STOKE: A Stochastic Optimizer

STOKE is a prototype implementation of a stochastic optimizer. STOKE differs from a traditional compiler by using random search to discover the space of optimal assembly code sequences, including those that are an incorrect implementation of the input program. STOKE is designed to experiment with shortcuts through temporally incorrect transformations that can produce code sequences that a compiler that insists on preserving correctness might miss.

The code discovered by STOKE often occupies an entirely different code space than the code produced by a general purpose compiler. Often, it has the property of small equality preserving transformations that can connect it to the original code. It represents a completely distinct algorithm at the assembly level, one which might require that the input values be permuted and relocated to distinguish register locations to permit the use of hardware intrinsics or take advantage of some other complex features of a CISC instruction set.

STOKE search is divided into two distinct phases: a stochastic optimizer. STOKE differs from a numeric simulation and a ray tracer by providing end-to-end speedups of over 30% on a direct comparison against the outputs produced by the input program.

Verification

Because our ultimate goal is correctness, it is necessary to periodically verify results using a formal proof of equivalence. We use a standard technique for verifying that a split point must exist. STOKE uses a standard technique for verifying the equivalence of loop free programs. Both the input program and a candidate rewrite are verified for the equivalence of loop free programs. STOKE treats the difference in floating-point programs as an error function E(x) = |f(x)-f'(x)|. We use an SMT solver to prove that there does not exist an x that satisfies E(x) > E_ref.

Boundary Conditions: STOKE assumes that the input program and a rewrite must begin and execute in identical states (Figure 5 top).

Loop Splitting: STOKE examines test case logs and finds the locations where identical values appear in both executions (Figure 5 middle), whenever identical values appear, STOKE guesses that a split point must exist.

Proof: STOKE uses the boundary conditions and split points to attempt an inductive proof of correctness that the loop in proofs relating to the equivalence of programs that contain loops.

Floating Point

STOKE addresses this issue by using a relaxed definition of correctness. STOKE only penalizes candidate rewrites that produce results that exceed a user-defined upper bound on error. Using this modified definition, STOKE is able to discover sequences of transformations that result in optimized programs with weaker precision guarantees than the input program.

Experimental Results

Beginning from fixed-point binaries compiled with gcc -O3, STOKE is able to produce hit-wise correct optimizations that match or outperform the code produced by gcc and icc with full optimizations enabled, and in some cases expert hand-written code. The improvement often results from the space of possible rewrites that STOKE can explore. Although our technique does not provide a formal guarantee of correctness, it does provide strong evidence. For the benchmarks that we consider, we are unable to refute the resulting error bounds even after extensive further testing.

Figure 6. Floating-point instruction sets are complicated semantics and few opportunities for optimization exist for any particular implementation.

Figure 7. Relaxing the definition of correctness allows STOKE to increase the size of regions of correct implementation and find a path to previously inaccessible optimizations.