Flexible Non-intrusive Dynamic Instrumentation for WebAssembly

Ben L. Titzer
Carnegie Mellon University
btitzer@andrew.cmu.edu

Elizabeth Gilbert
Carnegie Mellon University
evgilber@andrew.cmu.edu

Bradley Wei Jie Teo
Carnegie Mellon University
bradleyt@andrew.cmu.edu

Yash Anand
Carnegie Mellon University
yashanan@andrew.cmu.edu

Kazuyuki Takayama
Carnegie Mellon University
ktakayam@andrew.cmu.edu

Heather Miller
Carnegie Mellon University
hmiller2@andrew.cmu.edu

Abstract

The ability to gain detailed insight into the dynamic execution of programs is a key strength of managed runtimes over hardware. Analyses such as code coverage, execution frequency, tracing, and debugging, are all made easier in a virtual setting. As a portable, native bytecode, WebAssembly offers inexpensive in-process sandboxing with high performance. Yet to date, Wasm engines have not offered much insight into executing programs, supporting at best bytecode-level stepping and basic source maps, but no instrumentation capabilities. In this work, we outline an innovative set of instrumentation primitives in a Wasm research engine that allow building complex dynamic analyses with low overhead. We show a variety of dynamic analyses and show how instrumentation mechanisms form a hierarchy that allows implementing high-level analyses in terms of low-level, programmable probes. We detail a fully-featured implementation of probes in a high-performance multi-tier Wasm engine, show novel optimizations specifically targeted to instrumentation overhead, and evaluate performance characteristics under load from various analyses. This paper shows that flexible dynamic instrumentation for WebAssembly can be supported with low overhead without impacting production performance.

1. Introduction

Programs have bugs and sometimes run slow. Understanding the dynamic behavior of programs is key to debugging, profiling, and optimizing programs so that they execute correctly and efficiently. Program behavior can be extremely complex with millions of interesting events [24] [22] [28] [18] [60]. Clearly manual inspection cannot scale and programmatic observation is required.

1.1. Monitors and `M-code`

Just as we use the common term *application* to refer to a self-contained program, we will use the term *monitor* to refer to a self-contained *analysis* which monitors an application and observes execution events and internal states. For example, a profile monitor might count the number of runtime iterations of a loop or the execution count of basic blocks. A monitor may *instrument* programs using various mechanisms, before or during execution, execute runtime logic during program execution, and generate a post-execution report. Of particular interest is a monitor's additional runtime storage and logic, which we refer to as *monitor data* and *monitor code*. For example, a profile monitor's data includes counters and the monitor code includes the updates and reporting of counters. Monitor code may be written in a high-level language and compiled to a lower form. We will use `M-code` to refer to the actual monitor code that will be executed at runtime. `M-code` may take many forms, including injected bytecode, source code, or machine code, utilities fixed in the engine itself such as tracing modes, or extensions to the engine written in its implementation language.

While most monitors aim to observe program behavior without changing it, the implementation technique may not always guarantee this. For example, injecting `M-code` by overwriting machine code in a native program is fraught with peril because native programs can observe machine-level details such as reading their own code as data. Some approaches, such as emulation, are inherently *non-intrusive* as they fully virtualize the execution model and `M-code` can operate outside of this virtual execution context.

1.1.1. Intrusive approaches

A monitor is *intrusive* if it alters program behavior in an observable way. Intrusiveness is fundamentally a property of the monitor together with the chosen technique for implementing instrumentation. For example, instrumenting a native program that reads its own code as data could be intrusive if done with code injection, but non-intrusive with an emulator. Relatedly, a monitor can also *perturb* a program's performance characteristics such as execution time and memory consumption without altering program behavior.

Instrumentation implementations that risk intrusiveness include static and dynamic code rewriting techniques.

Static rewriting. If the execution platform does not directly offer debugging and inspection services, *static rewriting* is often used where source code, bytecode, or machine code is injected directly into the program before execution. Static rewriting has its advantages:

- no support from the execution platform is necessary; will work anywhere

- not limited by the instrumentation capabilities of underlying execution platform; can do anything
- inserted `M-code` can be small and inline; approaches minimal overhead
- instrumentation overhead is fixed before runtime; no dynamic instrumentation costs

However, static rewriting can also have disadvantages:

- `M-code` intrudes on the state space of the code under test; easier to break the program
- for machine- and bytecode level instrumentation, `M-code` must necessarily be low-level; more tedious to implement
- offline instrumentation must instrument all possible events of interest; cannot dynamically adapt
- source-level locations and mappings are altered by added code; additional mapping needed
- `M-code` perturbs performance in subtle and potentially complex ways; unpredictable performance impacts
- pervasive instrumentation could massively increase code size; binary bloat
- some information is only dynamically discoverable; could miss libraries, indirect calls, and generated code
- for machine code, the binary may need to be reorganized to fit instrumentation; may not always be possible

Given these properties, this approach is frequently used and is demonstrated in the source-level tool Oron [37], the bytecode-level tools BISM [47] and Wasabi [31], and the machine-code-level tool EEL [30], among others.

Dynamic rewriting. In contrast to static rewriting, *dynamic rewriting* allows a monitor to add `M-code` at runtime. This remedies some of the static rewriting disadvantages:

- can discover information only available at runtime
- can instrument 100% of the code
- does not require recompilation or relinking of binary
- potentially less code bloat
- can dynamically adapt to program behavior, instrumenting more or less
- implementation technique may be able to preserve some of the original addresses

However, it can have its own disadvantages:

- more instrumentation cost is paid at runtime
- may make the execution platform more complicated, e.g. dynamic recompilation
- the monitor is heavily coupled to the framework used to implement dynamic instrumentation

Given these properties, this approach is frequently used and is demonstrated in the source-level tool Jalangi [46], the bytecode-level tool DiSL [35], the machine-code-level tools Dyninst [14], Pin [34] and Dtrace [25], among others (see Section 6).

1.1.2. Non-intrusive approaches

Several techniques exist that do not alter the program code or behavior; their logic runs non-intrusively outside of the program space. Debuggers for native binaries can use hardware-assisted techniques such as debug registers, JTAG, and process tracing APIs to debug programs directly on a CPU. Emulators can support debugging and tracing easily in their interpreters.

Typically non-intrusive native mechanisms are slow, imposing orders of magnitude execution time overhead. Yet profiling branches, method calls, loops, or memory accesses, requires a more high-performance mechanism. For limited cases such as profiling, Linux Perf [10] is a non-intrusive sampling profiler that can analyze programs directly running on the CPU. Valgrind [39] and QEMU [13] are emulators with analysis features and use dynamic binary translation via JIT compilation to reduce overheads.

Managed runtime environments offer high-performance implementations of languages and bytecode. Many offer APIs for observing and interacting with a running program. Support for instrumentation can be standard tracing modes, APIs for bytecode injection, or hot code reload (i.e. swapping the entire code of a function or class at a time). For example, the Java Virtual Machine offers JVMTI [3], and the .NET platform offers the .NET profiling API [11]. JVMTI offers both intrusive (dynamic bytecode rewriting with `java.lang.instrument`) and non-intrusive (`Agents` [1]) mechanisms.

The flexible sensor network simulator Avrora [53], powered by a microcontroller emulator, allows monitors to attach `M-code` to code and memory locations as well as clock events [54]. The `M-code` is written in Java and therefore runs outside of the emulated CPU. Its cycle-accurate interpreter runs microcontroller code faster than realtime on desktop CPUs without the need for a JIT.

Emulator and VM-based approaches still have the disadvantages of dynamic rewriting, but have more advantages:

- non-intrusive instrumentation
- simplifies source-level address mapping
- no need to reorganize binaries
- can reuse existing JIT in engine; no instrumentation-specific JIT
- does not require writing `M-code` in a low-level language
- does not perturb memory consumption

1.2. WebAssembly

WebAssembly [26], or Wasm for short, is a portable, low-level bytecode that serves as a compilation target for many languages, including C/C++, Rust, AssemblyScript, Java, Kotlin, OCaml, and many others. Initially released for the Web, it has since seen uptake in many new contexts such as Cloud [56] and edge computing [40, 1], IoT [32, 33], and embedded and industrial [38] systems. Wasm is gaining momentum as the

[1] In JVMTI, users can write *Agents* in native code that fire when *Events* occur in the running application. They then interface with the program to query state or control the execution itself. JVMTI is a part of the larger Java Platform Debugger Architecture (JPDA) that offers Java-level debugging interfaces and remote (socket-based) debugging interfaces, via the Java Debug Wire Protocol (JDWP).

central sandboxing technology in many new computing platforms, as its execution model naturally separates program state during execution. The format is designed to be load- and run-time efficient with many high-performance implementations. Wasm comes with strong safety guarantees, starting with a strict formal specification [6], a mechanically-proven sound type system [58], and implementations being subjected to verification [16].

Yet to date, no standard APIs for Wasm instrumentation exist, as production Wasm engines have focused on delivering the best performance for their domain. Only intrusive instrumentation techniques exist today. To work around the lack of standard APIs for instrumentation, several offline bytecode rewriting tools have emerged [31] [44].

Wasm engines achieve near-native performance through AOT or JIT compilation. Compilation is greatly simplified (over dynamic binary translation) as Wasm's code units are modules and functions rather than unstructured, arbitrarily-addressable machine code. While Wasm JITs give excellent performance, some engines such as JavaScriptCore [4] and wasm3 [2] employ interpreters either for startup time or memory footprint. Interpreters also help debuggability and introspection; recent work [52] outlined a fast in-place interpreter design in the Wizard Research Engine.

1.3. Our contributions

In this work, we describe the first dynamic, non-intrusive instrumentation framework for WebAssembly and detail its implementation in the Wizard [51] Research engine. Unlike previous work, this framework is inherently non-intrusive and imposes zero overhead when not in use, thus making it suitable for adoption in production engines. We show how to implement efficient support for *probes* in a multi-tier Wasm engine and how to build useful, complex analyses from this basic building block, including tracing, profiling, and debugging. In contrast to Pin [34] and DynamoRIO [17], our work makes only minor additions to the existing execution tiers of the Wasm engine (a few hundred lines of code), rather than a new, purpose-built JIT (tens of thousands of lines of code). We further show novel optimizations that reduce the overhead of common instrumentation tasks and evaluate their effectiveness on a standard suite of benchmarks while placing our system's performance in context with related work.

2. Non-intrusive instrumentation in Wizard

High-performance virtual machines optimize execution time by cheating. JIT compiler optimizations make more efficient use of underlying hardware resources and skip some execution steps of the abstract machine. For example, not every update of a local variable or operand stack value is modeled at runtime, but function-local storage is virtualized and register allocated. Yet monitoring a program for dynamic analysis inherently observes intermediate states of a program, rather than just its final outcome. Dynamic analyses typically observe states of the abstract machine, so VMs that support introspection must materialize the abstract states whenever requested.

Where to instrument programs? Most monitors instrument code to observe the flow of execution or program data. With code instrumentation, locations in the original program code (e.g. bytecode offset, address, line number) become the natural points of reference. This makes it intuitive to use an instrumentation API to attach `M-code` to program locations which will fire when that point is reached during execution.

Monitoring in Wizard with probes. A dynamic analysis for Wizard involves writing a `Monitor` in Virgil [50] against an engine API. With this API, everything about a Wasm program's execution can be observed *on demand*, including any/every bytecode executed, any/every internal state computed, and all interaction with the environment. Monitors observe execution by inserting probes that fire *callbacks* (i.e. `M-code`) before specified events or states occur. Callbacks are dynamic logic but their `M-code` can be statically compiled into the engine. When compiled, their `M-code` is efficient machine code that the engine invokes directly from either the interpreter or JIT-compiled code.

Probes are maximally general. While many systems [12, 61] offer event traces that can be analyzed asynchronously or offline, probes are more fundamental. Probes can *generate* event traces or react to program behavior, but event traces do not offer the ability to influence execution, an inherent capability of synchronous probes.

Figure 1 illustrates the probe hooks offered by Wizard and their implementation in the interpreter.

2.1. Global Probes

The simplest type of probe is a *global probe*, which fires a callback *for every instruction executed by the program*. Global probes are at the bottom of the probe hierarchy because they are *complete*, i.e. can implement any analysis since they can execute arbitrary logic at any point in execution. A global probe is the easiest way to implement tracing, counting, or the `step-instruction` operation of a debugger.

Global probes are easy to implement in an interpreter; its main loop or dispatch sequence simply contains a check for any global probe(s) and calls them at each iteration. They also are the slowest `M-code` because, even with a JIT, they effectively reduce the VM to an interpreter[2].

Despite their inefficiency, global probes are still useful. Unfortunately, the simple implementation technique of an extra check per interpreted instruction imposes overhead even when not enabled, which tempts VMs to have different production and debug builds. A key innovation in Wizard (Section 4) is to implement global probes with *dispatch table switching*, which

[2]E.g. in a compile-only engine, a compilation mode which inserts a call to fire global probes before *every* instruction suffices, but bloats generated code and has marginal performance benefit over an interpreter.

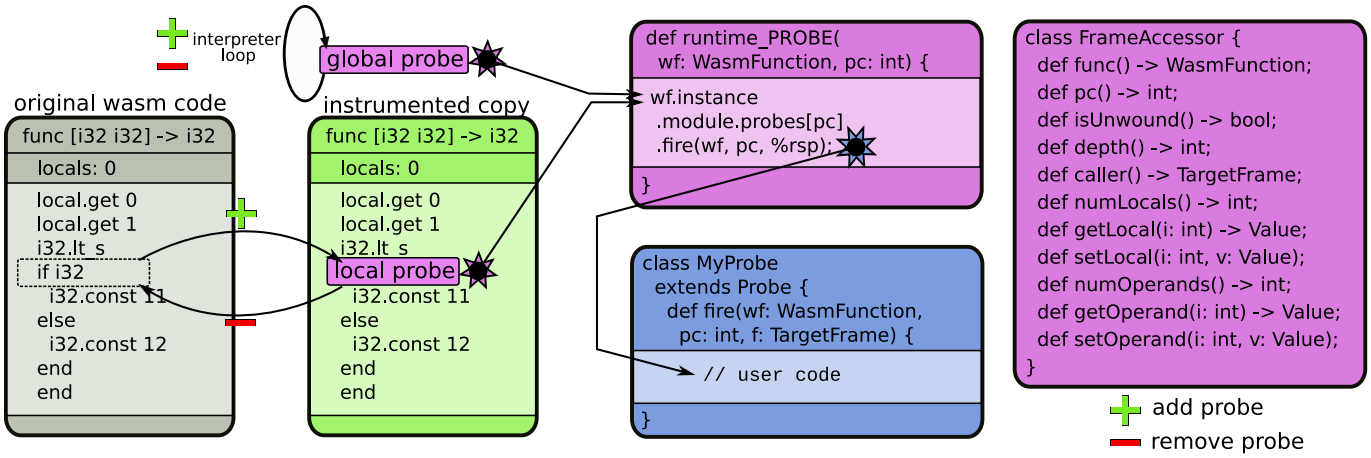

Figure 1: Illustration of instrumentation in the interpreter. Global probes can be inserted into the interpreter loop and local probes are implemented via bytecode overwriting. The `FrameAccessor` API allows a probe programmatic access to the state in the Wasm frame.

imposes zero overhead when global probes are not enabled, obviating the need for a separate debug build.

2.2. Local Probes

Many dynamic analyses are *sparse*, only needing to instrument a subset of code locations. For this reason, Wizard also allows *local probes* to be attached to specific locations in the bytecode. At runtime, the engine fires local probes just *before* executing the respective instruction. Each Wasm instruction can be identified uniquely by its module, function, and byte offset from the start of the function, making the triple `(module, funcdecl, pc)` a natural location identifier in the API. Since local probes only fire when reaching a specific location, they are more convenient for implementing analyses such as branch profiling, call graph analysis, code coverage, breakpoints, etc.

Local probes can be significantly more efficient than global probes for several reasons:

- zero overhead for uninstrumented instructions
- efficient implementation in interpreter and compilers
- compilers can optimize around local probes

Like global probes, local probes are *complete*; both can be implemented in terms of each other. The difference is in their execution efficiency.

2.3. The `FrameAccessor` API

While many analyses need only the sequence of program locations executed, more advanced dynamic analyses like taint tracking, fuzzing and debugging observe program states. To allow probe callbacks access to program state, they receive not only the program location, but also a lazily-allocated object with an API for reading state, called the *`FrameAccessor`*.

The `FrameAccessor` hides engine implementation details from callbacks by exposing a façade [23], with methods for each kind of state that is accessible. Rather than exposing the machine-level details of frames, which often differ between execution tiers and engine versions, the accessor object provides a stable, uniform interface. `FrameAccessor` objects are key in a multi-tier system, where, due to dynamic optimization and deoptimization, the engine may change the frame representation during the execution of a function.

A `FrameAccessor` object represents a single stack frame and is allocated when a callback first requests state other than the easily-available `WasmFunction` and `pc`. Importantly, the identity of this object is observable to probes so that they can implement higher-level analyses across multiple callbacks, which we will see shortly. At the implementation level, execution frames maintain the mapping to their accessor by storing a reference in an additional slot (i.e. one machine word), called the *accessor slot*.

Stackwalking and callstack depth. The `FrameAccessor` API allows walking up the callstack to callers so monitors can implement context-sensitive analyses and stacktraces. The depth of the call stack alone is also often useful for tracing or context-sensitive profiling, so `FrameAccessor` objects include a `depth()` method, which a VM can implement slightly more efficiently.

Dangling accessor objects. `FrameAccessor` objects are allocated in the engine's state space (for Wizard, the managed heap), and since probes are free to store references to them across multiple callbacks, it is possible that the accessor object outlives the execution frame that it represents[3]. While the accessor object itself will be eventually reclaimed, it is problematic if `M-code` accesses frames that have been unwound. We identified a number of implementation mechanisms to protect the runtime system from buggy monitors. Our solution is to minimize checks in the interpreter and compiled code and favor checks at the `FrameAccessor` API boundary. This relies on stack frame layout invariants: function entry clears the accessor slot, the first request for the `FrameAccessor` materi-

[3]With ownership, as in Rust, lifetime annotations can statically prevent a `FrameAccessor` object from escaping from a *single* callback, yet some monitors legitimately want to track frames across multiple callbacks.

alizes the object, and subsequent accessor calls compare the accessor slot to a cached stack pointer in the object. To make these checks bulletproof to monitor bugs, `FrameAccessor`s should be invalidated on return, e.g. with a runtime check[4].

2.4. Consistency guarantees

Many analyses can be implemented by making use of dynamic probe insertion and removal. Other analyses, particularly debuggers, could make modifications to frames that alter program behavior. When do new probes and frame modifications take effect? Providing consistency guarantees is key to making reliable analyses that compose well.

2.4.1. Deterministic firing order

What should happen if a probe p at location L fires and inserts another probe q at the same location L? Should the new probe q also fire before returning to the program, or not? Similarly, if probes p and q are inserted on the same event, is their firing order predictable?

We found that a guaranteed probe firing order is subtly important to the correctness of some monitors (e.g. the function entry/exit utility shown in Section 2.5). For this reason, our system guarantees three *dynamic probe consistency* properties:

- **Insertion order is firing order**: Probes inserted on the same event E fire in the same order as they were inserted.
- **Deferred inserts on same event**: When a probe fires on event E and inserts new probes on the same E, the new probes do not fire until the next occurrence of E.
- **Deferred removal on same event**: When a probe fires on event E and removes probes on the same E, the removed probes *do* fire on this occurrence of E but not subsequent occurrences.

2.4.2. Frame modifications

As shown, the `FrameAccessor` provides a mostly *read-only* interface to program state. Since monitors run in the engine's state space, and not the Wasm program's state space, by construction this guarantees that monitors do not alter the program behavior. However, some monitors, such as a debugger's `fix-and-continue` operation, or fault-injection, do intentionally change program state.

For an interpreter, modifications to program state, such as local variables, require no special support, since interpreters typically do not make assumptions across bytecode boundaries. For JIT-compiled code, any assumption about program state could potentially be violated by `M-code` frame modifications. Depending on the specific circumstance, continuing to run JITed code after state changes might exhibit unpredictable program behavior[5].

It's important for the engine to provide a consistency model for state changes made through the `FrameAccessor`. When monitors explicitly *intend* to alter the program's behavior, it is natural for them to expect state changes to take effect immediately, *as if* the program is running in an interpreter. Thus, our system guarantees:

- **Frame modification consistency**: State changes made by a probe are immediately[6] applied, and execution after a probe resumes with those changes.

Thus, Wizard's consistency guarantees include not only executing the original program according to Wasm's semantics, but *also* executing the program under observation and state changes done through instrumentation. With these guarantees, probes from multiple monitors do not interfere, making monitors *composable* and *deterministic*.

2.5. Function Entry/Exit Probes

Probes are a low-level, instruction-based instrumentation mechanism, which is natural and precise when interfacing with a VM. Yet many analyses focus on function-level behavior and are interested in calls and returns. Instrumentation hooks for function entry/exit make such analyses much easier to write.

At first glance, detecting function entry can be done by probing the first bytecode of a function, and exit can be detected by probing all **`return`**s, **`throw`**s, and **`br`**s that target the function's outermost block. However, some special cases make this tricky. First, a function may begin with a **`loop`**; the entry probe must distinguish between the first entry to a function, a backedge of the loop, and possible (tail-)recursive calls. Second, local exits are not enough: frames can be unwound by a callee throwing an exception caught higher in the callstack.

Should the VM support function entry/exit as special hooks for probes? Interestingly, we find this is not strictly necessary. This functionality can be built from the programmability of local probes and offered as a library. There are even several possible implementation strategies: 1) using entry probes that push the `FrameAccessor` objects onto an internal stack, with exit probes popping from the stack; 2) sampling the stack depth via the `FrameAccessor`'s `depth()` method; or 3) by instrumenting, and thus ignoring, loop backedges. Thus, function entry/exit reside above global/local probes in the hierarchy of instrumentation mechanisms.

2.6. After-instruction

Some analyses, such as branch profiling or dynamic call graph construction, are naturally expressed as `M-code` that should run *after* an instruction rather than *before*. For example, profiling which functions are targets of a **`call_indirect`** would be easiest if a probe could fire *after* the instruction is executed and a frame for the target function has been pushed onto the execution stack. However, the API has no such functionality.

[4] Or a return guard trampoline, which avoids any runtime overhead.

[5] True even for baseline compilers like Wizard's compiler, which perform limited optimizations like register allocation and constant propagation.

[6] Note: frames are by-definition thread local; races can only exist if the monitor itself is multi-threaded and `FrameAccessor` objects are shared racily.

Should the VM support an "after-instruction" hook directly? Interestingly, we find that like function entry/exit, the unlimited programmability of probes allows us to invoke `M-code` *seemingly after* instructions. For example, suppose we want to execute probe p after a **`br_table`** (i.e. Wasm's switch instruction). A probe q_p executed *before* the **`br_table`** can use the `FrameAccessor` object to read the top (**`i32`** value) of the operand stack, determine where the branch will go, and dynamically insert probe p at that location. Another option, since **`br_table`** has a limited set of local targets, would be to insert probes into all such targets once and use a state variable to distinguish reaching each target from the **`br_table`** versus another path. However, other instructions like **`call_indirect`** have an unlimited set of targets, so a dynamic probe insertion is the most general solution to implement "after instruction". Thus after-instruction resides above global/local probes in the instrumentation mechanism hierarchy.

3. The Monitor Zoo

The wide variety and ease with which analyses are implemented[7] showcases the flexibility of having a fully-programmable instrumentation mechanism in a high-level language. Users activate monitors when invoking Wizard with flags (e.g. `wizeng --monitors=MyMonitor`), which instrument modules at various stages of processing before execution and may generate post-execution reports. Examples of monitors we have built include a variety of useful tools.

The **Trace monitor** prints each instruction as it is executed. While many VMs have tracing flags and built-in modes that may be spread throughout the code, Wizard already offers the perfect mechanism: the global probe. Instruction-level tracing in Wizard simply uses one global probe. Other than a short flag to enable it, there is nothing special about this probe; it uses the standard `FrameAccessor` API as it prints instructions and the operand stack.

The **Coverage monitor** measures code coverage. It inserts a local probe at every instruction (or basic block), which, when fired, sets a bit in an internal datastructure and then *removes itself*. By removing itself, the probe will no longer impose overhead, either in the interpreter or JITed code. Eventually, all executed paths in the program will be probe-free and JITed code quality will asymptotically approach zero overhead. This is a good example of a monitor using dynamic probe removal.

The **Loop monitor** counts loop iterations. It inserts `CountProbe`s at every loop header and then prints a nice report. This is a good example of a counter-heavy analysis.

The **Hotness monitor** counts every instruction in the program. It inserts `CountProbe`s at every instruction and then prints a summary of hot execution paths. Another example of a counter-heavy analysis.

The **Branch monitor** profiles the direction of all branches. It instruments all **`if`**, **`br_if`** and **`br_table`** instructions and uses the top-of-stack to predict the direction of each branch. It is a good example of non-trivial `FrameAccessor` usage.

The **Memory monitor** traces all memory accesses. It instruments all loads and stores and prints loaded and stored addresses and values. Another good example of non-trivial `FrameAccessor` usage.

The **Debugger REPL** implements a simple read-eval-print loop that allows interactive debugging at the Wasm bytecode level. It supports breakpoints, watchpoints, `single-step`, `step-over`, and changing the state of value stack slots. It primarily uses local probes but uses a global probe to implement single-step functionality. This monitor is a good example of dynamic probe insertion and removal. It is also the only monitor (so far) that modifies frames.

The **Calls monitor** traces function calls, including Wasm and host calls, and can be configured with various filters. It uses local probes at all direct and indirect callsites and specific support for attaching to host calls.

The **Call tree profiler** measures execution time of function calls and prints self and nested time using the full calling-context tree. It can also produce flame graphs. It inserts local probes at all direct and indirect callsites and all return locations [8]. It is a good example of a monitor that measures non-virtualized metrics like wall-clock time.

4. Optimizing probe overhead

Optimizations in Wizard's interpreter and JIT compiler reduce overhead for both global and local probes. We define *overhead* as the execution time spent in neither application code nor `M-code`, but either in transitions *between* application and `M-code` or additional work in the runtime system and compiler.

4.1. Optimizing global probes in the interpreter

Global probes, being the most heavyweight instrumentation mechanism, are supported only in the interpreter. It is straightforward to add a check to the interpreter loop that checks for any global probes at each instruction. However, the naive approach imposes overhead on all instructions executed, even if global probes are not enabled. One option to avoid overhead when global probes are disabled is to have two different interpreter loops, one with the check and without, and dynamically switch between them. This comes at some VM code space cost, since it duplicates the entire interpreter loop and handlers. Another approach described in [52] avoids the code space cost by maintaining a pointer to the *dispatch table* in a register and switches modes by changing the register. Both approaches are suitable for production, as they allow the VM to support global probes while imposing no overhead when disabled.

[7] Most monitors required a dozen or two lines of instrumentation code; in fact, most lines are usually spent on making useful presentation of the data!

[8] Wizard has preliminary support for the proposed Wasm exception handling mechanism, but does not yet have monitoring hooks for unwind events.

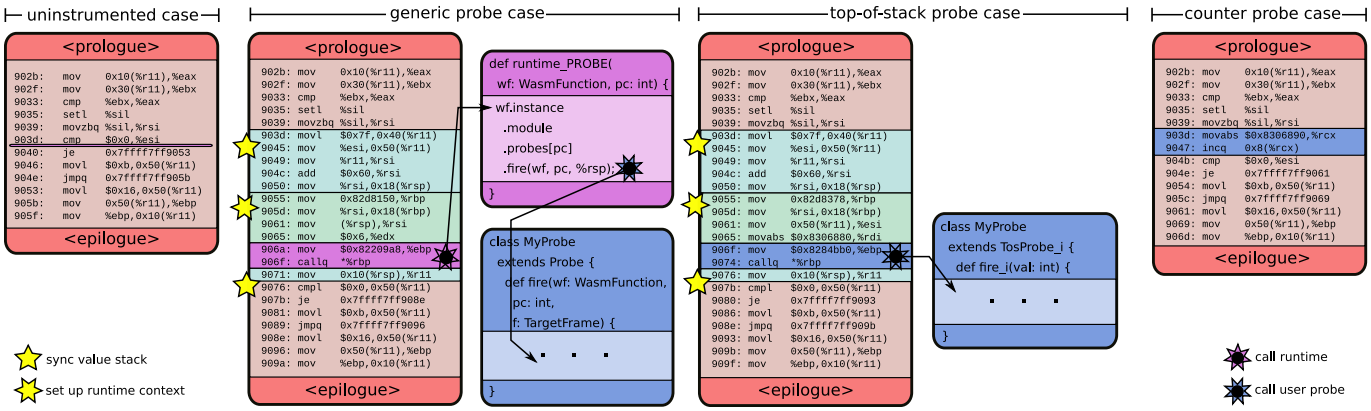

Figure 2: Code generated by Wizard's baseline JIT for different types of `M-code` implemented with probes. The machine code sequence for generic probes is more general than for probes that only need the top-of-stack value, versus a fully-intrinsified counter probe.

4.2. Optimizing local probes in the interpreter

Local probes are supported in both Wizard's interpreter and baseline JIT. In the interpreter, local probes impose no overhead on non-probed instructions by using in-place bytecode modifications. With *bytecode overwriting*, the insertion of a local probe at a location L overwrites the original opcode at L with an otherwise illegal **`probe`** opcode. The original unmodified opcode is saved on the side. When the interpreter reaches a **`probe`** opcode, the Wasm program state (e.g. value stack) is already up-to-date; it saves the interpreter state, looks up the local probe(s) at the current bytecode location, and simply calls that `M-code` callback. In Wizard, since the callback is compiled machine code, the overhead is a small number of machine instructions to exit the interpreter context and enter the callback context. After returning from `M-code`, the original opcode at L is loaded (e.g. by consulting an unmodified copy of the function's code) and executed. Removing a probe is as simple as copying the original bytecode back; the interpreter will no longer trip over it. Overwriting has two primary advantages over bytecode injection; the original bytecode offsets are maintained, making it trivial to report locations to `M-code`, and insertion/removal of probes is a cheap, constant-time operation. Consistency is trivial; the bytecode is always up-to-date with the state of inserted instrumentation.

4.3. Local probes in the JIT

In a JIT compiler, local probes can be supported by injecting calls to `M-code` into the compiled code at the appropriate places. Since probe logic could potentially access (and even modify) the state of the program through the `FrameAccessor`, a call to unknown `M-code` must checkpoint the program and VM-level state. For baseline code from Wizard's JIT, the overhead of making such a call is a few machine instructions more than a normal call between Wasm functions [9]. The presence of probes has little impact on compile speed and the code quality of uninstrumented regions. Bytecode overwriting benefits more than the interpreter. The **`probe`** opcode marks instrumented instructions, avoiding additional checks in the JIT's bytecode parsing loop. Supporting probes adds less than one hundred lines of code to Wizard's JIT.

4.4. JIT-intrinsification of probes

While probes are a fully-programmable instrumentation mechanism to implement unlimited analyses, there are a number of common building blocks such as counters, switches, and samplers that many different analyses use. For logic as simple as incrementing a counter every time a location is reached, it is highly inefficient to save the entire program state and call through a generic runtime function to execute a single increment to a variable in memory. Thus, we implemented optimizations in Wizard's JIT to *intrinsify* counters as well as probes that access limited frame state.

Figure 2 shows machine code sequences for a sample Wasm function with different probes attached. We see here how Wizard's baseline JIT optimizes different kinds of probes. At the left, we have uninstrumented code. For the generic probe case, the compiler calls a generic runtime routine which calls the user's probe. For the next more specialized case, the top-of-stack, we can skip the runtime call overhead and the cost of reifying an expensive `FrameAccessor` object on every probe call by instead directly passing the top-of-stack value to the probe's `fire` method. This is just an example of a more general pattern where a fixed set of known quantities from the frame can be directly passed from the JITed code to `M-code`. Lastly, for the counter probe, we see that Wizard's JIT simply inlines an increment instruction to a *specific* `CountProbe` object without looking it up.

4.5. Monitor consistency for JITed code

We just saw how a JIT can inline `M-code` into the compiled code. However, `M-code` can change as probes can be inserted and removed during execution, making compiled code that has been specialized to `M-code` out-of-date. This problem can be addressed by standard deoptimization techniques such as

[9] Primarily because the calling convention models an explicit value stack.

on-stack-replacement back to the interpreter and invalidating relevant machine code.

4.6. Strategies for multi-tier consistency

There are several different strategies for guaranteeing monitor consistency in a multi-tier engine like Wizard.

1. When instrumentation is enabled, disable the JIT.
2. When instrumentation is enabled, disable only *relevant* JIT optimizations.
3. Upon frame modification, recompile the function under different assumptions about frame state and perform on-stack-replacement from JITed to JITed code.
4. Upon frame modification, perform on-stack-replacement from JITed code to the interpreter.

We observe strategy 1) while heavyweight, is dead-simple to implement for engines with interpreters. For a production Wasm engine that wishes to adopt our instrumentation API, this achieves functional correctness at little engineering cost, leaving instrumented performance as a later product improvement. It provides the key consistency guarantees but may be too slow to run often or in production. We would expect strategy 2) to fare better by eliminating interpreter dispatch cost. But, ironically, this strategy is actually a lot of work in practice, since it introduces modes into the JIT compiler and optimizations must be audited for correctness. The compiler logic becomes littered with checks to disable optimization and ultimately the JIT defaults to emitting very pessimistic code. Strategy 3) has other implications for JIT compilation. In particular, it requires the JIT to be able to support arbitrary OSR locations[10], which is also significant engineering work.

In Wizard, we chose strategy 4. Frame modifications trigger immediate deoptimization of *only that frame*, rewriting it in place to return to the interpreter. In the *dynamic* tiering configuration mode, sending an execution frame back to the interpreter due to frame modification doesn't banish it there forever; if it remains hot, it will be recompiled under new assumptions[11]. This means frame modification support requires the interpreter; Wizard will not allow modifications in the JIT-only config.

5. Evaluation

In this section, we evaluate several aspects of the performance of the monitoring API over several sets of benchmarks and with several engine implementation strategies.

[10]Most JITs that allow tier-up OSR into compiled code only do so at loop headers. Depending on the JIT design, OSR entries might require significant IR transformations such as loop peeling. Difficult.

[11]Pathological cases can occur where hot frames are repeatedly modified, constantly transferring between interpreter and JITed code. This phenomenon can occur in dynamic language implementations such as JavaScript engines and a typical fix is to simply limit the number of times a function can be optimized and offer user diagnostics.

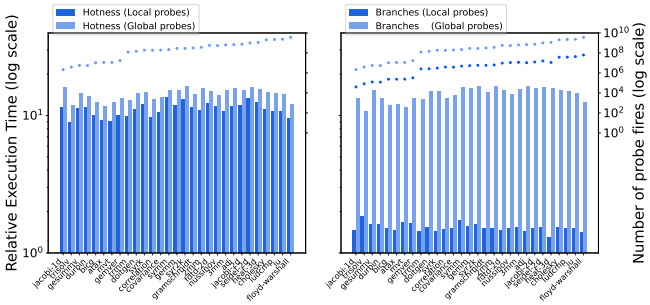

Figure 3: Relative execution time for the hotness monitor (left) and branch monitor (right), when implemented with local probes and when implemented with a global probe. Points above the bars denote number of probe fires.

5.1. Evaluation setup

We evaluate the performance of Wizard by executing Wasm code under various execution strategies (interpreter or JIT) and monitors, measuring total execution time of the entire program, including engine startup and program load.

We chose the "hotness" and "branch" monitors (described in Section 3). The hotness monitor instruments every instruction with a local `CountProbe`, which is representative of monitors with many simple probes. The branch monitor probes branch instructions and tallies each destination by accessing the top of the operand stack. Compared to the hotness monitor, probes in the branch monitor are more sparse but more complex.

These monitors were chosen because they strike a balance between being powerful enough to capture insights about the execution of a program, yet simple enough to be implemented in other systems. They are also likely to instrument a nontrivial portion of program bytecode.

Benchmark Suites. We run Wasm programs from three benchmark suites: PolyBench/C [42] with the `medium` dataset, Ostrich [27] and Libsodium [21]. We average execution time over 5 runs.

Given instrumented execution time T_i and uninstrumented execution time T_u, we define *absolute overhead* as the quantity $T_i - T_u$ and *relative execution time* as the ratio T_i/T_u. We report relative execution time for Wizard's interpreter, Wizard's JIT (with and without intrinsification), DynamoRIO, Wasabi, and bytecode rewriting in Figures 6 and 7.

5.2. Global vs local probes

Global probes can emulate the behavior of local probes, but impose a greater performance cost by introducing checks at every bytecode instruction. We compare two implementations of the branch and hotness monitors, one using a global probe and the other using local probes. Both are executed in Wizard's interpreter, since Wizard's JIT doesn't support global probes. The results can be found in Figure 3. For the hotness monitor, since the number of probe fires is the same for local and global probes, the relative overhead is similar across all programs.

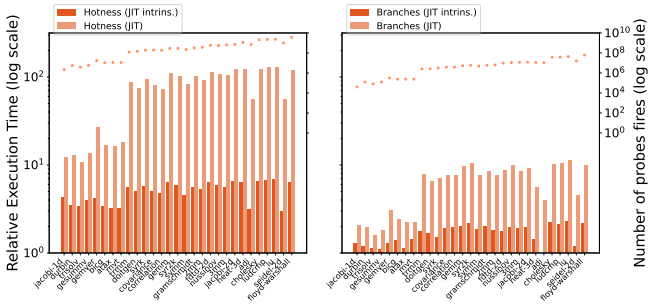

Figure 4: Relative execution times for the hotness (left) and branch monitors (right), with and without probe intrinsification. Ratios are relative to uninstrumented JIT execution time. Points above the bars denote number of probe fires.

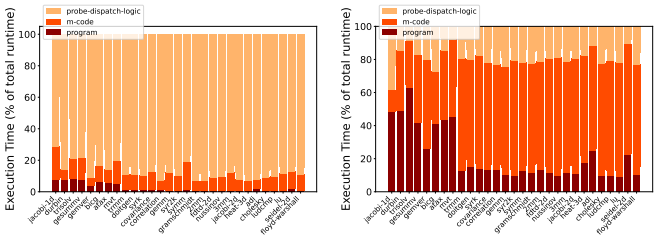

Figure 5: Execution time decomposition of hotness (left) and branch monitors (right) into `M-code` and probe dispatch overhead with and without probe intrinsification.

For the branch monitor, local probes on branch instructions have relative execution times $< 2.18\times$, whereas it is between $< 16.2\times$ for global probes.

5.3. JIT optimization of count and operand probes

Section 4 describes how Wizard's JIT intrinsifies some types of probes to reduce overhead. We evaluate the effectiveness of JIT-intrinsification in Figure 4 and report the relative execution time of instrumented over non-instrumented.

For the hotness monitor, which counts the execution frequency of every instruction[12] we observed relative execution times between 7.02–134×. This is due to the high cost of switching between JIT code and the engine at every instruction. With intrinsification, the same monitor imposes relative execution times between 2.17–7.71×.

We performed a similar experiment to evaluate the effectiveness of JIT-intrinsification of top-of-stack operand probes by measuring execution times with the branch monitor. We see that intrinsification improves relative execution times from $< 16.6\times$ to $< 2.81\times$. The improvement is less dramatic than for `CountProbe`s, which are fully inlined, as a call into the probe's `M-code` remains (see Figure 2).

We further decompose the runtime of the benchmarks into the time spent in the program's JIT-compiled code (T_{JIT}), time in `M-code` (T_M), and time in the probe dispatch logic (T_{PD}). This decomposition is done by recording:

1. The uninstrumented execution time of code in the JIT, which approximates T_{JIT};
2. The instrumented execution time with empty probes (probes with empty `fire` functions), which approximates $T_{\text{PD}} + T_{\text{JIT}}$;
3. The instrumented execution time with functional probes, which gives $T_{\text{PD}} + T_M + T_{\text{JIT}}$.

The results of this analysis for the branch and hotness monitors are in Figure 5. Execution time without JIT intrinsification is shown as the entire bar for each program. The cross-hatched portions of each bar represent the execution time saved by intrinsification. For the non-intrinsified branch monitor, the overhead $T_{\text{PD}} + T_M$ is dominated by `M-code`. In the intrinsified case, the overhead is dominated by probe dispatch, and the `M-code` overhead is reduced substantially: calling the top-of-stack operand probe's `M-code` still requires significant spilling on the stack and a call, contributing to runtime overhead. The `M-code` overhead no longer includes time for construction of the `FrameAccessor` as it is not necessary.

As for the non-intrinsified hotness monitor, the overhead is dominated by the probe dispatch overhead as probes are simpler but fired more frequently. In the intrinsified case, there is almost no `M-code` overhead as counter probes do not have custom `fire` functions; the counter increment is entirely inlined. The remaining probe dispatch overhead comes from the monitor setup and reporting.

5.4. Interpreter vs. JIT

We find that the relative overhead of Wizard's interpreter is much lower than the JIT, for two reasons: the interpreter runs much slower, and less additional work is done in checkpointing state. In contrast, calls to local probes in the JIT require checkpointing to support the `FrameAccessor` API. Data in figures 6 and 7 show that, for the branch monitor, the relative execution time in the interpreter is $< 2.18\times$ as compared to $< 17\times$ in Wizard's JIT. In the higher-workload hotness monitor, this difference is even more stark: the relative execution time in the interpreter is 7.00–13.5× as compared to 7.02–134× in the JIT. Although relative execution times differ substantially, absolute overhead between the two modes is comparable: for the branch monitor, the mean overhead in the interpreter is 2.55s and 2.27s in the JIT.

5.5. Comparison with bytecode rewriting

Bytecode rewriting is an example of static instrumentation described in Section 1.1.1. Using Walrus [7], a Wasm transformation library written in Rust, we implemented the hotness and branch monitors as bytecode rewriters [8]. For the hotness monitor, we inject counting instructions before each insruction, and for the branch monitor, before each branching instruction. Counters are stored in memory, necessitating loads and stores. We evaluated the performance of the transformed

[12] Obviously, it is more efficient to count basic blocks. We chose to count every instruction in order to *maximize* instrumentation workload.

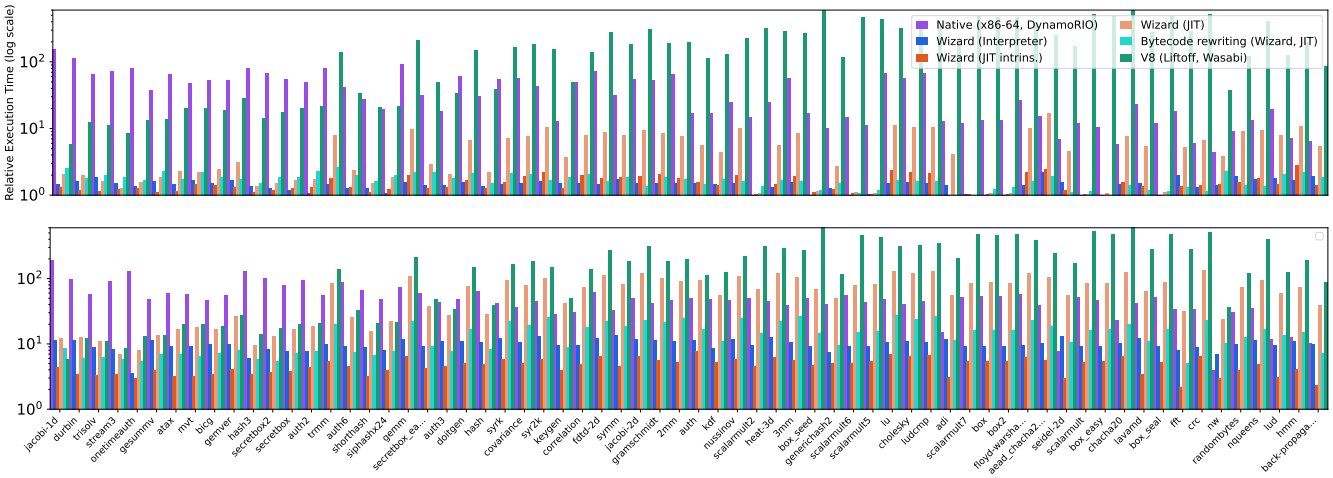

Figure 6: Relative execution times of the hotness monitor (bottom) and branch monitor (top) in Wizard, Wasabi, and DynamoRIO across all programs on all suites, sorted by absolute execution time. Ratios are relative to uninstrumented execution time.

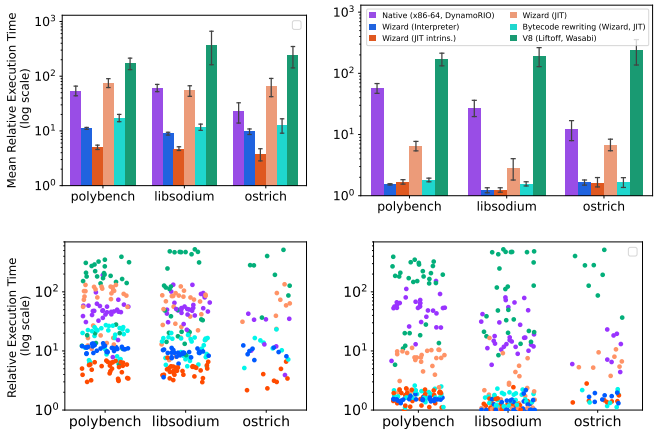

Figure 7: Mean relative execution times of the hotness monitor (left) and branch monitor (right) in Wizard, Wasabi, and DynamoRIO across the three suites. Ratios are relative to uninstrumented execution time.

Wasm bytecode when run in Wizard's JIT, and compared it to their respective monitors in Wizard.

From Figure 7, we observe that the intrinsified JIT execution time is lower than bytecode rewriting for both monitors.

5.6. Comparison with Wasabi

Wasabi is an alternative dynamic instrumentation tool that runs analyses on Wasm bytecode using a JavaScript engine. Since Wasabi instrumentation must be written in JavaScript, it requires a Wasm engine that also runs JavaScript, such as V8 [5]. For this comparison, we use V8 in its single-pass compiler mode (**`v8-liftoff`**) which is more similar to Wizard's JIT than its default configuration with an optimizing compiler.

While a full engine comparison between Wizard and V8 is beyond the scope of this paper, we measured the execution time of Wizard and **`v8-liftoff`** (not shown). We found that on average, Wizard's JIT is 45–55% faster than **`v8-liftoff`**, and for slower items, it is generally no more than 10–20% slower. Thus, relative instrumentation overheads in the two systems have comparable baselines.

Figure 6 includes data for Wasabi on **`v8-liftoff`**. Wasabi instrumentation is vastly slower than Wizard instrumentation, due to the overhead of calling JavaScript functions. On average, a branch monitor in Wasabi slows execution 4.73–375×, compared to 1.04–16.6× for Wizard's JIT.

5.7. Comparison with DynamoRIO

We also compare with machine code instrumentation. We cannot make a direct comparison, so instead, we compile the same benchmark programs to x86-64 assembly and instrument them with DynamoRIO, with analogous machine-code hotness and branch monitors.

The results are shown in Figures 6 and 7. Executing the native programs instrumented with a DynamoRIO hotness monitor is about 3.91–192× slower than without instrumentation. The hotness monitor has a substantial relative overhead because, among other things, DynamoRIO inserts instructions to spill and restore `EFLAGS` for each counter increment. On the other hand, the DynamoRIO branch monitor slows execution time from 4.40–153×, which varies a lot. This is likely because our DynamoRIO monitor is implemented with a function call at every basic block, which DynamoRIO can sometimes inline, since it works at the machine code level. Its default inlining heuristics seem to give rise to unpredictable overheads.

6. Related work

Techniques for studying program behavior have been the subject of a vast amount of research. Techniques differ in *when* to instrument (statically or dynamically), at *which level* (source, IR, bytecode, or machine code), and *what* mechanism is used

to do so. A key issue is that the analysis and the chosen mechanism work together to analyze a program's behavior *intrusively* or *non-intrusively*, yet some mechanisms provide safer program observation, making it easier to implement non-intrusive analyses.

6.1. Code injection

A very common approach is to inject `M-code` directly into programs [59], either statically or dynamically. Injecting code into a program can be done *inline* (directly inserted into code, often requiring binary reorganization), with *trampolines* (jumps to out-of-line instrumentation code), or a mix of both.

Static. Early tools for Java static bytecode instrumentation include Soot [55] and Bloat [41]. Later, with the rise of Aspect-Oriented Programming (AOP), tools emerged to target *joinpoints*, such as DiSL [35], AspectJ [29] and BISM [47]. Oron [37] reduced the performance overhead of JavaScript source-level instrumentation by targetting AssemblyScript and compiling the instrumented program to Wasm for execution. For Wasm, tools are now emerging such as the aspect-oriented [44], and Wasabi [31], which injects trampolines into Wasm bytecode that call instrumentation code provided as JavaScript.

Dynamic. FERRARI [15] statically instruments core JDK classes while dynamically instrumenting all others at the bytecode level using `java.lang.instrument`. SaBRe [9] injects instrumentation at load-time (also done by BISM) thus paying the rewriting performance hit once at startup rather than continuously during execution. DTrace [25, 20], inspired by Paradyn [36] and other tools, enables tracing at both the user and kernel layer of the OS by operating inside the kernel itself and uses dynamically-injected trampolines. Dyninst [14] interfaces with a program's CFG and maps modifications to concrete binary rewrites. A user can tie `M-code` to instructions or CFG abstractions (e.g. function entry/exit). It can do this statically or at any point during execution and changes are immediate. Recent research in this direction [19] [57] [43] focuses on reducing instrumentation overhead with a variety of low-level optimizations.

6.2. Recompilation.

Compiled programs can be *recompiled* to inject code using several techniques. For example, their compiled code can be *lifted* to a higher-level IR, which is then instrumented and recompiled.

Static. Early examples of static lifting for instrumentation include ATOM [49], followed by EEL [30] with finer-grained instrumentation. Etch [45], through observing an initial program execution, discovered dynamic program properties to inform static instrumentation. Other examples include Vulcan [48], which injects code into lifted Win32 binaries.

Dynamic. Dynamic recompilation of compiled code can be done with a JIT compiler. For example, both DynamoRIO [17] and Pin [34] use dynamic recompilation of native binaries to implement instrumentation. They differ somewhat on subtle implementation details, how `M-code` is injected, and performance characteristics, but fundamentally work by recompiling machine code for a given ISA to the same ISA. Their JIT compilers are purpose-built for instrumentation and basic-block and trace-cache based. Though they take great care to be non-intrusive in most situations, since they run code in the original process and potentially reorganize binaries, they can be intrusive, particularly if `M-code` is supplied as low-level native code.

6.3. Emulation

QEMU [13] is a widely-used CPU emulator. It includes an interpreter for several different ISAs and completely virtualizes a user-space process while supporting unintrusive instrumentation. Valgrind [39], primarily used as a memory debugger, is similar. As emulators, both can run a guest ISA on a different host ISA, and both use JIT compilers to make emulation fast. Thus, their JIT compilers are not necessarily "purpose-built" for instrumentation, but for cross-compilation. Avrora [53], a microcontroller emulator and sensor network simulator, provides an API to attach `M-code` to clock events, instructions, and memory locations.

6.4. Direct engine support

Runtime systems can be designed with specific support for instrumentation, enabling intelligent optimization and fine-grained control of all program state and control flow. The JVM TI [3] allows Java bytecode instrumentation and also *agents* to be written against a lower-level internal engine API. In .NET [11], users build profiler DLLs that are loaded by the CLR into the same process as a target application. The CLR then notifies the profiler of events occuring in the application through a callback interface.

7. Conclusion and Future Work

In this paper, we showed the first non-intrusive dynamic instrumentation framework for Wasm in a multi-tier Wasm research engine that imposes zero overhead when not in use. Modifications to the interpreter and compiler tiers of Wizard are minimal; just a few hundred lines of code. Novel optimizations reduce instrumentation overhead and perform well for sparse analysis and acceptably well for heavy analysis.

While probes offer a complete instrumentation mechanism for code, many analyses instrument other events, such as accesses to memory locations, traps, etc. As we saw with function entry/exit and after-instruction hooks, libraries can implement higher-level hooks *using* probes; but if directly supported by the engine, these hooks can be implemented more efficiently, e.g. hardware watchpoints for memory accesses.

In this work, we showed monitors written against Wizard's engine APIs in a high-level language. Generic probes use runtime calls to compiled `M-code`. Massive speedups are

possible from intrinsifying *certain* probes by inlining all or part of their `M-code`. What if `M-code` was instead supplied in an *IR* the JIT could *inline*? We plan to explore *Wasm bytecode* as just that IR.

References

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

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

[3] Java Virtual Machine Tools Interface. https://docs.oracle.com/javase/8/docs/technotes/guides/jvmti/, 2021. (Accessed 2021-07-29).

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

[5] V8 development site. https://v8.dev, 2021. (Accessed 2021-07-29).

[6] WebAssembly specifications. https://webassembly.github.io/spec/, 2021. (Accessed 2021-07-29).

[7] Walrus: A webassembly transformation library. https://github.com/rustwasm/walrus, 2023.

[8] Wasm bytecode instrumenter. https://github.com/yashanand1910/wasm-bytecode-instrumenter, 2023.

[9] Paul-Antoine Arras, Anastasios Andronidis, Luís Pina, Karolis Mituzas, Qianyi Shu, Daniel Grumberg, and Cristian Cadar. Sabre: load-time selective binary rewriting. *International Journal on Software Tools for Technology Transfer*, 24(2):205–223, Apr 2022.

[10] Linux Wiki Authors. Linux perf main page. https://perf.wiki.kernel.org/index.php/Main_Page, 2012. (Accessed 2023-8-4).

[11] .NET Wiki Authors. The .NET Profiling API. https://learn.microsoft.com/en-us/dotnet/framework/unmanaged-api/profiling/profiling-overview, 2021. (Accessed 2023-8-4).

[12] David F. Bacon, Perry Cheng, and David Grove. Tuningfork: A platform for visualization and analysis of complex real-time systems. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, OOPSLA '07, page 854–855, New York, NY, USA, 2007. Association for Computing Machinery.

[13] Fabrice Bellard. Qemu: A generic and open source machine emulator and virtualizer. http://qemu.org, 2020. (Accessed 2023-8-07).

[14] Andrew R. Bernat and Barton P. Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE '11, page 9–16, New York, NY, USA, 2011. Association for Computing Machinery.

[15] Walter Binder, Jarle Hulaas, and Philippe Moret. Advanced java bytecode instrumentation. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, PPPJ '07, page 135–144, New York, NY, USA, 2007. Association for Computing Machinery.

[16] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. Provably-safe multilingual software sandboxing using WebAssembly. In *Proceedings of the USENIX Security Symposium*, August 2022.

[17] D Bruening, T Garnett, and S Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* IEEE Comput. Soc, 2003.

[18] Rodrigo Bruno, Duarte Patricio, José Simão, Luis Veiga, and Paulo Ferreira. Runtime object lifetime profiler for latency sensitive big data applications. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[19] Buddhika Chamith, Bo Joel Svensson, Luke Dalessandro, and Ryan R. Newton. Living on the edge: Rapid-toggling probes with cross-modification on x86. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 16–26, New York, NY, USA, 2016. Association for Computing Machinery.

[20] Greg Cooper. Dtrace: Dynamic tracing in oracle solaris, mac os x, and free bsd by brendan gregg and jim mauro. *SIGSOFT Softw. Eng. Notes*, 37(1):34, jan 2012.

[21] Frank Denis, 2021.

[22] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for java. *SIGPLAN Not.*, 38(11):149–168, oct 2003.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Pearson Education, 1994.

[24] Manuel Geffken and Peter Thiemann. Side effect monitoring for java using bytecode rewriting. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, page 87–98, New York, NY, USA, 2014. Association for Computing Machinery.

[25] Brendan Gregg and Jim Mauro. *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD*. Prentice Hall Press, USA, 1st edition, 2011.

[26] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 185–200, New York, NY, USA, 2017. Association for Computing Machinery.

[27] David Herrera, Hanfeng Chen, Erick Lavoie, and Laurie Hendren. Numerical computing on the web: Benchmarking for the future. In *Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages*, DLS 2018, page 88–100, New York, NY, USA, 2018. Association for Computing Machinery.

[28] Saba Jamilan, Tanvir Ahmed Khan, Grant Ayers, Baris Kasikci, and Heiner Litz. Apt-get: Profile-guided timely software prefetching. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 747–764, New York, NY, USA, 2022. Association for Computing Machinery.

[29] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, page 327–353, Berlin, Heidelberg, 2001. Springer-Verlag.

[30] James R. Larus and Eric Schnarr. Eel: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, page 291–300, New York, NY, USA, 1995. Association for Computing Machinery.

[31] Daniel Lehmann and Michael Pradel. Wasabi: A framework for dynamically analyzing webassembly. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 1045–1058, New York, NY, USA, 2019. Association for Computing Machinery.

[32] Borui Li, Hongchang Fan, Yi Gao, and Wei Dong. 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*, MobiSys '21, page 487–488, New York, NY, USA, 2021. Association for Computing Machinery.

[33] Renju Liu, Luis Garcia, and Mani Srivastava. Aerogel: Lightweight access control framework for webassembly-based bare-metal iot devices. In *2021 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 94–105, 2021.

[34] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.

[35] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. Disl: A domain-specific language for bytecode instrumentation. In *Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development*, AOSD '12, page 239–250, New York, NY, USA, 2012. Association for Computing Machinery.

[36] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, nov 1995.

[37] Aäron Munsters, Angel Luis Scull Pupo, Jim Bauwens, and Elisa Gonzalez Boix. Oron: Towards a dynamic analysis instrumentation platform for assemblyscript. In *Companion Proceedings of the 5th International Conference on the Art, Science, and Engineering of Programming*, Programming '21, page 6–13, New York, NY, USA, 2021. Association for Computing Machinery.

[38] Otoya Nakakaze, István Koren, Florian Brillowski, and Ralf Klamma. Retrofitting industrial machines with webassembly on the edge. In Richard Chbeir, Helen Huang, Fabrizio Silvestri, Yannis Manolopoulos, and Yanchun Zhang, editors, *Web Information Systems Engineering – WISE 2022*, pages 241–256, Cham, 2022. Springer International Publishing.

[39] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, jun 2007.

[40] Manuel Nieke, Lennart Almstedt, and Rüdiger Kapitza. Edgedancer: Secure mobile WebAssembly services on the edge. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*, EdgeSys '21, page 13–18, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Nathaniel John Nystrom. Bytecode-level analysis and optimization of java classes. Master's thesis, Purdue University, August 1998.

[42] Louis-Noel Pouchet, May 2016.

[43] David Georg Reichelt, Stefan Kühne, and Wilhelm Hasselbring. Towards solving the challenge of minimal overhead monitoring. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*, ICPE '23 Companion, page 381–388, New York, NY, USA, 2023. Association for Computing Machinery.

[44] João Rodrigues and Jorge Barreiros. Aspect-oriented webassembly transformation. In *2022 17th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–6, 2022.

[45] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and optimization of win32/intel executables using etch. In *Proceedings of the USENIX Windows NT Workshop on The USENIX Windows NT Workshop 1997*, NT'97, page 1, USA, 1997. USENIX Association.

[46] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 488–498, New York, NY, USA, 2013. Association for Computing Machinery.

[47] Chukri Soueidi, Ali Kassem, and Yliès Falcone. Bism: bytecode-level instrumentation for software monitoring. In *Runtime Verification: 20th International Conference, RV 2020, Los Angeles, CA, USA, October 6–9, 2020, Proceedings 20*, pages 323–335. Springer, 2020.

[48] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical report, Microsoft Research, 2001.

[49] Amitabh Srivastava and Alan Eustace. Atom: A system for building customized program analysis tools. New York, NY, USA, 1994. Association for Computing Machinery.

[50] Ben L. Titzer. Harmonizing classes, functions, tuples, and type parameters in Virgil III. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 85–94, New York, NY, USA, 2013. Association for Computing Machinery.

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

[52] Ben L. Titzer. A fast in-place interpreter for webassembly. *Proc. ACM Program. Lang.*, 6(OOPSLA2), oct 2022.

[53] Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, IPSN '05, page 67–es. IEEE Press, 2005.

[54] Ben L. Titzer and Jens Palsberg. Nonintrusive precision instrumentation of microcontroller software. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '05, page 59–68, New York, NY, USA, 2005. Association for Computing Machinery.

[55] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, CASCON '10, page 214–224, USA, 2010. IBM Corp.

[56] Kenton Varda. WebAssembly on Cloudflare Workers. https://blog.cloudflare.com/webassembly-on-cloudflare-workers/. (Accessed 2021-07-06).

[57] Mingzhe Wang, Jie Liang, Chijin Zhou, Zhiyong Wu, Xinyi Xu, and Yu Jiang. Odin: On-demand instrumentation with on-the-fly recompilation. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 1010–1024, New York, NY, USA, 2022. Association for Computing Machinery.

[58] Conrad Watt. Mechanising and verifying the WebAssembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, page 53–65, New York, NY, USA, 2018. Association for Computing Machinery.

[59] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. From hack to elaborate technique—a survey on binary rewriting. *ACM Comput. Surv.*, 52(3), jun 2019.

[60] Zhiqiang Zuo, Kai Ji, Yifei Wang, Wei Tao, Linzhang Wang, Xuandong Li, and Guoqing Harry Xu. Jportal: Precise and efficient control-flow tracing for jvm programs with intel processor trace. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 1080–1094, New York, NY, USA, 2021. Association for Computing Machinery.

[61] Zhiqiang Zuo, Kai Ji, Yifei Wang, Wei Tao, Linzhang Wang, Xuandong Li, and Guoqing Harry Xu. Jportal: Precise and efficient control-flow tracing for jvm programs with intel processor trace. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 1080–1094, New York, NY, USA, 2021. Association for Computing Machinery.