Partial Evaluation, Whole-Program Compilation

Or, We Have a Compiler at Home

CHRIS FALLIN, Fastly, 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 that can result in serious 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; also, 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; for example, the interpreter tier. In this work, we introduce a partial evaluator that can derive compiled code "for free" by specializing an interpreter with its bytecode. This transform operates on the interpreter body at a basic-block IR level and is applicable to almost unmodified existing interpreters in systems languages such as C or C++. 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 with only three hours' effort. Finally, we outline an approach to carry this work further, deriving more of the capabilities of a JIT backend from first principles while retaining semantics-preserving correctness.

1 Introduction

Most dynamic language runtimes start as interpreters, for their numerous initial advantages: interpreters are easier to develop and extend than compilation-based alternatives; 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, 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 corresponding 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.

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 [33]. The three major JavaScript engines (V8, SpiderMonkey and JSC) regularly have CVEs resulting from subtle JIT bugs (e.g. [2] as one recent example in V8). Even coarse-grained sandboxing, such as V8's "ubercage" [23], does not protect against correctness bugs that can violate isolation when multiple server-side tenants share one VM via isolates [9], or when multiple requests with different users' data are processed by one engine. Separately, the continued prevalence of JS engines' security bugs indicates that full language-semantics correctness is difficult to maintain; and, even if a bug does not yield a sandbox escape, it can be catastrophic to individual applications when miscompilations alter the application's logic.

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 as these languages are used to implement the underpinnings of modern infrastructure.

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

In this paper, we explore a technique – partial evaluation – to derive a language runtime's compiler backend automatically from an interpreter that already exists. This is an old technique, going back to Futamura [18, 19], with numerous modern implementations [7, 10, 29, 32, 44]. We observe, however, that most extant tools today require the interpreter to be *developed for the purpose* – in other words, written in terms of a framework (as with Graal/Truffle [44]), or in terms of a semantic-definition DSL [7, 29], or in a special restricted language [10]. Our contribution is to design 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). Our tool is an open-source, industrial-strength compiler that we call Rufus (renamed for double-blind review). In its initial form, it processes interpreters compiled to WebAssembly [24], as a portable and easily transformable IR that many compiler toolchains can target, but without loss of generality it should work on any optimizing compiler IR that uses basic blocks, such as LLVM [27], with some fairly 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 three hours' effort. There exists a long tail of language implementations that have only interpreters and that could benefit from this technique. Even established language runtimes can benefit on new platforms not supported by existing their JIT backends: for example, the SpiderMonkey JS engine (§6) has no JIT tiers when compiled to run on a server-side Wasm platform (i.e. sandboxed within a Wasm module), whereas Rufus 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 plausible future path for carrying this approach forward to include profile feedback in a *semantics-preserving* way, showing how one might derive a competitive JIT from language semantics (in an interpreter) from first principles.

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: that is the *Futamura projection*.

2.1 The Futamura Projection

Futamura [18, 19] 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 less tractable than the first projection, and we will not describe them further.

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. A partial evaluator performs a **bytecode-erased compilation** if the resulting specialized program does not load data from the bytecode stream; hence it should have program points that statically correspond to the *interpreted* program rather than the *interpreter*, and should no longer dynamically dispatch behavior based on the original bytecode.[1]

That is, we consider the result a desirable compilation if it replaces the interpreter's control flow with the native control-flow graph of the interpreted program. This is both itself a speedup (in our observations, often 1.5-2x) and a substrate for further optimizations: each instance of an opcode becomes its own static code, we can optimize it separately, and together with the opcodes around it.

```
1 void interpret(bytecode_t* pc,
2                Value* stack) {
3   while (true) {
4     switch (*pc++) {
5     case OP_add:
6       Value v1 = *stack++;
7       Value v2 = *stack++;
8       *--stack = value_add(v1, v2);
9       break;
10    /* ... other opcodes ... */
11    }
12  }
13 }
```

Fig. 1. An sketch of an interpreter loop written in C.

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.

```
1 void interpret_specialized_func0(
2     bytecode_t* _pc_unused,
3     Value* stack) {
4   Value v1 = stack[0];
5   Value v2 = stack[1];
6   stack[1] = value_add(v1, v2);
7   return;
8 }
```

Fig. 2. Compiled code resulting from constant propagation of `interpret` from Fig. 1 on one opcode.

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 [25], which specifies that a partial evaluator should "remove all computational overhead caused by interpretation."

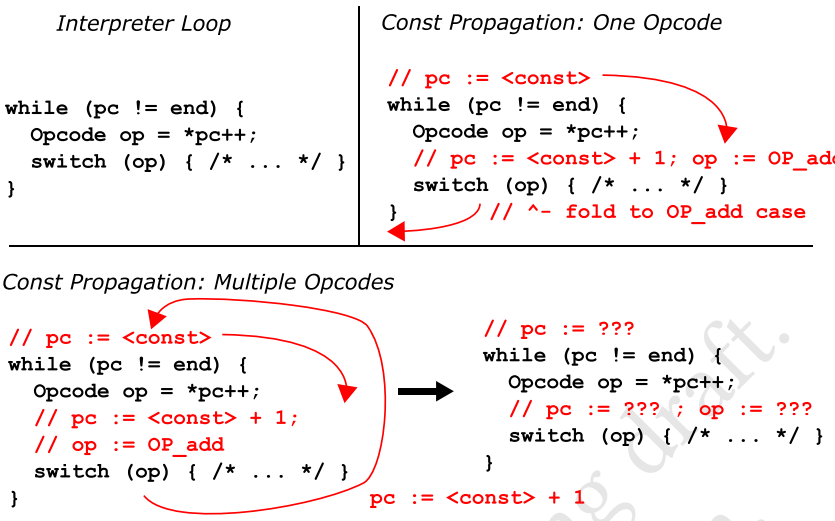

Fig. 3. An illustration of constant propagation over an interpreter loop: with one iteration, we can deduce constant values, but multiple iterations cause the analysis to degrade to "unknown" because all iterations are considered together.

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 [4], 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 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 [10], is to detect *hot loops* that result from loops in the interpreted program, and trace the interpreted execution of these loops, instantiating and specializing the interpreter loop once for each opcode in the loop.

This approach resolves the above limits but has 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 [20].

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.

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

We have argued that to produce a *bytecode-erased compilation* – that is, to produce compiled code that has a separate program point for each bytecode, turning the transitions in interpreter state into true control-flow edges – 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. For purposes of double-blind review, we will call this system Rufus.

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 [27] (§3.6). This transform is relatively small for its power, measuring at 5KLoC of Rust.

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 interpreter loop iteration. 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 iterated dataflow analysis is dynamic and expandable. This will be illustrated by an example below in Fig. 6.

This annotation is lightweight and minimal, yet it unlocks an entire specialization pipeline: it drives code duplication exactly and 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 of this. In other words, by avoiding the "meet-over-all-paths" trap that we described in §2.3, we achieve a *bytecode-erasing compilation* that produces an output control-flow graph that follows the bytecode rather than the interpreter.

```
1  void interpret(bytecode_t* pc) {
2   while (true) {
3     switch (*pc++) {
4       /* ... */
5     }
6     // Update analysis context:
7     // backedge reaches loop
8     // header in a new context,
9     // maintaining constantness
10    // of pc.
11    update_context(pc);
12  }
13 }
```

Fig. 4. Annotations to context-specialize analysis of an interpreter function.

Note that context may be nested: our actual intrinsics include `push_context()` and `pop_context()`, allowing, e.g., value-specialization or manual loop unrolling to occur inside of the main interpreter loop unrolling.

Furthermore, note that this requires the context value (`pc` here) to 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 workable. (What CFG should result in the compiled code? Will there be an edge to every block?) To support computed-gotos with a known list of destinations (e.g., switches or exception handlers), we allow for value specialization (§3.3).

Finally, note that this intrinsic is not load-bearing for correctness: it splits constant-propagation context, but the Rufus transform is sound regardless of how many separate contexts are used to analyze duplicates of code. The worst that happens with an arbitrarily wrong context is that specialization collapses back to "nothing is known" and the result is the original interpreter body. Separately, we provide an intrinsic that *asserts compile-time constantness* to help debug such *performance* issues.

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.

The algorithm operates over the generic (input) function in an SSA-based IR containing basic blocks, and is driven by a worklist of blocks to specialize. Blocks in the generic function are specialized *per context* into blocks in the specialized function. We keep a mapping from ⟨basic block, context⟩ tuples to specialized blocks, and likewise for SSA value numbers. We specialize one block at a time, performing constant-propagation analysis, const-folding and branch-folding as we flow forward. We track and update the *current context* as flow-sensitive analysis state, updated as necessary by intrinsics. When we reach a branch instruction, look up target block(s) in the current context. If already processed, create an edge to the corresponding specialized block. Otherwise, enqueue the block in context on a worklist. We provide this algorithm as pseudocode in Fig. 5.

In Fig. 6 we show an example of a specialization of a simple interpreter (supporting three opcodes, 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 specialization provides the semantic information that the bytecode is constant. Note, however, that no other knowledge of *interpreters*, per-se, is needed: this is a fully general transform for duplicating and constant-specializing code.

The analysis is worklist-driven and runs until fixpoint, but in practice in most cases, makes one pass over the bytecode, emitting the portion of the interpreter-switch corresponding to each opcode.

```
worklist = []    # Worklist of (Ctx, Block)
blockmap = {}    # Map from (Ctx, Block) to SpecializedBlock
valuemap = {}    # Map from (Ctx, Value) to SpecializedValue
valuestate = {}  # Map from SpecializedValue to CpropAnalysisState
# ... Dependency management to re-enqueue blocks omitted
# ... Flow-sensitive state management omitted

def partially_evaluate(func, args):
  for (arg, cprop_state) in args:
    valuestate[arg] = cprop_state

  initial_ctx = create_root_context()
  worklist.append((initial_ctx, func.entry_block))
  while len(worklist) > 0:
    (ctx, block) = worklist.pop()
    partially_evaluate_block(func, ctx, block)

def partially_evaluate_block(func, ctx, block):
  specialized_block = blockmap[(ctx, block)] or create_new_block(ctx, block)
  # We may be revisiting due to updated abstract state; empty the block.
  clear_block(specialized_block)

  # Partially evaluate and transcribe over the instructions.
  for inst in func[block]:
    specialized_args = [valuemap[(ctx, value)] for value in func.inst_args(inst)]
    (specialized_inst, abstract_state, ctx) = partially_evaluate_inst(
        func, ctx, specialized_block, inst, specialized_args,
        [valuestate[spec_arg] for spec_arg in specialized_args])
    valuestate[specialized_inst] = abstract_state
    append_to_block(specialized_block, specialized_inst)

  # Evaluate the terminator (branch) targets, enqueueing more blocks.
  # Omitted: if conditional or switch and constant selector, branch-fold.
  set_terminator(specialized_block,
    [evaluate_target(ctx, target) for target in terminator_targets(func, block)])

def partially_evaluate_inst(func, ctx, specialized_block,
                            inst, specialized_args, abs_states):
  if inst is intrinsic 'update_context':
    ctx = abs_states[0]  # Assert this is a constant; we do not support runtime ctx
  elif ...: # Handle other intrinsics
  else:
    abstract_state = constant_propagate(inst_opcode(func, inst), abs_states)
    if abstract_state is constant c:
      inst = create_const_value(specialized_block, c)
    else:
      inst = clone_inst(inst, specialized_block)
  return (inst, abstract_state, ctx)

def evaluate_target(ctx, target):
  specialized_target = blockmap[(ctx, target)] or create_new_block(ctx, target)
  # ... Meet flow-sensitive state into entry state (omitted) ...
  if newly created or entry state changed:
    worklist.push((ctx, target))
  return specialized_target
```


Fig. 5. Pseudocode for the main specialization (Futamura projection) algorithm.

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.

The resulting compiled code contains a control-flow graph that corresponds to the *interpreted program*, with its loop (the JMP backedge), rather than the interpreter. Thus, we have a *bytecode-erased compilation* as a result of a partial evaluation. This is an instance of a first Futamura projection.

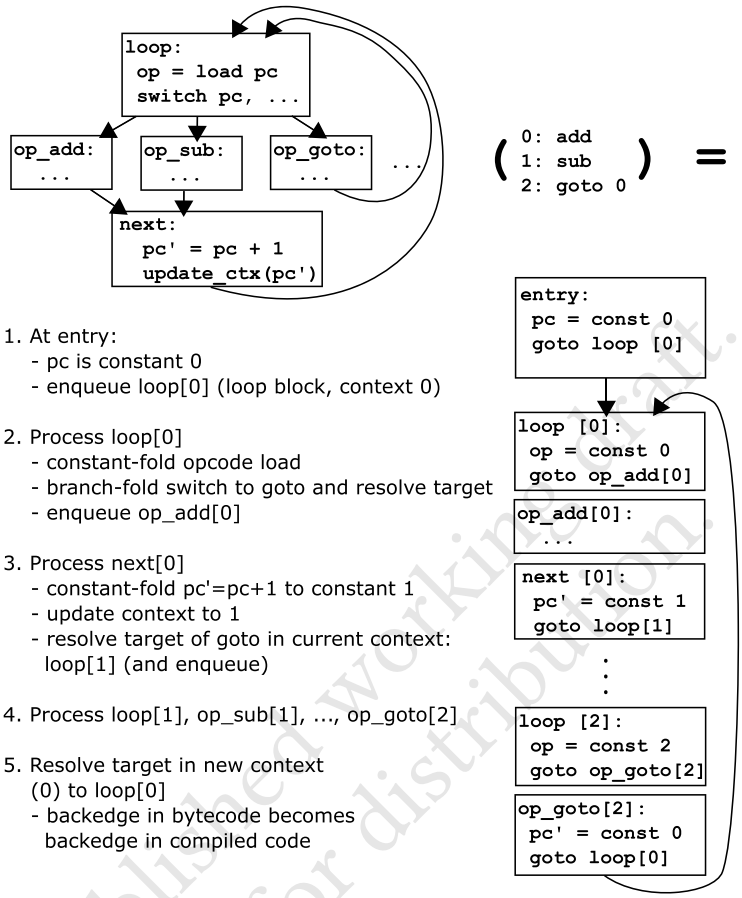

Fig. 6. An example of a partial evaluation of a simple interpreter on a three-opcode interpreted program.

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 (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`?

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 Rufus transform, but in the end, we decided this was a philosophical dead-end: it is better for the transform

targets (from e.g. switch statements). 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"* [26]). The idea is that rather than a scalar context (e.g., an interpreter PC), we add a sub-context index, so specialization maps key 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`.

3.4 Maintaining Static Single Assignment (SSA) Form

There is one optimization that is critical to grant the Rufus 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 Rufus 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. Thus, we must somehow repair the SSA, or ensure by construction that we do not violate this invariant.

A simple solution is to run the Rufus transform on a restricted form of SSA that uses values only in the blocks in which they are defined, and otherwise passes all values across control-flow edges explicitly with ϕ-nodes (or equivalently, block parameters). This guarantees correctness because it trivially removes any dependence on inter-block dominance relations. However, while this solution is correct, it 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).

We implement an analysis that finds a "minimal CFG cut" across which all values need to be made explicit block parameters. Intuitively, this cut is around where paths from different contexts may merge, e.g., the interpreter backedge: these points are where subgraphs of the original CFG are "glued together" to form a new overall shape.

This analysis operates by, in a fixpoint loop, for each basic block, finding the "highest same-context ancestor" (HSCA) in the dominance tree. Each block flows its HSCA outward on CFG edges; blocks that have update-context intrinsics have themselves as HSCA. If an HSCA flows into a block and does not dominate that block, the block becomes a cut-point and becomes its own HSCA. Otherwise, a block's HSCA is the domtree-join (lowest common ancestor) of all inbound HSCA values. Once we find all cut-points, we update these blocks to have block parameters for all live values flowing in.

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

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.

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 Rufus 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: *Run-time*, *SpecializedConst(value)*, or *SpecializedMemory(data)*. The first means the value is not known at compile time (no constant-folding occurs), and the latter two specialize on either a constant value or constant data at the given pointer, respectively. In essence, the specialization request makes the *promise* that the function parameter or the memory contents will have those values at invocation time: the semantics-preserving specialization is with respect to this promise. In order to retain function-pointer type compatibility, each specialized function continues to have 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 Rufus "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 [16] snapshot tool. Note, however, that this is not fundamental to the Rufus transform.

When integrated into a Wasm-snapshot build workflow, the top-level interface to our tool is a function 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 Rufus 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.

3.6 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.

[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 Rufus tool itself.

- 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.

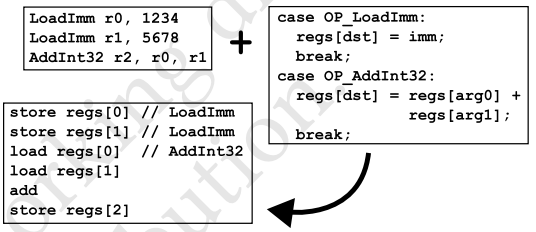

Fig. 7. Partial evaluation by itself removes dispatch overhead, but preserves load/store semantics of interpreter state data structures, leading to inefficiency.

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 proved intraprocedurally. Ideally we would like to indicate some other way to the partial evaluator that these values can be stored in true locals (i.e., SSA values in the Rufus transform result) rather than memory.

4.1 Virtualized Registers

In our tool, we allow the interpreter to communicate this intent via intrinsics. Specifically, we provide 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 always 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 have state that is necessarily exposed to the rest of the runtime. For example, in a garbage-collected platform, a GC

might need to inspect an array of local-variable values in order to mark them as rooted or update them after a compaction.

As above, we wish to lift the original in-memory storage to SSA when possible. However, these values will need to be written back to memory at certain points, and their new values reloaded. 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 kind of state operates like a write-back cache: the transform will perform true loads when necessary, and will generate stores at "flush" intrinsics. The intrinsics for in-memory locals are `read(index, address)`, `write(index, address, value)`, and `flush()`.

Beyond indexed locals, many interpreters also present a *stack VM* abstraction with opcodes that push and pop operands and results. To support this, we also provide `push(address, value)`, `pop(address)`, `read_stack(depth, address)`, and `write_stack(depth, address, value)` intrinsics. These perform an abstract interpretation of stack state, falling back to true loads when needed, and generating stores at flush-points as above.

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 when in-memory state is observable, a flush has occurred. Other design points might also be possible: for example, a new intrinsic or mode in our tool that flushes at every callsite, or that tracks escaped pointers to the state in some other way.

4.3 Discussion: Semantics-Preservation and Polyfills

These intrinsics differ from the initial function-specialization transform in §3 in two ways: (i) they grant the Rufus tool permission to diverge in semantics in controlled ways (*lazy flushing* of in-memory state with user-denoted sync 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.

Note that we have carefully designed the signatures so that polyfills are possible: the in-memory state intrinsics take canonical address arguments and can thus fall back to true loads and stores. The virtualized register intrinsics (§4.1) could be rewritten to loads and stores to an array allocated on the stack. For pragmatic reasons we have no 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 Rufus, we integrate it first into a minimal example interpreter—a small 64-bit unsigned integer register machine named *Min*. Min has 10 instructions that operate on a program counter *pc*, an array of indexed registers *registers*, and a special accumulator register *acc*. 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 summarized in Figure 9.

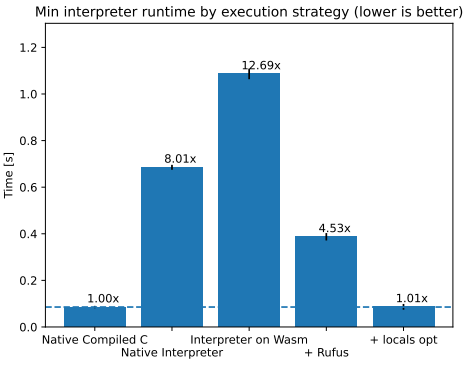

Fig. 8. Benchmark of the loop program with different execution strategies.

```
uint64_t Execute(uint64_t *program) {
  uint64_t accumulator = 0;
  uint64_t registers[256] = {0};
  uint32_t pc = 0;
  PUSH_CONTEXT(pc);  // NEW
  while (true) {
    uint64_t op = program[pc++];
    switch (op) {
    case LOAD_IMMEDIATE: {
      accumulator = program[pc++];
      break;
    }
    case STORE_REG: {
      uint64_t idx = program[pc++];
      registers[idx] = accumulator;
      break;
    }
    // ...
    case ADD: {
      uint64_t idx1 = program[pc++];
      uint64_t idx2 = program[pc++];
      accumulator = registers[idx1] +
        registers[idx2];
      break;
    }
    // ...
    } // end switch
    UPDATE_CONTEXT(pc); // NEW
  } // end while
  POP_CONTEXT(); // NEW
}
```

Fig. 9. *Min* bytecode interpreter in C. The listing is shortened for this paper but in total is 63 lines of code. Lines marked NEW are the added Rufus annotations.

```
#define REG_AT(idx) (IsSpecialized ? \
      read_reg(idx) :                \
      registers[idx])
#define REG_AT_PUT(idx, val)   \
  if (IsSpecialized) {         \
    write_reg(idx, val);       \
  } else {                     \
    registers[idx] = val;      \
  }

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

Fig. 10. We modify the macros to read and write registers to conditionally use Rufus'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.

The first step to Rufus-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 meaning when the program is being partially evaluated, we invoke the annotations with macros that are conditionally defined only in a build for Rufus.

Just these annotations alone will confer a performance boost; they unroll the interpreter loop into guest language control flow and allow for Rufus's optimizer "see through" the interpreter into the guest language. We can, however, do better: we can also use Rufus's interpreter state optimizations (§4).

To allow Rufus to optimize *registers* and remove loads/stores, in Fig. 10 we replace direct array accesses like `registers[idx]` with macros `REG_AT(idx)` and `REG_AT_PUT(idx, value)`

and define them to use `read_reg(idx)` and `write_reg(idx, value)` in a variant of `Execute` passed to the partial evaluator. Now we can unroll the "infinite register" bytecode that reads from and writes to an array 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. 8 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 Rufus 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 Rufus-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.

Min is an excellent introduction to Rufus because a first-year graduate student learned about the idea of Rufus, designed an interpreter, and then directly applied Rufus with minimal tutelage—all in the space of four hours. This indicates to the authors that Rufus is not only useful, but immediately applicable to many extant projects.

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 [3] 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 [6] which embeds SpiderMonkey to target Wasm-first platforms, where run-time code generation (JIT) is not supported. Rufus-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 [13]; in this mode, it runs only with its interpreter tiers, because Wasm does not support run-time code-generation. SpiderMonkey supports inline caches (ICs) in a fully interpreted mode using the *Portable Baseline Interpreter* (PBL) [15]; we take this as our baseline. We augment StarlingMonkey's compilation flow that uses the Wizer [16] snapshotting tool: once a snapshot of the engine with the user program's bytecode is taken, with function specialization requests enqueued in the heap image, Rufus processes those specialization requests and appends new Wasm functions to the module.

There is one further hurdle we must address: IC bytecode is generated only at run-time, when behaviors are observed, while our tool requires bytecode to be present in the snapshot for AOT processing. To address this dependency, we augment SpiderMonkey with an *IC corpus* mechanism that builds a pre-collected set of IC bodies into the binary, available in a lookup table.

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.

6.1 Changes to the Interpreter

In order to permit the Portable Baseline Interpreter (PBL)'s two interpreter loops – for JS and IC bytecode – to be partially evaluated, several minor changes were necessary. First, we had to

[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 [43]) 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`.

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 Rufus'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 restarts, 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.

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

6.2 Performance Results

In Fig. 11, we show the performance of our modified SpiderMonkey engine on the Octane benchmark suite [34], reporting throughput (inverse run time, i.e., speed) data for these configurations, all as Wasm modules running on Wasmtime [5]:

- *Generic Interp*: default (generic) interpreter;
- *Interp + ICs*: Portable Baseline (PBL) interpreter with inline caches;
- *Rufus'd*: AOT-compiled JS via partially-evaluated PBL interpreter;
- *Rufus'd + state opt*: same, with state optimizations (§4) – our final configuration.

The speedup of our approach is properly seen as the delta from *Interp + ICs* to *Rufus'd + state opt*. 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 *Rufus'd* configuration to *Rufus'd + state opt*, 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 there are more often GC safepoints in between that force values back into memory.)

6.3 Comparison to Native Execution

In order to judge the relative speedups attained by Rufus, and also eventual upper bounds, it is interesting to compare analogous execution strategies in a native-code configuration, with JIT

[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.

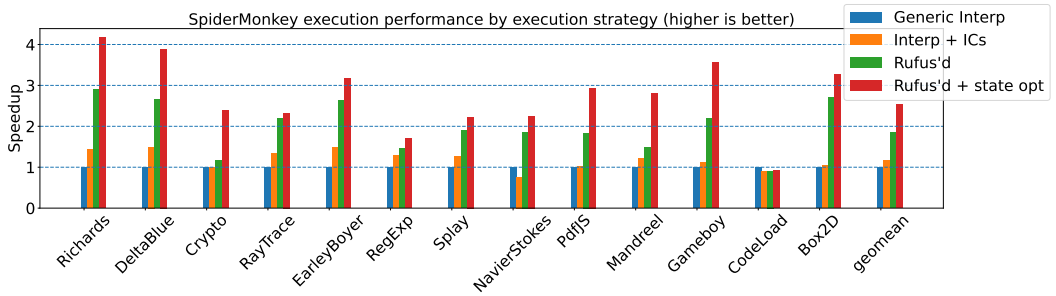

Fig. 11. Performance results of Octane benchmark suite on SpiderMonkey engine, with interpreter in a Wasm module (without and with ICs), and Rufus-compiled code (without and with interpreter-state optimizations).

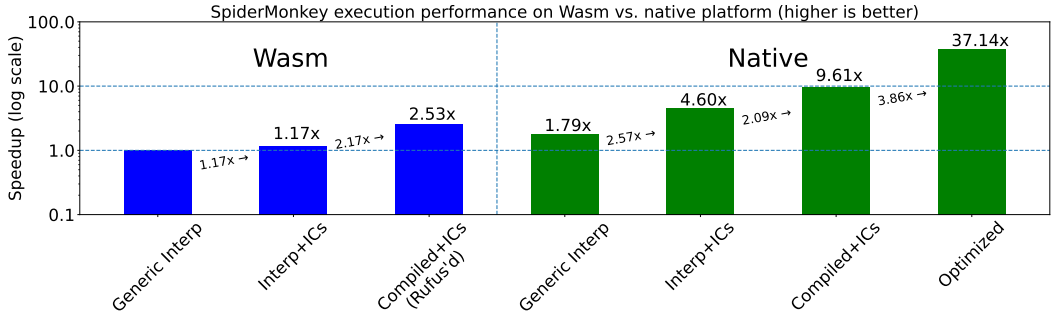

Fig. 12. SpiderMonkey configurations running on top of a Wasm engine vs. SpiderMonkey as a native build on the same system. This shows how (i) inline-cache fastpaths, (ii) compilation of JS bytecode and inline caches separately, and (iii) optimized compilation of both together (native only) result in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

backends. Note that we do not intend to compare Rufus'd 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 §6.2 on Wasm alongside four configurations running natively (e.g. directly on x86_64) on the same system:

- *Generic Interp*: the same generic interpreter as in the Wasm case (`js -no-ion -no-baseline -no-blinterp`);
- *Interp + ICs*: SpiderMonkey's baseline interpreter, which interprets JS bytecode but runs *compiled* IC stubs (`js -no-ion -no-baseline`);
- *Compiled + ICs*: SpiderMonkey's baseline compiler, which compiles JS bytecode and ICs, comparable to the optimization level of Rufus'd code (`js -no-ion`);
- *Optimized*: a fully optimized combination of JS bytecode with type-specialized cases inlined, yielding maximal performance (`js` default native engine).

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 Rufus 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 runtime 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 Rufus: 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.

6.4 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 Rufus of the entire Octane benchmark suite (7.5MiB or 337KLoC 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. With more optimization work in our tooling, including our Wasm compiler backend, we believe the size of generated bytecode could be decreased substantially.

6.5 Transform Speed

Compiling the above Wasm module takes, in total, 350 seconds of CPU time (44.16 seconds wallclock parallelized over specialization requests on a 12-core machine). This indicates a compilation speed of slightly under 1KLoC/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, with which no author had any familiarity, to Wasm and applied Rufus-based partial evaluation in the space of three hours. We split the process into the following chunks:

Support Wasm. Porting the interpreter to Wasm took approximately 45 minutes. 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].

[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.

Support Wizer-based snapshotting. After the initial Wasm port, we spent another 45 minutes adding snapshotting. 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 pushes that module's `main` function (a convention we arbitrarily chose) onto the call stack. The resume function—the new C `_start`—calls this function.

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 Rufus 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 Rufus 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 implementation simple.) When we make the Rufus request, we also pass it the address of *spec* field for Rufus to fill in later, after Wizer has run.

We tested this step before we added annotations to the interpreter: at this point, Rufus 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 had been filled in. We also ensured that the interpreter calls itself for each Lua call, rather than handling the call opcode "inline".

After a short period of debugging, 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 Rufus'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. We expect this to take not significantly more time for programmers who are already familiar with the code.

8 Related Work

Partial Evaluation: There is a rich pre-existing literature on partial evaluation, going back at least to Futamura [18, 19]. Jones [26] provides a comprehensive overview of the field. Several aspects of the Rufus transform, such as its use of value specialization ("The Trick"), are standard techniques for partial evaluators; and others implement optimizations such as virtualized handling of interpreter state (e.g., PyPy [10]) as well. Our particular tool differs primarily in targeting WebAssembly, and possibly in its particular basic block- and SSA-oriented specialization transform.

DyC [21, 22] is a run-time optimization system for C that performs partial evaluation. It targets use-cases where run-time data could be used to greatly simplify or specialize a program's logic: for example, simulators (for a particular configuration) or numeric code (for known dimensions or parameters), in addition to interpreters. It provides annotations (e.g., `make_static()` to perform "The Trick" of value specialization) and performs binding-time analysis; the work demonstrates the ability to unroll interpreter loops, as we do. However, the system appears to be significantly more complex, relying on heuristics and analysis to handle interprocedural specialization, overlapping specialization regions, caching of specializations, and more. In contrast, our tool (Rufus) is only 5K LoC, and provides a comprehensible semantic model for its transform.

GraalVM and Truffle [44] are a JIT compiler backend and language runtime framework (respectively) that perform the first Futamura transform. The Truffle ecosystem supports high-performance execution of Ruby [38, 44], JavaScript [44], and other popular languages.

Truffle users rely on a Truffle-provided framework that supports AST-walking interpreters. Users' AST node classes must inherit from Truffle classes. In other words, the interpreter must be developed specifically for Truffle. The engine also includes support for run-time optimization and de-optimization based on changing or violated assumptions, which is very useful for dynamic language support. The tradeoffs come in terms of significant added complexity, and long warmup times. More recently, Truffle has been used to build interpreters for bytecode (e.g., TruffleWasm [37]), though by building AST nodes for bytecode instructions.

GraalVM also supports a "Native Image" feature that shares some motivation (startup latency, etc) with the Wizer [16] WebAssembly snapshotting tool (used as part of the workflow in our case studies), but it goes further: it runs static initializers, then uses a closed-world assumption and a loop of (optionally context-sensitive) points-to analysis and optimization to specialize Graal IR. Then it produces a small native binary. Taken together with a Truffle-based interpreter, this presents another way to build an AOT compiler for a language from an interpreter.

PyPy [10] is a Python implementation built on top of the RPython meta-tracing JIT compiler infrastructure. RPython has a similar API to our annotation infrastructure. Constructing an instance of the `JitDriver` class requires annotating which variables are constant for the execution of a particular instruction and which are not; `JitDriver.jit_merge_point` unrolls the interpreter loop using an arbitrary context (ordinarily the `pc` value). RPython also allows for optimizing interpreter state, as we do. RPython is different, however, because (i) interpreters must be written in RPython, a restricted subset of Python 2.7, and (ii) the interpretation and compilation strategy is based on a linear program trace instead of an entire method or CFG. Additionally, it is not possible to use RPython for AOT compilation.

BuildIt [12] is a C++ library for partial evaluation of C++ programs. It provides similar annotations in the form of templated types—`static_var` and `dyn_var`—for partitioning variables into compile-time constants and run-time data, respectively. BuildIt generates C++ code. With BuildIt, it is possible to do similar unrolling of interpreter loops and promotion of local variables, but it is easier to go wrong; mis-use of `static_var` (analogous to our `push_context`) can lead to semantic differences other than performance. Additionally, unlike Rufus, BuildIt relies on the user to provide a C++ compiler to compile the generated code.

Lightweight Modular Staging (LMS) [35, 36] 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 [41]. Like BuildIt, it is general-purpose and requires annotations like `Rep` (comparable to BuildIt's `static_var`). Unlike BuildIt, it can target more language backends. Using this library requires using Scala as the host language.

Deegen [7] aims to generate a fast interpreter, baseline JIT, and—eventually—an optimizing JIT from a description of language semantics. Deegen provides a C++ DSL for describing interpreter semantics in the form of an infinite register bytecode VM. It has APIs for defining opcodes, type-specialized variants of opcodes, inline caches, a type lattice, "slow paths", and more. Similarly to PyPy, it cannot be used for AOT compilation; it intentionally burns constants into the generated code for better performance.

SemPy [29] and Static Basic Block Versioning [30] (SBBV) are works by Melançon, Feeley, and Serrano that seem to be building toward a similar goal of deriving 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 to a semantics with

many dynamic type checks: it is an algorithm for finding a set of type-specialized contexts ahead-of-time to generate code for (without knowing what the real types will be), while applying a technique to limit combinatorial blowup. This is an alternative to our approach to dynamic types in SpiderMonkey: we observe that AOT compilation is possible when inline caches (ICs) are used if all type-driven dynamism is pushed to late-binding (run-time-filled) IC callsites, while SBBV allows optimization beyond that level, permitting type-specialized variants of code with strongly typed values fully unboxed. These are complementary, and an SBBV-like algorithm could in theory be applied to a Rufus'd interpreter body with intrinsics denoting type-specialization points.

LuaAOT: We apply Rufus to a Lua interpreter to AOT-compile Lua bytecode. LuaAOT [32] is a purpose-built AOT compiler framework that, superficially, works similarly: 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 implementation; in contrast, our Lua modification achieves 1.84× speedup (on top of WebAssembly, though in principle Rufus is not limited to Wasm, as we noted in §3) with a `+173 -57`-line diff.

AOT JS Compilers: While AOT compilation of JavaScript is not the focus of our work, our evaluation on SpiderMonkey does yield such a compiler. Several other options exist. Hopc [39, 40], Porffor [28], Static Hermes [31], and Static TypeScript [8] are all compilers that accept JavaScript, TypeScript, or some annotated 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 solution. On the other hand, they can perform optimizations in a more straightforward way, potentially yielding higher ultimate performance.

9 Future Work

9.1 Profile-Guided Inlining and Semantics-Preserving Optimizations

The level of optimization that the Rufus 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, inline body, else call-indirect). In this way, just as for the basic Rufus transform, *semantics are fully preserved*. The Winliner tool [17] prototypes this optimization strategy. This parallels how SpiderMonkey's WarpMonkey [14] backend generates its optimizing compiler input from inlined ICs.

Such inlining would remove the IC indirect-call overhead, and it indirectly creates opportunity: it places *boxing* and *unboxing* operations 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 [11, 42]. Overall, this is a further step toward a goal of *safe dynamic-language compilers*, where "safe" means that the only "load-bearing" semantics are described in the interpreter body.

9.2 Specializing at Run-time

While our initial prototype is used to derive AOT compiler backends, we are also interested in generating JIT backends. Nothing prevents the Rufus transform from operating at run-time.

Operating at run-time places stresses on the performance of the specialization algorithm itself. In principle, the *second* Futamura transform could generate an efficient compiler directly from the interpreter, without the specializer (the Rufus tool) remaining at all. This is out of reach today, but remains an interesting end-goal.

References

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

[2] 2024. CVE-2024-4761. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2024-4761

[3] 2024. SpiderMonkey JavaScript Engine. https://spidermonkey.dev/.

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

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

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

[7] Anonymous Authors. 2025. Deegen: A Compiler Generator for Dynamic Languages. *PLDI* (2025).

[8] 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

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

[10] 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

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

[12] 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

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

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

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

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

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

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

[19] 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

[20] 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

[21] 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

[22] 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

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

[24] 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

[25] 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.

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

[27] 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

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

[29] 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

[30] 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

[31] Tzvetan Mikov. [n. d.]. 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

[32] 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

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

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

[35] 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

[36] 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

[37] 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

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

[39] 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

[40] 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

[41] 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

[42] 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

[43] WebAssembly. [n. d.]. Optimizer and compiler/toolchain library for WebAssembly. https://github.com/WebAssembly/binaryen

[44] 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