Go 1.22 inlining overhaul

Austin Clements, with contributions from Damien Neil, Matthew Dempsky, Michael Pratt, and Than McIntosh
Last update: Jul 20, 2023

The Go compiler's inliner has never been particularly good. It wasn't until Go 1.12, released in 2019, that the Go compiler supported inlining more than leaf functions, and we've slowly chipped away at more limitations of the inliner over the years (it started inlining functions with for loops in early 2021!). Go 1.20, released in February 2023, added support for basic profile-guided inlining, the most significant change to Go's inlining policy since 1.12.

The backend of the inliner—the component that implements the decisions made by the inlining policy—received a major overhaul with unified IR in Go 1.20. In Go 1.21, the old backend was deleted entirely, which eliminated almost all backend limitations that have suffused the inlining policy for years.

Our current inlining policy remains built on a foundation that is becoming increasingly strained as we add things like PGO, is increasingly anchored in past backend limitations, and it continues to use an overly simplistic cost model driven by an overly simplistic scheduler. Between unified IR and the untapped possibilities of PGO, I believe there's now a significant opportunity to improve the inlining policy, resulting in significant performance improvements for Go applications, and reducing the effort and expertise needed to write highly efficient Go code.

The rest of this document lays out a set of considerations for a redesign of Go's inlining policy.

Obvious improvements

If we know there's only one call to a function and it's possible to inline, inline it. A trivial case of this is: `func() {...}()`. This would also be easy to analyze for unexported functions. There are some potential downsides to doing this in the general case: it may present an undesirable performance cliff where adding a second reference to a function suddenly prevents inlining, or doing it only for unexported functions violates our goal of not having performance penalties on package boundaries.

Heuristic improvements

Inlining heuristics determine whether or not to inline a given call edge. Our current inliner uses a simple cost model where it computes the "hairiness" of a function, which is roughly the number of AST nodes in it, and inlines anything under a certain threshold.

Prefer inlining if it enables follow-up optimizations. The current heuristic solely models the *costs* of inlining, and not the *value* of inlining. This makes it extremely conservative. In effect, this models the only value as eliminating call overhead, but most of the value of inlining comes from enabling further optimizations, particularly constant propagation (and subsequent dead-code elimination), call devirtualization, and escape analysis. If we could model this value, we could accept cases with a medium cost but a high value. Cherry proposed a simple but expensive way to do this: make two copies of a caller, one in which we do inlining and another in which we don't, pass both through further phases of the compiler, and at some point after key optimizations have been applied, pick between them. Alternatively, we could attempt to model (perhaps approximately) these key optimizations directly in the inlining policy.

Better PGO heuristics. Currently, PGO essentially works by raising the hairiness threshold for hot functions. Even this fairly rudimentary approach achieves a 3–4% speedup, but with further work I'm sure we could improve this. Improving the inliner scheduling is one thing that would help us make better use of PGO data. Another possibility is that we use PGO not just to directly drive inlining decisions, but to direct where the compiler spends time applying more costly inlining heuristics, such as trying follow-up optimization passes.

Call site-aware heuristics. The current inliner is completely insensitive to the caller: a callee is either inlined everywhere or nowhere. The call site, however, can significantly affect the value of a particular inlining decision. For example, it's generally more valuable to inline a call that's performed in a loop because the cumulative overhead of that call is higher, and because it may enable loop-invariant code motion. Many other follow-up optimizations are also sensitive to the caller: constant propagation depends on constant arguments at a call site, and escape analysis depends on whether a value further escapes the caller. One case where non-trivial constant propagation should encourage inlining is when closures are passed to a function. For example, sort.Search is a small function that is almost always called with a function literal. Ideally we would inline sort.Search into such callers and perhaps even inline the function literal, resulting in optimal code at no cost to expressiveness.

Consider performing partial escape analysis before inlining. Currently, we perform inlining before escape analysis because it significantly affects the results of escape analysis. However, it would be valuable to have information from escape analysis available to inlining. It might be possible to perform partial escape analysis first to produce data flow graphs that can then be quickly combined following inlining and other basic optimizations (like dead code elimination) before being finalized into function-level escape summaries.

Consider moving inlining to SSA. The current inliner works on our AST representation, which makes it difficult to accurately model costs because some simple ASTs produce a large amount of code and some complex ASTs optimize to very little code. Moving it to SSA would enable a more accurate cost model. However, there are many complications to doing this, so it may not be worthwhile: we would have to either move escape analysis to SSA or find a way to perform escape analysis before inlining (see above); SSA compilation is currently entirely parallel, and this would add significant ordering constraints (though we have experimented with SCC ordering of SSA compilation and found it has little negative impact); and we currently have no way to serialize SSA to the export data.

Scheduling improvements

The scheduler determines the order in which inlining considers call edges. Our current inliner does a strict bottom-up traversal of the call graph (technically, of the strongly-connected components graph), inlining callees into callers until a threshold is reached, then starting over with the next caller up the chain. Even in a fixed-threshold model, this is suboptimal, and it leads to unstable results where small changes to functions near the leaves of the call graph can lead to completely different inlining boundaries as you go up the call chain. This algorithm originated before we supported mid-stack inlining, and it still reflects this limited model of inlining.

Cost-based inline scheduling. The current bottom-up approach is suboptimal and unstable. An obvious possibility is to compute the local cost of every function, then start with the lowest-cost functions and inline those into their parents (recomputing the cost of the combined function) and keep going from there until every remaining call edge would exceed the inlining threshold. This is very similar to building a Huffman tree. It would have dramatically better stability and would likely result in more optimal results. It may also be a prerequisite for certain heuristic improvements. For example, a common pattern in Go is to split computations into a small fast path function and a large slow path function, where the fast path is intended to be inlined and calls the large function if the fast path conditions aren't satisfied. In a bottom-up schedule, there's always a danger that strong heuristics will allow inlining the large slow path function into the fast path function (for example, if we unconditionally inline functions with a single call site) but the combined function will not be inline-able, defeating the whole purpose. A cost-driven scheduler will consider inlining the fast path into its callers before considering inlining the slow path into the fast path. Cost-driven scheduling would also handle calls within strongly-connected components much better than our current approach (e.g., #58905).

Call site-aware scheduling. Much like call site-aware heuristics, the scheduler could also benefit from being call site-aware. For example, it could start by inlining the highest value call sites in a caller and stop once the caller goes over a size threshold (at which point i-cache pressure increases).

References

CL 451356 (PS 1) encountered two bugs that prevented inlining and make a cleaner sync.Once* API slower than the hard-to-use API. (TODO: File actual bugs for these.)

#52193

#38992

CL 459037 duplicates sort.Search logic into slices.BinarySearch and gets a substantial performance improvement.

CL 495595 disallows inlining from a norace package into a race package (in -race mode) because we track this at the package level and lose this information across inlining. This is an example of a larger problem where we track information at the source function or package level, which is almost always hostile to inlining. We may want to push more of this into IR nodes so it doesn’t block inlining.