Building a better Go linker

Austin Clements
2019-09-12
golang.org/s/better-linker

Motivation

As the scale of software built in Go grows, both the memory consumption and performance of the Go linker have become a problem for our users. We've had a fire drill after each of the past several Go releases where slight increases in binary sizes increase the Go linker's memory footprint past critical limits and result in even longer builds. This has been observed in spades within Google, as well as externally (#26186, #26132). Linker inefficiency has blocked rolling out API improvements within Google that increase the load on the linker, and has interfered with us rolling out improvements to debuggability (#26318). Link time is an unavoidable part of build latency, since the linker is on the critical path of even the smallest incremental build and hence a key component of maintaining Go's famed build speed. The scale of systems built in Go will only continue to grow and exacerbate these issues.

This is also a matter of project health. The linker has not had a maintainer for a long time, and the codebase requires significant ramp-up time even to make seemingly small changes and fix seemingly small issues. The linker was originally written in C and converted semi-automatically to Go and there has been minimal clean-up effort since. The original linker was also simpler than it is now and its implementation fit in one Turing award winner's head, so there's little abstraction or modularity. Unfortunately, as the linker grew and evolved, it retained its lack of structure, and our sole Turing award winner retired.

However, I believe there is a great deal of opportunity here. We understand the problem space far better now. There are features of Go that make it amenable to very efficient linking algorithms, none of which we take advantage of right now. Go also controls its object file format, giving us the opportunity to innovate and co-design the compiler, object format, and linker.

Background

The Go linker's overall architecture is similar to a standard C linker, but it has several unusual properties that are useful to be aware of.

First, Go has its own object format. While one could imagine using a standard object file format, the main Go toolchain uses a custom linker for many other reasons, so there's little cost to having an object format tailored to Go, and a lot of potential upside.

The basic unit of the Go object format is a symbol, which generally represents a package-level object like a function or package variable. Each symbol has a name, a kind (data, function, etc.), contents, and a set of relocations. Function symbols carry additional metadata, most of which the linker ultimately provides to the runtime. There are also naming conventions for finding some metadata; for example, the DWARF function debugging information entry (DIE) for function symbol `X` is named `go.info.X`.

The Go linker does symbol-level reachability analysis, rather than section-level analysis like most linkers (though many compilers can be configured to emit a section per function, generally with various trade-offs). Mostly, this just follows the relocation graph, but special care is taken with methods. If the binary does not use reflection method calls, the linker performs a global program analysis to find all reachable interface types and discard methods that don't match any signatures in reachable interface types. This optimization actually applies rather broadly; for example, it applies to every program in cmd/, though it only reduces the size of the compile binary by 4%. If the binary does use reflection method calls, then the linker must keep all methods of any reachable type, since the program can call any method by constructing its string name.

The Go linker is responsible for constructing the runtime's (somewhat-misnamed) "pcln" table. This contains metadata for all functions in the final binary, including information for symbolic stack traces and data for the garbage collector such as stack maps. Doing this in the linker lets it retain metadata only for reachable functions, and gives it the opportunity to further optimize some metadata. For example, it eliminates unused file paths from the file path table and rewrites all indexes into this table.

Likewise, the Go linker constructs the final binary's DWARF tables from pieces provided by the compiler. This is complicated by the fact that DWARF is oriented around traditional compilation units, and thus not designed with symbol-level reachability in mind. For example, the DWARF line table is constructed entirely in the linker from the runtime pcln table once it knows what functions are retained.

Finally, the Go linker must produce a binary the system can run. It has two different modes for doing this. For "pure Go" binaries, it can emit a native binary in several formats. This allows developers to program in Go without installing a system toolchain. However, even pure Go binaries typically touch small amounts of C code—for example, the net package on Linux must use the system libraries for name resolution—so the linker has just enough built-in support to read and link against some known system libraries. For "cgo" binaries, which may make arbitrary use of C libraries, the Go linker links all of the Go code into a single native object file and then invokes the system linker to produce the final binary.

An analysis of the linker

As a case study, we'll consider linking cmd/compile, which is the largest binary in the standard distribution. For reference, we use tag go1.13beta1 – the latest beta release as of this writing – on linux/amd64 on a 2 core, 4 thread 2.6GHz Skylake. Many of the measurements in this section are based on the linker stats patch at golang.org/cl/185990/2 and the output of `go build -ldflags -stats=- -o /dev/null cmd/compile`. CPU profiles are based on `perflock go build -ldflags -cpuprofile=/tmp/cpu.prof -o /dev/null cmd/compile`.

Memory footprint is proportional to the sum of inputs and output

A key structural issue with the current linker is that it expects to do everything in memory. It deserializes all of the input objects into the heap and produces the output in memory as well. As a result, its peak memory footprint includes the entirety of the inputs (even if it eliminates most symbols as unreachable). Many of the linker's issues revolve around this design choice.

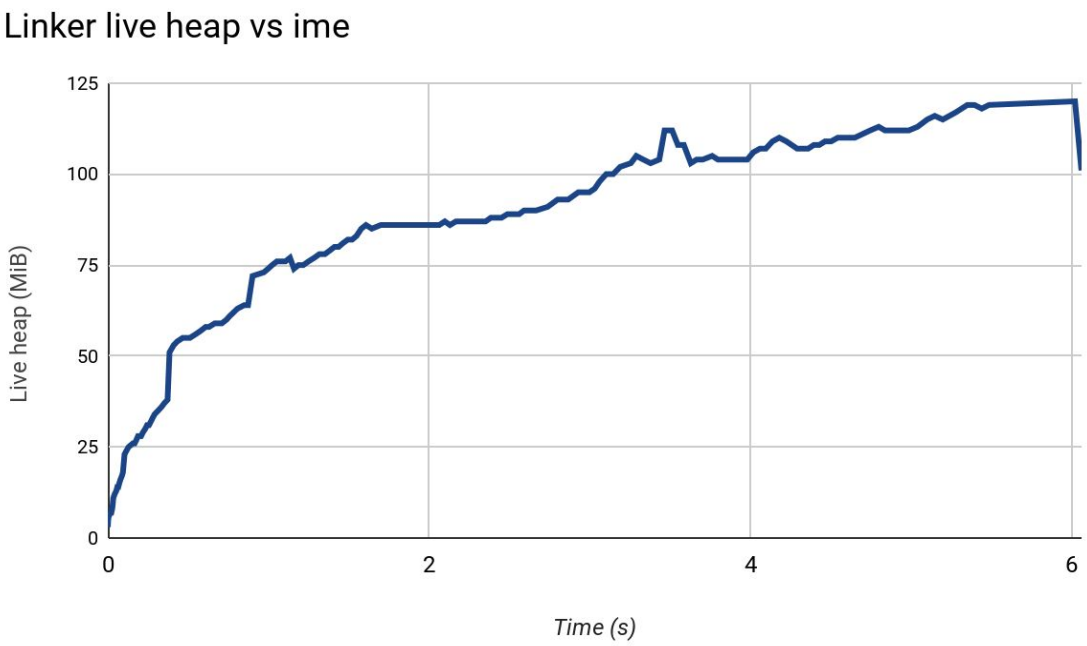

Figure 1. The linker's live heap size grows nearly monotonically, indicating that nearly all allocations are long-lived. The time scale on this graph is longer than a regular run because of measurement overhead.

Even though the cmd/compile binary is only 24MiB, the linker reads in 56MiB[1] of object files, takes roughly one second to run, allocates a total of 229MiB,[2] and reaches a peak RSS of 278MiB[3] with a peak live heap of 120MiB.[4] Its live heap grows nearly monotonically throughout the duration of the link, as shown in figure 1, indicating that the majority of allocations remain live forever.[5] This monotonic growth is a worst-case situation for garbage collector performance: the linker spends 23.5% of its cycles in GC marking, which is quite high, yet only 5% of its cycles allocating.

This behavior is after recent significant improvements: as of Go 1.13, we mmap most symbol data and symbol names, copy symbol data to an mmapped output file, and apply relocations directly to the mmapped output. While this currently suffers from the limitations of being bolted on, the approach of mmaping the input read-only, mmaping the output read/write, copying symbol data to the output, and applying relocations in the output has significantly reduced the linker's footprint. For cmd/compile, this keeps 35 MiB[6] of symbol data out of the Go heap, though the linker still synthesizes an additional 8 MiB[7] of symbol data for the output, which does live in the heap.

We should work to retain and improve this structure. For example, one serious remaining limitation is that all metadata in the object file is still deserialized into memory, including relocations and funcdata and pcdata tables. Reading these directly out of the input would further reduce the linker's memory footprint.

In-memory structures are inefficient

Some key object metadata has a large deserialized footprint. In Go 1.13, `sym.Symbol` is 176 bytes, every function symbol has an additional 248 byte `sym.FuncInfo` object, and `sym.Reloc` is 32 bytes. This is after efforts to reduce the size of these structures, which brought `Symbol` down from 248 bytes in Go 1.10 and `Reloc` down from 48 bytes in Go 1.12.[8]

To make this worse, the linker reads in all `Symbols` and `Relocs` before reachability analysis *and* the unreachable ones can't be garbage collected. For symbols marked as "duplicates allowed," only one instance of each symbol is allocated, but the `Relocs` for every instance are retained. This problem is worse in larger programs, where a larger fraction of "duplicates allowed" symbols are eliminated as actual duplicates. When linking cmd/compile, the linker reads in 171,523[9] symbols and retains 160,276[10] non-duplicate symbols (27 MiB of `Symbol`

---

[1] objfile:bytes
[2] Measured with compilebench -run Link.
[3] Measured with go build -x -work cmd/compile, then perflock benchcmd LinkCmdCompile <link-cmd>
[4] Measured with GOGC=1 GODEBUG=gcstoptheworld=1,gctrace=1 perflock <link-cmd>
[5] sed '/^gc/!d; s/.*@\([^s]*\)s.*->\(.*\)->.*/\1\t\2/' gctrace
[6] objfile:rsData - afterLoadLib:symbol contents in heap.
[7] final:symbol contents in heap
[8] ptype $(gover 1.10 tool -n link) cmd/link.*\\.Symbol$
[9] objfile:symbols

objects) and 1,121,659[11] relocations (34 MiB of `Reloc` objects). Together, this metadata consumes 51% of the live heap, and stays live for almost the entire duration of the link.

Only 56% of non-DWARF symbols in cmd/compile are actually reachable, but the linker spent the CPU and I/O to parse all Symbols and Relocs, and since all Symbols and Relocs are bulk-allocated, the unreachable symbols can't be discarded by the garbage collector.

The linker's handling of symbol names is also inefficient. Go symbol names include entire package paths, which makes them quite large in comparison with, say, typical C symbol names (though perhaps peanuts compared to typical C++ symbol names). The inputs to linking cmd/compile contain 7 MiB[12] of symbol name data, or 12.5% of the total input data. In Go 1.13, we significantly reduced the memory impact of symbol names by mmaping most symbol name data. However, these long symbol names still incur CPU time for name lookup and matching. Furthermore, owing to symbols synthesized during the link, there is still 502 KiB[13] of symbol name data allocated in the heap by the end of the link.

The toolchain's current approach to DWARF exacerbates every aspect of this problem. Since the linker constructs the final DWARF, the compiler must communicate the information the linker needs to do this. Currently, this is done through specially-named symbols. For example, for every function symbol pkg/path.F, the compiler will produce four additional symbols named like go.info.pkg/path.F, go.loc.pkg/path.F, etc. These symbols comprise 40% of the symbols read in by the linker, and 55% of the relocations, and amplify the impact of long package paths in symbol names by a factor of five (though the contents of these names is mmapped). Furthermore, this approach of associating these symbols using only a naming convention messes up reachability analysis. Unlike other symbols, which are marked reachable if the linker will need them, these symbols are generally *not* marked reachable. This doesn't currently matter since the linker retains all symbols in memory, so DWARF construction can find them anyway, but two wrongs don't make a right. This non-orthogonality has frequently led to confusion and interferes with improvements to reachable symbol handling.

The Go object format is sequential

The Go object file format is deeply sequential, with limited support for random or on-demand access. There is no symbol index. Everything is encoded in a variable-length format (individual fields are varint-encoded, and different symbol types have different fields). And each symbol in the symbol table specifies its length, but the offset of its data is implied from the cumulative sum of lengths. As a result, object files must be read in their entirety before any linking. This approach may make sense in a linker that assumes everything is in memory, but precludes many significant optimizations.

---

[10] afterLoadLib:symbols defined
[11] objfile:relocations
[12] Sum of objfile:symsize
[13] final:symbol names in heap

This variable-length format is also inefficient to decode. For example, the linker spends 5%[14] of its time just reading varints from Go object files.

The Go object file format does have some redeeming qualities. For example, all raw byte data is packed together contiguously, which includes symbol contents and pcdata tables. Hence, this region can be mmapped and skipped over by the loader.

Long-term maintainability

Finally, we consider the overall maintainability of the linker. While the linker evolves at a slower pace than much of the Go toolchain, it does still evolve, and many changes to the Go compiler require corresponding changes in the linker. As implemented, the linker seriously lacks modularity, and, since the linker has no owner who understands its global structure, changes to it incur a high engineering cost.

Building a better linker

I believe there is significant opportunity to build a better linker for Go that is dramatically more efficient in both CPU and memory. There are also deeper opportunities to change the way software is constructed, which we cover in the next section.

A few basic optimization principles underlie much of what we propose in this section:

1. Shift work from the linker to the compiler. The linker is a bottleneck of every build graph, even for the smallest incremental change. Unlike the linker, many instances of the compiler can run in parallel or even be distributed. And the results of the compiler can be cached. This section discusses several components of the linker that could be moved to the compiler, including symbol resolution (which the compiler already does, but the linker redoes) and more metadata generation.
2. Avoid strings. Strings are space-ineffficient, and comparing, hashing, and constructing strings are all CPU-inefficient operations. Strings also lead to inefficient and cache-unfriendly data structures. Currently, a string-indexed symbol table lies at the heart of the linker. This section proposes instead using a dense numbering of symbols and various algorithmic improvements that derive from that.
3. Load as little into the heap as possible. In the previous section, we saw that the linker eagerly loads a lot of its input into the heap *and retains it*. This results in high memory use, and the retention translates into high garbage collector CPU use. This section proposes lazily reading as much as possible from the inputs and discarding it as quickly as possible. In part, this involves algorithmic changes. It also involves moving per-symbol data out of the Symbol structure and into temporary data structures whose lifetimes can be minimized.

[14] perflock go build -ldflags -cpuprofile=/tmp/cpu.prof -o /dev/null cmd/compile

Since the object file format defines and constrains much of the design of the linker, I propose that we start by developing a new Go object file format. This also allows a more incremental approach since we can switch to a more flexible object file format with minimal changes to the rest of the linker, while opening up optimization opportunities. This section is organized around algorithmic changes to the linker, but many of these influence the design of the object file format.

Use symbol indexes instead of names

The Go linker currently assumes that an external symbol referenced by a package could be defined anywhere. This is necessary in linkers for C-like languages, where external symbols could be defined in any compilation unit, but because of Go's package DAG, the majority of cross-package symbol name resolution has already been done by the Go compiler. This offers a significant and unique opportunity for the Go linker: it could largely avoid working with symbol names (which are long strings) and instead work in terms of compact, dense indexes.

Figure 2 shows an example of the structures that would be involved. In this approach, the compiler emits a table of imports and a dense numbering to the symbols declared in a package to that package's object file. All symbol references in an object file would be a tuple of (import index, symbol index), where the import index specifies a package via the *referencing* package's import table, and the symbol index refers to the *referenced* package's symbol definition table. When the linker reads these in, it assigns a global base index to each package based on the number of symbols defined by that package and remaps local references in each object file into this global, dense numbering scheme.

Package A	
Imports	
[0]	(self)
Definitions	
[0]	X
[1]	Y
[2]	Z

Package B	
Imports	
[0]	(self)
[1]	A
Definitions	
[0]	F
[1]	G

Global package base	
A	0
B	3

Figure 2: Two example packages with import and definition tables. The table on the right shows the linker's global package offsets computed from the number of symbols defined in each package. The reference (1, 0) from package B refers to symbol A.X and would be assigned global index 0, and the reference (0, 1) from package B (symbol B.G) would

be assigned global index 4. None of this symbol resolution involves the names in the definitions table.

Hence, I propose that the initial object loading phase of the linker should simply read the imports table and the number of symbol definitions in each object and construct the global numbering.

To detect skew between an importing package and the imported package, each package should also contain a fingerprint, which should be replicated in the import table. When reading packages, the linker should check the fingerprint in the import table against the imported package. This fingerprint does not have to be strong, since it would only serve to detect errors in the build system.

Using dense symbol indexing has many advantages:

1. It would significantly reduce time spent processing symbol names: 8%[15] of the linker's CPU cycles (11% of non-GC cycles) are spent just reading in symbol names and references.
2. It allows for space- and time-efficient data structures in the rest of the linker; for example using bitmaps for reachability. Since it would be easy to create structures that parallel the symbol table, they would also be much easier to create and discard only as needed. In contrast, one of the reasons the `Symbol` type is so large right now is because it records information that's only used for certain phases of the link.
3. Finally, using indexes instead of pointers is much cheaper for GC marking, and will likely reduce the substantial fraction of the linker's cycles that are spent marking.

In this model, the linker doesn't need to deal with symbol names or symbol name lookup at all except 1) when printing diagnostics, 2) for "linknamed" symbols, and 3) for "dupok" symbols (potentially). Since diagnostics are rare, this suggests that symbol names should be kept in a separate table of the object file that can be ignored by the linker until something goes wrong, and that the object file does not need a way to efficiently look up a symbol by name. Figure 3 gives a simple example of what this table could look like. We discuss linknamed and dupok symbols below.

Name offsets	
[0]	0x00
[1]	0x09
[2]	0x17
[3]	0x20

Name data	
0x00	`main.mai`
0x08	`nmain.pa`
0x10	`rseArgsm`
0x18	`ain.load`

[15] objfile.(*ObjReader).readRef

Figure 3. Example of compact symbol name table representation for symbols “main.main”, “main.parseArgs”, and “main.load”. This allows for random access to symbol names when needed, while keeping the names separate from the bulk of the object file.

Symbols that require special resolution

There are some symbols the compiler cannot pre-resolve. “Linknamed” symbols and symbols defined in assembly can only be resolved via their names. However, these symbols are rare and can be identified at their definition site, so the linker could maintain a name table just for these, which it can discard after object loading.

“Dupok” symbols are more troublesome. These allow multiple (matching) definitions of a symbol under the same name, as long as all of the definitions are dupok. When linking cmd/compile, the linker reads 96,151[16] dupok symbols (56% of all symbols read), of which it discards 18,389[17] as duplicates. Roughly 50% of dupok symbols are related to type descriptors. Since multiple packages can define the same (unnamed) type, the symbol is named after the type definition itself and does not include a package path. Half of the type descriptor symbols are simply for names of types, fields, and methods, which we could represent differently. Roughly 20% of dupok symbols are DWARF-related and could be eliminated with a better symbol side-band data representation (discussed in a later section). Finally, roughly 13% of dupok symbols are strings, which may be worth handling separately from other symbols. The remaining dupok symbols are scattered across many uses.

I propose that, initially, we continue to resolve dupok symbols by name. This will still require reading these symbols’ names, but we can avoid reading their other metadata. We can assume that if one definition of a symbol is dupok, then all of its definitions are dupok. If the cost of dealing with the symbol names remains a problem, we should reduce the number of dupok symbols and then consider making dupok symbols content-addressed. For content-addressed symbols, a strong 128-bit (16 byte) hash should suffice to resolve birthday paradox issues, while being significantly smaller than type symbol names, which average 36 bytes.[18]

Type descriptor symbols already have a type hash and their names are only used for debugging purposes, so we could treat these as a special case.

Strings are already a special case. In order to properly deduplicate strings by content in a name-based system, the string symbol’s content is part of the symbol name. If we add content-addressed symbols, strings would be just another use of this.

---

[16] objfile:dupok
[17] objfile:dupSymbols
[18] go build -gcflags all=-S cmd/compile |& grep '^type\.' | grep dupok | sed 's/ SRODATA.*//' | awk '{print length($0)}' | dist

Given the advantages of dense symbol numbering, we want to incorporate these special types of symbols into the dense numbering. There are many potential ways to do this. For example, we could cluster all named symbols under their own pseudo-package, which would give them their own numbering space within each package, and then collect all unique symbols at the end of the global dense numbering constructed by the linker. Once these symbols have been globally numbered, the tables used for resolving special symbol can be discarded.

Compute reachability prior to loading symbols

After object loading, the linker computes symbol reachability. We should arrange for this step to be done without reading symbols first.

To do this efficiently without loading symbols, we must arrange the object file so the linker can first read in the *identity* of all symbols as well as the reachability graph. The identity of symbols is simply the numbering discussed in the previous section. The linker will need to resolve the identity of named and dupok symbols before reachability analysis, but should be able to discard these resolution tables before computing reachability.

The edges of the reachability graph are primarily determined by relocations. The linker will need to access these relocations mostly randomly, but will only need to access each symbol's relocations at most once. Given the memory cost of reading all relocations in, and the low overhead of random access on modern I/O devices, I propose that the linker stream each symbol's relocations from the mmapped object files as it walks the reachability graph and should *not* cache these in the heap. This will require streaming the relocations a second time when later applying relocations, but if there is enough memory, the kernel will keep this data in the page cache, so this will require no additional I/O.

Given the Go package DAG, we can maintain good spatial locality of reference despite random access to the relocations table by biasing the graph flood toward staying within a package before traversing into another package. Because of named symbols, we can't strictly depend on the package DAG to know that we're done with a package's relocations, but as a locality optimization it should be quite effective.

Given a dense symbol numbering, reachability can be computed with just a single bit per symbol plus a queue. In the object file, we can pack all relocations together and use an offset table to read them efficiently on demand, as shown in figure 4. A sketch of this algorithm is given in figure 5.

Relocation offsets	
[0]	0

Relocation data			
0	R_ADDR	25/4	Sym 1+0

[1]	2
[2]	3

1	R_ADDR	30/4	Sym 2+0
2	R_CALL	10/4	Sym 3+0
3	R_CALL	42/4	Sym 3+0

Figure 4. Example relocation offsets and data tables. Structurally, this is the same as the name table, except the unit is a relocation record rather than a byte.

```
// Use a min-heap work queue to cluster accesses within each
// package (assuming indexes are assigned in package DAG order).
var work minHeap
// Reachable is a bitmap marking reachable symbols.
reachable := makeBitmap(nSyms)
mark := func(symIndex int) {
   if !reachable.has(symIndex) {
      work.push(symIndex)
      reachable.set(symIndex)
   }
}
// Flood the relocation graph, starting at main.main.
// In reality there will be more roots.
mark(lookupSymIndex(“main.main”))
for !work.empty() {
   symIndex := work.popMin()
   reachable.set(symIndex)
   pkg := symToPackage(symIndex)
   // Read relocation offsets for symIndex and symIndex+1,
   // then read the relocations between these offsets.
   relocs := pkg.readRelocs(symIndex)
   for i := 0; i < relocs.n; i++ {
      reloc := relocs.get(i)   // Read reloc i of this sym
      mark(reloc.symIndex)
   }
}
```

Figure 5. A sketch of the reachability algorithm using one bit per symbol and random access into the packed relocation data. The algorithm works best when symbol indexes are assigned in package DAG order and sequentially within each package. It uses a min-heap to cluster accesses to a given package. It accesses relocations in a way that can avoid memory allocation in the loop.

Some aspects of reachability involve more than the relocation graph. As described above, reachability analysis handles methods and interface tables specially, and this handling is different depending on whether a binary uses reflection calls. This involves recognizing and decoding interface type symbols. Currently, these are recognized by symbol name, so to avoid loading symbol names the compiler will need to specially mark these symbols. Since the flood needs to look at relocations anyway, this could be done with a no-op relocation on the symbol. To decode the type symbol, the linker must be able to load some symbol data before reachability is complete. This shouldn't be a problem if the object file format is randomly accessible.

Once the linker has determined the set of reachable symbols, it can load the minimal symbol metadata necessary for the following phases, such as the symbol's kind (for assigning it a section during layout).

It's tempting to compute a new dense numbering over just the reachable symbols so later linker phases can use even more densely indexed data structures. However, later phases still need to perform random access into the object file, so keeping a simple mapping to the original symbol indexes is likely more valuable.

Represent side-band data explicitly and efficiently

After computing reachability, many of the linker phases generate new data and symbols for the output file. For example, the linker constructs several runtime data structures as well as DWARF debug info. To support this, the compiler passes side-band data to the linker that doesn't appear directly in the final output, but contributes to these generated sections and symbols.

Currently, the linker has several different ways of representing side-band information. For text symbols, the symbol itself carries a dozen extra metadata fields, which would make random access into the symbol table difficult, including an extensible set of pcdata and funcdata tables, which use their own ad hoc representation in the object file. For DWARF, extra information is passed to the linker through specially-named symbols, which pollute the symbol space, interfere with reachability analysis, and introduce overhead into parts of the linker that these symbols never participate in.

I propose the object file format have an extensible way to associate general-purpose side-band information with a symbol, and that we use this mechanism for all of these purposes.

In terms of reachability, this strictly simplifies things: in all cases, if the symbol itself is reachable, then all of its side-band data should be considered reachable, at least in the sense that the linker will need to access it (though it may or may not appear in the final binary).

There are, however, two advantages to our current use of symbols to carry DWARF data: symbols are our abstraction for data that can be placed in the output file (which is useful for

some, though not all DWARF symbols), and symbols can carry relocations, which are used extensively on DWARF data. But symbols also carry a lot of baggage that isn't necessary for DWARF symbols, such as a name, a Go type, and fields for tracking the corresponding output symbol, and the association of the primary symbol to its DWARF data is done only through a naming convention.

However, the rest of this proposal should significantly reduce the cost of symbols and make symbols without names a reasonable option. Hence, I propose we embrace the use of symbols for side-band data, but instead of associating them through naming conventions, use unnamed symbols and introduce a separate table for explicitly associating a *primary symbol* with its *side-band symbols*.

Do less in the linker

In some cases, the linker simply does more work than necessary to generate new data, and we should shift this work into the compiler where it can be better parallelized across packages and reused across builds. Historically, the linker did far more than it does now: it used to even do final code generation, which meant anything that depended on PC values had to happen in the linker, and vestiges of that architecture remain.

There are two obvious things that could be shifted into the compiler or eliminated: much of DWARF generation, and pclntab rewriting.

Generate more DWARF in the compiler

The linker spends 13%[19] of its CPU cycles (17% of non-GC cycles) constructing DWARF data, not including time to compress it. Roughly half of this is spent generating DWARF line tables from the Go pclntab runtime line tables. We could instead generate a DWARF line table fragment for each function in the compiler and simply combine them in the linker. One slight complication is that DWARF line tables are per compilation unit, not per function, but the linker can attach the appropriate compilation unit header and assemble only the fragments for retained function symbols. Each fragment can start with a DW_LNE_set_address opcode with the relocated address of the function and end with a DW_LNE_end_sequence op code, which will put the state machine into a known state for the next fragment.

Besides line tables, the linker also generates .debug_info definitions for global variables and Go types. These could also be done in the compiler, though the advantage of generating DWARF definitions for Go types in the linker is that it saves a large amount of potentially duplicated work. Even though this requires the linker to parse the Go reflection structures, it may make more sense to leave DWARF type generation in the linker.

[19] ld.dwarfGenerateDebugInfo + ld.dwarfGenerateDebugSyms

Avoid rewriting pcln tables

The linker also spends 6%[20] of its CPU cycles (8% of non-GC cycles) constructing the pcln table. This includes symbolic traceback information as well as critical GC metadata. The fragments of this are mostly generated by the compiler, but the linker rewrites the line tables eliminate unused source file names (e.g., if all symbols from that file have been eliminated). This is almost certainly not worth the cost. Even if the space-savings is worthwhile, we should simply eliminate unused file paths without renumbering them. Ideally, the linker should not contain a pcdata table encoder/decoder at all.

Copy symbols directly to the output

After synthesizing fresh symbols and performing layout, the linker is finally ready to relocate and emit symbols into the output file.

The linker was originally designed to do all of this in the heap, but thanks to some clever tricks can now largely avoid copying symbols through the heap. We should retain and improve on this design.

In particular, the linker should mmap all of the input object data and its output file, copy symbol data directly from the input to the output after layout, and apply relocations directly in the mmapped output file. Relocations should be applied to each symbol immediately after copying it to maintain temporal locality.

One complication to this approach is that the linker may have to limit the number of simultaneously mapped files, as Linux limits the number of mappings to 65,530 by default. For example, the gold linker discards and re-maps object files during especially large links to avoid this limit. This suggests that we need an mmap manager with the ability to unmap and re-map object files. This is most compatible with an object reader API that returns copies of data from the object file. Any APIs that return references to the object file will need a way to explicitly dismiss those references.

This approach uses minimal memory while allowing the kernel maximum flexibility to swap out its page cache. In particular, this approach never modifies the input files (in fact, they can be mmapped in shared, read-only mode), so pages of the input file can be freely discarded from the page cache by the kernel. Pages of the output file can be flushed to disk by the kernel, but even under high memory pressure, this is unlikely to incur much more write I/O than necessary anyway.

---

[20] ld.(*Link).pclntab

Switch to DWARF 5

The other parts of this proposal should significantly reduce the linking cost of DWARF. However, they do nothing to reduce the sheer number of relocations associated with DWARF data, which accounts for over half of the total relocations when linking cmd/compile, and are a problem endemic to DWARF. It also doesn't reduce the time spent compressing DWARF sections, which accounts for 23%[21] of the linker's CPU cycles (31% of non-GC cycles).

Ultimately solving these problems likely requires turning to DWARF5 (see #26379). DWARF5 has three significant advantages: 1) it introduces new attribute forms with position-independent encodings, 2) it has far more compact encodings for some large debug info sections, and 3) it supports "DWARF fission".

The position-independent encodings in DWARF5 would significantly reduce the relocation load on the linker, and likely allow DWARF fragments produced by the compiler to be copied by the linker without any rewriting. This may even allow DWARF compression to occur in the compiler rather than the linker, which would reduce this significant source of CPU time as well as creating more opportunities to use mmapped data directly.

One complication to using DWARF5 position-independent encodings is that they are relative to the DWARF compilation unit. Currently, the linker creates a DWARF compilation unit for each package. But the layout *within* a compilation unit isn't determined until link time in the Go toolchain. It may be necessary to place each function into a different DWARF compilation unit (a 12 byte overhead per function), or to leave unused entries in the .debug_addr table (a 4 byte overhead for each address in discarded functions; though this may be much less if compressed).

DWARF5 fission would also enable a more radical departure where the DWARF data is written to separate DWARF objects by the compiler, bypassing the linker entirely. These DWARF objects can separately be linked into "DWARF packages." This would complicate the Go build and deployment process, as the DWARF would now be in a separate file, which could create confusion when moving binaries around. However, it would significantly reduce link times as well as binary sizes, while still giving users an easy way to access debug info.

I propose we should first take advantage of the new position-independent forms and compact tables in DWARF5, while still embedding the debug info in the binary, and then consider moving to DWARF objects.

---

[21] zlib.(*Writer).Write

Improve internal concurrency

Finally, the linker currently has minimal internal concurrency. This is particularly unfortunate given that there's typically no concurrency at the build system level once it reaches the link step.

While many linker phases depend on the complete output of the previous phase, many individual phases could be parallelized. For example, object files could be loaded concurrently; reachability could be computed with a parallel algorithm; and reachable symbols can be loaded, copied to the output, and relocated concurrently. In fact, one of the few phases that would be difficult to parallelize would be layout, but this takes so little time it doesn't even appear in the profile.[22]

Beyond a better linker

Our priority is to fix the linker as it stands, but it's worth keeping in mind more innovation we could bring to Go and designing with this in mind.

Both Russ Cox and Ian Lance Taylor have proposed that a Go binary should be able to directly load a Go object file into its running image. Ian has proposed that in the context of unit tests it should be possible to link and execute a test in a single step, rather than producing a binary that will be discarded almost immediately. Russ has proposed this in the context of interpreters and JITs that leverage the performance of the Go compiler, and as a more stable, portable replacement for plugin. This is similar to dynamic linking, but fluidly fits into the build process, rather than requiring special steps to produce a shared object that works very differently from a regular object. In this model, producing a static binary is simply an optional extra step. One major difference with Ian's "fast mode" linking is that it doesn't perform symbol-level reachability and instead keeps each object file contiguous so the linker only needs to generate a (contiguous) "links" section. We should consider how we could support such a mode in the future as we're designing the object format.

David Chase suggested a further opportunity for fast linking: if all code is position-independent and we retain entire packages, then all regular symbol references can be done with nothing more than the base address of the package that contains the symbol. For example, if package B imports package A, then package B can statically know the offsets of all text and data symbols within package A. Hence, package B could call any function in package A given just the base address of package A's text. This would make offset tables incredibly small, though would make cross-package symbol references more expensive.

In a similar vein, Austin Clements has proposed that, to be useful to a wide range of tools, any package for reading object files needs to implement symbol relocation just like a linker. For

[22] ld.(*Link).textaddress + ld.(*Link).dodata

example, debug/elf.(*File).DWARF contains a minimal, incomplete implementation of ELF relocation because parsing a DWARF section generally isn't possible without first applying relocations. But this is private to the debug/elf package, even though this is a problem for any tool that examines section contents. Instead, processing relocations should be a fundamental part of any package for working with object files. At its core, this problem is the same as applying relocations in a linker and as loading object files into an address space, and we should consider how to unify all of these.

Finally, Ian has proposed that it's time for the world to have a new object file format and that we could be the ones to create it. All popular object file formats were designed in the 1980's and software development has changed a lot since then.