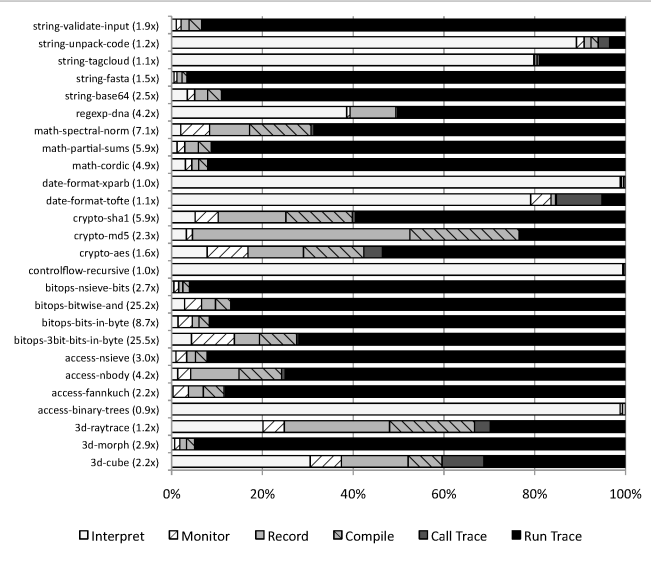


Figure 12. Fraction of time spent on major VM activities. The speedup vs. interpreter is shown in parentheses next to each test. Most programs where the VM spends the majority of its time running native code have a good speedup. Recording and compilation costs can be substantial; speeding up those parts of the implementation would improve SunSpider performance.

inner loops become hot first), leading to much greater tail duplication.

YETI, from Zaleski et al. (19) applied Dynamo-style tracing to Java in order to achieve inlining, indirect jump elimination, and other optimizations. Their primary focus was on designing an interpreter that could easily be gradually re-engineered as a tracing VM.

Suganuma et al. (18) described region-based compilation (RBC), a relative of tracing. A region is an subprogram worth optimizing that can include subsets of any number of methods. Thus, the compiler has more flexibility and can potentially generate better code, but the profiling and compilation systems are correspondingly more complex.

Type specialization for dynamic languages. Dynamic language implementors have long recognized the importance of type specialization for performance. Most previous work has focused on methods instead of traces.

Chambers et. al (9) pioneered the idea of compiling multiple versions of a procedure specialized for the input types in the language Self. In one implementation, they generated a specialized method online each time a method was called with new input types. In another, they used an offline whole-program static analysis to infer input types and constant receiver types at call sites. Interestingly, the two techniques produced nearly the same performance.

Salib (17) designed a type inference algorithm for Python based on the Cartesian Product Algorithm and used the results to specialize on types and translate the program to C++.

McCloskey (14) has work in progress based on a language-independent type inference that is used to generate efficient C implementations of JavaScript and Python programs.

Native code generation by interpreters. The traditional interpreter design is a virtual machine that directly executes ASTs or machine-code-like bytecodes. Researchers have shown how to generate native code with nearly the same structure but better performance.

Call threading, also known as context threading (8), compiles methods by generating a native call instruction to an interpreter method for each interpreter bytecode. A call-return pair has been shown to be a potentially much more efficient dispatch mechanism than the indirect jumps used in standard bytecode interpreters.

Inline threading (15) copies chunks of interpreter native code which implement the required bytecodes into a native code cache, thus acting as a simple per-method JIT compiler that eliminates the dispatch overhead.

Neither call threading nor inline threading perform type specialization.

Apple's SquirrelFish Extreme (5) is a JavaScript implementation based on call threading with selective inline threading. Combined with efficient interpreter engineering, these threading techniques have given SFX excellent performance on the standard SunSpider benchmarks.

Google's V8 is a JavaScript implementation primarily based on inline threading, with call threading only for very complex operations.

9. Conclusions

This paper described how to run dynamic languages efficiently by recording hot traces and generating type-specialized native code. Our technique focuses on aggressively inlined loops, and for each loop, it generates a tree of native code traces representing the paths and value types through the loop observed at run time. We explained how to identify loop nesting relationships and generate nested traces in order to avoid excessive code duplication due to the many paths through a loop nest. We described our type specialization algorithm. We also described our trace compiler, which translates a trace from an intermediate representation to optimized native code in two linear passes.

Our experimental results show that in practice loops typically are entered with only a few different combinations of value types of variables. Thus, a small number of traces per loop is sufficient to run a program efficiently. Our experiments also show that on programs amenable to tracing, we achieve speedups of 2x to 20x.

10. Future Work

Work is underway in a number of areas to further improve the performance of our trace-based JavaScript compiler. We currently do not trace across recursive function calls, but plan to add the support for this capability in the near term. We are also exploring adoption of the existing work on tree recompilation in the context of the presented dynamic compiler in order to minimize JIT pause times and obtain the best of both worlds, fast tree stitching as well as the improved code quality due to tree recompilation.

We also plan on adding support for tracing across regular expression substitutions using lambda functions, function applications and expression evaluation using `eval`. All these language constructs are currently executed via interpretation, which limits our performance for applications that use those features.

Acknowledgments

Parts of this effort have been sponsored by the National Science Foundation under grants CNS-0615443 and CNS-0627747, as well as by the California MICRO Program and industrial sponsor Sun Microsystems under Project No. 07-127.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Any opinions, findings, and conclusions or recommendations expressed here are those of the author and should