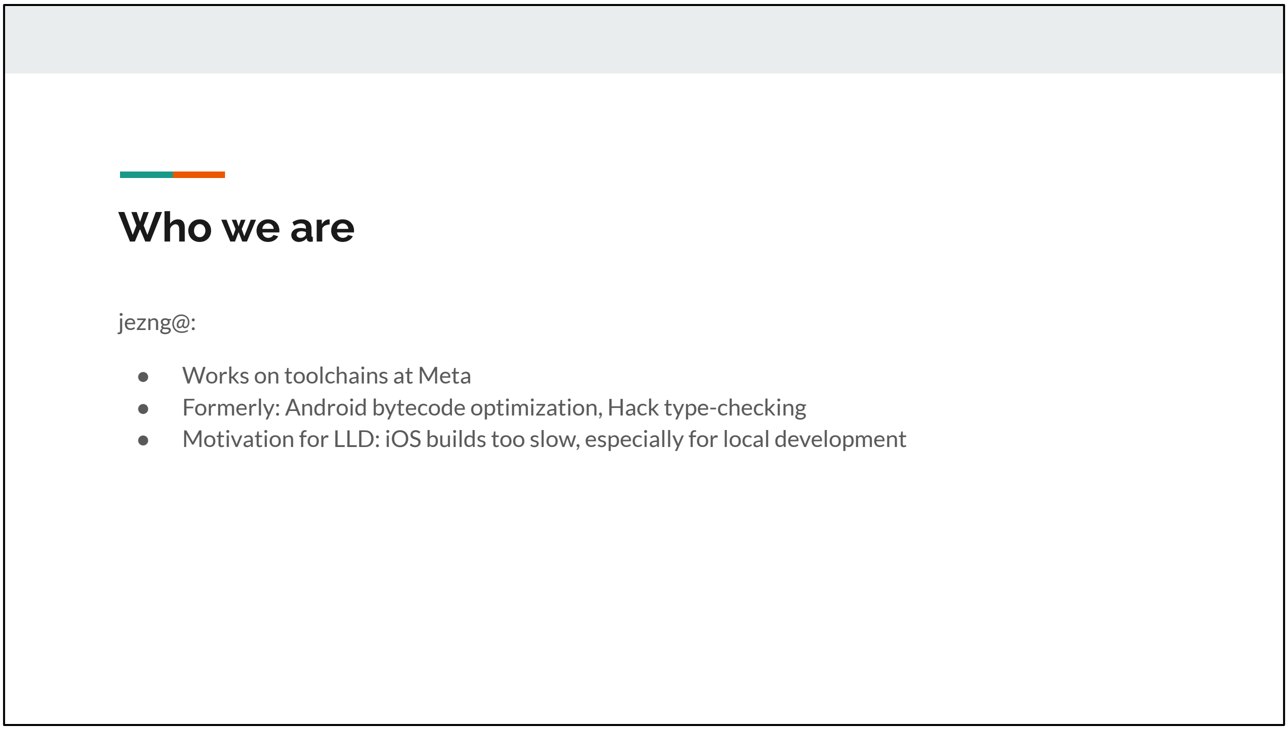
LLD for Mach-O: The Journey

Nico Weber <thakis@chromium.org>

Jez Ng <jezng@fb.com>

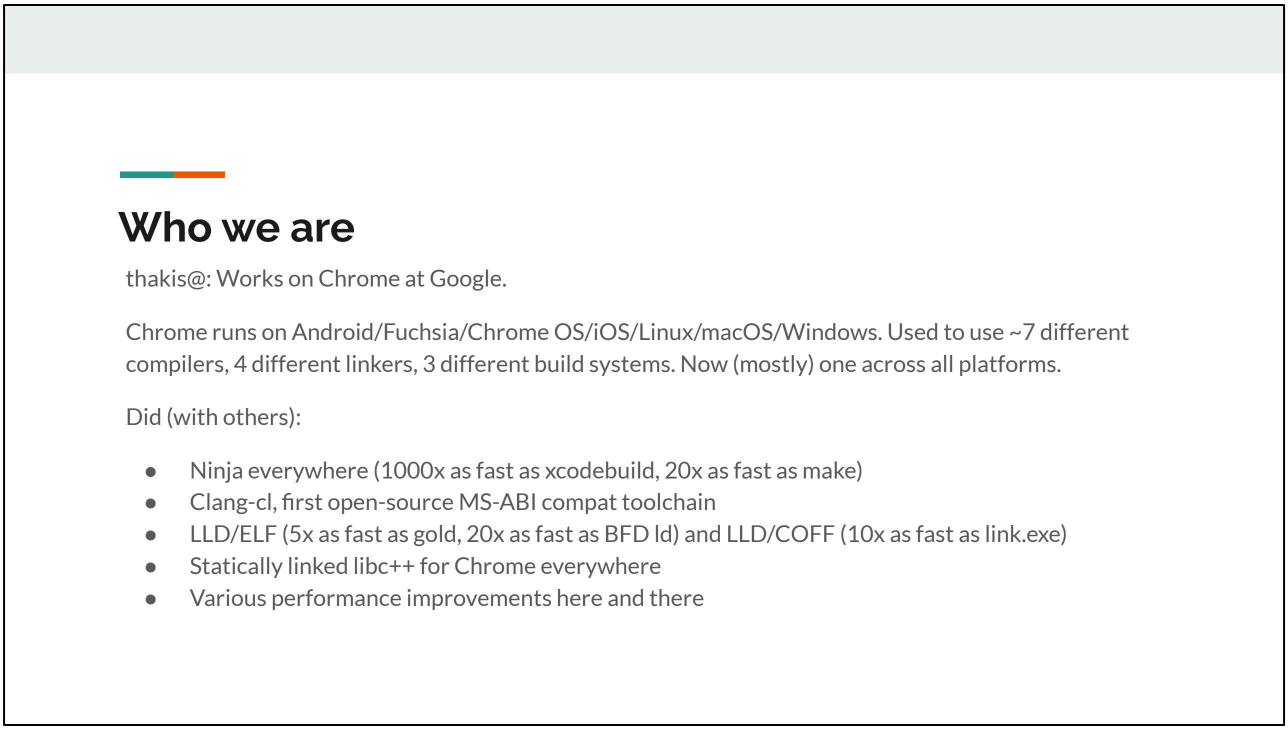


Who we are

jezng@:

- Works on toolchains at Meta
- Formerly: Android bytecode optimization, Hack type-checking
- Motivation for LLD: iOS builds too slow, especially for local development

(jez: jez)



Who we are

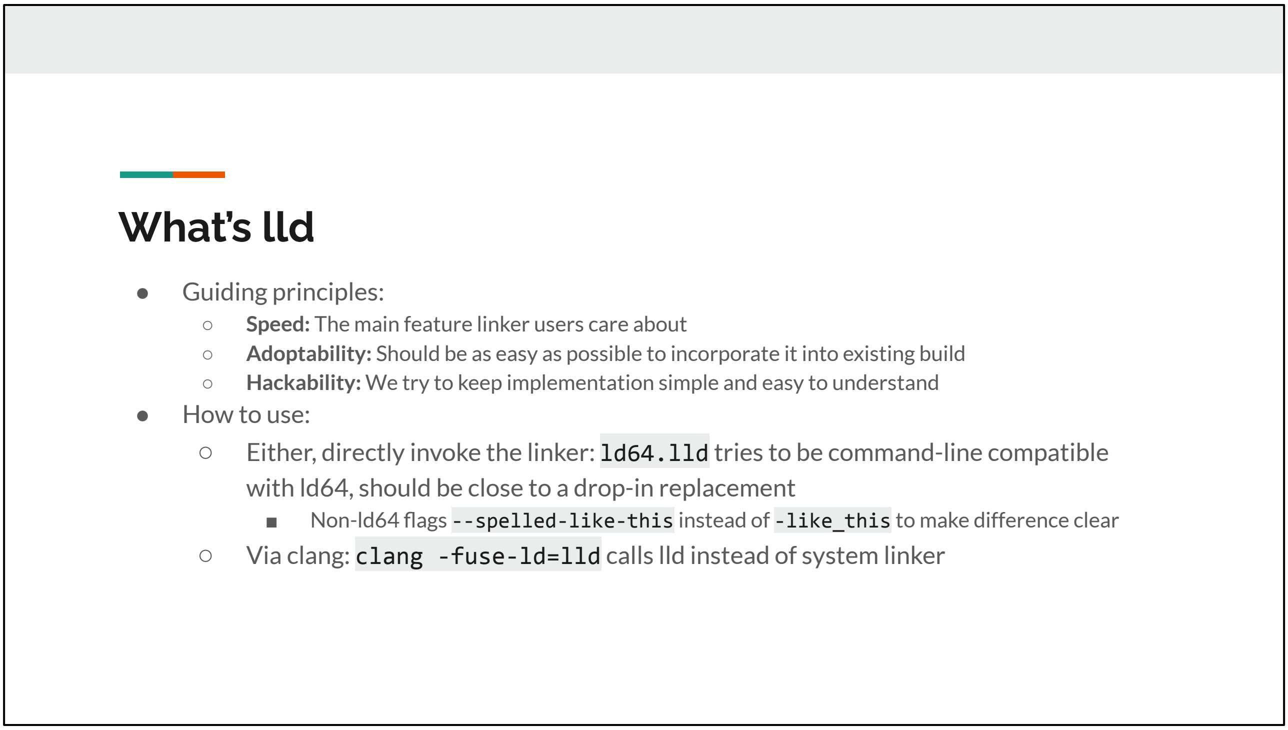
thakis@: Works on Chrome at Google.

Chrome runs on Android/Fuchsia/Chrome OS/iOS/Linux/macOS/Windows. Used to use ~7 different compilers, 4 different linkers, 3 different build systems. Now (mostly) one across all platforms.

Did (with others):

- Ninja everywhere (1000x as fast as xcodebuild, 20x as fast as make)
- Clang-cl, first open-source MS-ABI compat toolchain
- LLD/ELF (5x as fast as gold, 20x as fast as BFD ld) and LLD/COFF (10x as fast as link.exe)
- Statically linked libc++ for Chrome everywhere
- Various performance improvements here and there

I’m Nico, I work on chrome. Chrome runs on many different and we used to use many different compilers, linkers, build systems, c++ standard libraries, and that was a bit of a drag. So we set out to fix that, and we’re mostly done with that. We often ended up with something that was at least an order of magnitude faster than the old thing.



What's lld

- Guiding principles:
  - **Speed:** The main feature linker users care about
  - **Adoptability:** Should be as easy as possible to incorporate it into existing build
  - **Hackability:** We try to keep implementation simple and easy to understand
- How to use:
  - Either, directly invoke the linker: `ld64.lld` tries to be command-line compatible with ld64, should be close to a drop-in replacement
    - Non-ld64 flags `--spelled-like-this` instead of `-like_this` to make difference clear
  - Via clang: `clang -fuse-ld=lld` calls lld instead of system linker

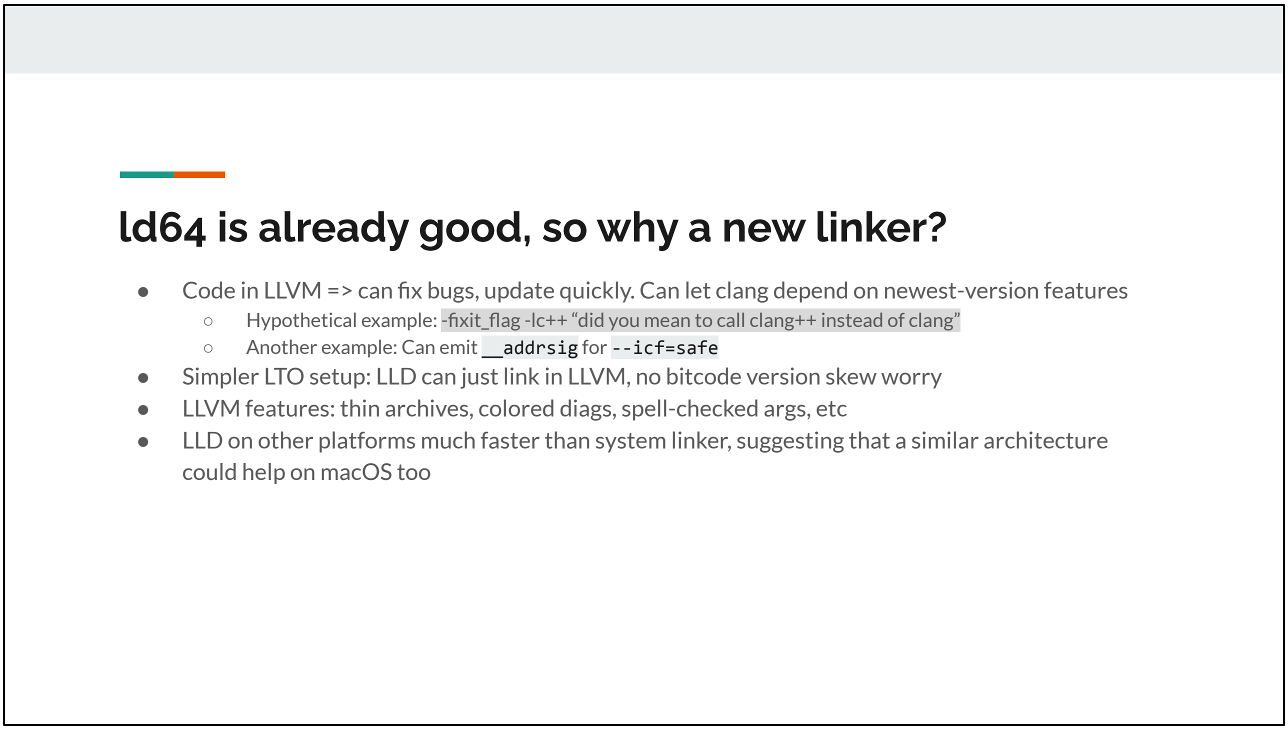
Lld is a linker. I won't tell you what a linker does. Lld makes a few assumptions: It thinks that speed and simplicity are important. (For that reason, it assumes that inputs are usually valid. It's not designed to be used as a library.)

Lld supports different flavors: Darwin, Visual Studio's link, elf, etc.

The COFF linker tries to be command-line compatible with link.exe, the linker in Visual Studio. The Mach-O version tries to be command-line compatible with ld64, the Xcode linker, etc. So you can use your existing build and just change which linker you call. Or just add `-fuse-ld=lld` to your ldflags. Really easy to try out.

Lld can link COFF and ELF and wasm and now also Mach-O binaries, but it's really several different linkers in one binary rather than a single linker. The ELF linker assumes ELF input files and ELF semantics, the COFF linkers assumes COFF, and the Mach-O linker assumes Mach-O. They each have their own symbol table and symbol classes, which means each port looks familiar to someone who knows that file format.

(There's an "old" lld design, and the "old" lld/mach-o port is the last remnant of that. We finally deleted that a few months ago.)



ld64 is already good, so why a new linker?

- Code in LLVM => can fix bugs, update quickly. Can let clang depend on newest-version features
  - Hypothetical example: -fixit_flag -lc++ “did you mean to call clang++ instead of clang”
  - Another example: Can emit `__addrsig` for `--icf=safe`
- Simpler LTO setup: LLD can just link in LLVM, no bitcode version skew worry
- LLVM features: thin archives, colored diags, spell-checked args, etc
- LLD on other platforms much faster than system linker, suggesting that a similar architecture could help on macOS too

ld64 is the linker we’ve had the least problems with in chrome, by far. It’s fast, it’s stable, it’s deterministic -- so why bother?

Main reason is integration with LLVM. We already build a bunch of LLVM binaries -- clang, and lld on other platforms -- and deploy them several times a month.

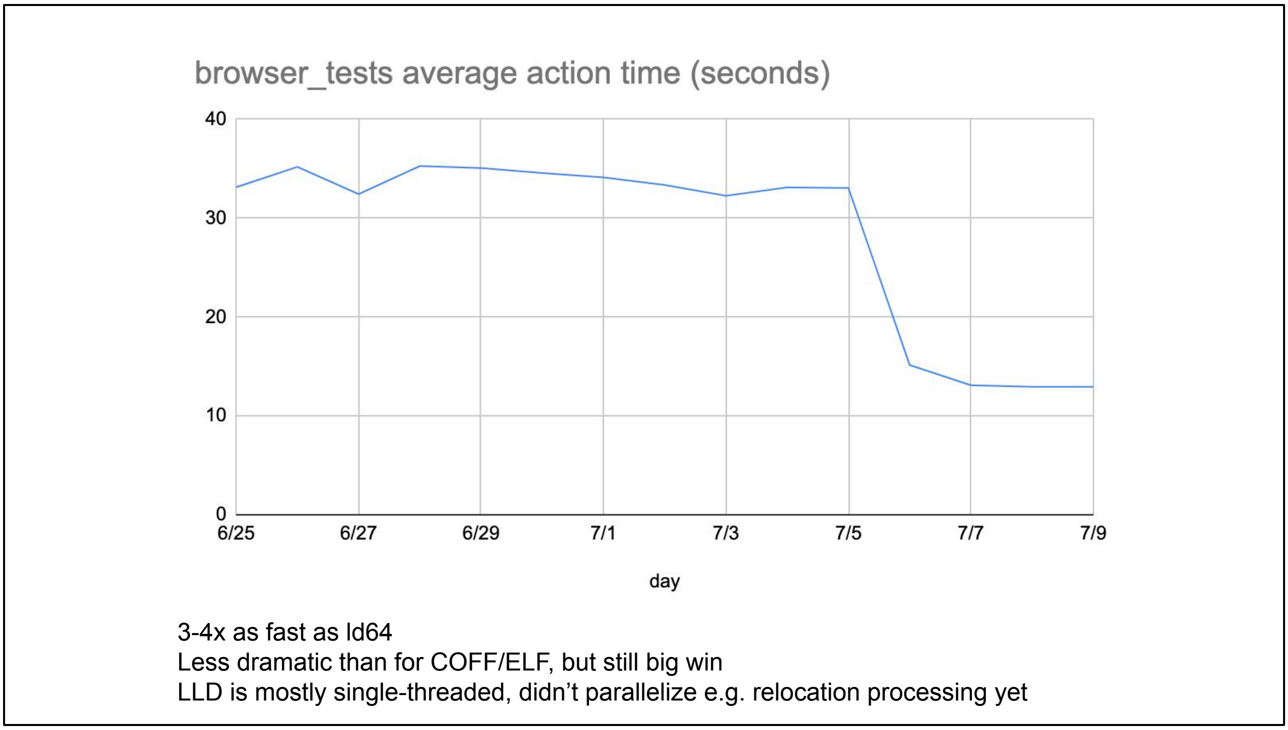
So for us this makes it easy to to fix problems in the linker and then deploy the new version very quickly.

Since clang and lld can assume to be built from the same revision, this makes it easy to add features that straddle the compiler-linker boundary. Here are two examples.

Also nice for LTO.

And the usual LLVM niceties.

Coming back to speed for a bit: We thought ld64 was very fast. But:



3-4x as fast as ld64
Less dramatic than for COFF/ELF, but still big win
LLD is mostly single-threaded, didn’t parallelize e.g. relocation processing yet

Lld is 3-4x as fast as ld64. Not the order of magnitude win we’ve seen elsewhere, but still pretty good!

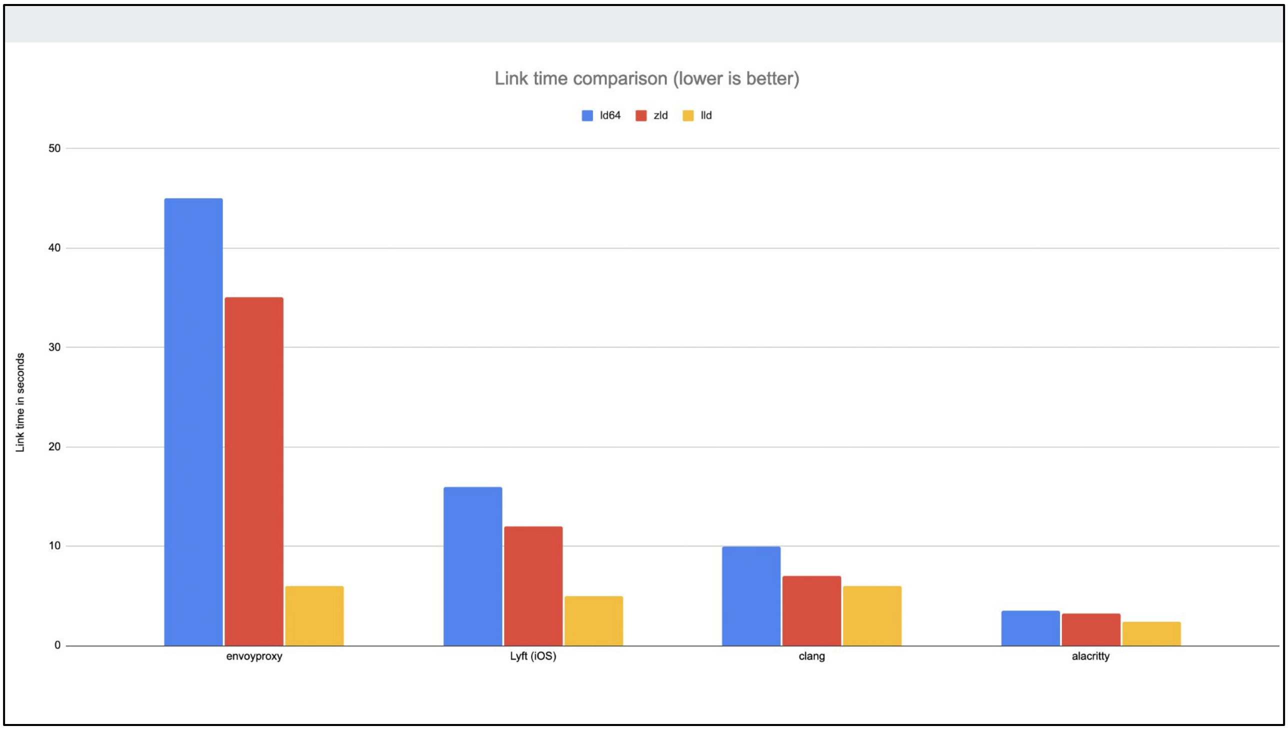
Note: Chart is zero-based.

This is for linking browser_tests, the biggest executable in chrome. This is on a busy bot with a mostly cold cache. Locally see 4x with warm cache for chrome. Less speedup for smaller binaries, but still noticeable. E.g. building check-lld which links a bunch of binaries goes through links noticeably faster.

Someone on my team reported that loading chrome into Instruments now takes 4 minutes instead of 10 minutes previously. I was both terrified and delighted to hear that. Some people are more patient than I am.

And haven’t even optimized all that much yet.

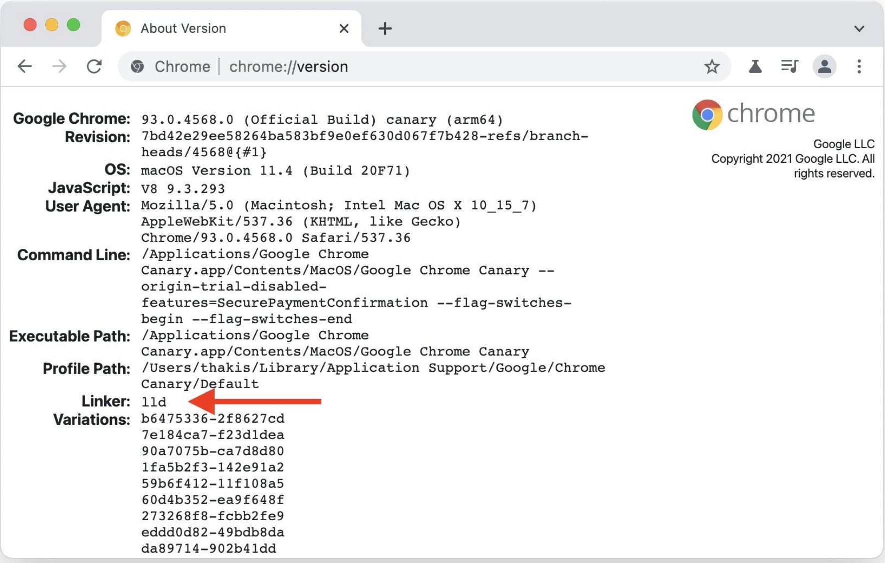
V1 of chrome/mac’s lld doc said “probably not faster”
V2 said “faster for now, but might become slower as more stuff is implemented
V3 says “faster” – not enough features missing to make lose that much perf



Numbers collected by Keith Smiley at Lyft.

These are debug builds.

(Zld is Michael Eisel's fork of ld64 that uses SwissTables for hash tables instead of STL, different hash functions, etc. "only meant for debug builds", aspirations for incremental linking. It is _not_ the Zig language's linker, which is also called zld.)



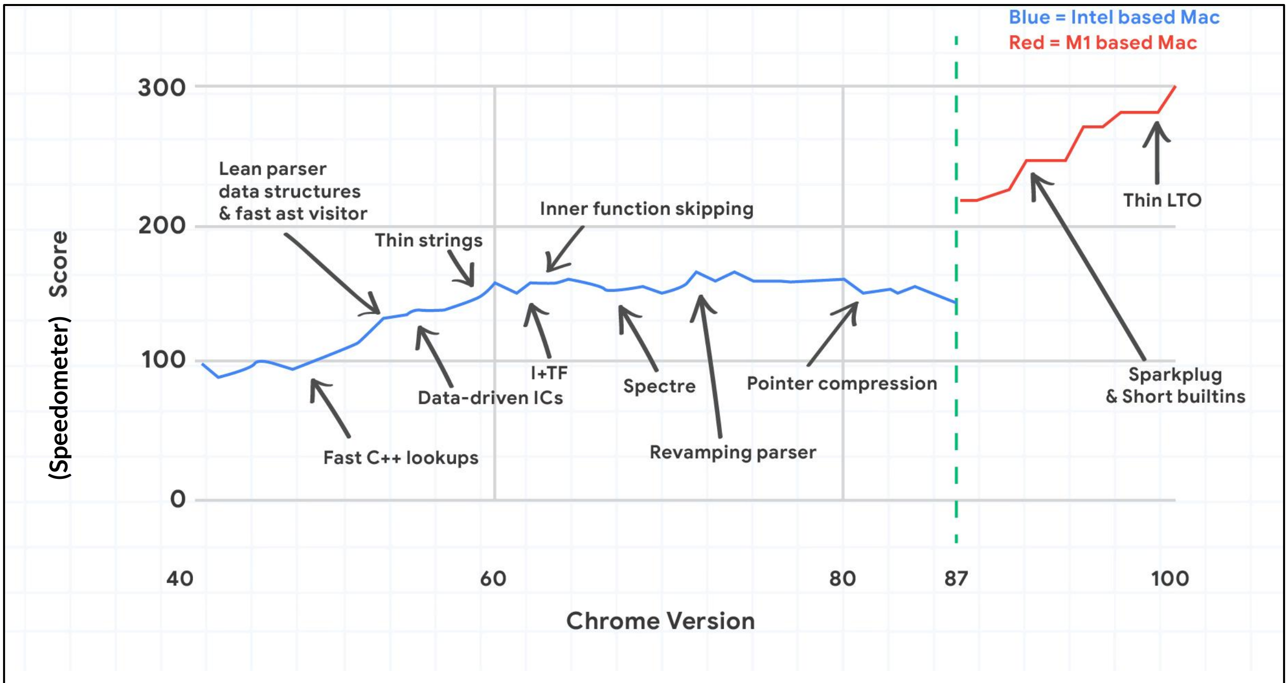
```
    Google Chrome: 93.0.4568.0 (Official Build) canary (arm64)
         Revision: 7bd42e29ee58264ba583bf9e0ef630d067f7b428-refs/branch-
                   heads/4568@{#1}
               OS: macOS Version 11.4 (Build 20F71)
       JavaScript: V8 9.3.293
       User Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
                   AppleWebKit/537.36 (KHTML, like Gecko)
                   Chrome/93.0.4568.0 Safari/537.36
     Command Line: /Applications/Google Chrome
                   Canary.app/Contents/MacOS/Google Chrome Canary --
                   origin-trial-disabled-
                   features=SecurePaymentConfirmation --flag-switches-
                   begin --flag-switches-end
  Executable Path: /Applications/Google Chrome
                   Canary.app/Contents/MacOS/Google Chrome Canary
     Profile Path: /Users/thakis/Library/Application Support/Google/Chrome
                   Canary/Default
           Linker: lld
       Variations: b6475336-2f8627cd
                   7e184ca7-f23d1dea
                   90a7075b-ca7d8d80
                   1fa5b2f3-142e91a2
                   59b6f412-11f108a5
                   60d4b352-ea9f648f
                   273268f8-fcbb2fe9
                   eddd0d82-49bdb8da
                   da89714-902b41dd
```

Google LLC
Copyright 2021 Google LLC. All rights reserved.

Chrome builds, all tests build, all tests pass. Chrome 95+ (late 2021) linked with lld. Default linker for Chrome/Mac.

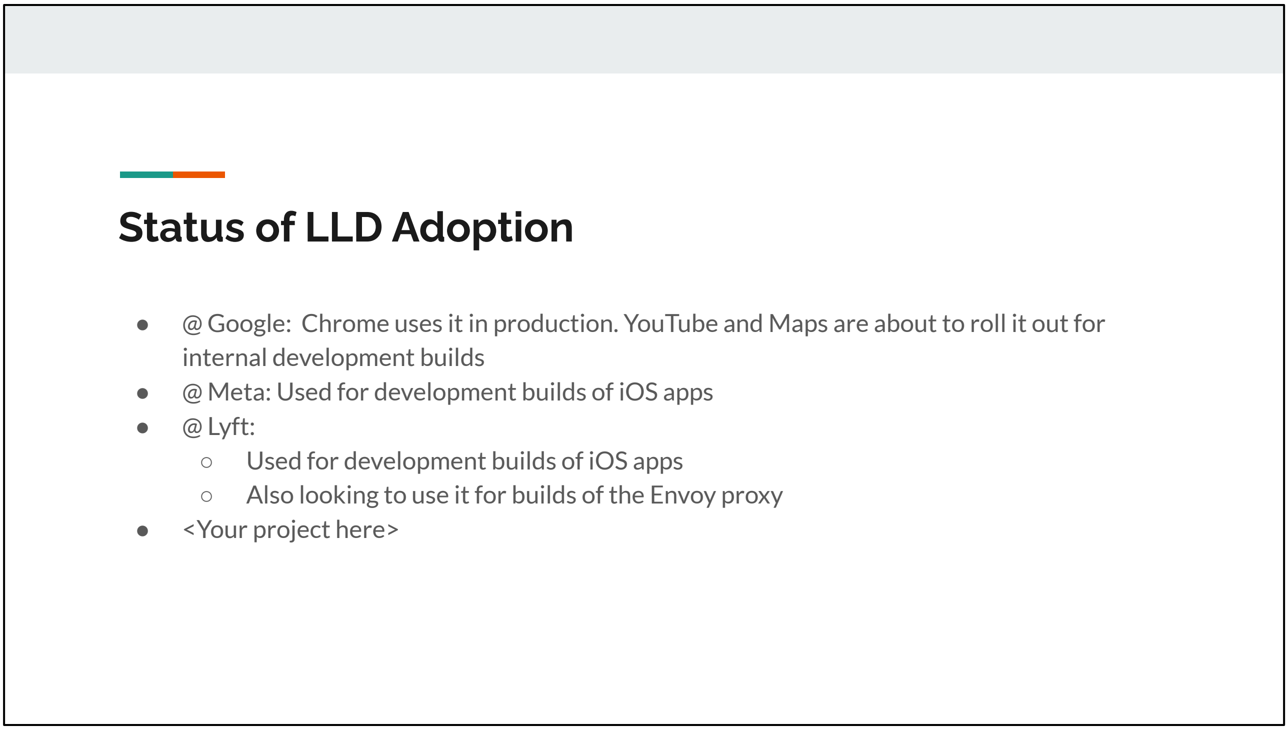
Performance seems comparable. Binary size with ICF turned on is 8 MB / 3.3% smaller.

Shipped to stable in m95.



https://blog.chromium.org/2022/03/how-chrome-became-highest-scoring.html

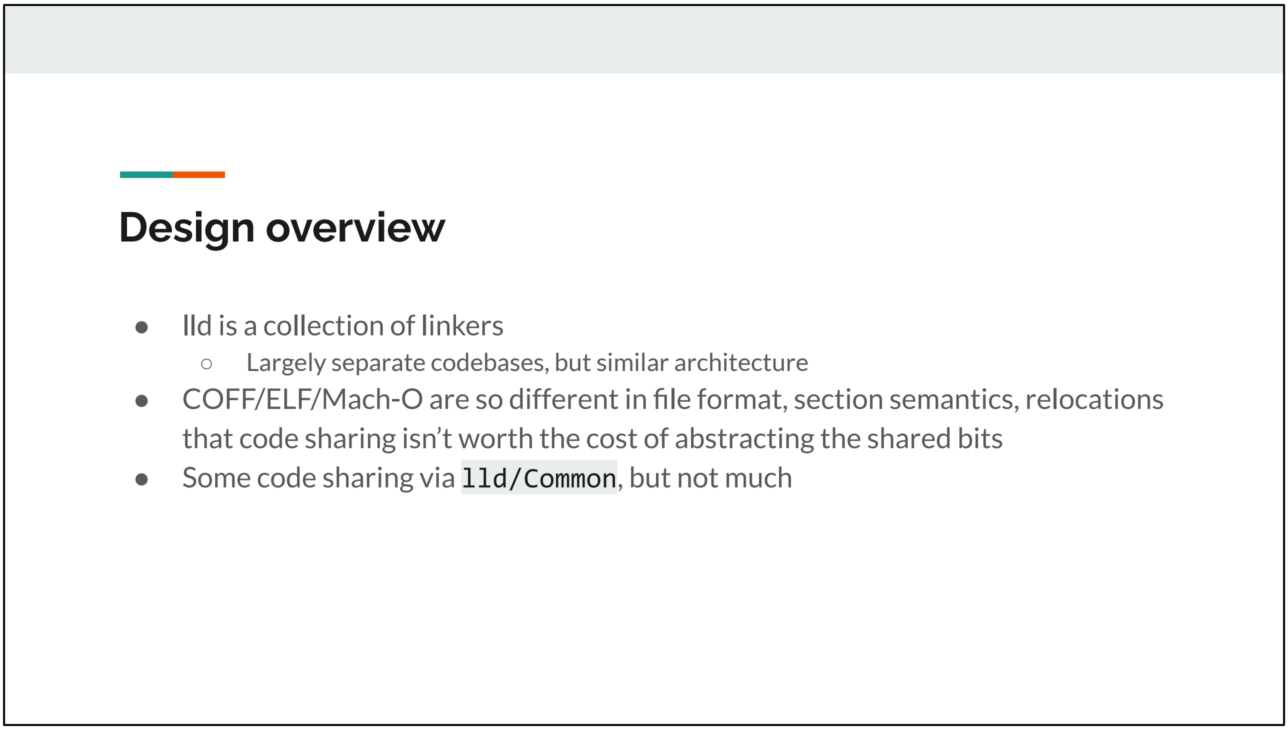
ThinLTO shipped in m99 (early 2022).



Status of LLD Adoption

- @ Google: Chrome uses it in production. YouTube and Maps are about to roll it out for internal development builds
- @ Meta: Used for development builds of iOS apps
- @ Lyft:
  - Used for development builds of iOS apps
  - Also looking to use it for builds of the Envoy proxy
- <Your project here>

Who’s using this?

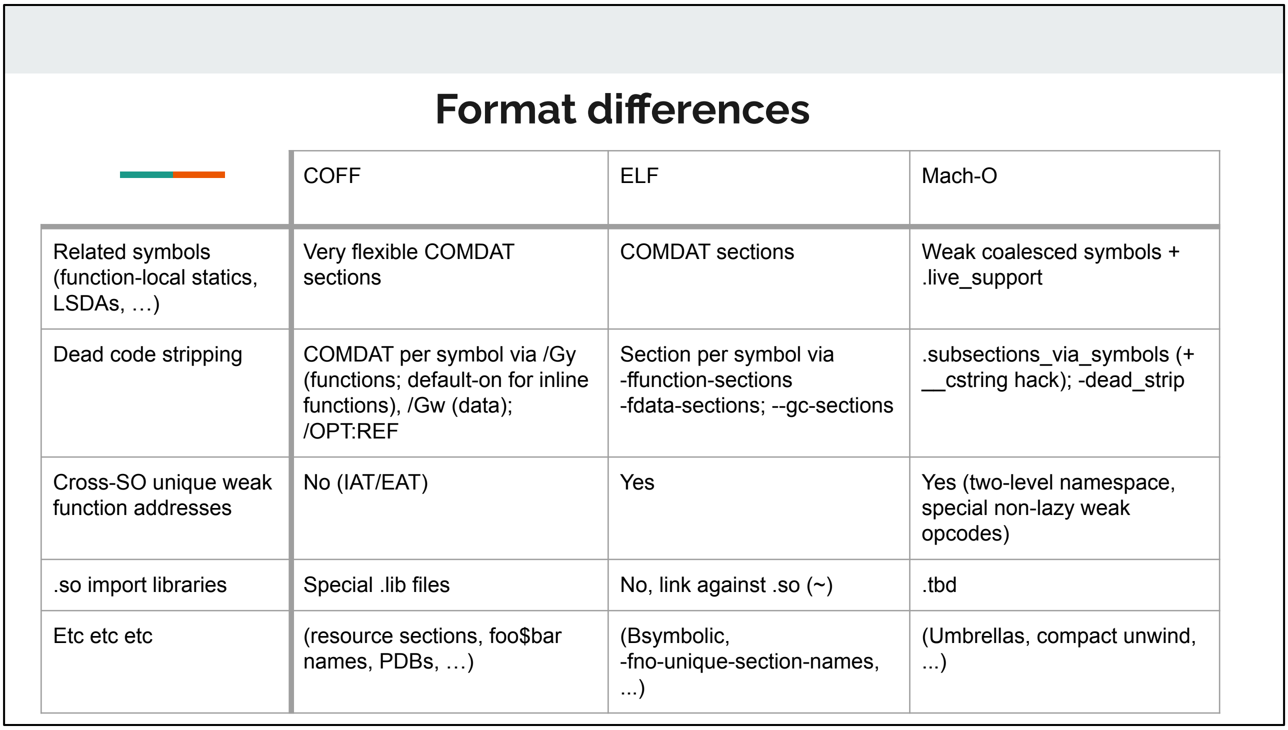


Design overview

- lld is a collection of linkers
  - Largely separate codebases, but similar architecture
- COFF/ELF/Mach-O are so different in file format, section semantics, relocations that code sharing isn't worth the cost of abstracting the shared bits
- Some code sharing via `lld/Common`, but not much

Thanks Nico. Now, I'll give a high level overview of the design.

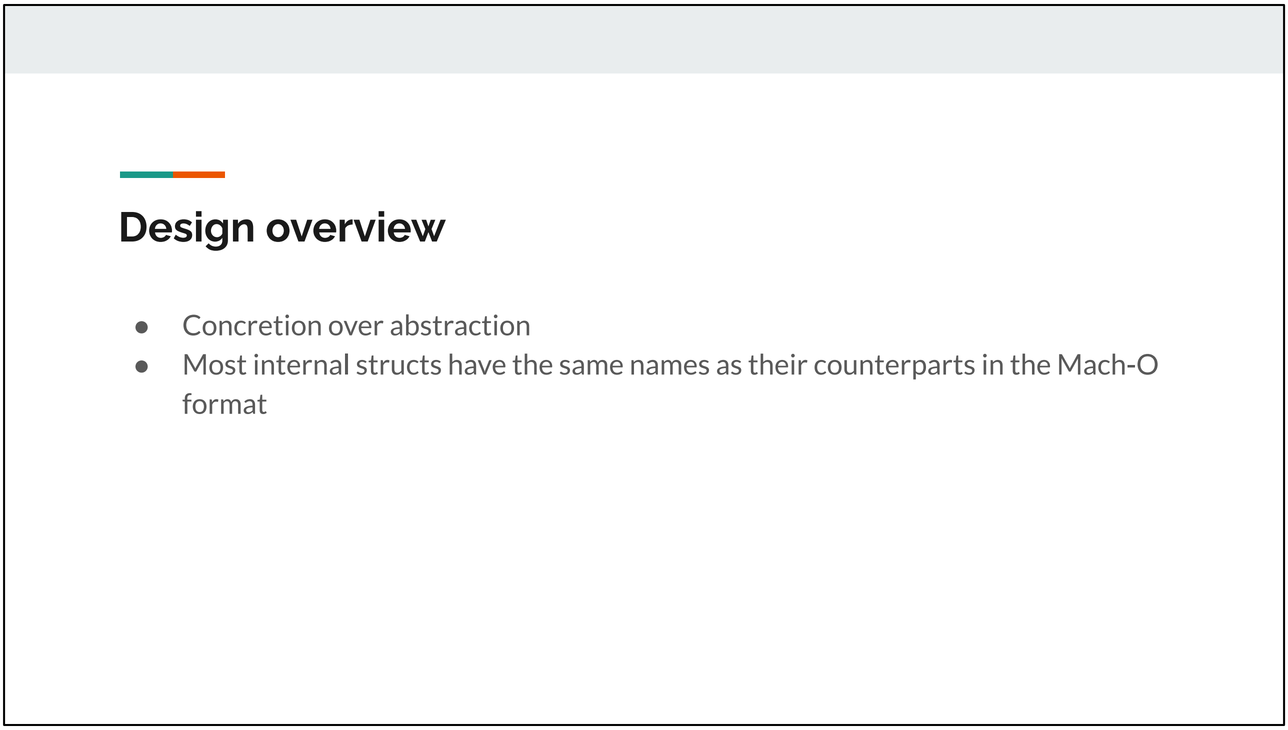
Code-wise, LLD is a collection of three largely separate linkers. The reason is that the COFF, ELF, and Mach-O formats have *so* many differences that most code sharing isn't worth it. All we share is basic utility code.



Format differences

	COFF	ELF	Mach-O
Related symbols (function-local statics, LSDAs, …)	Very flexible COMDAT sections	COMDAT sections	Weak coalesced symbols + .live_support
Dead code stripping	COMDAT per symbol via /Gy (functions; default-on for inline functions), /Gw (data); /OPT:REF	Section per symbol via -ffunction-sections -fdata-sections; --gc-sections	.subsections_via_symbols (+ __cstring hack); -dead_strip
Cross-SO unique weak function addresses	No (IAT/EAT)	Yes	Yes (two-level namespace, special non-lazy weak opcodes)
.so import libraries	Special .lib files	No, link against .so (~)	.tbd
Etc etc etc	(resource sections, foo$bar names, PDBs, …)	(Bsymbolic, -fno-unique-section-names, ...)	(Umbrellas, compact unwind, ...)

Here is a truncated list of all the differences between the formats. Don't worry, I know there's a lot on this slide, but I'm not going to dive into each of them. I just want to underscore how difficult it would be to have a common abstraction over all these differences.

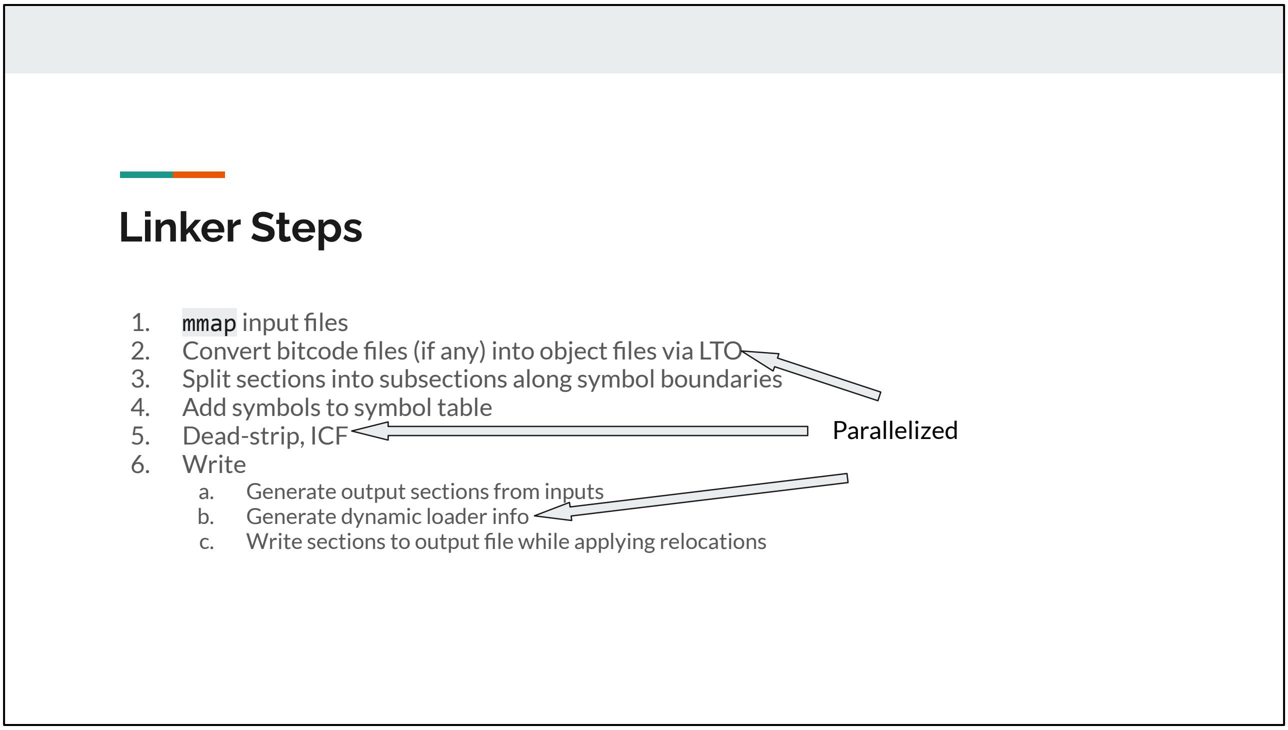


Design overview

- Concretion over abstraction
- Most internal structs have the same names as their counterparts in the Mach-O format

The bottom line is that LLD has chosen to favor concretion over abstraction. This means less indirection, less marshaling of data, and therefore, better linker performance.

Additionally, most of our internal data structures have the same names as their counterparts in the Mach-O file format. This ties back to our goal of hackability: someone familiar with the file format but not the LLD codebase should nonetheless be able to ramp up quickly.



Linker Steps

1. `mmap` input files
2. Convert bitcode files (if any) into object files via LTO
3. Split sections into subsections along symbol boundaries
4. Add symbols to symbol table
5. Dead-strip, ICF
6. Write
   a. Generate output sections from inputs
   b. Generate dynamic loader info
   c. Write sections to output file while applying relocations

Parallelized

Next, I want to talk about some quirks of the Mach-O format, but first, let me give you an overview of the linker's operations to show you how things fit together.

The operation of the linker can be roughly broken down into these six steps. First, we read in our inputs and perform LTO. Then, we split each object file's sections along symbol boundaries; this splitting is something unique to the Mach-O format, and I will speak more about that shortly.

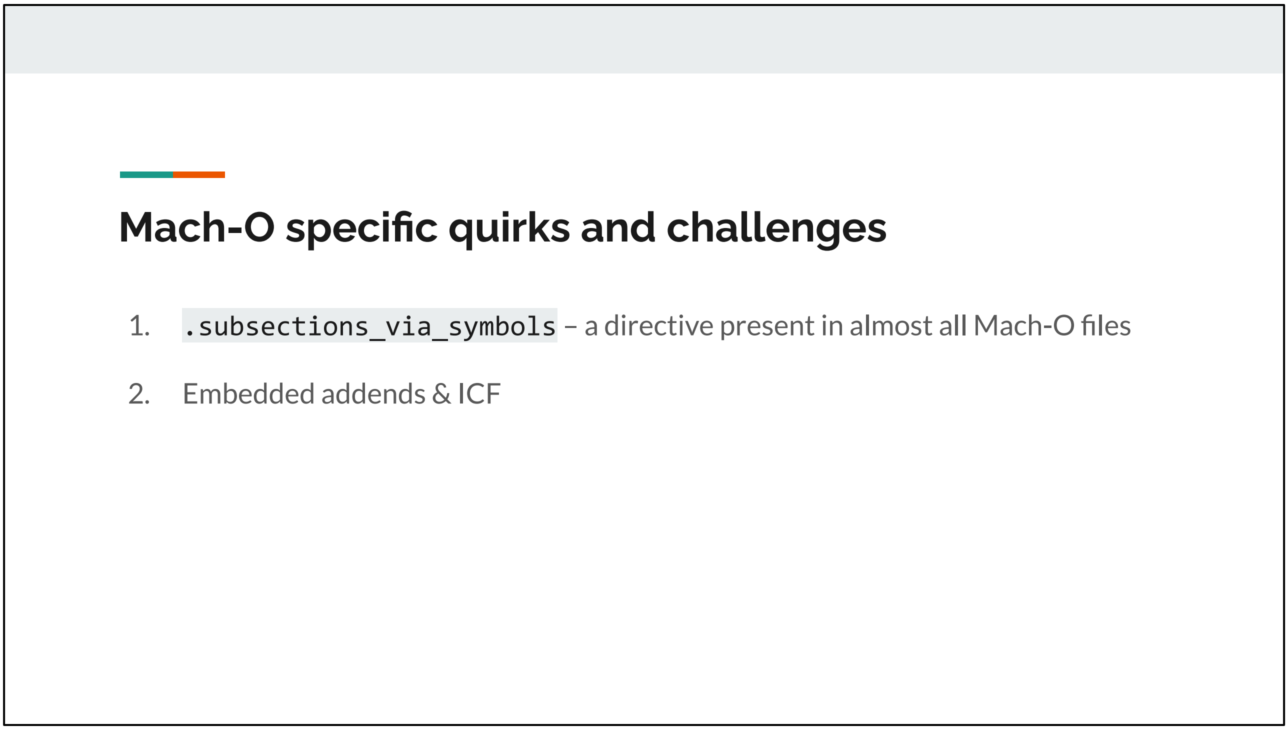
The fourth step is to perform symbol resolution, where we load various files from archives to satisfy our symbol references.

Assuming we've resolved all symbol references successfully, we now have enough information to emit a working binary. However, for release builds, we want to make sure the binary we emit is as small as possible. So we follow up by performing the linker's two most important optimizations: stripping dead code and identical code folding, or ICF for short.

Finally we write our output binary.

CLICK

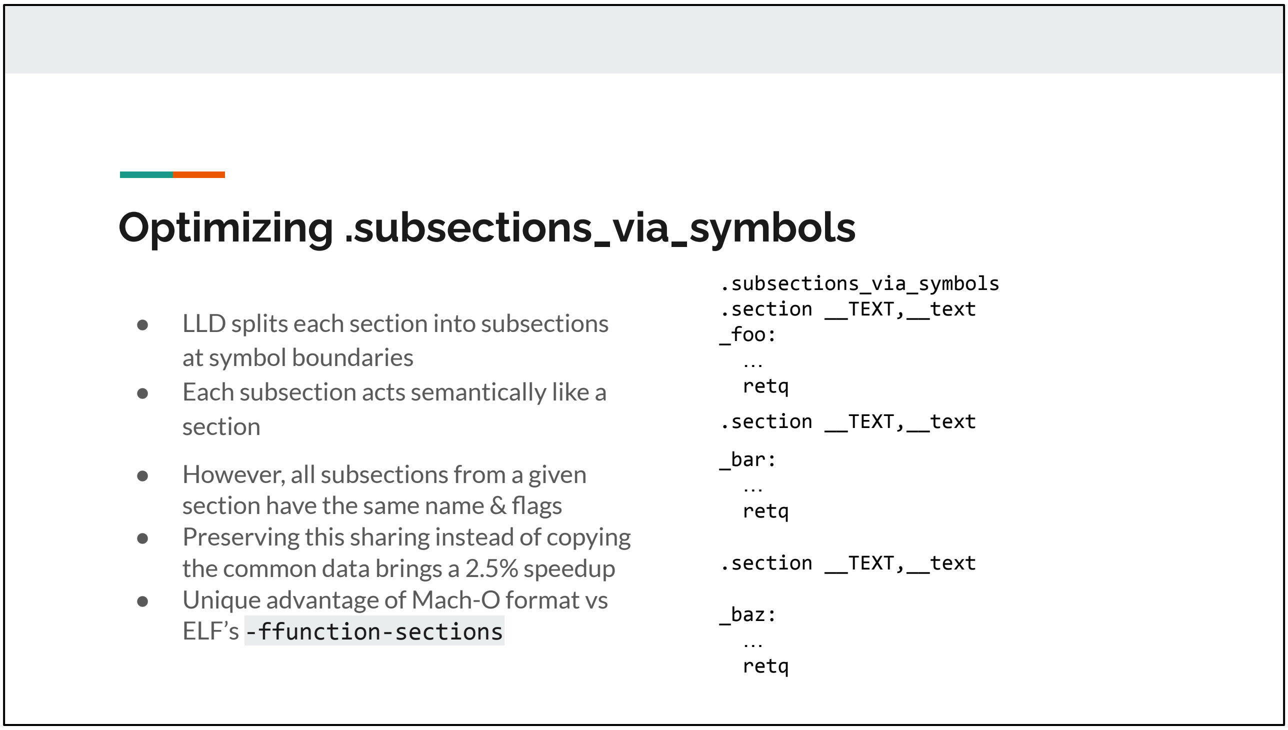
Now as you can see, we've only parallelized some of these steps. Most, if not all, of the others can be parallelized too, so there's definitely quite a lot more performance juice left to be squeezed.



Mach-O specific quirks and challenges

1. `.subsections_via_symbols` – a directive present in almost all Mach-O files
2. Embedded addends & ICF

Now that you have a sense of what the linker does, let me dive into two of Mach-O's format-specific quirks.



Optimizing .subsections_via_symbols

- LLD splits each section into subsections at symbol boundaries
- Each subsection acts semantically like a section
- However, all subsections from a given section have the same name & flags
- Preserving this sharing instead of copying the common data brings a 2.5% speedup
- Unique advantage of Mach-O format vs ELF's `-ffunction-sections`

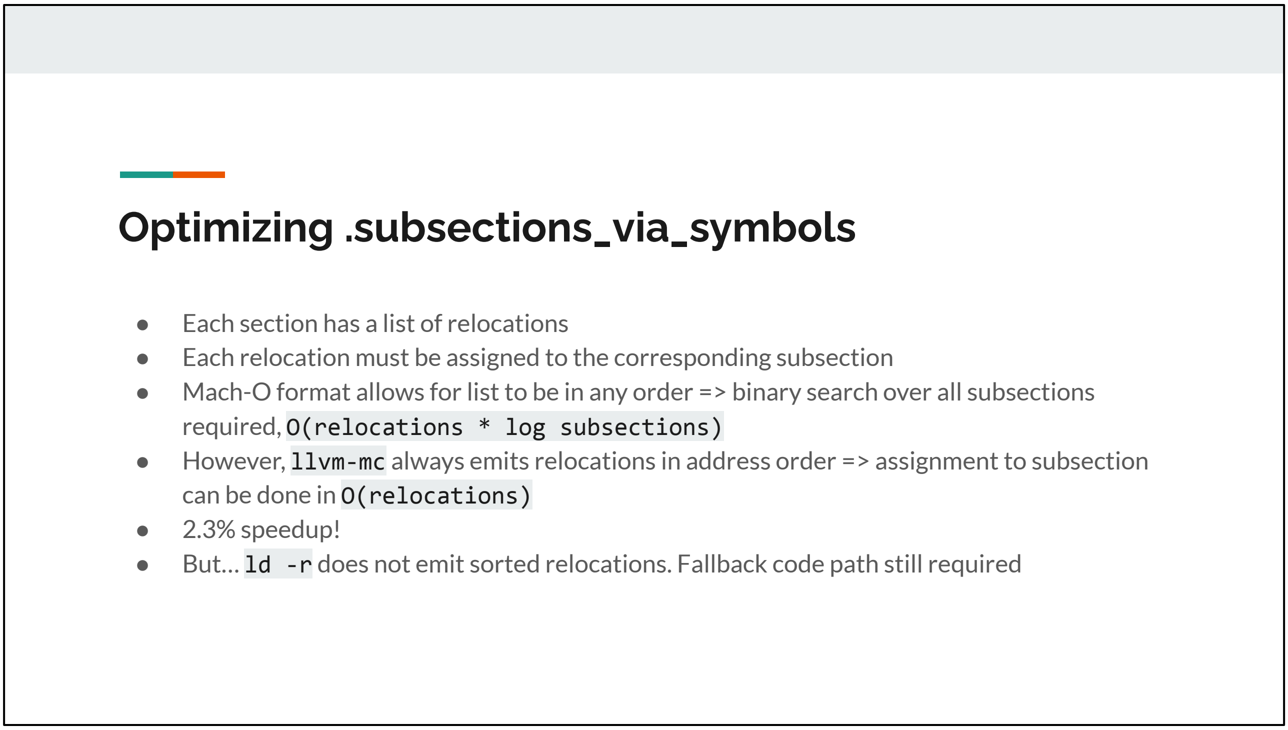
```
.subsections_via_symbols
.section __TEXT,__text
_foo:
  …
  retq
.section __TEXT,__text

_bar:
  …
  retq

.section __TEXT,__text

_baz:
  …
  retq
```

.subsections_via_symbols is a Mach-O specific directive that tells the linker it can break a section along symbol boundaries. Roughly speaking, it means that each function ends up in its own section. On the right, we have an assembly snippet containing three different symbols. The subsections_via_symbols directive causes the linker to implicitly insert each one into its own section, like so: CLICK

This allows us to optimize functions piecemeal: we can remove a function if it is dead, merge two identical functions, etc. Without this directive, we would only be able to remove a function if all the other functions in the same section were also dead, which almost never happens in practice.

In our initial implementation of this feature, we created a new section struct for every subsection, copying all the section metadata each time. Later we improved this by having each subsection share a common metadata struct. The memory / copying savings from this gave us a 2.5% speedup.

.subsections_via_symbols is truly one of the unique advantages of the Mach-O format; ELF does not have this feature, and so is forced to emulate it at the compiler level using the `-ffunction-sections` flag, which explicitly creates a new section for each function symbol. But that results in more object file data for the linker to ingest.



Optimizing .subsections_via_symbols

- Each section has a list of relocations
- Each relocation must be assigned to the corresponding subsection
- Mach-O format allows for list to be in any order => binary search over all subsections required, `O(relocations * log subsections)`
- However, `llvm-mc` always emits relocations in address order => assignment to subsection can be done in `O(relocations)`
- 2.3% speedup!
- But… `ld -r` does not emit sorted relocations. Fallback code path still required

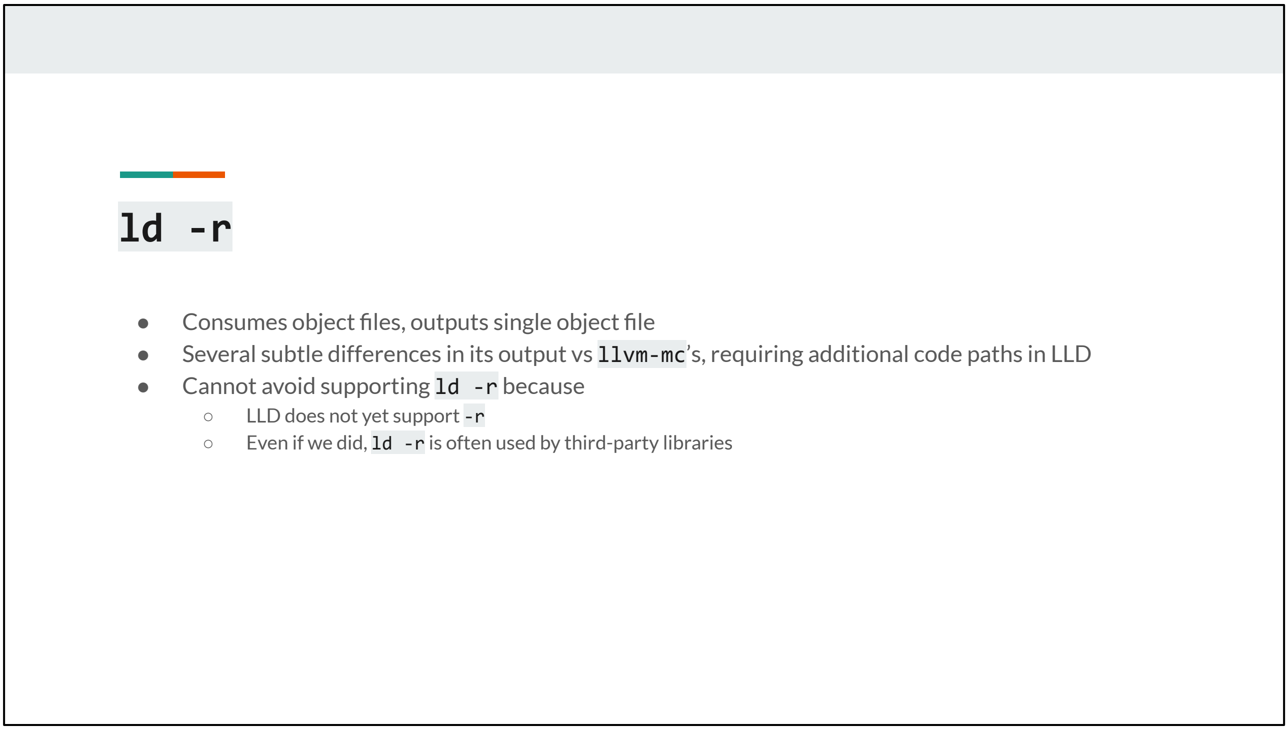
Another clever optimization we did involves the assignment of relocations to their respective subsections.

Each section has an associated list of relocations. When we split that section up, we end up with a list of subsections sorted by address. However, we have no such ordering guarantees for the list of relocations. Therefore, our initial implementation performed a binary search over all subsections for every given relocation.

Later on, however, I noticed that MC *always* emits relocations in address order. Thus, in practice, we can perform relocation assignment in O(N) time. Making this change gave us another 2.3% speedup!

However… ld dash R does not emit sorted relocations, so a fallback to the binary search code path is still required.

Now you may be wondering, what is ld dash R? That is what I'm going to talk about next.



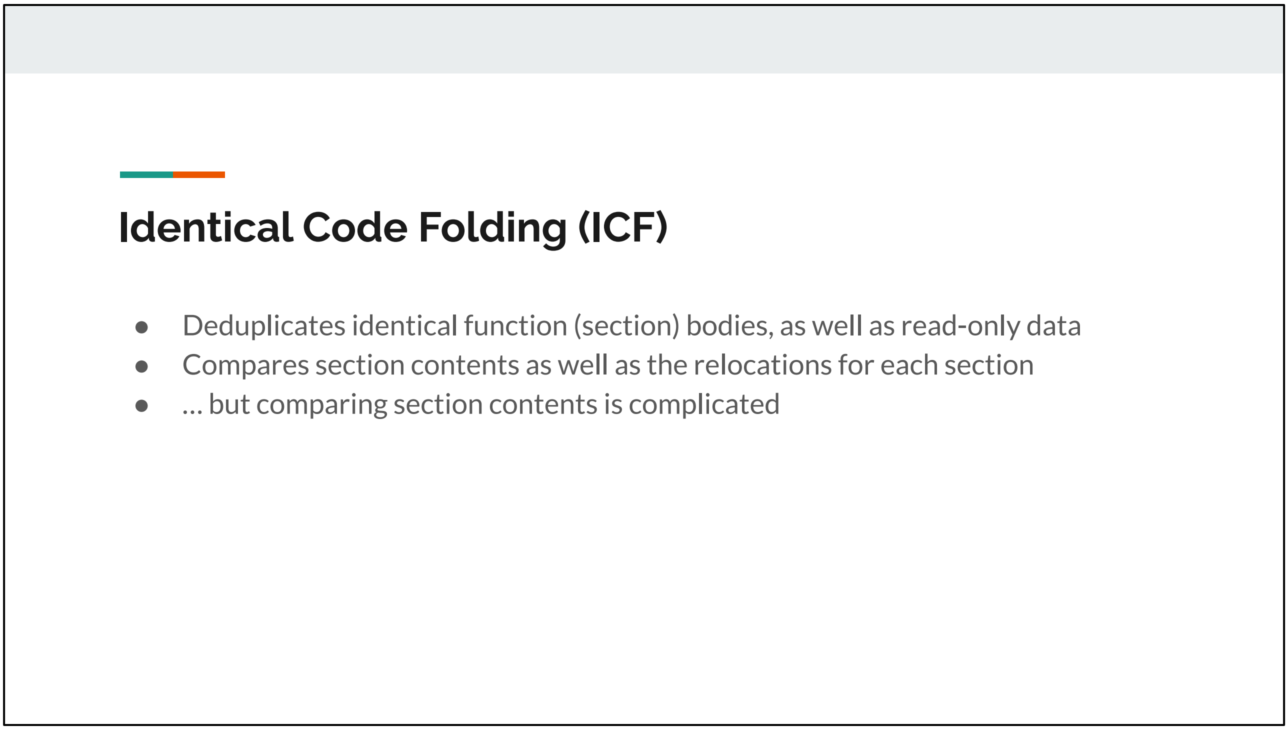
`ld -r`

- Consumes object files, outputs single object file
- Several subtle differences in its output vs `llvm-mc`’s, requiring additional code paths in LLD
- Cannot avoid supporting `ld -r` because
  - LLD does not yet support `-r`
  - Even if we did, `ld -r` is often used by third-party libraries

The -r flag tells the linker to consume several object files and combine them into a single output object file, which can of course be used as input into another link step. As you might expect, it’s typically used to package libraries. Unfortunately, ld64’s -r output has several subtle differences compared to MC’s. As a result, LLD has to use several additional code paths to handle ld64’s quirks.

We would love to drop this support at some point, but it probably won’t happen any time soon. First, LLD itself does not yet support dash R. And *even* if we did, LD dash R’s output is often encountered when importing third-party libraries whose builds the user doesn’t control.

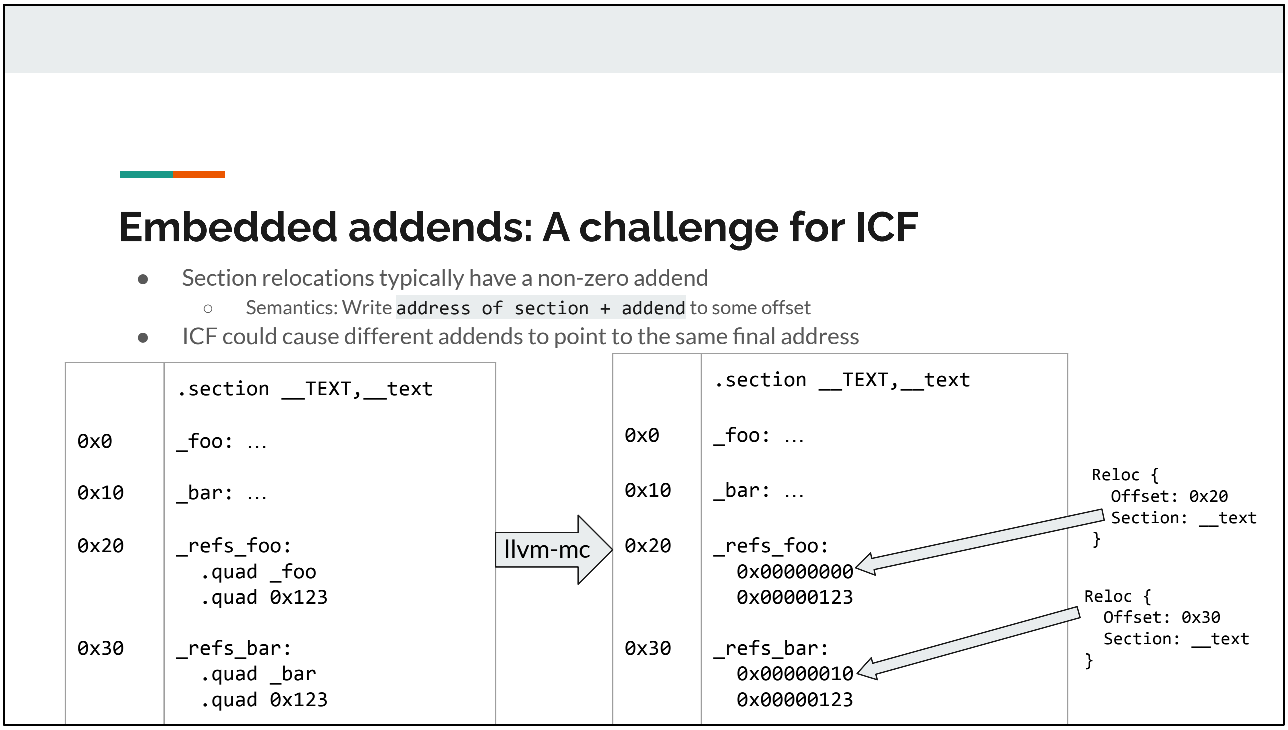
All that said, we would love to discuss some of these differences with Apple linker folks. Perhaps they can be resolved in a future version of ld64, enabling us to further simplify our code.



Identical Code Folding (ICF)

- Deduplicates identical function (section) bodies, as well as read-only data
- Compares section contents as well as the relocations for each section
- … but comparing section contents is complicated

Next, I would like to talk about the challenges we've faced with identical code folding. To recap, here's a brief description of what ICF does. I'm not going into details about the algorithm, but the core idea is that we compare section contents and relocations to determine which sections are identical.



Embedded addends: A challenge for ICF

- Section relocations typically have a non-zero addend
  - Semantics: Write `address of section + addend` to some offset
- ICF could cause different addends to point to the same final address

```
        .section __TEXT,__text

0x0     _foo: …

0x10    _bar: …

0x20    _refs_foo:
          .quad _foo
          .quad 0x123

0x30    _refs_bar:
          .quad _bar
          .quad 0x123
```

llvm-mc

```
        .section __TEXT,__text

0x0     _foo: …

0x10    _bar: …

0x20    _refs_foo:
          0x00000000
          0x00000123

0x30    _refs_bar:
          0x00000010
          0x00000123
```

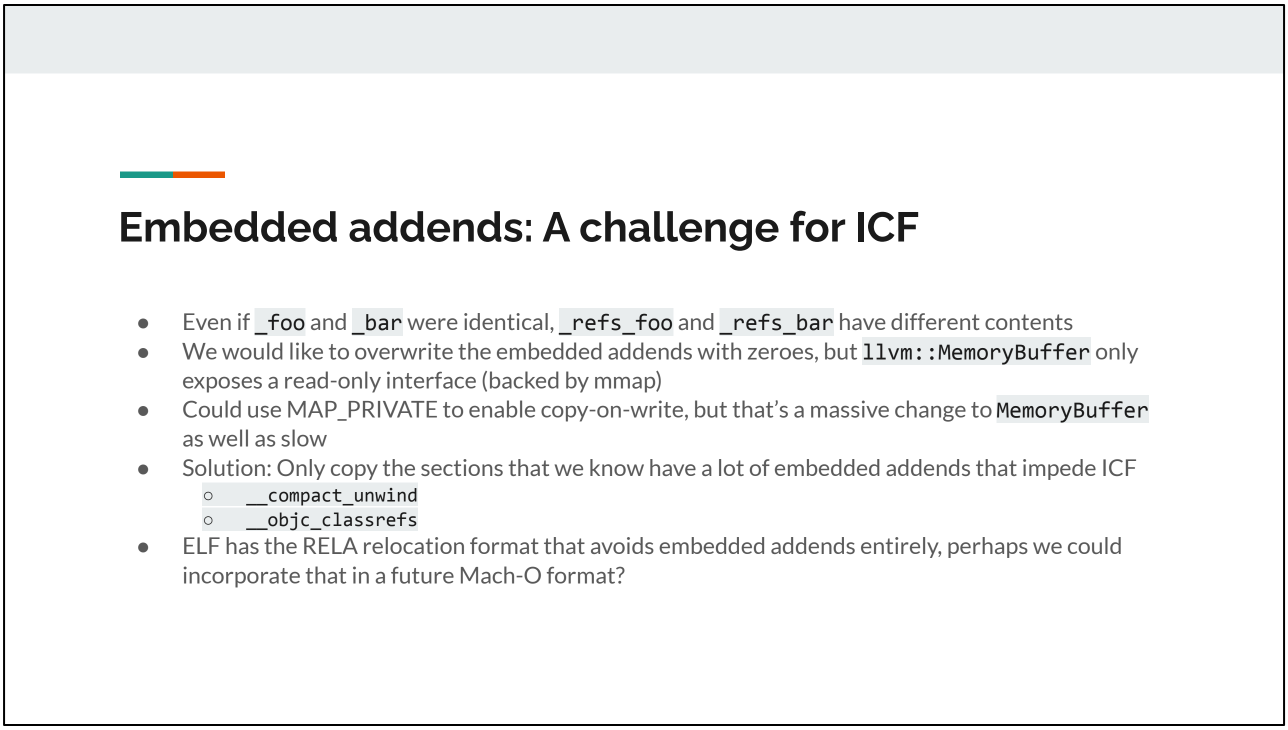
```
Reloc {
  Offset: 0x20
  Section: __text
}
```

```
Reloc {
  Offset: 0x30
  Section: __text
}
```

What complicates ICF is the fact that relocation data may be embedded in the section contents. Let me illustrate with an example. In the code snippet below, we have two functions foo and bar, and two more functions that reference them. We can see that if foo and bar are identical, then refs_foo and refs_bar must also be identical, and ICF should be able to merge them.

However, let us see what happens when we run this code through MC. We get the object file on the right. Refs_foo and refs_bar each have an associated relocation. However, the relocation structs themselves don't contain the addresses of foo and bar. Instead, these addresses are stored in the section data as embedded addends. Since bar is at address 0x10, refs_bar has that as its embedded addend.

As a result, if we naively compare the section contents, we will incorrectly conclude that refs_foo and refs_bar can never be identical.

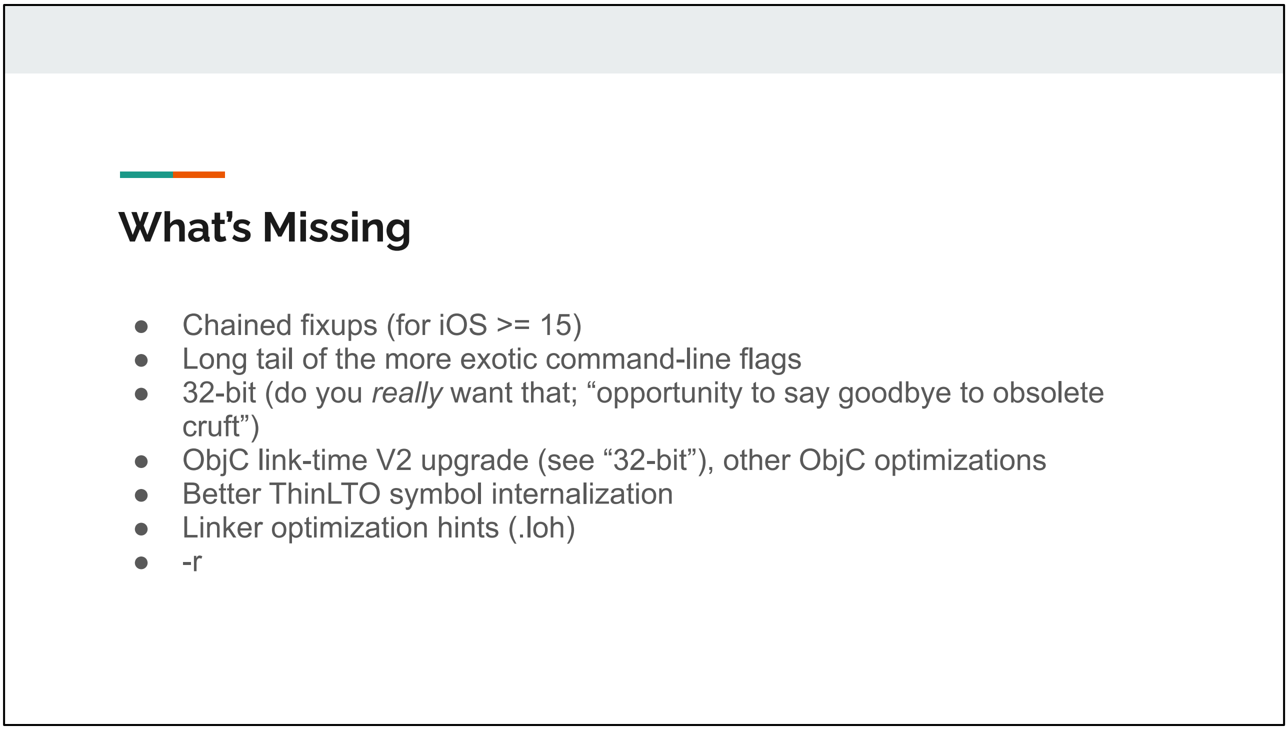


Embedded addends: A challenge for ICF

- Even if `_foo` and `_bar` were identical, `_refs_foo` and `_refs_bar` have different contents
- We would like to overwrite the embedded addends with zeroes, but `llvm::MemoryBuffer` only exposes a read-only interface (backed by mmap)
- Could use MAP_PRIVATE to enable copy-on-write, but that's a massive change to `MemoryBuffer` as well as slow
- Solution: Only copy the sections that we know have a lot of embedded addends that impede ICF
  - `__compact_unwind`
  - `__objc_classrefs`
- ELF has the RELA relocation format that avoids embedded addends entirely, perhaps we could incorporate that in a future Mach-O format?

So how can we work around this problem? Well, we could extract the embedded addends into a separate structure, and overwrite their locations in the section data with zeroes. That would allow us to compare section data without worrying about the differences in addends. However, there are two catches to this: first, the MemoryBuffer API only exposes a read-only interface backed by mmap. We could modify it to use copy-on-write instead, but we would still incur a significant performance overhead.

Our solution to this was to copy just the sections for which we know have a lot of embedded addends that impede ICF. This gives us the most bang for our [performance] buck, so to speak. While this works well enough, perhaps we should look to ELF for a nicer long-term solution. ELF has the RELA relocation format that avoids embedded addends entirely. If Apple engineers are willing, perhaps we could use a similar approach in a future Mach-O format.



What's Missing

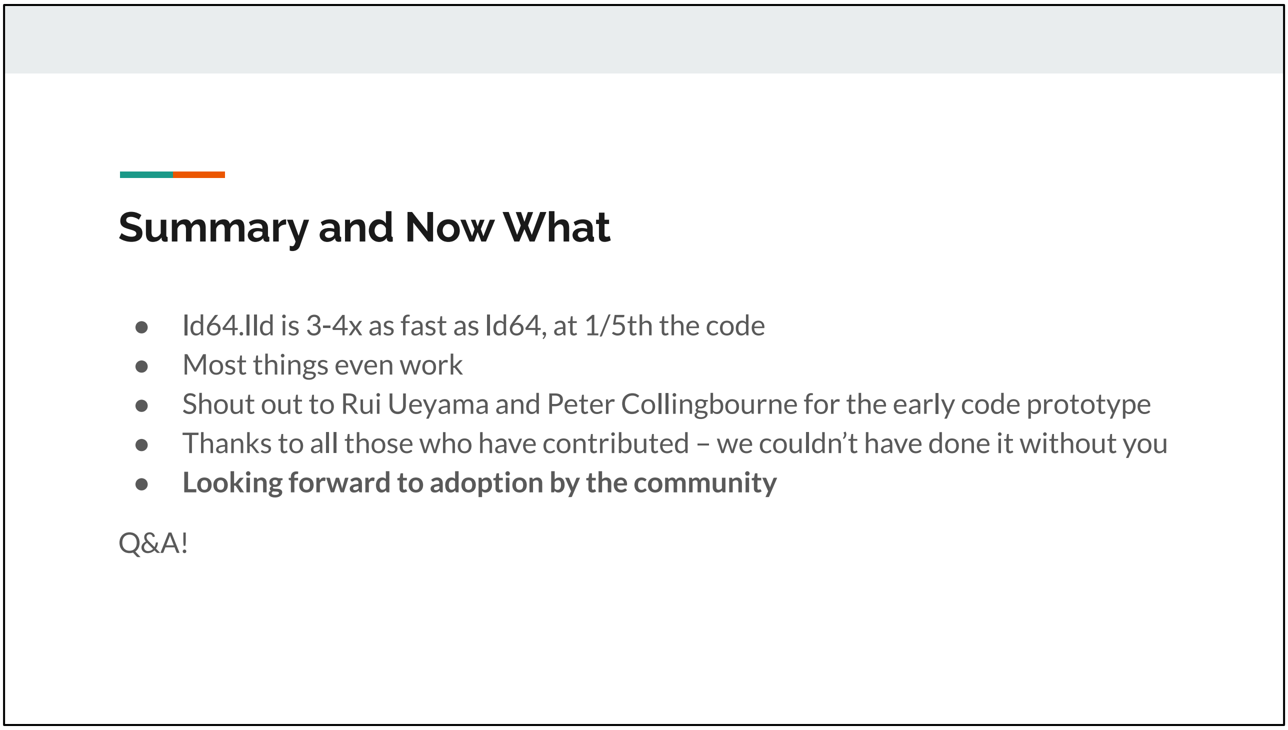
- Chained fixups (for iOS >= 15)
- Long tail of the more exotic command-line flags
- 32-bit (do you *really* want that; “opportunity to say goodbye to obsolete cruft”)
- ObjC link-time V2 upgrade (see “32-bit”), other ObjC optimizations
- Better ThinLTO symbol internalization
- Linker optimization hints (.loh)
- -r

Alright, having talked about all the things LLD can do, let me talk about some things it can’t.

Here’s a brief list of some ld64 features that we know are lacking in LLD. In general, we haven’t prioritized them because they are either for target platforms that are too new or too old.

For instance, chained fixups are a more compact way of encoding fixups, but they can only be used if you are targeting a minimum iOS version of 15. We’ll eventually want to support this, of course.

On the other hand, features that are only needed for old target platforms may never be supported. A prime example is 32-bit x86 support. The code complexity it would add just doesn’t seem worth it.



Summary and Now What

- ld64.lld is 3-4x as fast as ld64, at 1/5th the code
- Most things even work
- Shout out to Rui Ueyama and Peter Collingbourne for the early code prototype
- Thanks to all those who have contributed – we couldn’t have done it without you
- **Looking forward to adoption by the community**

Q&A!

And that’s where we are. In summary, LLD is about 3-4 times as fast as ld64, at approximately 1/5th the code. Most things work pretty well, at least for the use cases we’ve had.

I would like to give a shout out to Rui and to Peter for their early code prototype of LLD’s Mach-O backend, which served as a great launch-off point for the linker we have today.

I would also like to thank all those who have contributed – there are so many of you – we couldn’t have done it without your help.

Finally, we look forward to LLD’s adoption by the community. We would love it if you could try incorporating it into your build systems and reporting any bugs and performance improvements that you find.

All right, let’s open it up to the floor for questions and answers. Thank you!