| | Loops | Trees | Traces | Aborts | Flushes | Trees/Loop | Traces/Tree | Traces/Loop | Speedup |
|---|---|---|---|---|---|---|---|---|---|
| 3d-cube | 25 | 27 | 29 | 3 | 0 | 1.1 | 1.1 | 1.2 | 2.20x |
| 3d-morph | 5 | 8 | 8 | 2 | 0 | 1.6 | 1.0 | 1.6 | 2.86x |
| 3d-raytrace | 10 | 25 | 100 | 10 | 1 | 2.5 | 4.0 | 10.0 | 1.18x |
| access-binary-trees | 0 | 0 | 0 | 5 | 0 | - | - | - | 0.93x |
| access-fannkuch | 10 | 34 | 57 | 24 | 0 | 3.4 | 1.7 | 5.7 | 2.20x |
| access-nbody | 8 | 16 | 18 | 5 | 0 | 2.0 | 1.1 | 2.3 | 4.19x |
| access-nsieve | 3 | 6 | 8 | 3 | 0 | 2.0 | 1.3 | 2.7 | 3.05x |
| bitops-3bit-bits-in-byte | 2 | 2 | 2 | 0 | 0 | 1.0 | 1.0 | 1.0 | 25.47x |
| bitops-bits-in-byte | 3 | 3 | 4 | 1 | 0 | 1.0 | 1.3 | 1.3 | 8.67x |
| bitops-bitwise-and | 1 | 1 | 1 | 0 | 0 | 1.0 | 1.0 | 1.0 | 25.20x |
| bitops-nsieve-bits | 3 | 3 | 5 | 0 | 0 | 1.0 | 1.7 | 1.7 | 2.75x |
| controlflow-recursive | 0 | 0 | 0 | 1 | 0 | - | - | - | 0.98x |
| crypto-aes | 50 | 72 | 78 | 19 | 0 | 1.4 | 1.1 | 1.6 | 1.64x |
| crypto-md5 | 4 | 4 | 5 | 0 | 0 | 1.0 | 1.3 | 1.3 | 2.30x |
| crypto-sha1 | 5 | 5 | 10 | 0 | 0 | 1.0 | 2.0 | 2.0 | 5.95x |
| date-format-tofte | 3 | 3 | 4 | 7 | 0 | 1.0 | 1.3 | 1.3 | 1.07x |
| date-format-xparb | 3 | 3 | 11 | 3 | 0 | 1.0 | 3.7 | 3.7 | 0.98x |
| math-cordic | 2 | 4 | 5 | 1 | 0 | 2.0 | 1.3 | 2.5 | 4.92x |
| math-partial-sums | 2 | 4 | 4 | 1 | 0 | 2.0 | 1.0 | 2.0 | 5.90x |
| math-spectral-norm | 15 | 20 | 20 | 0 | 0 | 1.3 | 1.0 | 1.3 | 7.12x |
| regexp-dna | 2 | 2 | 2 | 0 | 0 | 1.0 | 1.0 | 1.0 | 4.21x |
| string-base64 | 3 | 5 | 7 | 0 | 0 | 1.7 | 1.4 | 2.3 | 2.53x |
| string-fasta | 5 | 11 | 15 | 6 | 0 | 2.2 | 1.4 | 3.0 | 1.49x |
| string-tagcloud | 3 | 6 | 6 | 5 | 0 | 2.0 | 1.0 | 2.0 | 1.09x |
| string-unpack-code | 4 | 4 | 37 | 0 | 0 | 1.0 | 9.3 | 9.3 | 1.20x |
| string-validate-input | 6 | 10 | 13 | 1 | 0 | 1.7 | 1.3 | 2.2 | 1.86x |

**Figure 13.** Detailed trace recording statistics for the SunSpider benchmark set.

mean). We exclude `regexp-dna` from the following calculations, because most of its time is spent in the regular expression matcher, which has much different performance characteristics from the other programs. (Note that this only makes a difference of about 10% in the results.) Dividing the total execution time in processor clock cycles by the number of bytecodes executed in the base interpreter shows that on average, a bytecode executes in about 35 cycles. Native traces take about 9 cycles per bytecode, a 3.9x speedup over the interpreter.

Using similar computations, we find that trace recording takes about 3800 cycles per bytecode, and compilation 3150 cycles per bytecode. Hence, during recording and compiling the VM runs at 1/200 the speed of the interpreter. Because it costs 6950 cycles to compile a bytecode, and we save 26 cycles each time that code is run natively, we break even after running a trace 270 times.

The other VMs we compared with achieve an overall speedup of 3.0x relative to our baseline interpreter. Our estimated native code speedup of 3.9x is significantly better. This suggests that our compilation techniques can generate more efficient native code than any other current JavaScript VM.

These estimates also indicate that our startup performance could be substantially better if we improved the speed of trace recording and compilation. The estimated 200x slowdown for recording and compilation is very rough, and may be influenced by startup factors in the interpreter (e.g., caches that have not warmed up yet during recording). One observation supporting this conjecture is that in the tracer, interpreted bytecodes take about 180 cycles to run. Still, recording and compilation are clearly both expensive, and a better implementation, possibly including redesign of the LIR abstract syntax or encoding, would improve startup performance.

Our performance results confirm that type specialization using trace trees substantially improves performance. We are able to outperform the fastest available JavaScript compiler (V8) and the fastest available JavaScript inline threaded interpreter (SFX) on 9 of 26 benchmarks.

## 8. Related Work

**Trace optimization for dynamic languages.** The closest area of related work is on applying trace optimization to type-specialize dynamic languages. Existing work shares the idea of generating type-specialized code speculatively with guards along interpreter traces.

To our knowledge, Rigo's Psyco (16) is the only published type-specializing trace compiler for a dynamic language (Python). Psyco does not attempt to identify hot loops or inline function calls. Instead, Psyco transforms loops to mutual recursion before running and traces all operations.

Pall's LuaJIT is a Lua VM in development that uses trace compilation ideas. (1). There are no publications on LuaJIT but the creator has told us that LuaJIT has a similar design to our system, but will use a less aggressive type speculation (e.g., using a floating-point representation for all number values) and does not generate nested traces for nested loops.

**General trace optimization.** General trace optimization has a longer history that has treated mostly native code and typed languages like Java. Thus, these systems have focused less on type specialization and more on other optimizations.

Dynamo (7) by Bala et al, introduced native code tracing as a replacement for profile-guided optimization (PGO). A major goal was to perform PGO online so that the profile was specific to the current execution. Dynamo used loop headers as candidate hot traces, but did not try to create loop traces specifically.

Trace trees were originally proposed by Gal et al. (11) in the context of Java, a statically typed language. Their trace trees actually inlined parts of outer loops within the inner loops (because