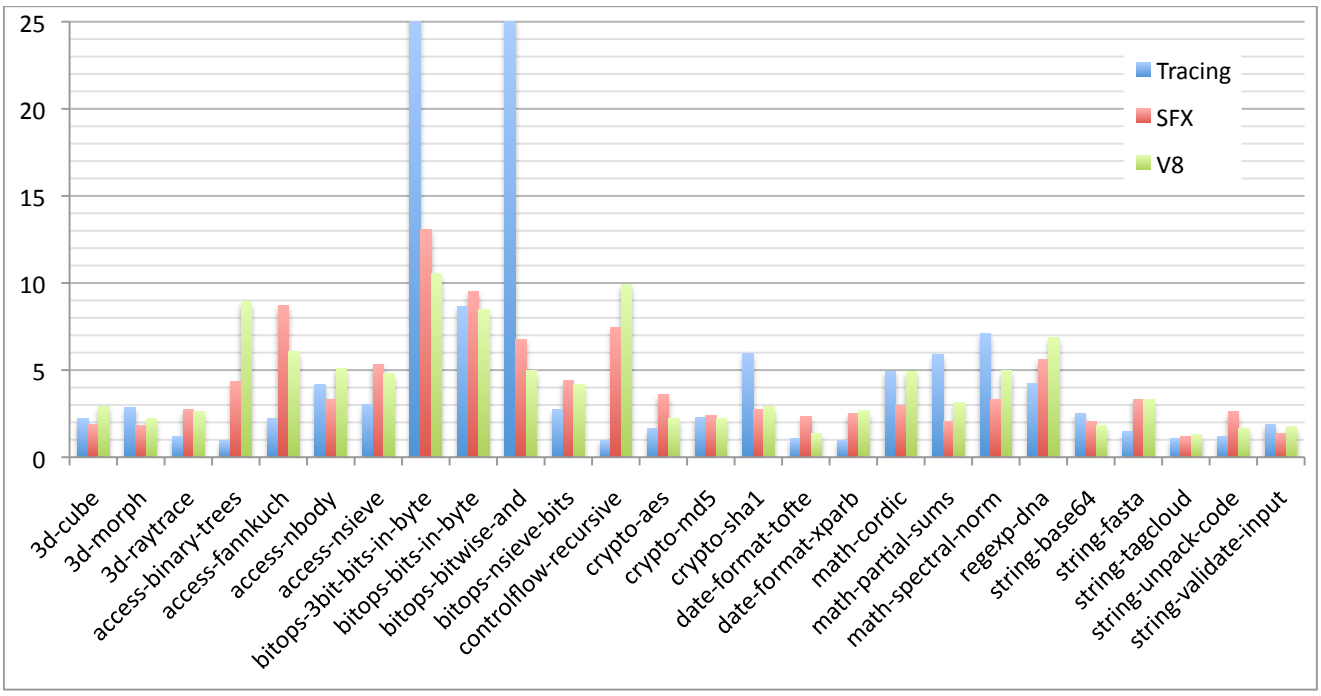

Figure 10. Speedup vs. a baseline JavaScript interpreter (SpiderMonkey) for our trace-based JIT compiler, Apple's SquirrelFish Extreme inline threading interpreter and Google's V8 JS compiler. Our system generates particularly efficient code for programs that benefit most from type specialization, which includes SunSpider Benchmark programs that perform bit manipulation. We type-specialize the code in question to use integer arithmetic, which substantially improves performance. For one of the benchmark programs we execute 25 times faster than the SpiderMonkey interpreter, and almost 5 times faster than V8 and SFX. For a large number of benchmarks all three VMs produce similar results. We perform worst on benchmark programs that we do not trace and instead fall back onto the interpreter. This includes the recursive benchmarks `access-binary-trees` and `control-flow-recursive`, for which we currently don't generate any native code.

In particular, the `bitops` benchmarks are short programs that perform many bitwise operations, so TraceMonkey can cover the entire program with 1 or 2 traces that operate on integers. TraceMonkey runs all the other programs in this set almost entirely as native code.

`regexp-dna` is dominated by regular expression matching, which is implemented in all 3 VMs by a special regular expression compiler. Thus, performance on this benchmark has little relation to the trace compilation approach discussed in this paper.

TraceMonkey's smaller speedups on the other benchmarks can be attributed to a few specific causes:

- The implementation does not currently trace recursion, so TraceMonkey achieves a small speedup or no speedup on benchmarks that use recursion extensively: `3d-cube`, `3d-raytrace`, `access-binary-trees`, `string-tagcloud`, and `controlflow-recursive`.
- The implementation does not currently trace `eval` and some other functions implemented in C. Because `date-format-tofte` and `date-format-xparb` use such functions in their main loops, we do not trace them.
- The implementation does not currently trace through regular expression `replace` operations. The replace function can be passed a function object used to compute the replacement text. Our implementation currently does not trace functions called as replace functions. The run time of `string-unpack-code` is dominated by such a `replace` call.
- Two programs trace well, but have a long compilation time. `access-nbody` forms a large number of traces (81). `crypto-md5` forms one very long trace. We expect to improve performance on this programs by improving the compilation speed of nanojit.
- Some programs trace very well, and speed up compared to the interpreter, but are not as fast as SFX and/or V8, namely `bitops-bits-in-byte`, `bitops-nsieve-bits`, `access-fannkuch`, `access-nsieve`, and `crypto-aes`. The reason is not clear, but all of these programs have nested loops with small bodies, so we suspect that the implementation has a relatively high cost for calling nested traces. `string-fasta` traces well, but its run time is dominated by string processing builtins, which are unaffected by tracing and seem to be less efficient in SpiderMonkey than in the two other VMs.

Detailed performance metrics. In Figure 11 we show the fraction of instructions interpreted and the fraction of instructions executed as native code. This figure shows that for many programs, we are able to execute almost all the code natively.

Figure 12 breaks down the total execution time into four activities: interpreting bytecodes while not recording, recording traces (including time taken to interpret the recorded trace), compiling traces to native code, and executing native code traces.

These detailed metrics allow us to estimate parameters for a simple model of tracing performance. These estimates should be considered very rough, as the values observed on the individual benchmarks have large standard deviations (on the order of the