A fast in-place interpreter for WebAssembly

BEN L. TITZER, Carnegie Mellon University, USA

WebAssembly (Wasm) is a compact, well-specified bytecode format that offers a portable compilation target with near-native execution speed. The bytecode format was specifically designed to be fast to parse, validate, and compile, positioning itself as a portable alternative to native code. It was pointedly *not* designed to be interpreted directly. Instead, design considerations at the time focused on competing with native code, utilizing optimizing compilers as the primary execution tier. Yet, in JIT scenarios, compilation time and memory consumption critically impact application startup, leading many Wasm engines to later deploy baseline (single-pass) compilers. Though faster, baseline compilers still take time and waste code space for infrequently executed code. A typical interpreter being infeasible, some engines resort to compiling Wasm not to machine code, but to a more compact, but easy to interpret format. This still takes time and wastes memory. Instead, we introduce in this article a fast in-place interpreter for WebAssembly, where no rewrite and no separate format is necessary. Our evaluation shows that in-place interpretation of Wasm code is space-efficient and fast, achieving performance on-par with interpreting a custom-designed internal format. This fills a hole in the execution tier space for Wasm, allowing for even faster startup and lower memory footprint than previous engine configurations.

CCS Concepts: • **Software and its engineering → Interpreters**.

Additional Key Words and Phrases: WebAssembly, virtual machines, runtime systems, interpreters, performance

1 INTRODUCTION

Emerging first for the Web in 2017 [37], WebAssembly is a portable, low-level compilation target supported in all major browsers. Originally designed as a successor to asm.js [66], which allowed C/C++ to be compiled to JavaScript, it supplanted other technologies such as Native Client [65] as the new best target for native compilation to the Web. Since that time, WebAssembly has seen rapid uptake in a number of new spaces, including cloud computing [48], digital contracts, edge computing [46][7], IOT [42], and embedded systems [36].

A key design criteria for Wasm was offering performance competitive with native code. Initially, top-tier performance was considered paramount, and approaching native code performance to compete with technologies like Native Client was realized by reusing the optimizing JIT compiler infrastructure in browsers. Yet Wasm bytecode was also designed to be fast to parse, verify, and compile−criteria validated during the design process by building single-pass validators and single-pass decoding to SSA compiler IR to minimize upfront costs in the compilation pipeline.

However, despite minimizing bytecode parsing work by careful design, optimizing compilers inescapably take considerable time and memory to produce good native code, penalizing application startup in JIT scenarios. To address startup time problems, browsers prototyped separate, faster compilers during Wasm's design phase, validating that the same choices that enabled single-pass verification enabled single-pass compilation. Such often-termed "baseline" compilers spend far less compilation time, often 10×- 20× less, but produce code that typically runs 1.5× to 3× slower than an optimizing compiler. This represents a classic tradeoff space known to VMs for decades; more compilation time means better code quality. Today, all browser engines employ multiple Wasm compiler tiers to strive for *both* good startup time *and* high throughput.

Author's address: Ben L. Titzer, btitzer@andrew.cmu.edu, Carnegie Mellon University, 4665 Forbes Avenue, Pittsburgh, Pennsylvania, USA, 15213.

1.1 Whither the Interpreter?

Seemingly overlooked in this development arc is the obvious choice of using an *interpreter* to execute bytecode. After all, traditionally, virtual machines are developed with an interpreter first. There are a lot of advantages to interpreters.

(1) Since interpreters are easier to write, understand, and maintain, they allow more rapid experimentation in bytecode design.
(2) Since they need no translation or rewriting step, start up is fast.
(3) Bytecode is usually more compact than machine code, so interpeters generally use less memory than compilers.
(4) Debugging application code is easier, as the interpreter loop can be stopped at any instruction and program state inspected, altered, and resumed.
(5) Interpreters are easier to audit, since there is a fixed amount of code, and usually have fewer security vulnerabilities.
(6) Dynamic code generation is sometimes impossible, either because is not allowed on the platform, like iOS, or code space is limited.

For all these reasons, nearly all virtual machines, from pioneering work on Lisp to Smalltalk to Self, to today's broadly-accepted VMs such as the Java Virtual Machine, the CLR, Python, Ruby, and JavaScript, have an interpreter.

Why then, is Wasm any different? The answer is simply that efficient interpretation was explicitly *not* in the design criteria[1]. But some Wasm engines do indeed employ interpreters, such as JavaScriptCore, Chakra[2], and Wasm3. These engines use interpreters for exactly the advantages listed above. Yet none of these interpreters work directly on the original bytecode; all of them rewrite Wasm bytecode to a different internal format. Rewriting Wasm bytecode has similar disadvantages to baseline-compiling: it still takes time and memory.

1.2 The Final Tier is Shed

There is an important point missing in the Wasm virtual machine design tradeoff space. An *in-place* interpreter, i.e. one that interprets the original bytes of a binary module, would offer the best startup time and lowest memory consumption. For cold or never executed code, where the downside of the interpreter's much slower execution speed is outweighed by the major savings of avoiding translation time, it would be the optimal choice. Employed in concert with compilation tiers for hot code, such an interpreter would serve the role it does in other mature systems like JVMs. Is it possible to interpret Wasm in-place efficiently? Until now, this was thought infeasible. In this work, we solve this open problem and supply the missing point in the design tradeoff space: the first fast in-place interpreter for Wasm (empirical measurements in Figure 1&2).

---

[1]In fact, in a smoky back room, I probably declared, "Interpreters don't matter here."
[2]Though discontinued, ChakraCore was the first Wasm engine to feature a rewriting interpreter.

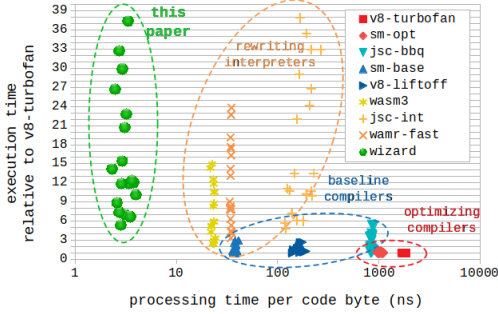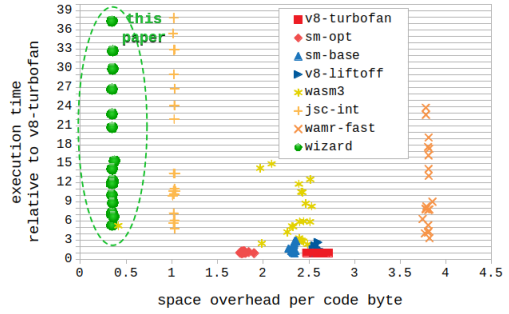Fig. 1. Execution time vs translation time.

Fig. 2. Execution time vs translation space.

We first identify the interpreter-crucial information that is missing in Wasm's bytecode design. This information, namely key control-flow and value stack information, is not actually missing, but rather *implicit*. Our insight is that the validation algorithm for Wasm bytecode *already* computes this information in its modeling the control and value stack during typechecking. All that remains is to distill a few key numbers into a compact side-table that is used during interpretation. The side-table is organized so that all accesses occur in constant ($O(1)$) time, and no searches of the table are necessary. Thus the interpreter always has relevant information directly at hand and behaves like a standard interpreter. Other details are important for making a fast interpreter as well, such as hand-coding key parts in assembly and combining exactly the right layout of value stack and virtual memory protections to robustly handle application stack overflow without needing any explicit checks.

With these new techniques, we have finally achieved an in-place interpreter for Wasm that is on par with state-of-the-art interpreters for other bytecodes and for Wasm interpreters using rewriting. This paper completes the triad of basic tier designs (interpreter, fast compiler, optimizing compiler) for Wasm. In a comical twist of fate, Wasm's tiers have arrived exactly backward!

1.3 Organization

The remainder of this paper is organized as follows. Section 2 recaps the design of Wasm's functions, stack machine, and control flow constructs, which are key to understanding why interpretation has been challenging until now. Section 3.1 shows the design of the side table used for the interpreter and how the validation algorithm already contains the key information necessary to emit the side table in a single pass. Section 3 details the interpreter implementation, including key assembly techniques to achieve the best-performing dispatch loop, and the design of the data structures necessary to make a fast operand and execution stack. Section 4 evaluates the interpreter on standard benchmarks and compares translation time, memory consumption, and execution time to JITs (4.4.1) and other interpreters (4.4.2) for Wasm. Prior work related to optimizing interpreters is summarized in Section 5, followed by the conclusion.

2 WASM DESIGN

Wasm provides a low-level programming model consistent with its original goal of a minimal, high-performance abstraction over hardware. Its principal features include:

- `i32`, `i64`, `f32`, `f64`, and `v128` primitive types
- an opaque reference type

- 32- and 64-bit integer arithmetic
- single- and double-precision floating point arithmetic
- 128-bit vector operations
- large byte-addressable memories with explicit load and store instructions
- functions with local variables
- direct and indirect function calls
- global variables

The Wasm binary format is designed to be compact yet fast to decode and validate in a single pass. This includes not just bytecode, but all constructs.

2.1 Modules and Instances

Wasm code is organized into *modules* which are in turn organized into a list of *sections*. Sections in a module declare functions, memories, tables, global variables and static data. Bytecode is grouped into *functions* with statically-typed *parameters*, *results*, and *local variables*. All operations in core Wasm manipulate only a module's own internal state. Modules must *import* functions (and memories, tables, etc) in order to access platform capabilities or state outside the module. Imports may be provided by the "host" environment, such as JavaScript and the Web, or from other modules.

A module is akin to an executable file, or part of one, rather than an executing program. To run, a module must be *instantiated*, supplying bindings for its imports. At instantiation time, a Wasm engine creates the state (tables, globals, and memories) declared by the module, with the result being called an *instance*. An instance may *export* its own functions, memories, tables, etc. to other modules or the host environment.

The primary dynamic storage of a Wasm program is typically one large, bounds-checked, byte-addressable *memory*, but global variables and tables of opaque host references can also be used. Future proposals will add first-class function references and garbage-collected objects to Wasm. These too are forms of local state and must be shared explicitly with other instances.

2.2 Bytecode design

Functions. All code[3] in Wasm is organized into functions. Functions each have a signature with a fixed number of parameter and result types, such as `[i32 f32 externref] -> [i32 i32]`. Execution of a Wasm program entails executing functions that may call each other, maintaining an execution stack[4] that stores their local variables and operands, and running their internal code. In a binary module, the body of a Wasm function begins by declaring the number and type of their additional local variables, followed immediately by the bytecode.

Stack machine. As is common for many bytecode designs, Wasm is a *stack machine*, meaning individual bytecode instructions take their operand values from an *operand stack* and push their results back. Local variables are separate. To be used, a local must be explicitly loaded onto or stored back from the operand stack. Implementations typically store them internally as a prefix of the operand stack, together referred to here and throughout as the *value stack*. The arguments of an outgoing function call become the first locals of the callee function.

Structured control flow. Unlike most bytecode designs, however, Wasm has *structured control flow constructs* such as blocks, ifs, loops, and switches that are encoded inline in the bytecode. We refer to them as *structured*, since they must be properly nested. This was a deliberate choice for compactness and to ensure that bytecode validation can be done in a single pass with minimal data

---

[3]other than trivial initializers for globals

[4]Note that the execution stack is not aliased by Wasm memory, thus not vulnerable to stack smashing

structures[5]. In contrast, a typical bytecode design with jumps usually requires more bytes to store and two passes to verify.

Direct interpretation not straightforward. Most bytecode formats can be interpreted directly in their binary form (i.e. in-place), with an instruction pointer stepping through the bytes of the original code. Jumps typically have an offset or instruction number of the target instruction directly in the bytes, allowing a constant-time adjustment of the instruction pointer. But Wasm is unusual in that a branch instruction specifies a target construct by *relative nesting depth*, transferring control to the beginning (in case of **loop**), middle (in the case of **if**) or end (for **block**, **if**, **else**) of the construct. Wasm is also unusual in that branches can also copy and pop values off the operand stack[6]. Thus a direct Wasm interpreter faces two unusual problems when executing a branch:

- can't quickly find the target bytecode offset, e.g. start of a **loop** or **end** of a **block**
- can't determine how many values to pop off the operand stack

To understand how this paper efficiently solves this open problem, we must first journey deep into how Wasm bytecode validation is done in production-quality engines.

2.3 Bytecode validation

Wasm, though low-level, is typed. A number of static checks ensure that a module is well-formed. Within a function, all instructions, including arithmetic, calls, control flow, etc. must be applied to the correct number and type of operands. All control flow constructs must be properly nested with no invalid branches. In the specification, this validation is expressed in a standard type system formalism. In engines, the algorithm is implemented as an abstract interpreter that models an *abstract* control and value stack. We describe such an implementation here to later make it clear how our modifications affect an already very efficient verification implementation.

Figure 3 illustrates the operation of a production quality Wasm code validator. Three primary data structures are used.

- The *module environment* models the types of functions, tables, globals, and number of memories of the enclosing module. The module environment is not mutated during verification of a function's bytecode.
- An *abstract control stack* models the nested control-flow constructs, keep tracking of where each starts and its expected parameter and result types[7]. This is used to properly match **block**, **if**, and **loop** starts with their **else** and/or **end**.
- An *abstract value stack* tracks abstract values for stack slots and local variables. In current Wasm, abstract values are simply *value types*, i.e. **i32**, **i64**, **externref**, etc. If a future proposal introduces flow-sensitive validation, the abstract values for locals would need to be extended to include initialization information.

The validation algorithm proceeds in a single forward pass over the bytecode, never needing to backtrack. For simple instructions like arithmetic or calls which only pop their operands from the value stack and push results back, the algorithm pops and checks required input types and pushes resulting output types. Control-flow instructions are validated using the control stack. For example, a **br** (branch) instruction references the target **block** or **loop** by relative nesting depth;

---

[5]Provably minimal, if a CFG is restructured from the known algorithm for optimal interval analysis, ensuring the validation metadata per control flow construct is discarded and reused as promptly as possible.

[6]Other stack machine designs, like JVM bytecode, allow values on the operand stack while branching, but stack heights and contents must match. Thus JVM branches cannot implicitly pop values.

[7]In Wasm, all control constructs can have data parameters and results, represented as a block signature. Thus a block can be seen as a super-instruction; it pops values off the stack, and every path to its end pushes the same number and type of results. This allows for very compact code, often obviating local variables. In practice, most blocks have an empty block signature.
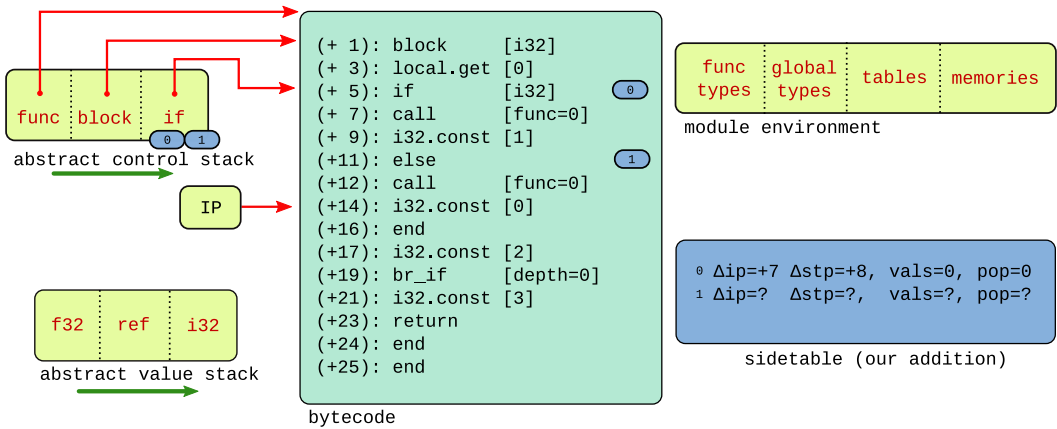
Fig. 3. Illustration of data structures used in production Wasm code validators.

the validator matches the opening construct by indexing into the control stack. Since, in Wasm **block**, **if**, and **loop** can have parameter and result types, the validator must check that the value stack contains the appropriate types expected for the target construct.

Importantly, an **end** bytecode closes a control-flow construct, either a **block**, **if**, **loop**, or the whole function. At **end**, all branches that can legally target the construct have been seen. Thus the implicit target bytecode offset of all branches *to this construct* are known, because that offset must be either the start of a **loop** or, for any other construct, the **end**, i.e. *here*. This information just needs to be saved somewhere easily accessible for the interpreter. After processing **end**, the algorithm pops the control stack entry and can reuse its storage space, which is optimally efficient.

3 INTERPRETER DESIGN

In this section, we present our fast in-place Wasm interpreter design.
The key enabling techniques are:

- an innovative side-table design which allows efficient access to missing branch information
- a highly efficient value stack organization for $O(1)$ local variable and operand access as well as zero-copy function calls

Additionally, we chose to implement the core logic of the interpreter in hand-written assembly language[8], which allows for near-perfect register allocation and unlocks all possible dispatch and organization techniques. We discuss the rationale for hand-written assembly at the end of this section.

3.1 Sidetable Design

As we've seen, Wasm control-flow bytecodes represent nested control constructs, rather than low-level jumps. However, an interpreter needs the bytecode offset of where to go if a branch is taken, ideally in $O(1)$ time. This includes not only explicit branches like **br**, **br_if**, **br_table**, but also the implicit branch in **else**. Note that **block**, **loop**, **end** and **return** are *not* branching bytecodes, since they either fall through or exit the function.

To supply the in-place interpreter with the missing information, the validation algorithm saves a portion of its work into a per-function side-table data structure separate from the original bytecode.

---

[8]More precisely, a macro assembler API in a high-level language that has methods to generate individual instructions, allowing it to be configurable in ways that typical textual assembly languages are not.

This side-table is a compact, highly-efficient mapping from branch origin to target offset, plus some additional stack manipulation information.
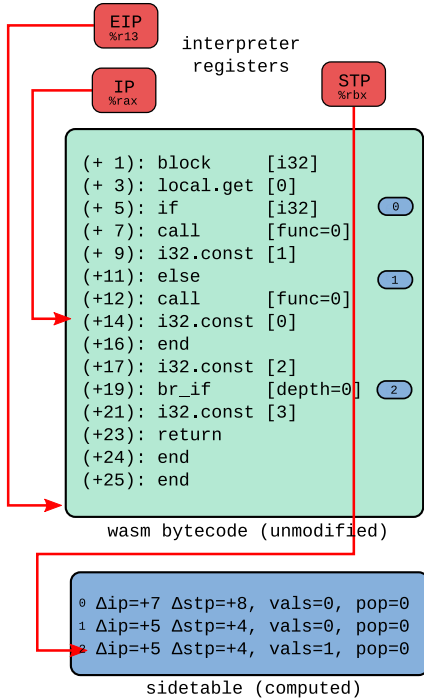

Fig. 4. Interpreter code and sidetable.

```
function_entry:
  IP  := &func->code->original[0];
  STP := &func->code->sidetable[0];
  decodeLocals();
  executeNextBytecode();

handle_BR:
  doControlTransferFromSTP();

handle_ELSE:
  doControlTransferFromSTP();

handle_BR_IF:
  var cond = popI32();
  if (cond != 0) {
    doControlTransferFromSTP();
  } else {
    IP = nextIP;
    STP = STP + #STP_entry_size;
    executeNextBytecode();
  }

handle_BR_TABLE:
  var key = popI32();
  var max = STP->maxcase;
  if (key > max) key = max;
  STP = STP + (1 + key) * #STP_entry_size;
  doControlTransferFromSTP();

def doControlTransferFromSTP() {
  moveValues(STP->valcnt, STP->popcnt);
  IP = IP + STP->delta_ip;
  STP = STP + STP->delta_stp;
  executeNextBytecode();
}
```

Listing 1. Interpreter pseudocode for branches.

To make the data structure time- and space-efficient, it consists of entries sorted by branch origin and omits non-branch instructions. It is emitted as a side-effect of the single-pass validation algorithm above. Because the validation algorithm already visits bytecodes in forward order, it simply emits branch entries as it goes, obviating a separate sorting step. Since only branches need entries, the sidetable is very small, often empty. Empirically, most Wasm functions are small with no control flow, so they have no side-table at all.

Every entry in the side-table is a 4-tuple of the form $\langle \Delta\textbf{ip}, \Delta\textbf{stp}, \textbf{valcnt}, \textbf{popcnt} \rangle$, where:

- $\Delta\textbf{ip}$ the amount to adjust the instruction pointer by if the branch is taken
- $\Delta\textbf{stp}$ the amount to adjust the *side-table pointer* by if the branch is taken
- **valcnt** the number of values that will be copied if the branch is taken
- **popcnt** the number of values that will be popped if the branch is taken

3.1.1 The Sidetable Pointer. Like most interpreters, our in-place interpreter maintains an instruction pointer (**IP**) into the bytecode during execution. It also maintains an *end* instruction pointer (**EIP**), which is used to check if the program falls off the end of the function, which is a legal implicit return in Wasm. To use the side-table, the interpreter simply maintains another pointer, the side-table pointer (**STP**), consulted when executing branches.

Figure 4 illustrates the interpreter state for the bytecode and sidetable, and Listing 1 illustrates how side-table entries are used by the interpreter during execution. The instruction pointer (**IP**)
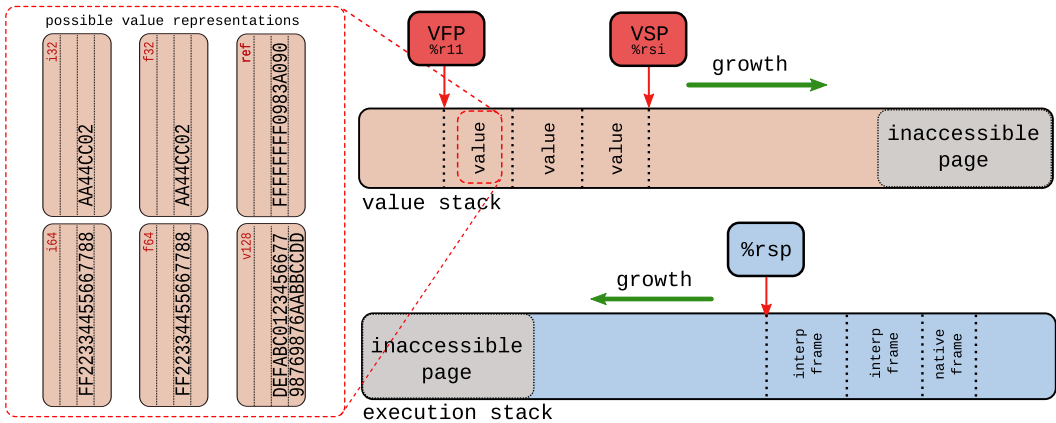
Fig. 5. Value stack and execution stack layout.

is initialized to point directly into the bytecode and the sidetable pointer (**STP**) points at the first side-table entry. Branch instructions make use of the **doControlTransferFromSTP()** subroutine which adjusts both the **IP** and **STP** based on the entry to which it points, as well as adjusting the value stack. Notice that a conditional branch that is not taken still must update the **STP** so that it points at the next entry for the next branch, if any, in the code. Note that **br_table** works much like a jump table, computing an index into the side-table and using the corresponding entry.

3.2 Value stack design

Since Wasm bytecode constitutes a stack machine, nearly all instructions access the value stack, making it crucial for interpretation speed. A single indirection to local variables and the top of the operand stack is ideal. Wasm's numbering scheme for locals is inspired by JVM bytecode. Parameter #0 is local #0, followed by declared locals, then the operand stack. Outgoing arguments of a call become the first local variables of the callee function's value stack. The JVM chose this design so interpreters could *overlap* the value stack of the callee frame with the operand stack of the caller, avoiding copying any argument values. This never panned out for Wasm until now. Our design succeeds in using the JVM trick to avoid copying arguments, but requires separating the value stack from the execution stack.

Figure 5 illustrates the value stack design. We separate the storage of Wasm program values from the control information of the interpreter itself. That is, the value stack contains only Wasm values, while the interpreter's call stack contains only control information, organized into one execution frame per Wasm call frame. As such, the value stack is a contiguous array of Wasm values which increases in size towards higher addresses. Though invisible to Wasm programs, and orthogonal to our design here, the interpreter frames are on the native stack and use the native stack pointer (e.g. **%rsp** on **x86-64**) which grows towards lower addresses.

3.2.1 Value tagging. Wasm values can be 32-bit or 64-bit integers, 32-bit or 64-bit floats, 128-bit SIMD vectors, or external references. We choose to store all Wasm values in the value stack as *unboxed*, so that the interpreter never needs to implicitly allocate a heap object[9]. This obviously makes sense for all primitive values, since they can share storage as raw bytes in the memory of the value stack.

---

[9]Boxing is a major overhead in dynamic language implementations and would be prohibitively expensive for Wasm, outweighing all optimizations described in this paper.
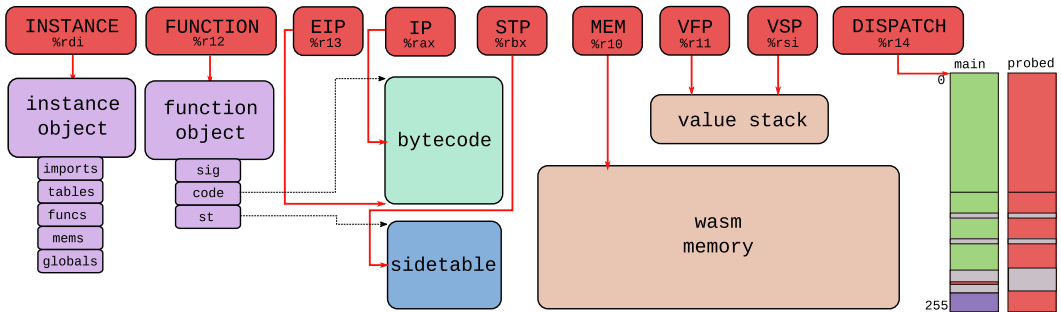
Fig. 6. All interpreter registers.

However, an engine may need to find **externref** values in a value stack during garbage collection. Natively-compiling Wasm engines use stackmaps [30]. But an interpreter is different, and typically cannot afford either space or time to precompute stackmaps. Instead, we chose to tag value stack entries with a 1-byte *type tag*, inflating entries to be 32 bytes, i.e. *really fat*[10]. Type tags are written into the value stack only when necessary (i.e. when initializing locals in the function prologue, not at all for primitive arithmetic, etc). To be clear, type tags are never needed for any dynamic check, since Wasm code is statically typechecked. They can be omitted if there is no garbage collector or it uses conservative stack scanning [28].

3.2.2 Stack overflow. Wasm engines must be robust to call stack overflow. The Wasm specification includes one test with unbounded recursive function calls. Every engine must fail gracefully, though the exact point where overflow is detected is not specified.

Checking for stack overflow should not ruin performance. A check on every value stack push would be prohibitive. Instead, we use a *guard page* at the end of the value stack and rely on an OS-level signal upon fault to catch and report stack overflow. A single guard page suffices, as the interpreter cannot inadvertantly stride over it; by design it never accesses arbitrarily far ahead. Similarly, interpreter execution frames (on the native stack) are all fixed size and another guard page[11] will cause a fault if the native execution stack overflows before the value stack.

3.2.3 Putting it all together. At this point, we've designed two critical data structures necessary to make an efficient in-place interpreter. Figure 6 completes the set of 9 state registers used by our interpreter implementation. In addition to the 3 "control" registers pointing into the bytecode and sidetable and the two "data" pointers into the value stack, the interpreter also requires:

- **MEM** a pointer to the start of the wasm memory[12]
- **FUNCTION** a reference to the current function
- **INSTANCE** a reference to the current *instance*
- **DISPATCH** for dynamically enabling per-instruction probing

3.3 Interpreter implementation in assembly

We chose to implement our interpreter in a new Wasm research engine, **wizard** [60]. While most of the engine is written in a portable, safe, high-level language [59], we use custom hand-written assembly for the fast interpreter. Using assembly or a custom code generation facility is relatively

---

[10]Aligning value stack entries on a power-of-two boundary allows for shift-based arithmetic when indexing.
[11]Using **sigaltstack** on POSIX platforms for signal handling.
[12]Though not shown, Wasm memories also have a guard region, obviating the need for an explicit bounds check.

common for production interpreters. For example, the Java HotSpot virtual machine generates its interpreter [1] using a macro assembler (at startup), V8's Ignition [14] interpreter is generated from hand-written TurboFan compiler IR (at build time), and JSC's LLint is written in a macro assembler language (build time). Several factors impinge on our decision.

(1) an interpreter is a small, important piece of code
(2) the bytecode format and semantics of Wasm are very stable
(3) compilers have trouble generating optimal code for interpreter loops
(4) we wish to study interpreter performance in detail
(5) key dispatch techniques are difficult to obtain from compilers
(6) developing and debugging assembly code is relatively time consuming

Key advantages of using assembly to implement an interpreter are 1) its code is not perturbed by changing compiler optimizations, 2) key interpreter state can be fixed to specific architectural registers, 3) threaded [20], indirect-threaded [29], and other dispatch techniques can be used, 4) handlers can be ordered and aligned cache line boundaries, 5) fast- and slow-paths can be organized inline or out-of-line, 6) all hardware instructions can be used, 7) self-modifying code and dispatch table swap techniques can be used, 8) error handling and hard cases can be factored from handlers, 9) very small resultant code footprint.

The interpreter that we present in this paper makes use of nearly all of these techniques. It is implemented using a macro assembler that generates **x86-64** machine code, has many switches to enable different features, and provides an instrumentation interface for profiling and debugging. It consists of 2800 source lines of code which generate approximately 14KiB of machine code and 7KiB of dispatch tables.

3.3.1 Dispatch tables and handlers. Wasm is a bytecode in the true sense of the word. An instruction is encoded as a byte-sized opcode, followed by zero or more immediates. Longer instructions are preceded by prefix byte. It is natural therefore to design a software interpreter around 256-entry *dispatch tables* that contain pointers to *handlers* that implement each bytecode. Figure 7 illustrates the dispatch table and handler organization for our fast Wasm interpreter.

The fast interpreter uses multiple dispatch tables, each of which points to a sequence of machine code called a *handler*. A *dispatch* through a table consists of loading the address at a particular index and indirectly jumping to that address. The first dispatch table, the *main* dispatch table, is used for the first byte of an instruction. Since the most important bytecodes in Wasm were assigned a non-prefixed opcode, the first dispatch through the main table normally lands in a handler that will directly execute the bytecode.

Prefix dispatch tables handle the tricky but rare corner cases. If the first byte of an instruction is a prefix, then the target address in the main table will be a *special handler* that loads the next byte in the instruction stream and dispatches through the appropriate table. There is still one more wrinkle, however. Wasm is unusual in that the opcode after a prefix can be a variable-length LEB [16], where the uppermost bit of the byte indicates continuation bytes follow. Thus, in prefix dispatch tables, entries for the upper 128 opcodes (i.e. where the upper bit is 1) point to another special handler that fully decodes the LEB and finally dispatches to an actual bytecode handler, using yet another dispatch table. Of current Wasm bytecodes, only the SIMD opcodes occupy the upper part of their prefix space (**0xFD**). Thus, in practice, Wasm opcodes normally require just one dispatch, two if prefixed, and maximum three if the LEB opcode is longer than 1 byte.

3.3.2 Direct vs threaded dispatch. Each bytecode handler consists of machine instructions that manipulate the value stack or module instance or perform control flow. A handler usually simply leaves the interpreter registers ready to start the next bytecode pointed to by **IP**, except for
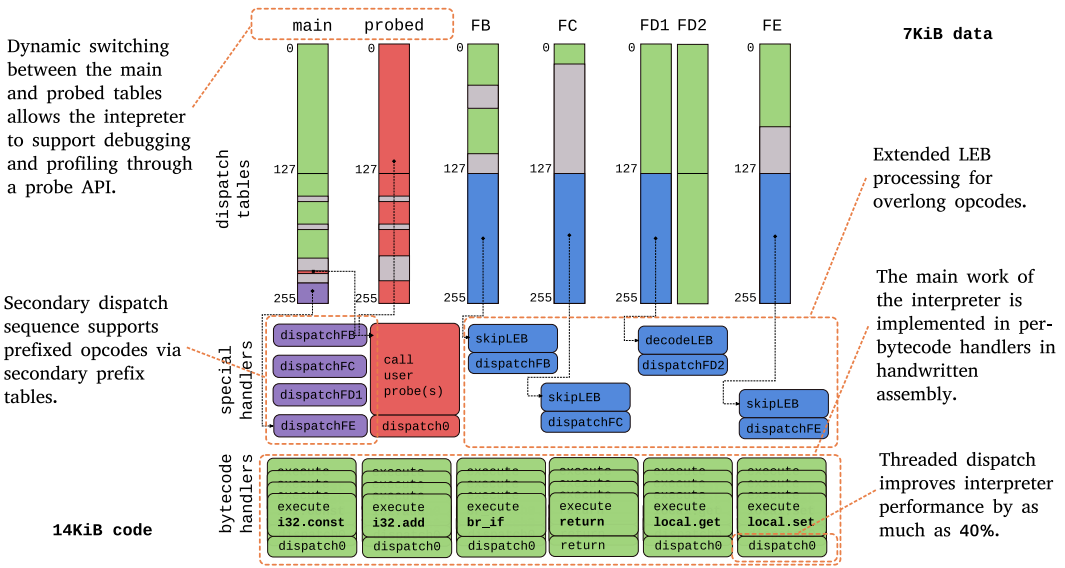
Fig. 7. Dispatch tables and handlers in the fast interpreter.

instructions like **unreachable** and **return**, which will terminate execution or return from the current interpreter frame, respectively.

Conceptually an interpreter has a single loop that repeatedly dispatches one instruction at a time, each handler jumping back to the loop header. Nowadays, a technique known as *threaded dispatch* [58] is often used, where instead of a jump, a copy of the dispatch code is inlined at the end of each handler. This is difficult to do in a high-level language[13], but easy in assembly. This saves a jump instruction and typically makes better use of the CPU's indirect branch predictor. We implemented both and report the performance difference in Section 4.

3.3.3 Debugging and instrumenting with probes. One of the motivating advantages for an interpreter tier in any virtual machine is ease of debugging and instrumentation. In particular, an interpreter implements an exact bytecode-by-bytecode emulation of the machine, offering the possibility of stopping before or after any bytecode and inspecting the virtual machine state. This is key to support both high and low-level debugging of a program as well as instrumentation such as profiling.

Our fast interpreter design has provisions for general instrumentation at the bytecode level. It offers the ability to insert *probes* [61] at either *local* locations in a program, or *global*, the interpreter loop itself. Both have different use cases. A probe is simply a callback to engine-level code that *fires* when the given bytecode is executed, or each time the interpreter loop is executed. In a probe callback, one can inspect virtual machine state through an engine API and then indicate if the

---

[13]Compilers generally will not transform a straight-forward loop-over-switch into threaded dispatch, as the necessary transformation, tail-duplicating the loop header for many hundreds of cases, is highly specific. Instead, we must arduously fill out manual dispatch tables and either use tail calls or, in C, the non-standard **gcc** "labels as values" [15] extension. The resulting control flow is complex, with multiple irreducible loops and hundreds of indirect branch targets, and may lead to spills across important instructions. To use tail calls, we must rewrite the entire interpreter as individual handler functions that end with a tail-call to the next handler (as is done in Ignition [14]), passing the entire interpreter state forward as arguments. Yet only some C compilers support a non-standard tail-call optimization. In any case, without a custom calling convention, we run out of registers and spill some interpreter state on the stack unnecessarily.

program should resume normally or do something else (typically, terminate). Probes are primitives from which debugging support (e.g. breakpoints), tracing support (e.g. logging), or profiling support (e.g. counters), are built.

Figure 7 shows how the fast interpreter supports probes.

- **Local probes**. For a probe inserted *at a particular instruction*, the original bytecode of the function containing the location is copied and the bytecode is overwritten with a special, normally-illegal bytecode, **PROBE**. Since illegal bytecodes will be rejected by the code validator, they will never appear in valid programs. Dispatching on **PROBE** will land in a special handler that looks up the user's callback associated with this bytecode, calls it, and then after, loads the *original* bytecode and dispatches through the **main** table.
- **Global probes**. For a probe inserted *into the global interpreter loop*, the interpreter switches *modes*, using the **probe** dispatch table for every instruction. Similar to the *local* probes, the interpreter looks up the global instrumentation, calls it, and then after, dispatches through the **main** table.

It's important to note that this design allows probes to be inserted and removed dynamically. This allows maximum flexibility to instrumentation code while allowing the interpreter to run at full speed otherwise. For example, suppose a user wants to trace the execution of just one particular function in a module. They could insert global instrumentation into the main interpreter loop and filter out all callbacks where the function of interest is not on the top of the stack. But this is inefficient; the interpreter will go through the **probed** table every time, issue calls into the runtime, into the user code, which inspects state, etc. A more efficient way is to insert a *local* probe into the interesting function. When the local probe is fired, it dynamically inserts the global probe, thereby getting called for every subsequent bytecode. When the function calls another, or when it returns, the global probe can be disabled, and everything goes back to full speed. The interpreter is careful to switch back to the **main** dispatch table whenever it detects global instrumentation is disabled, so it will always run fast when it has opportunity to do so.

3.4 Tuning

Considerable work has gone into designing efficient data structures and dispatch tables for the fast interpreter presented here. We'd be remiss in not enumerating the many other tuning strategies applied here and what we've learned. This was made easier by the fact that the fast interpreter presented here was implemented not in textual assembly language, but in a macro assembler framework we built in a high-level language. Thus configuration options were easy to introduce into the code that generates the interpreter, rather than relying on macro facilities in a textual assembly language.

- **Manual register allocation**. The fast interpreter state consists of 9 registers (Figure 6). All of today's 64 bit architectures have enough architectural registers that it is simply a matter of assigning interpreter registers to architectural registers. For **x86-64**, we chose register assignments carefully to eliminate **REX** prefixes for the most commonly-occurring registers. We did not measure alternative assignments, but suspect that CPU register renaming makes additional tuning moot.
- **Minimal dispatch sequence**. In addition to the choice between threaded and non-threaded dispatch, we experimented with dispatch table designs where entries were 2, 4, and 8 bytes. In the 2 byte design, entries are not direct addresses, but deltas that are applied to the start of a code region, requiring an additional **add** instruction in the dispatch sequence. In the 4 and 8 byte alternatives, the entries are direct addresses, constrained to be in the lower 4gb of address space in the former case. Of these alternatives, the **4 byte sequence is fastest**, often

as much as 10% faster. Unsurprisingly, we found **threaded dispatch is fastest**, on average 14%, maximum 29% faster.

- **Value tagging**. Our interpreter design uses value tags to find GC roots when necessary. We evaluated the alternative and found that **eliminating tags improves performance**, on average 8%, maximum 15% faster.
- **Really fat values**. SIMD values are 128 bits wide. When coupled with value tagging, values occupying 32 bytes of memory each in the value stack.
- **Inline/out-of-line LEB decoding**. The interpreter must dynamically decode the many immediates in Wasm that are encoded as variable-length LEBs, e.g. the index of a `local.get`. Often these variable-length immediates are just a single byte. We experimented with moving the uncommon case out-of-line and concluded that **out-of-line LEB slow cases** could be as much as 5% faster.
- **Memory #0 base pointer**. Wasm programs access memory very frequently. As described earlier, our implementation caches a pointer to the base of the (first) Wasm memory in an architectural register. We did not evaluate the performance impact of this optimization.
- **Handler alignment**. Bytecode handlers are short but critically important sequences of code. We suspected they may be subject to microarchitectural effects of instruction and trace caches arising from code alignment. We experimented with 1, 2, 4, 8, 16, and 32 byte alignment of handlers but **detected no statistically significant performance variation.**
- **Handler order**. We suspect that handler code order may introduce significant microarchitectural effects [45] [26]. However, we did not study the effect of handler order beyond simply emitting common bytecode handlers first.
- **Error case sharing**. Several Wasm bytecodes *trap* on error cases, like divide by zero, unrepresentable floats, etc. We factored the error handling paths in order to save space. Since traps usually terminate the program, the performance does not matter.
- **Call/entry/exit sharing**. We exploit commonalities among call bytecodes (`call`, `call_indirect`, and tail calls[14]), `return`/fall-through, as well as branches in order to save code space. This may have a small effect on performance, but we did not measure this.
- **Handler sharing**. We exploit the fact that some instructions end up with identical handler code and share the handlers (e.g. `block` and `loop`, some memory stores). We did not measure alternatives.

The above conclusions are supported by a performance evaluation of alternatives that is beyond the scope of this paper. To summarize those experiments, the overall difference between the best (tuned) and worst (untuned) interpreter performance is 20% to 60% across the benchmark suite. Interestingly, as we will see in the next section, this difference is enough to make our interpreter design meet and exceed existing (re-writing) interpreters in comparative performance tests.

4 EVALUATION

In this section, we evaluate our Wasm interpreter against many state-of-the-art Wasm engines. The goal is to assess our claim that an in-place interpreter supplies the missing point in the execution tier tradeoff space between translation time, space overhead, and execution time. In particular, an in-place interpreter should offer superior translation time and space overhead compared to other tiers. Ideally, such an interpreter should also have comparable execution time to existing interpreter tiers. Of course we expect that interpreters *should be* handily outclassed by JIT compilers for long-running programs, so we shouldn't expect to *replace* them. Our experiments quantify the tradeoff space empirically.

---

[14]From the `tail-call` Wasm proposal.

4.0.1   Wasm Engines in 2022. Today, five years after support was announced in 4 major browsers, engines are significantly different, and many new competitors have appeared. In particular, Web engines have evolved significantly from what has been reported in the literature to date. Today, all Web engines are sophisticated multi-tier systems.

- **V8** [11] - two compiler tiers. V8 eagerly compiles the entirety of a module with Liftoff [3], a baseline compiler, with many parallel threads. Upon completing baseline compilation, a module is ready to run. Optimized compilation with TurboFan [4], the optimizing compiler shared with JavaScript, is begun in parallel in the background. Optimized code gradually replaces baseline code as it is finished, until all functions in a module are fully optimized. Either compiler can be disabled with command-line flags.
- **SpiderMonkey** [10] - two compiler tiers. Similar to V8, with a baseline compiler first compiling a whole module and then an optimizing compiler in the background, patching in optimized code as it is completed. Either compiler can be disabled with flags.
- **JavaScriptCore** (**JSC**) [9] - one interpreter and two compiler tiers. Wasm modules are not eagerly compiled. Instead, individual functions are lazily translated to interpreter bytecode for LLint [6]. Dynamic counters tier-up hot functions from interpretation to **BBQ**, a fast compiler that has a minimal IR, and then to **OMG**, a fully optimizing compiler based on B3 [53]. Either compiler can be disabled with flags.
- **ChakraCore** [44] - one interpreter tier and one compiler tier. Now largely unsupported, Wasm modules were not eagerly compiled. Instead, individual functions were lazily translated to interpreter bytecode. Dynamic counters tier-up hot functions from interpretation to a fully optimizing compiler.

In addition to the rapid evolution of Web engines, a number of non-Web Wasm engines have appeared.

- **wasmtime** [13] - A standalone runtime implemented in Rust.
  One compiler tier. The optimizing Cranelift compiler can be used in JIT or AOT modes.
- **WAVM** [5] - A standalone runtime implemented using LLVM.
  One compiler tier. Consists primarily of a Wasm loader and the LLVM compiler backend and can only be used in AOT mode.
- **wamr** [17] - (WebAssembly Micro Runtime) a lightweight standalone runtime.
  Two interpreter tiers and one compiler tier, but only one at a time. The "classic" interpreter, we discovered, is an in-place interpreter that does not use a side table, but a control flow cache. The "fast" interpreter is a rewriting interpreter that reuses most of the standard Wasm bytecodes but rewrites control flow. Both interpreter loops are written in C and use **gcc** extensions for threaded dispatch, if possible. A one-pass JIT can be used instead.
- **wasm3** [8] - The self-proclaimed fastest WebAssembly interpreter[15].
  One interpreter tier. All design considerations emphasize interpreter performance. It is implemented using tail calls and relies on **gcc** tail-call optimization. The internal bytecode format consists of a list of function pointers to handlers, plus immediates, i.e. classic threaded code.
- **wasmer** [12] - A standalone runtime for lightweight containers.
  Three compiler tiers, one at a time. Packages two compilers from other projects: Cranelift (from **wasmtime**) or LLVM, and offers its own one-pass compiler.

4.0.2   Our Tier Choices. We chose only to compare against execution tiers that perform *dynamic* translation, intentionally omitting those engines performing static (AOT) translation. Further,

---

[15]It is; we confirm their measurements in our experiments. Hey, but speed isn't everything.

though this paper focuses on interpreter design, we include several experiments that compare engines with multiple tiers to understand their current tradeoffs. We therefore measure these execution tiering configuration:

- **wizard**- Our engine with its in-place interpreter that uses a sidetable.
- **wamr-classic** - The **WAMR** engine with its "classic" in-place interpreter.
- **wamr-fast** - The **WAMR** engine with its (now default) rewriting interpreter.
- **wasm3** - Default configuration of a rewriting interpreter.
- **v8-liftoff** - **V8** with only the Liftoff baseline JIT.
- **v8-turbofan** - **V8** with only the TurboFan optimizing JIT.
- **sm-base** - **Spidermonkey** with only the baseline compiler.
- **sm-opt** - **Spidermonkey** with only the optimizing compiler.
- **jsc-int** - **JSC** with JITs disabled, i.e. only the rewriting interpreter.
- **jsc-bbq** - **JSC** with the rewriting interpreter and **BBQ** quick JIT only.
- **jsc-omg** - **JSC** with the rewriting interpreter and **OMG** optimizing JIT only.

4.1 Benchmark setup

All tests were performed on a Linux 4.15 kernel on a machine with 32GiB of RAM and one Intel Core i7-8700K CPU @ 3.7GHz and the CPU governor set to "performance". Benchmarks used are the PolyBenchC-4.2.1 suite, with the MEDIUM dataset. We used V8 version `10.2.0`, JSC version (roughly 2022-02-01), and Spidermonkey version `C101.0a1`, all release builds built from source. Data was collected in two experiments; 100 uninstrumented runs of 10 engine configurations on the 24 benchmarks to gather execution time, and 100 instrumented runs of the same to collect translation time and space statistics. Every run represents a separate OS process. Execution times are of the complete process (i.e. not internally timed), while translation times are the sum over all Wasm functions translated in the run. All timing results are the average over the 100 runs, and when error bars are shown, they represent the 5th and 95th percentiles of the distribution.

4.2 Translation Time

All of our chosen execution tiers, except **wizard** and **wamr-classic**, the only other in-place interpreter of which we are aware, apply some form of translation to the input bytecode. Rewriters either generate an internal bytecode (interpreters) or machine code (compilers). Thus we measure the time taken for the respective translation by instrumenting the source code of each engine by adding time and space measurements. In the case of **wamr-fast**, the custom bytecode is generated as a side-effect of validation, so we measure the *additional* time for translation by subtracting the baseline validation time obtained from **wasmr-classic**, which has no translation time. **wizard** doesn't *translate*, but instead the reported time is the sidetable tracking and construction time, measured as the difference between validation time with and without sidetable tracking.

We plot the average translation time, divided by the number of input bytes translated. The ratio of translation time to input bytes normalizes differences in tiering strategy (e.g. lazy compilation) and benchmark size. Figure 8 gives the results (note Y-axis is logarithmic).

Here we focus on configurations that isolate individual tiers, rather than multi-tier adaptive configurations. The experimental results show a dramatic difference in translation time for these tiers, nearly 3 orders of magnitude. Though baseline compilers often differ from optimizing compilers by more than 10×, they are still 10× more expensive than rewriting interpreters. Yet there is overlap, as the rewrite time of **jsc-int**, an interpreter, is almost on-par with **v8-liftoff**, a baseline compiler. Similarly, the **jsc-bbq** quick compiler is closer to an optimizing compiler, as it uses an IR, unlike
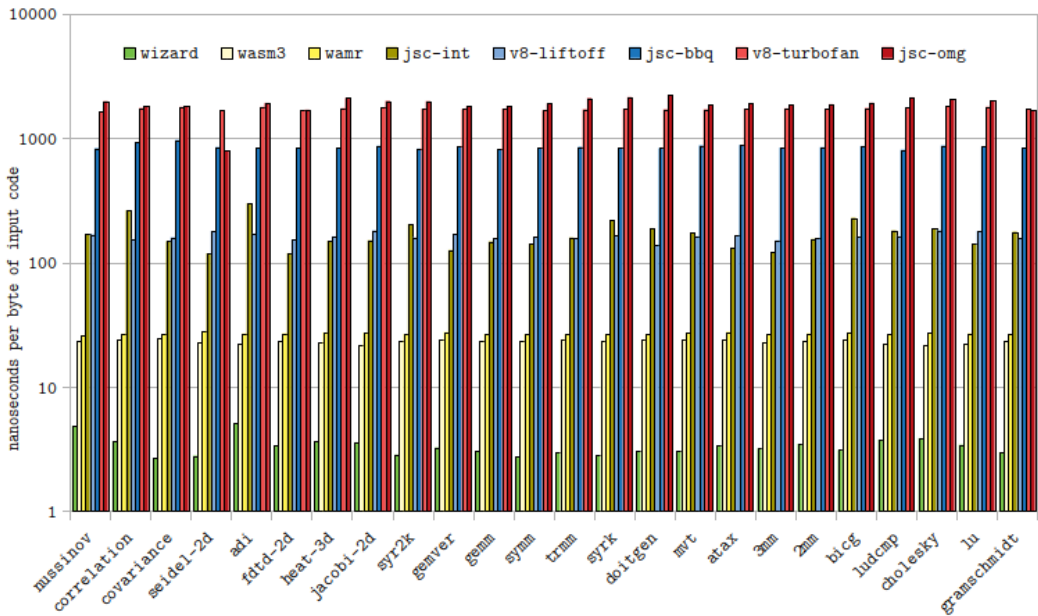
Fig. 8. Translation time normalized to bytes of input code.

v8-liftoff. The winner is clearly our in-place interpreter design, as the sidetable generation in **wizard** is a full order of magnitude cheaper than the cheapest rewriting interpreters.

4.3 Translation Space

Translation not only consumes time, but space. We measure the memory overhead of translation in terms of bytes generated per input byte of code. Here again, the ratio of output bytes to input bytes normalizes differences in tiering strategy (e.g. lazily compilation) and benchmark size. Note that we only measure the size of generated code (whether internal bytecode or machine code) and not additional metadata such as code objects or a source position table, which all rewriters need for debugging. We also do not measure per-module space costs. Thus this experiment *underestimates* space costs of rewriters when debugging is involved. Figure 9 gives the results.

There are several somewhat surprising results here. We find that some rewriting interpreters (such as **wamr**) can consume up to 4× as much space as the original bytecode, while others (such as **jsc-int**) consume about the same amount of space as the original bytecode. We believe this is because **jsc-int** uses an internal bytecode similar to JSC's JavaScript bytecode, which has been heavily tuned to reduce memory consumption on webpages. Also somewhat surprising is that JIT compilers, which generate machine code, do not necessarily consume more code space, in general, than rewriting interpreters, though **jsc-bbq** is an outlier. We discovered that **wasm3**, like **wamr**, often trades space for time–nearly all of its bytecode quantities are word-sized. Yet during rewriting, **wasm3** does a number of peephole-like optimizations, globally canonicalizes constants, and uses a register machine internally, all of which save space.

These measurements show the sidetable in **wizard** takes far less space, only about 30% additional space compared to the original bytecode, as it only requires entries for control-flow[16], and many

---

[16]Note that here, we measure **wizard** sidetable entries compressed to 2 bytes where possible, though this is not the default.
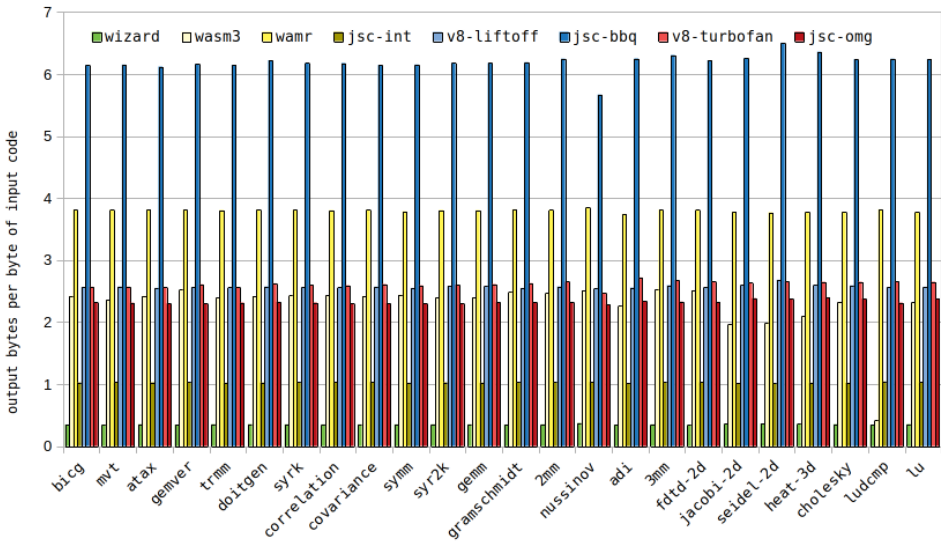
Fig. 9. Translation output bytes normalized to bytes of input code.

sidetables are empty. That is a full order of magnitude more space-efficient than the other tiers, which typically cost 2× to 4× the original bytecode's space.

4.4 Execution time

Figure 10 gives the absolute execution times of the benchmarks for the **v8-turbofan** and **wasm3** execution tiers. Execution times reported in other figures in this section are normalized to either of these two baselines.

4.4.1 Interpretation vs JIT compilation. Interpreters start up faster, but have slower execution time compared to JIT compilers. But how much? We measured the execution time of the benchmarks in the six single-JIT tier configurations and the *fastest* interpreter that we measured, **wasm3**. Since the execution time of benchmarks varies by two orders of magnitude, we normalize execution times for each benchmark to the corresponding execution time on **v8-turbofan**. Figure 11 gives the results.

On the horizontal axis, we sorted the benchmarks by their execution time on **wasm3** (same as in the table). As we can see, the shortest-running benchmarks on the left side of the graph do not run long enough to benefit from the work spent by the optimizing compiler, and all baseline compilers are faster, with the interpreter fastest. There is also a fixed cost of starting a relatively heavyweight JSVM, which contributes to the interpreter being fastest for the shortest 4 benchmarks. Moving to the right, as execution time increases with longer-running benchmarks, the fixed cost of startup and the

benchmark	v8-turbofan	wasm3	benchmark	v8-turbofan	wasm3	benchmark	v8-turbofan	wasm3
bicg	0.016571	0.007197	covariance	0.023764	0.072963	3mm	0.037620	0.203351
mvt	0.016617	0.007252	symm	0.023545	0.087849	fdtd-2d	0.029838	0.204593
atax	0.016593	0.007344	syr2k	0.024594	0.106093	jacobi-2d	0.031257	0.244652
gemver	0.016756	0.009553	gemm	0.024285	0.105961	seidel-2d	0.153029	0.349776
trmm	0.020411	0.042696	gramschmidt	0.028422	0.117820	heat-3d	0.038734	0.363499
doitgen	0.021046	0.054735	2mm	0.029670	0.122010	cholesky	0.078381	0.773620
syrk	0.020491	0.069131	nussinov	0.032324	0.189651	ludcmp	0.088084	0.893945
correlation	0.023820	0.068604	adi	0.071388	0.228267	lu	0.088218	0.876272

Fig. 10. Execution time of benchmarks (in seconds) to which Figures 11&12 are normalized.
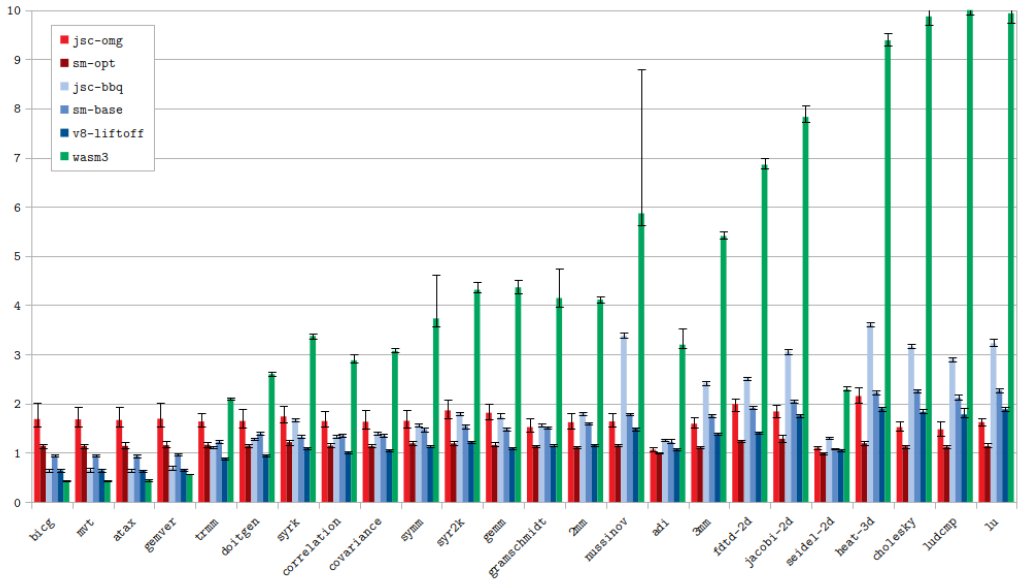
Fig. 11. Execution time (relative to **v8-turbofan**) of JITs versus the fastest Wasm interpreter.

cost of compilation are increasingly amortized. Thus the middle of the graph shows more balanced results; the interpreter falls behind but the baseline compilers, particularly **v8-liftoff**, remain competitive. Continuing on, the gap between optimizing compilers (particularly **v8-turbofan** and **sm-opt**) and the rest increases; execution time is now dominated by code quality. Baseline compilers level off, with **v8-liftoff** and **sm-base** around 2× slower and **jsc-bbq** closer to 3×. The common rule of thumb that interpreters are 10× slower than JITs turns out to be roughly accurate in the end, as there is a clear trend towards roughly 10× for **wasm3** vs **v8-turbofan** here. These results match our intuition, but better, *quantify* it, giving us fairly round numbers to reason with. They also reaffirm the need to have a broad picture of execution time:

- Each of (interpreter, baseline compiler, optimizing compiler) runs some benchmark fastest.

4.4.2 Interpreter showdown. Of course, this paper focuses on making a fast, in-place interpreter. Using similar benchmarking methodology, on the same benchmarks, we evaluated the five interpreter tiers. Figure 12 gives the results.

As we found that **wasm3** was consistently the fastest interpreter across the board, we chose to normalize all execution times in this graph to it. Here we find that **wamr-fast**, the configuration of the **wamr** engine with its rewriting interpreter, is consistently 2nd fastest, nearly always 1.5× to 1.7× slower than **wasm3**, while the others are typically 2×- 3× slower. We attribute this not only to **wasm3**'s threaded code dispatch technique, but its several bytecode-level optimizations. Our design, **wizard**, performs better than the other in-place interpreter design, **wasm-classic** in most situations. It performs nearly equivalent to **wamr-fast** on the 4 shortest-running benchmarks.

4.4.3 Multi-tier configurations in context. Note that our experiments in this section focused on isolating individual execution tiers in order to study their tradeoffs. In production configurations, all web engines run in multi-tier configurations, as described in Section 4.0.1. The space for tiering designs is vast, with many variables, such as laziness, concurrency, thresholds for tiering up, on-disk
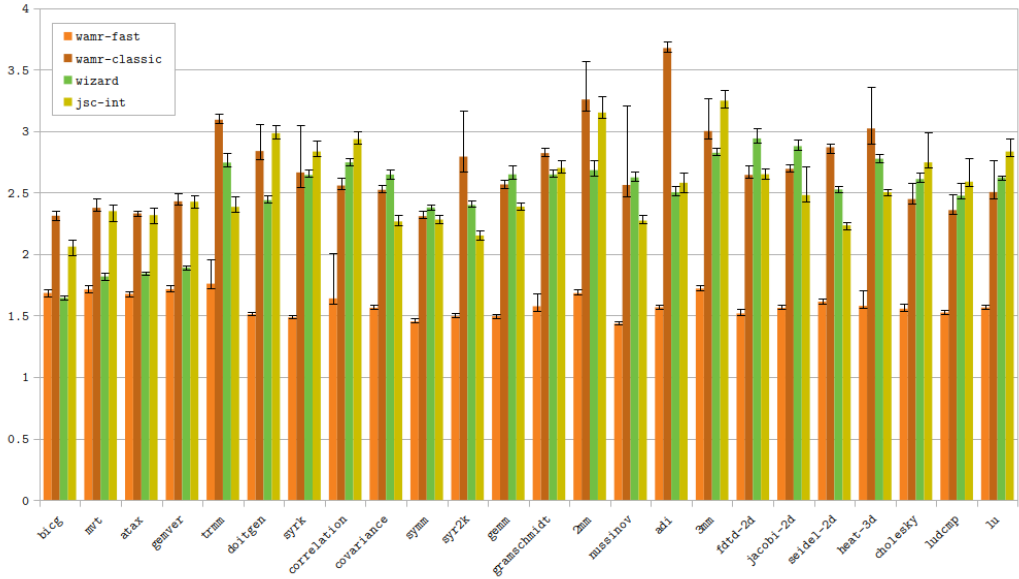
Fig. 12. Execution time (normalized to `wasm3`) of interpreters.

caching, etc. We ran additional experiments for engines in their default configuration, but we found the the results are complex enough to merit study in their own right–a topic that is unfortunately beyond the scope of this paper.

4.4.4 Avoiding pathological behavior. We discovered `wamr-classic` uses an in-place design just days before publication. Since then, we studied its implementation in earnest. It is written in C and uses `gcc` extensions to construct a jump table for threaded dispatch. The jump table is crucial for performance; disabling it via a configuration variable reduces performance by more than 2×, which reaffirms the need for this crucial optimization in interpreter design. Of course, we tested its fastest configuration, as 2× would have put it handily out of the running.

However we discovered that, instead of an $O(1)$ sidetable like our design, it uses both a *dynamic control stack* and a cache of control entries that help it find branch targets during runtime. The cache is a fixed-size, 128-element, 2-way set associative cache indexed by branch address. That amounts to a fixed cost per module, rather than per function (we did not include its space cost in Figure 9). A cache miss results in a slow path where the entire function may be rescanned to the end, repopulating the cache with new entries, including an entry for the current branch. The performance could be pathological for a large program with many branches, a fact obscured by the relatively small benchmark programs in our suite. To verify the pathological behavior, we constructed an adversarial program consisting of thousands of branches, approximating a larger program, and observed slowdowns of as much as 8×. This vulnerability was known to its authors and is one of the reason that `wamr` now ships the rewriting interpreter (`wamr-fast`) by default.

Given the potential pathological behavior of `wamr-classic`, we still believe that `wizard` represents the first viable *fast* in-place design.

5 RELATED WORK

Interpreters are as old as the hills. From the first popular interpreted language, Lisp [43] in 1960, to today's modern scripting and data manipulation languages like Python, Ruby, R, PHP, and MatLab, interpreter performance has been a key subject of interest. Many languages that are now fast through dynamic compilation were once primarily interpreted, such as Smalltalk, Self, Java, C#, OCaml, and JavaScript. Python is still most-often interpreted, and its performance is still of key concern [19] [68]. Because of their advantages, new interpreters continue to appear, even for languages previously compiled [38].

5.0.1   Fixed or flexible format? Research into interpreter techniques either assumes the code format to be *fixed*, such as standardized bytecode formats like the JVM, the CLR, Dalvik, and WebAssembly, where binary programs arrive metaphorically chiselled into stone, or *flexible*, where the format can be changed to suit the needs of a specific language or implementation. Clearly the larger design space of flexible formats affords more techniques, though key lessons hopefully improve the design of future fixed formats. For example, a key question of interest is whether a stack machine (such as the JVM or WebAssembly), or a register machine (such as the CLR or Dalvik bytecode), is inherently more efficient [58] [27]. Though it is too late to change the JVM, CLR, Dalvik, or Wasm, research here informs the next proposed format.

5.0.2   Interpreter dispatch techniques. As our own experimental results reaffirm, the dispatch sequence is critical to interpreter performance. If the format is flexible, e.g. if entirely in-memory, then threaded code [20] or more compact indirect threaded code [29] can be used. A large amount of work has aimed to improve the predictability of indirect branches [56] by exploiting the microarchitectural details of BTBs [25].

5.0.3   Superinstructions. For flexible formats, the number of dispatches can be reduced with superinstructions [54], replacing small opcodes with larger, combined opcodes. This can be done online[31], offline, or a mixture of both. Super-instructions can be formed from combinations of simpler instructions, often in embedded Java VMs [70], or by defining language-specific high-level operations like in CPython and JavaScript VMs.

5.0.4   Radically different interpreter IRs. Bytecode is not the only interpretable format. Abstract syntax trees or other intermediate representations can be interpreted directly in memory. Usually, speed is not the goal, though recently a fast AST-walking interpreter has been described for R [39]. Instead, interpreting IR has other benefits, e.g. proving the correctness of the interpreter, partial evaluation, and collapsing multiple levels of interpretation [18]. Truffle/Graal [64] uses ASTs, for example [47], as a way to express an interpreter for the Futamura [34] projection. Other graph-based IRs have been explored [62]. Some compilers have IR interpreters [2] for testing. A standard compilers course [50] includes interpreters for each IR during translation. None seem to compete with bytecode interpreters in speed.

5.0.5   Optimizing fixed format interpreters. A number of interpreter optimizations can still apply to fixed formats. The most common is duplicating the dispatch sequence at the end of every handler (referred to in this paper and elsewhere, if somewhat imprecisely, as a threaded interpreter or threaded dispatch), and is used in all the interpreters we tested. Threaded dispatch has even been applied to hosted interpreters on the JVM [57]. For stack machines, stack caching [51] [32] [49] VMs try to keep the top-of-stack cached in a register, reducing loads and stores to the value stack. It is possible to further duplicate [55] handlers to get more BTB entries [23]. Recent work has also proposed new hardware support for indirect speculation [69] [63] or to directly address BTB

entries [41]. Many JVMs mutate bytecode in-place [22] to replace symbolic references with indexes and offsets.

5.0.6 Interpreter generators. Writing a highly efficient interpreter remains a black art. It is challenging, sometimes impossible, to convince compilers for high-level languages to emit perfect interpreter code. A number of interpreter generation frameworks have been proposed, including Tiger [24], VMGen [33], and a JavaScript VM generator [40], that automate much of the tedious bookkeeping and broadly apply best practices.

5.0.7 Interplay with JITs and debugging. If the virtual machine is allowed to generate new machine code, then the entire VM design space is unlocked. Context threaded code [67], where code is represented as a sequence of calls to handlers, can be used. From there, selective inlining [52] of handlers can be applied to improve performance. Discussion of JIT designs is beyond the scope of the paper, yet their interactions with interpreters is of note. Trace compilation [35] is fed by collecting execution traces from an interpreter. Meta-tracing [21], i.e. tracing through the handler implementation, has been employed by the PyPy dynamic optimizer. Dynamic adaptive optimization with deoptimization is now standard in many virtual machines. The design of interpreter stack frames determines the cost and complexity of on-stack replacement. In-place interpretation has been shown to ease debugging support, since no mapping need be maintained from rewriting.

6 CONCLUSION

In-place interpretation has long been considered when designing fixed code formats such as the JVM, CLR, and Dalvik, since it is the most memory-efficient. Wasm is unique in that it was explicitly designed with near-native performance as the highest priority and engines shipped with optimizing compiler tiers first. Interpretation was not thought necessary, and in-place interpretation, stymied by structured control flow, was dismissed as either impossible or at least unneeded. Yet in the intervening five years, startup time and memory consumption have increased in priority, and interpretation of Wasm (by rewriting to an internal format) has become more widespread.

This paper restores the missing execution tier for Wasm, regaining the key property of efficient in-place interpretation that was thought lost. We presented the design and implementation of the first fast in-place interpreter for Wasm that utilizes a compact sidetable easily generated as a side-effect of the code validation algorithm. We measured an order of magnitude improvement in memory consumption and processing time over rewriting Wasm. With this open problem now solved, we believe that Wasm engines in the future will employ new interpreter tiers for improved startup time, reduced memory consumption, and improved debugging support.

REFERENCES

[1] 1998. Hotspot internals: interpreter. https://openjdk.java.net/groups/hotspot/docs/RuntimeOverview.html. https://openjdk.java.net/groups/hotspot/docs/RuntimeOverview.html (Accessed 2022-4-07).

[2] 2015. lli - directly execute programs from LLVM bitcode. https://llvm.org/docs/CommandGuide/lli.html. https://llvm.org/docs/CommandGuide/lli.html (Accessed 2022-4-12).

[3] 2018. https://v8.dev/blog/liftoff. https://v8.dev/blog/liftoff (Accessed 2022-4-07).

[4] 2018. TurboFan: V8's Optimizing Compiler. https://v8.dev/docs/turbofan. https://v8.dev/docs/turbofan (Accessed 2021-07-29).

[5] 2018. WAVM: a non-browser WebAssembly virtual machine. https://github.com/WAVM/WAVM. https://github.com/WAVM/WAVM (Accessed 2022-1-10).

[6] 2019. A New Bytecode Format for JavaScriptCore. https://webkit.org/blog/9329/a-new-bytecode-format-for-javascriptcore/. https://webkit.org/blog/9329/a-new-bytecode-format-for-javascriptcore/ (Accessed 2022-04-07).

[7] 2020. The edge of the multi-cloud. https://www.fastly.com/cassets/6pk8mg3yh2ee/79dsHLTEfYIMgUwVVllaa4/5e5330572b8f317f72e16696256d8138/WhitePaper-Multi-Cloud.pdf. https://www.fastly.com/cassets/6pk8mg3yh2ee/79dsHLTEfYIMgUwVVllaa4/5e5330572b8f317f72e16696256d8138/WhitePaper-Multi-Cloud.pdf (Accessed 2021-07-06).

[8] 2020. Wasm3: The fastest WebAssembly interpreter, and the most universal runtime. https://github.com/wasm3/wasm3. https://github.com/wasm3/wasm3 (Accessed 2021-08-11).

[9] 2021. JavaScriptCore, the built-in JavaScript engine for WebKit. https://trac.webkit.org/wiki/JavaScriptCore. https://trac.webkit.org/wiki/JavaScriptCore (Accessed 2021-07-29).

[10] 2021. SpiderMonkey: Mozilla's JavaScript and WebAssembly engine. https://spidermonkey.dev. https://spidermonkey.dev (Accessed 2021-07-29).

[11] 2021. V8 Development Site. https://v8.dev. https://v8.dev (Accessed 2021-07-29).

[12] 2021. Wasmer: A Fast and Secure WebAssembly Runtime. https://github.com/wasmerio/wasmer. https://github.com/wasmerio/wasmer (Accessed 2021-07-06).

[13] 2021. Wasmtime: a standalone runtime for WebAssembly. https://github.com/bytecodealliance/wasmtime. https://github.com/bytecodealliance/wasmtime (Accessed 2021-08-11).

[14] 2022. Ignition: a fast low-level interpreter. https://v8.dev/docs/ignition. https://v8.dev/docs/ignition (Accessed 2022-04-11).

[15] 2022. Labels as Values (GNU Compiler Collection). https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html. https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html (Accessed 2022-04-11).

[16] 2022. LEB128. https://en.wikipedia.org/wiki/LEB128. https://en.wikipedia.org/wiki/LEB128 (Accessed 2022-04-11).

[17] 2022. WebAssembly Micro Runtime (WAMR). https://github.com/bytecodealliance/wasm-micro-runtime. https://github.com/bytecodealliance/wasm-micro-runtime (Accessed 2022-04-11).

[18] Nada Amin and Tiark Rompf. 2017. Collapsing Towers of Interpreters. *Proc. ACM Program. Lang.* 2, POPL, Article 52 (dec 2017), 33 pages. https://doi.org/10.1145/3158140

[19] Gergö Barany. 2014. Python Interpreter Performance Deconstructed. In *Proceedings of the Workshop on Dynamic Languages and Applications* (Edinburgh, United Kingdom) *(Dyla'14)*. Association for Computing Machinery, New York, NY, USA, 1–9. https://doi.org/10.1145/2617548.2617552

[20] James R. Bell. 1973. Threaded Code. *Commun. ACM* 16, 6 (jun 1973), 370–372. https://doi.org/10.1145/362248.362270

[21] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems* (Genova, Italy) *(ICOOOLPS '09)*. Association for Computing Machinery, New York, NY, USA, 18–25. https://doi.org/10.1145/1565824.1565827

[22] Stefan Brunthaler. 2010. Efficient Interpretation Using Quickening. In *Proceedings of the 6th Symposium on Dynamic Languages* (Reno/Tahoe, Nevada, USA) *(DLS '10)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/1869631.1869633

[23] Kevin Casey, M. Anton Ertl, and David Gregg. 2007. Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters. *ACM Trans. Program. Lang. Syst.* 29, 6 (oct 2007), 37–es. https://doi.org/10.1145/1286821.1286828

[24] Kevin Casey, David Gregg, and M. Anton Ertl. 2005. Tiger – an Interpreter Generation Tool. In *Proceedings of the 14th International Conference on Compiler Construction* (Edinburgh, UK) *(CC'05)*. Springer-Verlag, Berlin, Heidelberg, 246–249. https://doi.org/10.1007/978-3-540-31985-6_18

[25] Po-Yung Chang, Eric Hao, and Yale N Patt. 1997. Target prediction for indirect jumps. *ACM SIGARCH Computer Architecture News* 25, 2 (1997), 274–283.

[26] Charlie Curtsinger and Emery D. Berger. 2013. STABILIZER: Statistically Sound Performance Evaluation. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 219–228. https://doi.org/10.1145/2451116.2451141

[27] Brian Davis, Andrew Beatty, Kevin Casey, David Gregg, and John Waldron. 2003. The Case for Virtual Register Machines. In *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators* (San Diego, California) *(IVME '03)*. Association for Computing Machinery, New York, NY, USA, 41–49. https://doi.org/10.1145/858570.858575

[28] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. 1989. Combining Generational and Conservative Garbage Collection: Framework and Implementations. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '90)*. Association for Computing Machinery, New York, NY, USA, 261–269. https://doi.org/10.1145/96709.96735

[29] Robert B. K. Dewar. 1975. Indirect Threaded Code. *Commun. ACM* 18, 6 (jun 1975), 330–331. https://doi.org/10.1145/360825.360849

[30] Amer Diwan, Eliot Moss, and Richard Hudson. 1992. Compiler Support for Garbage Collection in a Statically Typed Language. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation* (San Francisco, California, USA) *(PLDI '92)*. Association for Computing Machinery, New York, NY, USA, 273–282. https://doi.org/10.1145/143095.143140

[31] M. Anton Ertl and David Gregg. 2004. Combining Stack Caching with Dynamic Superinstructions. In *Proceedings of the 2004 Workshop on Interpreters, Virtual Machines and Emulators* (Washington, D.C.) *(IVME '04)*. Association for Computing Machinery, New York, NY, USA, 7–14. https://doi.org/10.1145/1059579.1059583

[32] M. Anton Ertl and David Gregg. 2004. Combining Stack Caching with Dynamic Superinstructions. In *Proceedings of the 2004 Workshop on Interpreters, Virtual Machines and Emulators* (Washington, D.C.) *(IVME '04)*. Association for Computing Machinery, New York, NY, USA, 7–14. https://doi.org/10.1145/1059579.1059583

[33] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. 2002. Vmgen: A Generator of Efficient Virtual Machine Interpreters. *Softw. Pract. Exper.* 32, 3 (mar 2002), 265–294. https://doi.org/10.1002/spe.434

[34] Y. Futamura. 1983. Partial computation of programs. *Lecture Notes in Computer Science* 147 (1983).

[35] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-Based Just-in-Time Type Specialization for Dynamic Languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 465–478. https://doi.org/10.1145/1542476.1542528

[36] Robbert Gurdeep Singh and Christophe Scholliers. 2019. WARDuino: A Dynamic WebAssembly Virtual Machine for Programming Microcontrollers. 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, 27–36. https://doi.org/10.1145/3357390.3361029

[37] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 185–200. https://doi.org/10.1145/3062341.3062363

[38] Xiaowen Hu, David Zhao, Herbert Jordan, and Bernhard Scholz. 2021. An Efficient Interpreter for Datalog by De-Specializing Relations *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 681–695. https://doi.org/10.1145/3453483.3454070

[39] Tomas Kalibera, Petr Maj, Floreal Morandat, and Jan Vitek. 2014. A Fast Abstract Syntax Tree Interpreter for R. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Salt Lake City, Utah, USA) *(VEE '14)*. Association for Computing Machinery, New York, NY, USA, 89–102. https://doi.org/10.1145/2576195.2576205

[40] Takafumi Kataoka, Tomoharu Ugawa, and Hideya Iwasaki. 2018. A Framework for Constructing Javascript Virtual Machines with Customized Datatype Representations. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing* (Pau, France) *(SAC '18)*. Association for Computing Machinery, New York, NY, USA, 1238–1247. https://doi.org/10.1145/3167132.3167266

[41] Channoh Kim, Sungmin Kim, Hyeon Gyu Cho, Dooyoung Kim, Jaehyeok Kim, Young H. Oh, Hakbeom Jang, and Jae W. Lee. 2016. Short-Circuit Dispatch: Accelerating Virtual Machine Interpreters on Embedded Processors. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 291–303. https://doi.org/10.1109/ISCA.2016.34

[42] Borui Li, Hongchang Fan, Yi Gao, and Wei Dong. 2021. ThingSpire OS: A WebAssembly-Based IoT Operating System for Cloud-Edge Integration. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services* (Virtual Event, Wisconsin) *(MobiSys '21)*. Association for Computing Machinery, New York, NY, USA, 487–488. https://doi.org/10.1145/3458864.3466910

[43] John McCarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM* 3, 4 (apr 1960), 184–195. https://doi.org/10.1145/367177.367199

[44] Microsoft. 2021. ChakraCore: a JavaScript engine with a C API. https://github.com/chakra-core/ChakraCore. https://github.com/chakra-core/ChakraCore (Accessed 2021-07-29).

[45] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. Producing Wrong Data without Doing Anything Obviously Wrong!. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS XIV)*. Association for Computing Machinery, New York, NY, USA, 265–276. https://doi.org/10.1145/1508244.1508275

[46] Manuel Nieke, Lennart Almstedt, and Rüdiger Kapitza. 2021. Edgedancer: Secure Mobile WebAssembly Services on the Edge. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking* (Online, United Kingdom) *(EdgeSys '21)*. Association for Computing Machinery, New York, NY, USA, 13–18. https://doi.org/10.1145/3434770.3459731

[47] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. 2018. GraalSqueak: A Fast Smalltalk Bytecode Interpreter Written in an AST Interpreter Framework. In *Proceedings of the 13th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems* (Amsterdam, Netherlands) *(ICOOOLPS '18)*. Association for Computing Machinery, New York, NY, USA, 30–35. https://doi.org/10.1145/3242947.3242948

[48] Mohammed Nurul-Hoque and Khaled A. Harras. 2021. Nomad: Cross-Platform Computational Offloading and Migration in Femtoclouds Using WebAssembly. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*. 168–178. https://doi.org/10.1109/IC2E52221.2021.00032

[49] Kazunori Ogata, Hideaki Komatsu, and Toshio Nakatani. 2002. Bytecode Fetch Optimization for a Java Interpreter. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California) *(ASPLOS X)*. Association for Computing Machinery, New York, NY, USA, 58–67. https://doi.org/10.1145/605397.605404

[50] Jens Palsberg. 2014. From Java to Mips in Four Nifty Steps. http://web.cs.ucla.edu/classes/spring11/cs132/kannan/index.html. http://web.cs.ucla.edu/classes/spring11/cs132/kannan/index.html (Accessed 2022-4-07).

[51] Jinzhan Peng, Gansha Wu, and Guei-Yuan Lueh. 2004. Code Sharing among States for Stack-Caching Interpreter. In *Proceedings of the 2004 Workshop on Interpreters, Virtual Machines and Emulators* (Washington, D.C.) *(IVME '04)*. Association for Computing Machinery, New York, NY, USA, 15–22. https://doi.org/10.1145/1059579.1059584

[52] Ian Piumarta and Fabio Riccardi. 1998. Optimizing Direct Threaded Code by Selective Inlining. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) *(PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 291–300. https://doi.org/10.1145/277650.277743

[53] Filip Pizlo. 2016. Introducing the B3 JIT Compiler. https://webkit.org/blog/5852/introducing-the-b3-jit-compiler/. https://webkit.org/blog/5852/introducing-the-b3-jit-compiler/ (Accessed 2022-4-12).

[54] Todd A. Proebsting. 1995. Optimizing an ANSI C Interpreter with Superoperators. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 322–332. https://doi.org/10.1145/199448.199526

[55] Gregory B. Prokopski and Clark Verbrugge. 2008. Analyzing the Performance of Code-Copying Virtual Machines. *SIGPLAN Not.* 43, 10 (oct 2008), 403–422. https://doi.org/10.1145/1449955.1449796

[56] Erven Rohou, Bharath Narasimha Swamy, and André Seznec. 2015. Branch prediction and the performance of interpreters — Don't trust folklore. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 103–114. https://doi.org/10.1109/CGO.2015.7054191

[57] Gülfem Savrun-Yeniçeri, Wei Zhang, Huahan Zhang, Eric Seckler, Chen Li, Stefan Brunthaler, Per Larsen, and Michael Franz. 2014. Efficient Hosted Interpreters on the JVM. *ACM Trans. Archit. Code Optim.* 11, 1, Article 9 (feb 2014), 24 pages. https://doi.org/10.1145/2532642

[58] Yunhe Shi, David Gregg, Andrew Beatty, and M. Anton Ertl. 2005. Virtual Machine Showdown: Stack versus Registers. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments* (Chicago, IL, USA) *(VEE '05)*. Association for Computing Machinery, New York, NY, USA, 153–163. https://doi.org/10.1145/1064979.1065001

[59] Ben L. Titzer. 2013. Harmonizing Classes, Functions, Tuples, and Type Parameters in Virgil III. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 85–94. https://doi.org/10.1145/2491956.2491962

[60] Ben L. Titzer. 2021. Wizard, An advanced WebAssembly Engine for Research. https://github.com/titzer/wizard-engine. https://github.com/titzer/wizard-engine (Accessed 2021-07-29).

[61] Ben L. Titzer and Jens Palsberg. 2005. Nonintrusive Precision Instrumentation of Microcontroller Software. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (Chicago, Illinois, USA) *(LCTES '05)*. Association for Computing Machinery, New York, NY, USA, 59–68. https://doi.org/10.1145/1065910.1065919

[62] Kevin Williams, Jason McCandless, and David Gregg. 2010. Dynamic Interpretation for Dynamic Scripting Languages. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Toronto, Ontario, Canada) *(CGO '10)*. Association for Computing Machinery, New York, NY, USA, 278–287. https://doi.org/10.

1145/1772954.1772993

[63] Kevin Williams, Albert Noll, Andreas Gal, and David Gregg. 2008. Optimization Strategies for a Java Virtual Machine Interpreter on the Cell Broadband Engine. In *Proceedings of the 5th Conference on Computing Frontiers* (Ischia, Italy) *(CF '08)*. Association for Computing Machinery, New York, NY, USA, 189–198. https://doi.org/10.1145/1366230.1366265

[64] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-Optimizing AST Interpreters. *SIGPLAN Not.* 48, 2 (oct 2012), 73–82. https://doi.org/10.1145/2480360.2384587

[65] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2010. Native Client: A Sandbox for Portable, Untrusted X86 Native Code. *Commun. ACM* 53, 1 (jan 2010), 91–99. https://doi.org/10.1145/1629175.1629203

[66] Alon Zakai. 2013. asm.js: an extraordinarily optimizable, low-level subset of JavaScript. http://asmjs.org. http://asmjs.org (Accessed 2021-07-29).

[67] Mathew Zaleski, Marc Berndl, and Angela Demke Brown. 2005. Mixed Mode Execution with Context Threading. In *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research* (Toronto, Ontario, Canada) *(CASCON '05)*. IBM Press, 305–319.

[68] Qiang Zhang, Lei Xu, Xiangyu Zhang, and Baowen Xu. 2022. Quantifying the interpretation overhead of Python. *Science of Computer Programming* 215 (2022), 102759. https://doi.org/10.1016/j.scico.2021.102759

[69] Massimiliano Zilli, Wolfgang Raschke, Reinhold Weiss, Johannes Loinig, and Christian Steger. 2015. Hardware/Software Co-Design for a High-Performance Java Card Interpreter in Low-End Embedded Systems. *Microprocess. Microsyst.* 39, 8 (nov 2015), 1076–1086. https://doi.org/10.1016/j.micpro.2015.05.004

[70] Massimiliano Zilli, Wolfgang Raschke, Reinhold Weiss, Christian Steger, and Johannes Loinig. 2015. A Light-Weight Compression Method for Java Card Technology. *SIGBED Rev.* 11, 4 (jan 2015), 13–18. https://doi.org/10.1145/2724942.2724944