value) to specialized SSA value numbers. ◁
12: valuemap ← {}
13: ▷ Map from specialized SSA value numbers to CPropLattice elements. ◁
14: valuestate ← {}
15: specialized ← empty function body
16: procedure Specialize(generic, specargs)
17:     ▷ Create arguments in specialized function and initialize their partial-
        evaluation state. ◁
18:     for all $arg_i$ ∈ generic.args do
19:         newarg ← AddArg(specialized, typeof($arg_i$))
20:         valuemap[(Root, $arg_i$)] ← newarg
21:         valuestate[newarg] ← $specarg_i$
22:     ▷ Start processing at the function entry block, in a root (empty) context. ◁
23:     push(worklist, (Root, generic.entry))
24:     ▷ Run the worklist until empty, specializing one block at a time. ◁
25:     while worklist is not empty do
26:         (ctx, block) ← pop(worklist)
27:         SpecializeBlock(ctx, block)
28: procedure AbsEval(inst, abstract_values)
29:     ▷ Omitted for space. Standard constant propagation: e.g.,
        add(Constant(1), Constant(2)) ⟶ Constant(3). ◁
30:     ▷ Abstract evaluation of loads at a ConstantMemory address reads the
        value at the pointer value's offset. ◁
31: procedure SpecializeBlock(ctx, block)
32:     if (ctx, block) ∈ blockmap then
33:         ▷ Clear contents of a block that we are re-processing. ◁
34:         b ← blockmap[(ctx, block)]
35:         ClearBlock(b)
36:     else            ▷ Create a new, empty block in the specialized function.
37:         b ← NewBlock(specialized)
38:         blockmap[(ctx, block)] ← b
39:     ▷ For each instruction in the generic version of this block, partially evalu-
        ate and transcribe the results into the specialized block. ◁
40:     for all inst ∈ block do
41:         specargs ← {valuemap[(ctx, value)]|value ∈ inst.args}
42:         argstate ← {valuestate[(ctx, value)] or Top|value ∈
            inst.args}
43:         if inst is push_context then
44:             ctx ← ctx ∗ Value(argstate[0])
45:         else if inst is pop_context then
46:             ▷ Remove last element of context: e.g., Root ∗ Value(..) ⟶ Root ◁
47:             ctx ← Truncate(ctx)
48:         else if inst is update_context then
49:             ctx ← Truncate(ctx) ∗ Value(argstate[0])
50:         else
51:             abstract_result ← AbsEval(inst, argstate)
52:             result ← CloneInst(inst, specargs)
53:             AppendInst(b, result)
54:             valuestate[result] ← abstract_result
55:     b.terminator ← SpecializeTerm(block.terminator, ctx)
56: procedure SpecializeTerm(terminator, context)
57:     Branch-fold terminator according to constant-propagation state in
        valuestate.
58:     terminator.targets ← {SpecializeTarget(context, target)|target ∈
        terminator.targets}
59:     return terminator
60: procedure SpecializeTarget(context, target)
61:     if (context, target.block) ∉ blockmap then
62:         b ← NewBlock
63:         blockmap[(context, target.block)] ← b
64:     else
65:         b ← blockmap[(context, target.block)]
66:     for all (arg, param) ∈ zip(target.args, b.params) do
67:         valuestate[param] ← valuestate[param] ⊔ valuestate[arg]
68:     if any abstract state input to b changed or b is new then
69:         Append(worklist, (context, target.block))
```



1. At entry:
   - pc is constant 0
   - enqueue loop[0] (loop block, context 0)
2. Process loop[0]
   - constant-fold opcode load
   - branch-fold switch to goto and resolve target
   - enqueue op_add[0]
3. Process next[0]
   - constant-fold pc'=pc+1 to constant 1
   - update context to 1
   - resolve target of goto in current context: loop[1] (and enqueue)
4. Process loop[1], op_sub[1], ..., op_goto[2]
5. Resolve target in new context (0) to loop[0]
   - backedge in bytecode becomes backedge in compiled code

Fig. 6. An example of a partial evaluation of a simple interpreter on a three-opcode interpreted program. The optimizer can subsequently merge together the blocks on the right-hand side, creating a compilation output that is *bytecode-erased*, with a CFG that corresponds to bytecode rather than the interpreter.

targets (from switch statements, for example). In essence, we cannot reify all control flow paths as branches in the interpreter if we do not have a static number of paths for one opcode.

Instead, we introduce another intrinsic to allow *splitting context on values* (in the partial evaluation literature, this is known as *"The Trick"* [29]). The idea is that rather than a scalar context (e.g., an interpreter PC), we add a sub-context index, so specialization maps are keyed on ⟨ basic block, context, value ⟩. We add an intrinsic `int32_t specialized_value(int32_t value, int32_t low, int32_t high)` that specifies a range of N possible values, and passes through a run-time value. At the intrinsic callsite, the block specialization generates control flow to N blocks, branching at *runtime* on `value`, then constant-propagating at *compile time* in each specialized path. The net result is that as long as we have a statically-enumerable list of possible values for a "next PC," we can support arbitrary control flow operators in bytecode such as `switch`.

The interpreter uses this intrinsic by passing the input to a conditional branch, or to a switch opcode, through the intrinsic. We show an example of use in Fig. 5.

3.4 Maintaining Static Single Assignment (SSA) Form

There is one optimization that is critical to grant the `weval` transform acceptable performance in practice. An SSA-based IR has the key invariant that a value can be used only in the subtree of the dominance tree below its definition—that is, in a block that is dominated by the block where it is defined. This invariant ensures that the value is always defined before it is used during program execution. Because the result of the `weval` specialization transform is a control-flow graph that resembles the *interpreted program's* control-flow graph rather than that of the interpreter, the def-to-use relationships in the IR of the interpreter body may no longer satisfy this invariant when transcribed over to the specialized function body by Algorithm 1. Thus, we must somehow repair the SSA, or ensure by construction that we do not violate this invariant.

We noted above one naïve solution: we could process a form of SSA that explicitly passes all live values across control-flow edges with block parameters (or ϕ-nodes). This guarantees correctness because it trivially removes any dependence on inter-block dominance relations. However, this leads to very high transform cost and overhead: in our experiments, up to a 5x increase in block parameter count, yielding very slow compilation of the result.

To understand our more efficient solution, let us start from what goes wrong if we process ordinary SSA (with use-def links across blocks) with Algorithm 1. If a value is defined before a context update intrinsic, and used afterward (e.g., a loop-carried dependence across interpreter loop iterations), then when we map uses of this value through specialized SSA value map, we will not find any definition in the *current* context. Furthermore, there is no obvious single "previous" context to fall back to: as described above, context cloning can lead to *arbitrary* CFGs. Fundamentally, any value that is live across an edge *between blocks in different contexts* may need a block parameter (or ϕ-node) to merge incoming edges from copies of the predecessor in different contexts. This corresponds directly to SSA construction over the interpreted bytecode's CFG.

We implement an analysis and transform that runs before Algorithm 1 and inserts these block parameters. In its simplest form, it finds blocks that follow context-update intrinsics and adds block parameters for all values flowing into these blocks. A further optimization finds points at which context-changes flow across non-dominating edges, e.g., loop backedges, rather than all blocks following a context-change. There are often fewer of the former.

Finally, note that the transform in Algorithm 1 omitted some dependency management that is necessary when SSA values are used across blocks: when the constant-propagation state for

to work for *any* code, optimized in *any* way (as far as practical). The calls to intrinsics will never be optimized away when compiling the interpreter, because they are external/imported functions; that is all that is necessary for correctness.

an SSA value is updated, any blocks that use that SSA value must be re-processed as well. In our implementation we track dependencies at a block level. A simple implementation could also re-enqueue the whole subtree of the dominance tree when a block is (re-)processed.

3.5 Interface: Semantics-Preserving Specialization

From the point of view of an interpreter and language runtime, how do we integrate a transform that operates on the interpreter itself, seemingly from outside the system? Furthermore, how do we reason about what the interface to this fragment of specialized code is, and how we can integrate it, i.e. invoke it in place of the original interpreter?

The key abstraction we provide is *semantics-preserving specialization*. The user of the `weval` tool can make *specialization requests* that reference a function (e.g., a generic interpreter) and include some constant arguments to that function. The request causes the partial evaluator tool to generate a new, specialized function. Each function argument is named in a specialization request with one of three modes: *RunTime*, *SpecializedConst(value)*, or *SpecializedMemory(data)*. *RunTime* means the value is not known at compile time, and the latter two specialize on either a constant value or constant data at the given pointer, respectively. The specialization request makes the *promise* that the function parameter or the memory contents will have those values at invocation time, and `weval` preserves the function's semantics as long as this promise holds. In order to retain function-pointer type compatibility, each specialized function continues to take parameters even for specialized arguments. The specialized function body simply ignores these parameters.

There are two general ways this API could be integrated into a system: *within* the execution universe of the program undergoing specialization, or *outside* of it. Both are reasonable for different design points. An interpreter that already has a separable frontend to parse and create bytecode might prefer to invoke `weval` "from the outside," appending new functions to an image of the runtime. On the other hand, when adapting an existing interpreter with no clear phase separation, it might make more sense to request a specialization "from the inside," directly providing data from the heap and receiving a function pointer in return. This could operate at run-time, with a JIT-compilation backend, or it could operate in a *snapshot* workflow: enqueue specialization requests, snapshot the program with its heap, append new functions to the snapshot, and restart. In our Wasm-based prototype, we take this latter approach, building on top of the Wizer [19] snapshot tool. Note, however, that this is not fundamental to the `weval` transform.

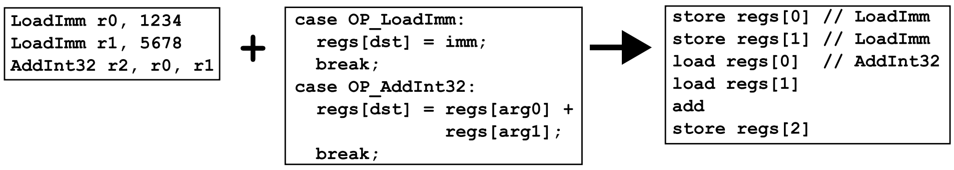
When integrated into a Wasm-snapshot build workflow, the top-level interface to our tool is a function accessible to the Wasm guest that has a signature like the following (slightly simplified):

```
template<typename... Args>
request_t* specialize(func_t* result, func_t generic, Args... args);
```

This enqueues the "request" at a well-known location in the Wasm heap so that the `weval` tool can find it; when the Wasm module snapshot is processed, the function pointer at `result` is updated to point to the appended function.

The integration into an interpreter then requires one to: (i) enqueue specialization requests when function bytecode is created; (ii) store a specialized-code function pointer on function objects; and (iii) check for and invoke this function pointer.[3] We will see objective measures of annotation overhead, including this "plumbing" to weave the specializations into the language runtime's execution, in the following sections.

[3]This is usually a conditional, and the original interpreter may still be present if the interpreted language allows, e.g., `eval()` at run-time, so not every function may be specialized—but this is outside the scope of the `weval` tool itself.



```
LoadImm r0, 1234
LoadImm r1, 5678
AddInt32 r2, r0, r1
```

```
case OP_LoadImm:
  regs[dst] = imm;
  break;
case OP_AddInt32:
  regs[dst] = regs[arg0] +
              regs[arg1];
  break;
```

```
store regs[0] // LoadImm
store regs[1] // LoadImm
load regs[0]  // AddInt32
load regs[1]
add
store regs[2]
```

Fig. 7. Partial evaluation by itself removes dispatch overhead, but preserves load/store semantics of interpreter state data structures, leading to inefficiency.

3.6 Generality Across IRs

We prototyped this transform on WebAssembly for pragmatic reasons (it was the platform that spawned the need for our tool) but we believe the transform is general. In brief, it will work on any IR and platform given these requirements:

- The possible control-flow edges need to be explicit—for example, the IR cannot have a computed-goto feature with "label address" operators. Otherwise, it would not be possible to resolve block targets in specialization contexts ahead of time.
- The IR needs to support arbitrary, e.g., irreducible, control flow: when driven by specialized-on bytecode, i.e. user-controlled data, invariants of the original CFG such as reducibility may be lost. In our prototype on Wasm, where the output format can only represent reducible CFGs, we implement special lowering for irreducibility.
- The platform and tool interface together need to have a way to expose "constant memory" to the transform. In our prototype, the interface allows specifying a function argument as "pointer to these constant bytes" (e.g., bytecode or interpreter configuration data), and this works from inside the Wasm module, referring to bytes in the Wasm heap snapshot. However, one could also imagine an externally-driven interface where the data is provided separately.

4 Handling Interpreter State Efficiently

As it stands so far, our partial evaluator can eliminate an interpreter's *dispatch overhead* by pasting together the parts of the interpreter's main loop that implement each opcode. However, these opcode implementations will still likely contain dynamic indirection to access the interpreted program's state. This is another source of overhead that differentiates interpreted execution from fully optimized compiled execution, and we wish to eliminate it as well.

As a simple example, consider a bytecode for a virtual register-based interpreter, together with opcode implementations, in Fig. 7. If we were to take the Futamura projection of this interpreter over the bytecode, we might obtain a compiled result like that in the figure.

The `regs` array accesses compile to loads and stores to offsets in the interpreter's state. A good alias analysis, combined with redundant load elimination, dead store elimination, and store-to-load forwarding optimizations, *might* be able to disambiguate these loads and stores. However, a realistic interpreter might have other features that interfere with this: for example, calls to other functions. These functions may not access `regs`, but this cannot be proven intraprocedurally. Ideally we would like to inform to the partial evaluator that these values can be stored in true locals (i.e., SSA values in the `weval` transform result) rather than memory.

4.1 Virtualized Registers

`weval` allows the interpreter to communicate which memory reads and writes correspond to true SSA locals via intrinsics. Specifically, it provides the intrinsics `load_register(index)` and `store_register(index, value)` that are semantically equivalent to loads and stores to a hidden array within the specialized function. The `index` parameter must be a constant (perhaps loaded from the constant bytecode) during specialization. (See §5 for an example that uses these intrinsics.) The specialization transform carries a map of indices to SSA values, and translates these intrinsics appropriately, reconstructing SSA by inserting block parameters at merge points where needed.

4.2 In-Memory State: Locals and Operand Stack

Non-escaping locals provide an important primitive, but interpreters sometimes mutate state that is visible to the rest of the runtime as well. For example, a language with GC might need to inspect local-variable values in order to mark them or update them after a compaction.

As above, we wish to lift the original in-memory storage to SSA when possible. However, these values need to be written back to memory at certain points, and their new values reloaded afterward. To support this, we provide two state abstractions that build on virtualized registers, but carry both the value *and* a canonical in-memory address. This state operates like a write-back cache: the transform will perform true loads when necessary, and will generate stores at "flush" intrinsics.

For the indexed-local case, `weval` provides the intrinsics `read(index, address)`, `write(index, address, value)`, and `flush()`. Many interpreters also implement a *stack VM* abstraction with opcodes that push and pop operands and results. `weval` thus provides the intrinsics `push(address, value)`, `pop(address)`, `read_stack(depth, address)`, and `write_stack(depth, address, value)`. These perform an abstract interpretation of stack state in lieu of explicit local indices.

Note that some care must be taken to ensure that `flush()` is invoked wherever the in-memory state might be observed. In our SpiderMonkey adaptation (§6) we built a C++ RAII mechanism to ensure this (exposing the ability to call the rest of the runtime only after interpreter state is flushed). Any interpreter that opts into these intrinsics will need to take care that a flush has occurred before in-memory state may be observed. Other design points might also be possible: for example, a new intrinsic or mode in our tool could be added that flushes at every callsite, or that tracks escaped pointers to the state in some other way.

4.3 Discussion: Semantics-Preservation and Polyfills

The locals and stack intrinsics differ from the initial function-specialization transform in §3 in two ways: (i) they grant the `weval` tool permission to diverge in semantics in controlled ways (*lazy flushing* of in-memory state with user-denoted synchronization points), and (ii) they are not simply intrinsics that can be removed ("hints") but must be replaced/polyfilled for ordinary execution of the original function body to work. This permission to diverge is fundamental for performance: the memory operations become a severe bottleneck otherwise.

The intrinsic signatures are carefully designed so that polyfills are possible: the in-memory state intrinsics take address arguments and can thus fall back to true loads and stores. The register intrinsics (§4.1) could be rewritten to loads and stores to an array. For pragmatic reasons we have not implemented these polyfills, and instead we generate two separate versions of the interpreter body function with and without state intrinsics, but this is not fundamental.

5 Case Study: Minimal Toy Interpreter

To give a feel for `weval`, we integrate it first into a minimal example interpreter—a small 64-bit register machine named *Min*. Min has 10 instructions that operate on a program counter *pc*, an array of indexed registers *regs*, and an accumulator register *accum*. Except for the `JMPNZ` instruction, the machine reads the instruction, increments the *pc*, executes the instruction, and returns to the top of the interpreter loop. The interpreter loop is shown in Figure 8.

The first step to `weval`ing an interpreter is adding a *context* annotation. To specialize a bytecode interpreter, we use the program counter—the `pc`—as the context. As the annotation only has

```
uint64_t Execute(uint64_t *program) {
  uint64_t accum = 0, pc = 0;
  uint64_t regs[256] = {0};
  PUSH_CONTEXT(pc);  // NEW
  while (true) {
    switch (program[pc++]) {
    case LOAD_IMMEDIATE:
      accum = program[pc++];
      break;
    case STORE_REG: {
      uint64_t idx = program[pc++];
      regs[idx] = accum;
      break; }
    case LOAD_REG: {
      uint64_t idx = program[pc++];
      accum = regs[idx];
      break; }
    case PRINT: {
      const char* msg =
        (const char*)program[pc++];
      printf("%s", msg);
      break; }
    case PRINT1:
      printf("%" PRIu64, accum);
      break;
    case HALT: return accum;
    case JMPNZ: {
      uint64_t addr = program[pc++];
      if (SPECIALIZE_VALUE(
            accum != 0, 0, 2)) // NEW
        pc = addr;
      break; }
    case INC: accum++; break;
    case DEC: accum--; break;
    case ADD: {
      uint64_t idx1 = program[pc++];
      uint64_t idx2 = program[pc++];
      accum = regs[idx1] + regs[idx2];
      break; }
    default: abort();
    } // end switch
    UPDATE_CONTEXT(pc); // NEW
  } // end while
  POP_CONTEXT(); // NEW
}
```

Fig. 8. *Min* bytecode interpreter in C. Lines marked NEW are the added weval annotations.

```
#define REG_AT(idx) (IsSpecialized ? \
      load_register(idx) :           \
      regs[idx])

#define REG_AT_PUT(idx, val)         \
  if (IsSpecialized) {               \
    store_register(idx, val);        \
  } else {                           \
    regs[idx] = val;                 \
  }

template <bool IsSpecialized>
uint64_t Execute(uint64_t *program) {
  // ...
      REG_AT_PUT(idx, accum);
  // ...
      accum = REG_AT(idx);
  // ...
}
```

Fig. 9. We modify the macros to read and write registers to conditionally use weval's register intrinsics. For non-fundamental reasons, we currently don't polyfill the intrinsics in our tool in non-specialized versions of the function, so we need to generate two versions of the interpreter function: one using the intrinsics, and one with a conventional register array. In order to create both alternatives, we use C++ template specialization to ensure this choice is made when the interpreter is compiled. In a pure C-based interpreter, one could put the interpret function in a separate file, redefine the macros twice (once for intrinsics and once without), and include the function body in both cases.

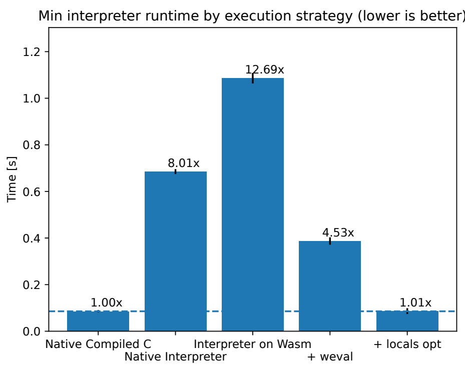


Fig. 10. Benchmark of the loop program with different execution strategies.

meaning when the program is being partially evaluated, we invoke the annotations with macros that are conditionally defined only in a build for `weval`.

These annotations alone will improve performance: they unroll the interpreter loop into guest language control flow and allow for `weval`'s optimizer to "see through" the interpreter into the guest language. We can, however, do better: we can also use `weval`'s state optimizations (§4).

To allow `weval` to optimize *registers* and remove loads/stores, in Fig. 9 we replace direct array accesses like `regs[idx]` with macros `REG_AT(idx)` and `REG_AT_PUT(idx, val)` and define them to use `load_register(idx)` and `store_register(idx, val)` in a variant of `Execute` passed to the partial evaluator. Now we can unroll the register bytecode into direct SSA dataflow. This avoids touching memory *and* gives more information to the optimizer.

As a benchmark, we write one program in Min that computes the sum of all integers from 0 to 100 million and prints it to *stdout*. In Fig. 10 we show performance of a Min program running on the C++ interpreter running on the host platform directly (as an x86_64 program), on the interpreter compiled to Wasm, and on that interpreter processed by `weval` and with interpreter-state optimizations applied (all Wasm variants running on Wasmtime), all compared to the equivalent program written in C.[4] We show that `weval`ing the interpreter beats native interpreter performance and unrolling local variables yields still more speedup, coming within 1% of the performance of the equivalent program written in C and compiled to native code.

6 Case Study: SpiderMonkey JavaScript Interpreter

In this section, we present our most significant real-world use-case: an application of our partial evaluator to the SpiderMonkey [4] JavaScript engine's interpreter, in order to derive compiled code directly from the interpreter semantics. This application of our tool has been merged into the StarlingMonkey JS runtime [8] which embeds SpiderMonkey to target Wasm-first platforms, where run-time code generation (JIT) is not supported. `weval`-based snapshot processing is used to provide its "ahead-of-time compilation" (AOT) feature.

The SpiderMonkey JavaScript engine, running on a native platform (e.g. x86 or ARM), has several interpreter tiers and several JIT-compilation tiers. It does not have support for ahead-of-time compilation. It has been ported to run within a WebAssembly module [15]; in this mode, it runs only with its interpreter tiers, because Wasm does not support run-time code-generation. SpiderMonkey supports inline caches (ICs) using a separate bytecode, CacheIR [17], to represent their logic, and the engine can execute these ICs in an interpreted mode using the *Portable Baseline Interpreter* (PBL) [18]. We take this as our baseline.

6.1 Ahead-of-Time Compilation

The StarlingMonkey runtime establishes a build workflow on top of SpiderMonkey that is *superficially* ahead-of-time: it loads the source into the engine, translating it to bytecode, then freezes Wasm execution using the Wizer [19] snapshotting tool. This snapshot, saved as a Wasm module, starts immediately interpreting bytecode when it is later executed.

In order to build *ahead-of-time compilation* on top of this, in a way that provides a *bytecode-erased compilation* per our definition, we need (i) to ensure all bytecode that will be interpreted at run-time is present in the snapshot, and (ii) process the snapshot with `weval`, creating and appending compiled function bodies to the Wasm module for each bytecode function.

[4]We find that a sufficiently advanced optimizer, such as the one present in Clang/LLVM, can completely unroll the loop into the closed-form $n(n+1)/2$. Adding such an optimizer (for example, Binaryen [46]) is future work and not relevant to the main claims of this paper. To keep the loop and local variables around for the benchmark, we annotate with `volatile`.

It would seem that the *specialization* aspect of dynamic-language compilation is at odds with AOT compilation: the PBL interpreter executes inline-cache bodies that are *generated at run-time* from the particular cases that the interpreter observes. Thus, the bytecode for these ICs is not available to compile via `weval` when processing the snapshot.

However, we observe that *most possible IC bodies* are known ahead-of-time. SpiderMonkey's set of possible ICs is technically infinite (up to bytecode length limits), because some ICs contain guard sequences that depend on user-controlled data structures (e.g. one guard per object in a prototype chain). But most are small and well-known, because the IC generator includes short handwritten sequences that directly correspond to the slow-path logic in C++. Note also that SpiderMonkey *separates data parameters from code* in ICs already, in order to allow code-sharing: that is, constants, such as object shape pointers in shape guards, are not baked into the IC but are loaded from a "stub data" region that is separate for each usage of the IC body.

We collect a corpus of known IC bodies by adding a mode to the interpreter to dump unknown ICs and running this over the unit-test suite—with the idea that any important ICs will be exercised in unit tests. We build this corpus into the engine binary, and update the IC attachment logic to look up generated IC cases in a hashtable of pre-loaded ICs first. This corpus contains the bytecode and allows `weval` to "pre-specialize" compiled IC bodies for all ICs in the corpus. We found that a corpus of 2320 IC bodies (most of them a dozen or fewer opcodes) covered all language features tested in the testsuite and gave 100% coverage of our benchmarks below.

Stated succinctly, the key insight is: ahead-of-time compilation of JavaScript is possible at this level because inline caches (ICs) allow dynamism in semantics to be pushed to late-binding run-time data changes (function pointer updates) rather than code changes.

6.2 Changes to the Interpreter

Next, we had to ensure that `weval` finds all bytecode JS function bodies and IC bodies in the snapshot and appends compiled Wasm functions to the module to replace these; we had to ensure the interpreters were compatible with `weval`; and we had to ensure that the engine would invoke these "specialized" functions rather than the interpreter.

In order to permit PBL's two interpreter loops—for JS and IC bytecode—to be partially evaluated, several minor changes were necessary. First, we had to ensure that one native function call frame (in the interpreter's implementation language, C++) corresponded to one JS function or IC stub call; this is what allows per-function specialization to work. The interpreter was originally written to perform JS calls and returns "inline," by pushing and popping JS stack frames as data without making C++-level calls. We modified the interpreter to recurse instead.

Second, as in the previous section, we added annotations to update context, and to optimize the storage of interpreter state. We used `weval`'s "registers" for CacheIR, which is a register-based IR; and "locals" and the virtualized operand stack for JS bytecode.

To handle several forms of non-local control flow, our modified interpreter loop tail-calls ("restarts") to a non-specialized version of itself—just for the active function frame—in several control flow situations that are nontrivial or inefficient to handle: `async` function resumes, which would imply multiple function entry points[5], and error cases (including exception throws), to minimize compiled code size. The engine supports all edge cases and retains 100% compatibility (continues to pass all tests), and only a negligible fraction of execution time is spent in interpreted (non-specialized) code in our benchmarks.

[5]It should be possible to either include a `switch` at the beginning of async functions to handle this, or more ambitiously, define new intrinsics that allow compiling coroutine-like code to WebAssembly's stack-switching proposal; we have not yet implemented either approach.

Table 1. Evaluation configurations for SpiderMonkey as a Wasm module.

Configuration	Description	ICs	Dispatch	Interpreter State
Generic Interp	Default (generic) interpreter	No	Dynamic	Dynamic (memory)
Interp + ICs	PBL interpreter	Yes	Dynamic	Dynamic (memory)
`weval`*ed*	AOT-compiled via `weval`ed PBL	Yes	Static	Dynamic (memory)
`weval`*ed + state*	Same, with state optimizations	Yes	Static	Static (Wasm locals)

Table 2. Evaluation configurations for native-JIT SpiderMonkey, as a comparison point to above. Note that the *Optimized* configuration does not yet have an equivalent in Table 1; see §9 for Future Work on this point.

Configuration	Description	ICs	Dispatch	Interpreter State
Generic Interp	Default (generic) interpreter	No	Dynamic	Dynamic (memory)
Interp + ICs	PBL interpreter	Yes	Dynamic	Dynamic (memory)
Compiled + ICs	Baseline compiler	Yes	Static	Dynamic (memory)
Optimized	IonMonkey optimizing backend	Inlined	Static	Static (regalloc)

Our patch to add these annotations and intrinsics amounted to `+1045 -2 lines`, including a vendored `weval.h`. The changes to the interpreter function itself amount to 133 lines of alternate macro definitions to swap in the intrinsics.

6.3 Performance Results

In Fig. 11, we show the performance of our modified SpiderMonkey engine on the Octane benchmark suite [37], reporting throughput (inverse run time, i.e., speed) data for the configurations listed in Table 1, all as Wasm modules running on Wasmtime [7].

`weval`'s speedup is seen as the delta from *Interp + ICs* to `weval`*ed + state*. This ratio is a 2.17× speedup; up to 2.93× on the best benchmark (Richards) and above 2× in all cases except RegExp (which depends heavily on the regular expression engine's interpreter loop which we have not modified) and CodeLoad (which tests the engine's code loading rather than execution speed).

Our use of interpreter state optimizations was motivated by the observation that loads and stores to locals and the operand stack are quite hot. By making use of these intrinsics, from the `weval`*ed* configuration to `weval`*ed + state*, we see a 1.37× speedup. Without this optimization, a number of benchmarks see almost no speedup at all (Crypto, Mandreel). Across all of Octane, the virtualized stack intrinsics elide 84% of 639K loads and 76% of 563K stores; the local intrinsics elide 14% of 149K loads and 5% of 74K stores. (Pushes and pops happen at every opcode, while JS locals are accessed less frequently and so GC safepoints are more likely to flush them to memory first.)

6.4 Comparison to Native Execution

In order to judge the relative speedups attained by `weval`, and also eventual upper bounds, we compare Wasm-based execution to analogous configurations in a native-code setting with JIT backends. Note that we do not intend to compare `weval`ed code inside a Wasm module *directly* to the native code—it encounters some overhead due to the Wasm sandbox—but rather, the progressive ratios of each step on the two platforms.

Fig. 12 shows three of the four configurations from §6.3 on Wasm (skipping plain `weval`*ed* without state optimizations) alongside four configurations running natively on the same system. The native configurations are designed to be analogous to the configurations running within a Wasm module (except for the last, *Optimized*), and are described in Table 2.

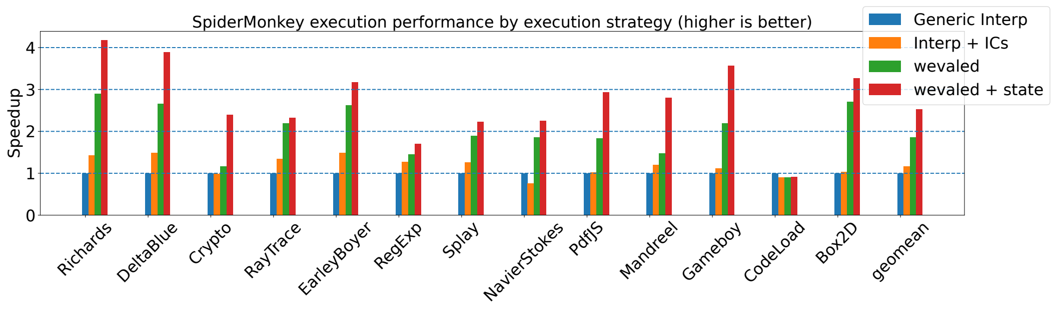


Fig. 11. Performance results of Octane benchmark suite on SpiderMonkey engine, with interpreter in a Wasm module (without and with ICs), and weval-compiled code (without and with interpreter-state optimizations).

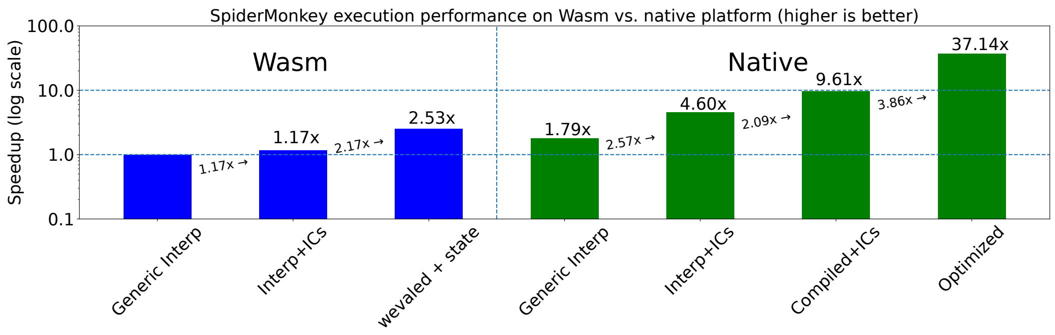


Fig. 12. SpiderMonkey configurations running on top of a Wasm engine vs. SpiderMonkey as a native build on the same system. This shows how (i) inline-cache fastpaths, (ii) compilation of JS bytecode and inline caches separately, and (iii) optimized compilation of both together (native only) result in progressive speedups.

We label speedup ratios between each successive pair of configurations. A few interesting comparisons can be made. First, by observing the second to third bar on each side, this plot shows that weval attains a similar speedup over the next lower tier (interpreter with ICs) as the native baseline compiler does. In both cases, we are removing the interpreter overhead but retaining run-time binding of behavior via IC stubs.

Second, we see that fully optimized native JIT execution is a significant speedup (3.86×) over baseline compilation. Note that the *Optimized* configuration "pulls out all the stops" and, in particular, takes advantage of being a *JIT compiler*: it type-specializes code. As we argue in §9, we believe there is a path for our AOT-based approach to adopt profile-guided inlining in a safe, principled way, possibly closing this gap. Nevertheless, the gap remains today.

Third, however, the overall speedup of the "baseline compiler-like" configurations—first bar to third bar—is still somewhat behind in weval: 2.53× over the generic interpreter, vs. 5.37× on native. In principle there is no difference between the optimizations that both configurations are capable of, and in fact profiling and examination of generated code largely bears this out: both are "baseline compilation" producing a skeletal compilation of JS bytecode that invokes ICs and plumbs values between them, and straightline compilations of IC opcodes. We believe the remaining inefficiencies lie mostly in the IC invocation efficiencies: the native baseline compiler can tightly control ABI and register allocation, keeping hot values in pinned registers and effectively doing interprocedural register allocation between the JS function body and ICs. In contrast, on a Wasm platform, control-flow integrity (CFI) checks make indirect calls much slower, and the Wasm engine's compiler does not otherwise perform any special register allocation or other optimizations.

6.5 Code Size

The Wasm module containing the entire SpiderMonkey JavaScript engine contains 8 MiB of Wasm bytecode initially, in 18080 functions. After AOT compilation with `weval` of the entire Octane benchmark suite (7.5 MiB or 337 KLoC of JS) together with the pre-collected corpus of 2320 ICs, there is 52 MiB of Wasm bytecode, with 5212 new functions from JS function bodies and 2320 new IC-stub functions. This code size is primarily due to two factors: AOT compilation forces us to compile *all* functions, rather than only the hot ones, as a JIT would; and many opcodes still compile to IC invocation sites, rather than single Wasm opcodes as would be the case with a lower-level source language. With more optimization work in our tooling, including the Wasm compiler backend that we use, we believe the size of generated bytecode could be decreased substantially.

6.6 Transform Speed

Compiling the above Wasm module takes, in total, 350 seconds of CPU time (44.16 wall-clock seconds parallelized over specialization requests on a 12-core machine). This indicates a compilation speed of slightly under 1 KLoC/second of JavaScript source. We believe this could be improved with further work: our Wasm compilation backend has not been heavily optimized. In order to improve compilation times in practice, we have added a cache that keys on input Wasm module hash plus the function specialization request's argument data; in practice, this works well to avoid redundant work for the unchanging AOT IC corpus, and helps with incremental compilation during development as well.

7 Case Study: PUC-Rio Lua Interpreter

To demonstrate generality of the tool across multiple real-world bytecode interpreters, we ported the original (PUC-Rio) Lua interpreter to Wasm and applied `weval`-based partial evaluation. We split the process into the following chunks:

Support Wasm. We performed a minimalistic port of the interpreter to compile for and run on a Wasm VM. For simplicity, we stubbed out (i.e. removed the source and added calls to `abort()`) some Linux-specific OS library functions; we also stubbed out exception handling because it uses the `setjmp` and `longjmp` C functions[6].

Support Wizer-based snapshotting. Next, we added support for snapshotting of the interpreter after loading a script and translating it to bytecode. This involves adding approximately 30 lines of C code near the C `main` function to expose two functions: `wizer_init` and `wizer_resume`. The initialization function runs the top-level Lua module and finds and saves that module's `main` function (a convention we arbitrarily chose). The resume function then calls this `main`.

Specialize functions. Supporting function specialization requires adding two pointer-sized fields to Lua's function object (`Proto`) struct: a specialized function pointer *spec* and a `weval` request pointer *req*. We create and fill in *req* when the function object is created in the parser. It must exist somewhere in the heap so that `weval` can find it, and we retain it so that it can be freed later on function destruction. (It is possible to instead use a side-table, but we chose to keep the

[6]Unfortunately, many WebAssembly runtimes do not *yet* support `setjmp` and `longjmp` but support is expected to land soon with the exception-handling extension [1]. For now, projects such as Emscripten handle exceptions by calling into the host JavaScript runtime and leaning on JavaScript exceptions.

implementation simple.) When we make the `weval` request, we also pass it the address of *spec* field for `weval` to fill in later, after the snapshot is taken and the transform has run.

We tested this step before we added annotations to the interpreter: at this point, `weval` specialization should produce the same interpreter function as output, because no context-specialization occurs. We also modified the interpreter function (`luaV_execute`) signature to pass in a bytecode parameter for specialization.

Annotate interpreter. PUC-Rio Lua uses macros instead of manual code duplication to implement much of its interpreter control-flow. This makes modifying the interpreter straightforward: we add a `push_context` to the top of the interpreter and an `update_context` to back edges.

Change call path. In order to reap the benefits of our specialized function pointers, we must call them. Lua has only two ways to call a managed function (outside the interpreter and inside the interpreter) and we modified both to call *spec* if it has been filled in. We also ensured that the interpreter calls itself for each Lua call, rather than handling the call opcode "inline."

With these changes, we had a working ahead-of-time compiler for Lua. Some trivial interpreter-heavy benchmarks produce the expected results, showing a 1.84× speedup. The resulting source tree has a diff in Lua C/header files (excluding `weval`'s and Wizer's headers, and build-system tweaks) of `+173 -57` lines. This includes the initial port to Wasm. Future work includes calling intrinsics to lift local variables or stack variables to Wasm locals.

8 Related Work

`weval` exists against a background of many prior partial-evaluation systems, and systems that derive compilers from interpreters in related ways (e.g., tracing). While this design space is broad, `weval`'s primary novelty lies in its combination of ahead-of-time focus and a design that allows application to existing, almost-unmodified, interpreters in systems languages such as C and C++ without rewriting to use an explicit framework. As we demonstrated in prior sections, this enables `weval` to be applied easily to existing, mature industrial language implementations.

Partial Evaluation: There is a rich pre-existing literature on partial evaluation, going back at least to Futamura [21, 22]. Jones [29] provides a comprehensive overview of the field. Several aspects of the `weval` transform, such as constant propagation and branch folding through interpreter dispatch, and the use of value specialization ("The Trick"), are standard techniques for partial evaluators. We compare explicitly to several prominent systems in this vein below.

GraalVM and Truffle [47] are a JIT compiler backend and language runtime framework (respectively) that partially evaluate on Java bytecode. The Truffle ecosystem supports Ruby [41, 47], JavaScript [47], and other languages.

Truffle users rely on a Truffle-provided framework to author AST-walking interpreters. Partial evaluation unrolls the AST interpreter's logic by constant-propagating and devirtualizing recursive calls in each AST node's execution method. Because of the interprocedural, recursive structure inherent in such an interpreter, Truffle requires careful attention to inlining and user annotations to mark an inlining boundary. In general, the interpreter must be developed specifically for Truffle. The system also supports run-time optimization and de-optimization, which is very useful for dynamic languages. However, these features also add significant complexity, and require long warmup times. Truffle has been adapted to bytecode-based inputs, e.g., TruffleWasm [40], but (at least in that work) via a translation to AST nodes. Truffle also has some support for AOT compilation [5]. However, many of its features work only in JIT mode, and its design is heavily optimized for JIT. In contrast, `weval` (i) is optimized for the ahead-of-time use-case, and (ii) is able to apply to existing mature industrial interpreters in C/C++, which were not written from scratch for the framework, with only a few lines of annotation.

PyPy [11] implements Python on top of the RPython meta-tracing JIT compiler. Meta-tracing differs from our approach because it fundamentally requires run-time information to profile and select hot traces. In some sense, `weval`'s approach is breadth-first, compiling whole function bodies ahead of time, while PyPy's is depth-first, compiling hot traces across function boundaries. PyPy also requires the interpreter to be written in RPython, a restricted subset of Python.
DyC [24, 25] is a run-time optimization system for C that performs partial evaluation. It provides annotations to perform control flow and value specialization and performs binding-time analysis. It can unroll interpreter loops, as we do. However, DyC appears to be significantly more complex than `weval`, relying on heuristics and analysis to handle interprocedural specialization, overlapping specialization regions, caching of specializations, and more. It also requires dynamic information, as it is a fundamentally run-time optimization. In contrast, our tool (`weval`) provides more predictable run-time performance, is applicable ahead-of-time, and does not require a runtime library.
BuildIt [13] is a C++ library for partial evaluation of C++ programs. It provides annotations for partitioning variables into compile-time constants and run-time data, and generates C++ code; it can also unroll interpreter loops. However, its use requires care to avoid miscompilation: misclassification of variables can lead to semantic differences. Also, practically, the annotation burden is much heavier: all variable bindings must be classified manually. Finally, the framework is language-specific. In contrast, `weval` can process any language that compiles to a CFG of SSA and can call C-like intrinsics.
Lightweight Modular Staging (LMS) [38, 39] is a library developed by Rompf and Odersky for partial evaluation of Scala programs. It has been used to great effect to, among other things, compile SQL queries to efficient code [44]. Using this library requires using Scala as the host language; like other related work and unlike `weval`, this language-level approach is difficult to apply to existing mature language implementations.
Deegen [48] aims to generate a fast interpreter and baseline JIT from language semantics. It provides a C++ DSL for describing language semantics, and has APIs for defining opcodes, type-specialized variants of opcodes, inline caches, a type lattice, "slow paths," and more. Similarly to PyPy, by design it cannot be used for AOT compilation. Its use requires writing the language implementation in terms of Deegen abstractions.
Basic Block Versioning (BBV) [14] implements block cloning in a way that is reminiscent of `weval`, but for a different purpose: its cloning context is a *type context* and is used to specialize code in a JIT for a language such as JavaScript. However, the core idea of *mapping context and block to specialized block* and *memoizing* this mapping remains. One could see BBV and `weval` as complementary: `weval` implements partial evaluation with block cloning to *translate the interpreter body into compiled user code*, while BBV can *optimize compiled user code* by further cloning blocks on possible type contexts.
SemPy [32] and Static Basic Block Versioning [33] (SBBV) also aim to derive a compiler from interpretable semantics using context-sensitive dataflow analyses and partial evaluation. SemPy records language semantics in canonical, interpreter-like form, but this form is developed explicitly for the purpose, i.e., is not a pre-existing interpreter. SBBV is a complementary technique that finds a set of type-specialized contexts ahead-of-time for which to generate code. The same downside applies here as other related work above: these frameworks require new expression of language semantics, and so unlike `weval`, are not applicable to existing engines.
LuaAOT [35] is a purpose-built AOT compiler framework that, superficially, works similarly to `weval`'s application to the Lua interpreter: it compiles bytecode by pasting together portions of the interpreter loop. However, its algorithm operates at the source (interpreter in C) level. The work claims 20–60% speedups (1.25×–2.5×) from a 500-LoC special-purpose implementation; in contrast,

our Lua modification achieves 1.84× speedup (on top of WebAssembly, though in principle `weval` is not limited to Wasm, as we noted in §3) with a `+173 -57`-line diff.

AOT JS Compilers: In this work we adapted SpiderMonkey to compile JavaScript ahead-of-time with the aid of `weval`. Several other works also address this problem. Hopc [42, 43], Porffor [31], Static Hermes [34], and Static TypeScript [9] are all compilers that accept JavaScript, TypeScript, or some subset thereof, and produce either native code or WebAssembly. The key distinction from our work is that these compilers are explicit: they are written as code transforms, not executable interpreters, and hence are harder to validate, debug or extend than an interpreter-based implementation, and they do not inherit the full language compatibility of an existing industrial implementation.

9 Future Work

9.1 Profile-Guided Inlining and Semantics-Preserving Optimizations

The level of optimization that the `weval` transform is able to provide in its current form (as a processing step on a program snapshot) is limited by its ahead-of-time-only design goal: it cannot optimize based on types in dynamic languages. To carry the goal of automatically deriving a compiler further, one ought to be able to derive type-specialization optimizations beyond ICs.

We believe that *profile-guided inlining* is a principled way to do this. One would start with an AOT compilation, observe and gather statistics on IC callsite targets, and eventually recompile indirect-call instructions into *guarded inlined functions* (if function pointer is X, run inlined body, else call-indirect). In this way, just as for the basic `weval` transform, *semantics are fully preserved.* The Winliner tool [20] prototypes this optimization strategy. This parallels how SpiderMonkey's WarpMonkey [16] backend generates its optimizing compiler input from inlined ICs.

Such inlining removes the IC indirect-call overhead, and it creates further opportunity: it places IC implementations, including *boxing* and *unboxing* operations and dynamic value type-checks, together in the same function body. Our tool could then incorporate special optimizations that—still preserving semantics—hoist type-checks upward, and eliminate boxing-unboxing pairs. All of these operations can be written as generic compiler rewrite rules—for example, SpiderMonkey's boxing is a form of NaN-boxing and so a partial-known-bits optimizer that understands known tag bits and conditional checks on them should achieve this [12, 45]. Overall, this is a further step toward a goal of *safe dynamic-language compilers* with correct-by-construction optimizations.

9.2 Specializing at Run-time

While `weval` today acts as an AOT compiler, we are also interested in JIT applications. Nothing prevents the `weval` transform from operating at run-time. Operating at run-time, however, places additional stresses on the performance of the specialization algorithm. Alternative designs that require more annotation overhead may allow for generation of a more efficient compiler (i.e., the *second* Futamura transform). Though the feasibility of this remains unclear, the efficiency of the `weval` transform is an important future goal for this and other reasons.

Acknowledgments

We wish to thank our fellow compiler engineers at Fastly and within the Bytecode Alliance, who were valuable sources of feedback throughout `weval`'s development. Luke Wagner, Andrew Brown, Dan Gohman, Till Schneidereit, Nick Fitzgerald, Jamey Sharp, Trevor Elliott, Jake Champion, Guy Bedford, and others provided an intellectually stimulating and vibrant environment in which to develop these ideas while solving real-world problems.

We wish to thank the Recurse Center for providing a place to learn, grow, and explore new ideas, and the many recursers for discussion, pair programming, walks, and late-night pizza.

References

[1] [n. d.]. WebAssembly exception-handling proposal. https://github.com/WebAssembly/exception-handling/

[2] [n. d.]. `weval` GitHub Repository. https://github.com/bytecodealliance/weval/

[3] 2024. CVE-2024-4761. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2024-4761

[4] 2024. SpiderMonkey JavaScript Engine. https://spidermonkey.dev/.

[5] 2025. Truffle AOT Tutorial. https://www.graalvm.org/latest/graalvm-as-a-platform/language-implementation-framework/AOT/

[6] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. 2006. *Compilers: principles, techniques, and tools, 2nd ed.* Addison Wesley.

[7] Bytecode Alliance. [n. d.]. Wasmtime WebAssembly virtual machine. https://wasmtime.dev

[8] Bytecode Alliance. 2024. StarlingMonkey JavaScript Runtime. https://github.com/bytecodealliance/starlingmonkey

[9] Thomas Ball, Peli de Halleux, and Michał Moskal. 2019. Static TypeScript: an implementation of a static compiler for the TypeScript language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Athens, Greece) *(MPLR 2019)*. Association for Computing Machinery, New York, NY, USA, 105–116. https://doi.org/10.1145/3357390.3361032

[10] Zach Bloom. 2018. Cloud Computing without Containers. https://blog.cloudflare.com/cloud-computing-without-containers/

[11] C F Bolz, A Cuni, M Fijalkowski, and A Rigo. 2009. Tracing the meta-level: PyPy's tracing JIT compiler. *ICOOOLPS* (2009). https://doi.org/10.1145/1565824.1565827

[12] CF Bolz-Tereick. 2024. A Knownbits Abstract Domain for the Toy Optimizer, Correctly. https://pypy.org/posts/2024/08/toy-knownbits.html

[13] Ajay Brahmakshatriya and Saman Amarasinghe. 2021. BuildIt: A Type-Based Multi-stage Programming Framework for Code Generation in C++. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 39–51. https://doi.org/10.1109/CGO51591.2021.9370333

[14] M. Chevalier-Boisvert and M. Feeley. 2015. Simple and Effective Type Check Removal through Lazy Basic Block Versioning. *ECOOP* (2015). https://doi.org/10.4230/LIPIcs.ECOOP.2015.101

[15] L Clark. 2021. Making JavaScript Run Fast on WebAssembly. https://bytecodealliance.org/articles/making-javascript-run-fast-on-webassembly.

[16] J de Mooij. 2020. Warp: Improved JS performance in Firefox 83. https://hacks.mozilla.org/2020/11/warp-improved-js-performance-in-firefox-83/

[17] Jan de Mooij, Matthew Gaudet, Iain Ireland, Nathan Henderson, and J. Nelson Amaral. 2023. CacheIR: The Benefits of a Structured Representation for Inline Caches. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Cascais, Portugal) *(MPLR 2023)*. Association for Computing Machinery, New York, NY, USA, 34–46. https://doi.org/10.1145/3617651.3622979

[18] C Fallin. 2023. Fast(er) JavaScript on WebAssembly: Portable Baseline Interpreter and Future Plans. https://cfallin.org/blog/2023/10/11/spidermonkey-pbl/

[19] N Fitzgerald. 2020. Wizer: The WebAssembly Pre-initializer. https://github.com/bytecodealliance/wizer.

[20] N Fitzgerald. 2023. The Winliner WebAssembly indirect call inliner. https://github.com/fitzgen/winliner

[21] Y Futamura. 1971. Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler. *Systems.Computers.Controls* 2, 5 (1971).

[22] Y Futamura. 1999. Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* 12 (1999). https://doi.org/10.1023/A:1010095604496

[23] A Gal, B Eich, M Shaver, D Anderson, D Mandelin, M R Haghighat, B Kaplan, G Hoare, B Zbarsky, J Orendorff, J Ruderman, E W Smith, R Reitmaier, M Bebenita, M Chang, and M Franz. 2009. Trace-based just-in-time type specialization for dynamic languages. *PLDI* (2009). https://doi.org/10.1145/1542476.1542528

[24] Brian Grant, Markus Mock, Matthai Philipose, Craig Chambers, and Susan J. Eggers. 2000. DyC: an expressive annotation-directed dynamic compiler for C. *Theor. Comput. Sci.* 248, 1–2 (Oct. 2000), 147–199. https://doi.org/10.1016/S0304-3975(00)00051-7

[25] Brian Grant, Matthai Philipose, Markus Mock, Craig Chambers, and Susan J. Eggers. 1999. An evaluation of staged run-time optimizations in DyC. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '99)*. Association for Computing Machinery, New York, NY, USA, 293–304. https://doi.org/10.1145/301618.301683

[26] Samuel Groß. 2021. V8 Sandbox – High-Level Design Doc. https://docs.google.com/document/d/1FM4fQmIhEqPG8uGp5o9A-mnPB5BOeScZYpkHjo0KKA8

[27] A Haas, A Rossberg, D L Schuff, B L Titzer, M Holman, D Gohman, L Wagner, A Zakai, and JF Bastien. 2017. Bringing the Web Up To Speed with WebAssembly. *PLDI* (2017). https://doi.org/10.1145/3062341.3062363

[28] Neil D. Jones. 1990. Partial evaluation, self-application and types. In *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming* (Warwick University, England). Springer-Verlag, Berlin, Heidelberg, 639–659.

[29] N D Jones. 1996. An Introduction to Partial Evaluation. *Comput. Surveys* 28, 3 (1996). https://doi.org/10.1145/243439.243447

[30] C. Lattner and V. Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. *CGO* (2004). https://doi.org/10.1109/CGO.2004.1281665

[31] Oliver Medhurst. [n. d.]. Porffor: A from-scratch experimental AOT JS engine, written in JS. https://github.com/CanadaHonk/porffor

[32] Olivier Melançon, Marc Feeley, and Manuel Serrano. 2023. An Executable Semantics for Faster Development of Optimizing Python Compilers. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering* (Cascais, Portugal) *(SLE 2023)*. Association for Computing Machinery, New York, NY, USA, 15–28. https://doi.org/10.1145/3623476.3623529

[33] Olivier Melançon, Marc Feeley, and Manuel Serrano. 2024. Static Basic Block Versioning. In *38th European Conference on Object-Oriented Programming (ECOOP 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 313)*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 28:1–28:27. https://doi.org/10.4230/LIPIcs.ECOOP.2024.28

[34] Tzvetan Mikov. 2023. Static Hermes: How to Speed Up a Micro-benchmark by 300x Without Cheating. https://tmikov.blogspot.com/2023/09/how-to-speed-up-micro-benchmark-300x.html

[35] Hugo Musso Gualandi and Roberto Ierusalimschy. 2021. A Surprisingly Simple Lua Compiler. In *Proceedings of the 25th Brazilian Symposium on Programming Languages* (Joinville, Brazil) *(SBLP '21)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3475061.3475077

[36] Johnathan Norman. 2021. Microsoft Edge: Super Duper Secure Mode. https://microsoftedge.github.io/edgevr/posts/Super-Duper-Secure-Mode/

[37] V8 Project. [n. d.]. Octane Benchmark Suite. http://chromium.github.io/octane/

[38] Tiark Rompf. 2016. *The Essence of Multi-stage Evaluation in LMS.* Springer International Publishing, Cham, 318–335. https://doi.org/10.1007/978-3-319-30936-1_17

[39] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *SIGPLAN Not.* 46, 2 (Oct. 2010), 127–136. https://doi.org/10.1145/1942788.1868314

[40] Salim S. Salim, Andy Nisbet, and Mikel Luján. 2020. TruffleWasm: a WebAssembly interpreter on GraalVM. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Lausanne, Switzerland) *(VEE '20)*. Association for Computing Machinery, New York, NY, USA, 88–100. https://doi.org/10.1145/3381052.3381325

[41] Chris Seaton. 2015. Specialising Dynamic Techniques for Implementing The Ruby Programming Language. PhD thesis, University of Manchester.

[42] Manuel Serrano. 2018. JavaScript AOT compilation. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages* (Boston, MA, USA) *(DLS 2018)*. Association for Computing Machinery, New York, NY, USA, 50–63. https://doi.org/10.1145/3276945.3276950

[43] Manuel Serrano. 2021. Of JavaScript AOT compilation performance. *Proc. ACM Program. Lang.* 5, ICFP, Article 70 (Aug. 2021), 30 pages. https://doi.org/10.1145/3473575

[44] Amir Shaikhha, Yannis Klonatos, and Christoph Koch. 2018. Building Efficient Query Engines in a High-Level Language. *ACM Trans. Database Syst.* 43, 1, Article 4 (April 2018), 45 pages. https://doi.org/10.1145/3183653

[45] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2022. Sound, Precise, and Fast Abstract Interpretation with Tristate Numbers. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 254–265. https://doi.org/10.1109/CGO53902.2022.9741267

[46] WebAssembly. [n. d.]. Binaryen: optimizer and compiler/toolchain library for WebAssembly. https://github.com/WebAssembly/binaryen

[47] T Würthinger, C Wimmer, C Humer, A Wöß, L Stadler, C Seaton, G Duboscq, D Simon, and M Grimmer. 2017. Practical Partial Evaluation for High-Performance Dynamic Language Runtimes. *PLDI* (2017). https://doi.org/10.1145/3062341.3062381

[48] Haoran Xu and Fredrik Kjolstad. 2024. Deegen: A JIT-Capable VM Generator for Dynamic Languages. https://doi.org/10.48550/arXiv.2411.11469 arXiv:2411.11469 [cs.PL]

Received 2024-11-11; accepted 2025-03-06