orphan:

Whole Module Optimization in Swift

State of Whole Module Optimization in Swift 2.0

Since Swift 1.2 / Xcode 6.3, the Swift optimizer has included support for whole module optimization (WMO).

To date (Swift 2.0 / Xcode 7), the differences in the optimization pipeline and specific optimization passes when WMO is enabled have been relatively minimal, and have provided high value at low implementation cost. Examples of this include inferring final on internal methods, and removing functions that are not referenced within the module and cannot be referenced from outside the module.

Additionally, compiling with WMO has some natural consequences that require no enhancements to passes. For example the increased scope of compilation that results from having the entire module available makes it possible for the inliner to inline functions that it would otherwise not be able to inline in normal separate compilation. Other optimizations similarly benefit, for example generic specialization (since it has more opportunities for specialize) and function signature optimization (since it has more call sites to rewrite).

Whole Module Optimization for Swift.Next

As it stands, WMO provides significant benefit with minimal complexity, but also has many areas in which it can be improved. Some of these areas for improvement are architectural, e.g. improvements to the pass management scheme, while others involve adding new interprocedural analyses and updating existing passes to make use of the results of these analyses.

Passes and Pass Management

Our current pass management scheme intersperses module passes and function passes in a way that results in module passes being run on functions that are partially optimized, which is suboptimal. Consider inlining as one example. If we run the inliner when callee functions are only partially optimized we may make different inlining decisions in a caller than if we ran it on a caller only after its callees have been fully optimized. This particular issue and others mentioned below are not specific to WMO per-se, but are more generally a problem for any interprocedural optimization that we currently do (most of which happen in per-file builds as well).

This also affects interprocedural optimizations where we can compute summary information for a function and use that information when optimizing callers of that function. We can obtain better results by processing strongly connected components (SCCs) of the call graph rather than individual functions, and running the full optimization pipeline on a given SCC before moving to the next SCC in a post-order traversal of the call graph (i.e. bottom-up). A similar approach can be taken for running transforms that benefit from information that is propagated in reverse-post-order in the call graph (i.e. top-down).

Processing one SCC at a time versus one function at a time is primarily for the benefit of improved analyses. For example consider escape analysis. If there is an SCC in the call graph and we process one function at a time, there are cases where we would have to be pessimistic and assume a value escapes, when in fact the value may be used within the SCC such that it never escapes. The same pessimism can happen in other analyses, e.g. dead argument analysis.

In our current set of passes, several are implemented as SILModuleTransforms but simply iterate over the functions in the module in whatever order they happen to be in and do not appear to benefit from being module passes at this time (although some would benefit from optimizing the functions in bottom-up order in the call graph). Other SILModuleTransforms currently walk the functions bottom-up in the call graph, and do gain some benefit from doing so.

Moving forward we should eliminate the notion of module passes and instead have SCC passes as well as the existing function passes. We should change the pass manager to run the SCC passes as a bottom-up traversal of the call graph. As previously mentioned we may also want to consider having the flexibility of being able to run passes in a top-down order (so we could create a pass manager with passes that benefit from running in this order, not because we would want to run any arbitrary pass in that order).

For each SCC we would run the entire set of passes before moving on to the next SCC, so when we go to optimize an SCC any functions that it calls (when going bottom-up - or calls into it when going top-down) have been fully optimized. As part of this, analyses that benefit from looking at an SCC-at-a-time will need to be modified to do so. Existing analyses that would benefit from looking at an SCC-at-a-time but have not yet been updated to do so can be run on each function in the SCC in turn, producing potentially pessimistic results. In time these can be updated to analyze an entire SCC. Likewise function passes would be run on each function in the SCC in turn. SCC function passes would be handed an entire SCC and be able to ask for the analysis results for that SCC, but can process each function individually based on those results.

TBD: Do we really need SCC transforms at all, or is it sufficient to simply have function transforms that are always passed an SCC, and have them ask for the results of an analysis for the entire SCC and then iterate over all functions in the SCC?

In some cases we have transforms that generate new work in a top-down fashion, for example the devirtualizer as well as any pass that clones. These can be handled by allowing function passes to push new work at the top of the stack of work items, and then upon finishing a pass the pass pipeline will be restarted with those new functions at the top of the work stack, and the previous function buried beneath them, to be reprocessed after all the callees are processed.

TBD: This may result in re-running some early passes multiple times for any given function, and it may mean we want to

front-load the passes that generate new callees like this so that they are early in the pipeline. We could also choose not to rerun portions of the pipeline on the function that's already had some processing done on it.*

SCC-based analyses

There are a variety of analyses that can be done on an SCC to produce better information than can be produced by looking at a single function at a time, even when processing in (reverse-)post-order on the call graph. For example, a dead argument analysis can provide information about arguments that are not actually used, making it possible for optimizations like DCE and dead-store optimization (which we do as part of load/store opts) to eliminate code, independent of a pass like function signature optimization running (and thus we eliminate a phase-ordering issue). It benefits from looking at an SCC-at-a-time because some arguments getting passed into the SCC might be passed around the SCC, but are never used in any other way by a function within the SCC (and are never passed outside of the SCC).

Similarly, escape analysis, alias analysis, and mod/ref analysis benefit from analyzing an SCC rather than a function.

This list is not meant to be exhaustive, but is probably a good initial set of analyses to plan for once we have the new pass framework up.

Incremental Whole Module Optimization

Building with WMO enabled can result in slower build times due to reduced parallelization of the build process. We currently parallelize some of the last-mile optimization and code generation through LLVM, but do not attempt to parallelize any of the SIL passes (see the next section).

With that in mind, it seems very worthwhile to examine doing incremental WMO in order to minimize the amount of work done for each compile.

One way to accomplish this is to serialize function bodies after canonical SIL is produced (i.e. after the mandatory passes) and only reoptimize those functions which change between builds. Doing just this, however, is not sufficient since we've used information gleaned from examining other functions, and we've also done things like inline functions into one another. As a result, we need to track dependencies between functions in order to properly do incremental compilation during WMO.

Some approaches to tracking these dependencies could be very burdensome, requiring passes to explicitly track exactly which information they actually use during optimization. This seems error prone and difficult to maintain.

Another approach might be to recompile adjacent functions in the call graph when a given function changes. This might be somewhat practical if we only have analyses which propagate information bottom-up, but it would be more expensive than necessary, and impractical if we also have analyses that propagate information top-down since it could result in a full recompile of the module in the worst case.

A more reasonable approach would be to serialize the results of the interprocedural analyses at the end of the pass pipeline, and use these serialized results to drive some of the dependency tracking (along with some manual tracking, e.g. tracking which functions are inlined at which call sites). These serialized analysis results would then be compared against the results of running the same analyses at the end of the compilation pipeline on any function which has changed since the previous compile. If the results of an analysis changes, the functions which use the results of that analysis would also need to be recompiled.

TBD: Properly tracking dependencies for functions generated from other functions via cloning. Is this any different from tracking for inlining?

Parallel Whole Module Optimization

We could also explore the possibility of doing more work in parallel during WMO builds. For example, it may be feasible to run the SIL optimization passes in parallel. It may also be feasible to do IRGen in parallel, although there are shared mutating structures that would need to be guarded.

It's TBD whether this is actually going to be practical and worthwhile, but it seems worth investigating and scoping out the work involved to some first-level of approximation.