Introduction

We describe an optimization pipeline built for our implementation of the ekcc program. We will discuss the pipeline itself and our testing strategy for the pipeline. We then examine the effects of different optimization strategies on compilation time and program speed, and the trade-offs between these two.

Optimization Pipeline

Our optimization pipeline consists of one analysis pass to perform basic alias analysis, and twelve code transformation passes. Each of these passes can be separately included in the optimization pipeline via a command-line argument to the compiler.

We divided the twelve code transformation passes into four separate groups based on their high-level effects on the generated IR:

- Those affecting function execution (argument promotion and function inlining)
- Those affecting control flow and dead code (CFG simplification, dead code elimination, and dead prototype stripping)
- Those doing mostly-local algebraic transformations on the bytecode itself (instruction combining and promotion of memory accesses to register usage)
- Miscellaneous transformations primarily affecting loops and constants (induction variable simplification, loop vectorization, expression reassociation, SCCP transformations, and dead argument elimination once the pipeline has completed)

Because the alias analysis pass does not directly change the generated IR, this pass was included in our tests for each other grouping, but not tested on its own.

Of these groupings, we anticipated that those including function inlining, control flow, and memory vs. register usage would have the largest effects on overall running time, so these were tested separately from one another. We also tested the entire pipeline by combining each of these four groups.

We tried repeating the optimizations a second time as well, to see if we could do something with fixed-point. We got very limited gains from this with a tradeoff of massively increased pipeline runtime, almost 10x on the all-optimizations pipeline. Bigger and more complex code would probably see more speedup, but our code was mostly pretty linear with one-deep function calls, so multiple passes didn't do a ton (2.1x speedup to 2.2x speedup, or 1.1 to 1.3 in another case, were some of our biggest gains across all tests).

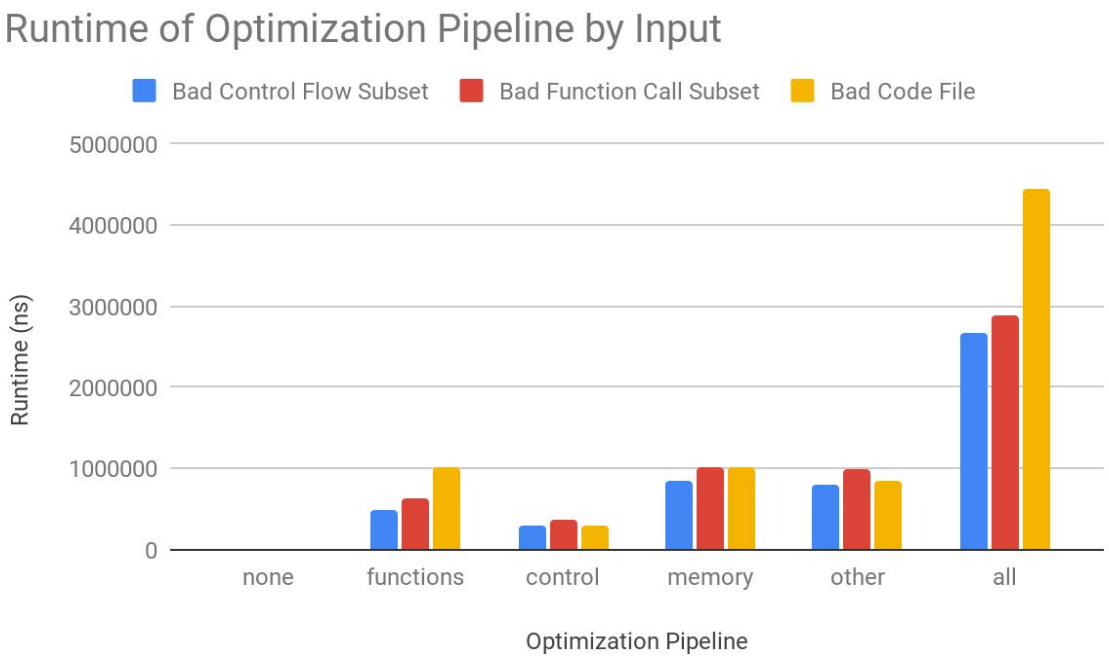
Test Programs

For our tests, we tried three different ek files, all derived from one large file where we came up with every possible "bad coding practice" that we could imagine that a compiler might catch. We targeted every single one of our optimizations, and added tests that would optimize or not depending on the order of optimizations (for example, there were some parts in the file that would be slow if inlining happened before constant propagation, or if constant propagation

happened before inlining). The functions we called were intended to be extremely slow if LLVM did not optimize the targeted part of the code, and extremely quick if it did – the optimization targets were placed in ways that caused the program to either do a lot of busywork or not, depending on whether the optimization took place.

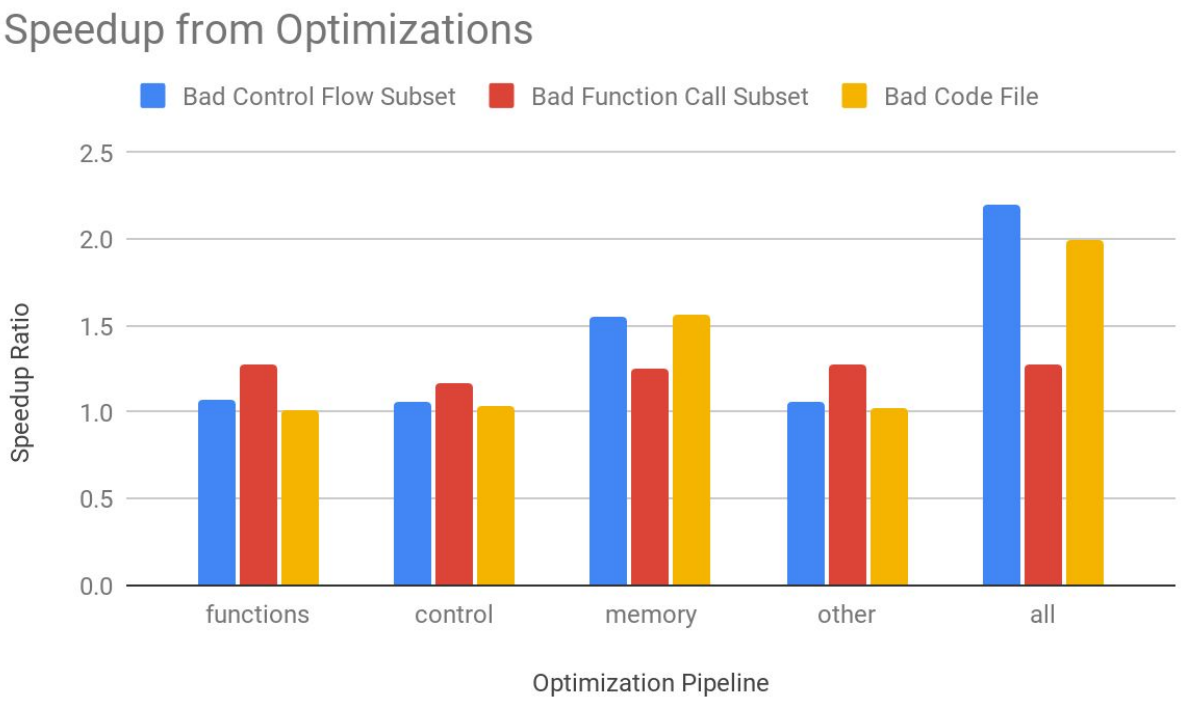
We then tested each of the four optimization groupings discussed above, as well as no optimizations at all and the full pipeline, on each input program.

Runtime of Optimization Pipeline: Data and Analysis



For the most part, the amount of time it took to run the transformations scaled with the amount of code, and did so additively across the optimizations. The first two test files are a subset of the third, so they are less than or equal to the runtime of the third. Functions and Control scaled with the amount of functions and if/while statements in the code as a proportion of the total. Memory generally scaled with the amount of referencing and math done, and other scaled with the usage of loops. All took place after memory-to-register stuff and function inlining, so with all, there was much more referencing and inlining, and therefore many more instructions to look over at the memory and "other" stages, where algebraic transformations would happen. This means LLVM had to look separately at all the inlined functions' instructions, so the time increased quite a bit for longer code. In the future it might be better to do the other pass, then inline, then do it again- this might actually speed things up significantly because the inlined code would end up smaller and simpler.

Speedup: Data and Analysis



Almost everything sped up some with all transformations. Function inlining alone and algebraic analysis alone did not help much, because there were many function calls throughout that would only benefit from algebraic or control flow passes after inlining and memory analysis. The memory-to-register stuff really helped with speed after alias analysis. Control flow significantly reduced code size, but did not really help with runtime, because it was mostly pruning away quick but useless checks and dead code.

Trade-offs Between Compile Time and Speed

We note that while two of our test cases gained roughly an additional 50% speedup in running time by using the full pipeline instead of the next-best single grouping, every one of our test cases required between 3 and 4 times the amount of time to complete the full optimization pipeline as opposed to the next-best grouping.

These data suggest that in situations where compilation time and overall running time are both major concerns, it could be beneficial to restrict the optimization pipeline to memory-to-register promotion and, for programs which make repeated, small function calls, function inlining. In situations where the running time of the program is the primary concern, especially if this time is very long or the program needs to be run many times, it may be preferable to run the complete optimization pipeline.