6.5 Calling External Functions

Like most interpreters, SpiderMonkey has a foreign function interface (FFI) that allows it to call C builtins and host system functions (e.g., web browser control and DOM access). The FFI has a standard signature for JS-callable functions, the key argument of which is an array of boxed values. External functions called through the FFI interact with the program state through an interpreter API (e.g., to read a property from an argument). There are also certain interpreter builtins that do not use the FFI, but interact with the program state in the same way, such as the `CallIteratorNext` function used with iterator objects. TraceMonkey must support this FFI in order to speed up code that interacts with the host system inside hot loops.

Calling external functions from TraceMonkey is potentially difficult because traces do not update the interpreter state until exiting. In particular, external functions may need the call stack or the global variables, but they may be out of date.

For the out-of-date call stack problem, we refactored some of the interpreter API implementation functions to re-materialize the interpreter call stack on demand.

We developed a C++ static analysis and annotated some interpreter functions in order to verify that the call stack is refreshed at any point it needs to be used. In order to access the call stack, a function must be annotated as either FORCESSTACK or REQUIRESSTACK. These annotations are also required in order to call REQUIRESSTACK functions, which are presumed to access the call stack transitively. FORCESSTACK is a trusted annotation, applied to only 5 functions, that means the function refreshes the call stack. REQUIRESSTACK is an untrusted annotation that means the function may only be called if the call stack has already been refreshed.

Similarly, we detect when host functions attempt to directly read or write global variables, and force the currently running trace to side exit. This is necessary since we cache and unbox global variables into the activation record during trace execution.

Since both call-stack access and global variable access are rarely performed by host functions, performance is not significantly affected by these safety mechanisms.

Another problem is that external functions can reenter the interpreter by calling scripts, which in turn again might want to access the call stack or global variables. To address this problem, we made the VM set a flag whenever the interpreter is reentered while a compiled trace is running.

Every call to an external function then checks this flag and exits the trace immediately after returning from the external function call if it is set. There are many external functions that seldom or never reenter, and they can be called without problem, and will cause trace exit only if necessary.

The FFI's boxed value array requirement has a performance cost, so we defined a new FFI that allows C functions to be annotated with their argument types so that the tracer can call them directly, without unnecessary argument conversions.

Currently, we do not support calling native property get and set override functions or DOM functions directly from trace. Support is planned future work.

6.6 Correctness

During development, we had access to existing JavaScript test suites, but most of them were not designed with tracing VMs in mind and contained few loops.

One tool that helped us greatly was Mozilla's JavaScript fuzz tester, JSFUNFUZZ, which generates random JavaScript programs by nesting random language elements. We modified JSFUNFUZZ to generate loops, and also to test more heavily certain constructs we suspected would reveal flaws in our implementation. For example, we suspected bugs in TraceMonkey's handling of type-unstable loops and heavily branching code, and a specialized fuzz tester indeed revealed several regressions which we subsequently corrected.

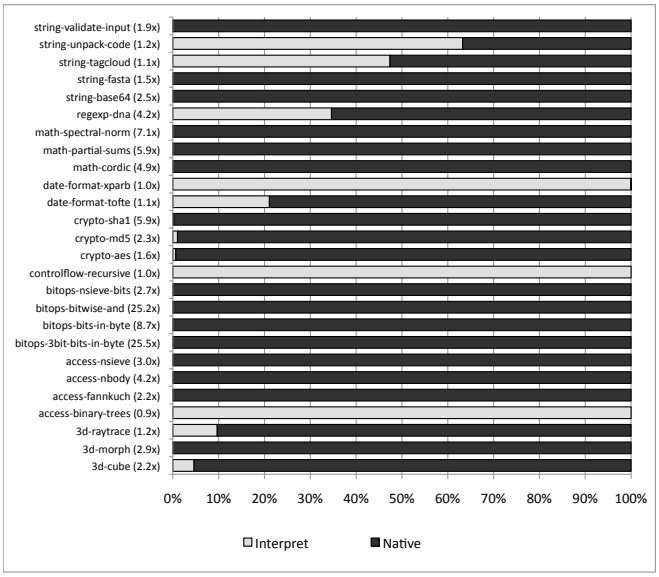

Figure 11. Fraction of dynamic bytecodes executed by interpreter and on native traces. The speedup vs. interpreter is shown in parentheses next to each test. The fraction of bytecodes executed while recording is too small to see in this figure, except for `crypto-md5`, where fully 3% of bytecodes are executed while recording. In most of the tests, almost all the bytecodes are executed by compiled traces. Three of the benchmarks are not traced at all and run in the interpreter.

7. Evaluation

We evaluated our JavaScript tracing implementation using SunSpider, the industry standard JavaScript benchmark suite. SunSpider consists of 26 short-running (less than 250ms, average 26ms) JavaScript programs. This is in stark contrast to benchmark suites such as SpecJVM98 (3) used to evaluate desktop and server Java VMs. Many programs in those benchmarks use large data sets and execute for minutes. The SunSpider programs carry out a variety of tasks, primarily 3d rendering, bit-bashing, cryptographic encoding, math kernels, and string processing.

All experiments were performed on a MacBook Pro with 2.2 GHz Core 2 processor and 2 GB RAM running MacOS 10.5.

Benchmark results. The main question is whether programs run faster with tracing. For this, we ran the standard SunSpider test driver, which starts a JavaScript interpreter, loads and runs each program once for warmup, then loads and runs each program 10 times and reports the average time taken by each. We ran 4 different configurations for comparison: (a) SpiderMonkey, the baseline interpreter, (b) TraceMonkey, (d) SquirrelFish Extreme (SFX), the call-threaded JavaScript interpreter used in Apple's WebKit, and (e) V8, the method-compiling JavaScript VM from Google.

Figure 10 shows the relative speedups achieved by tracing, SFX, and V8 against the baseline (SpiderMonkey). Tracing achieves the best speedups in integer-heavy benchmarks, up to the 25x speedup on `bitops-bitwise-and`.

TraceMonkey is the fastest VM on 9 of the 26 benchmarks (`3d-morph`, `bitops-3bit-bits-in-byte`, `bitops-bitwise-and`, `crypto-sha1`, `math-cordic`, `math-partial-sums`, `math-spectral-norm`, `string-base64`, `string-validate-input`).