# lld: A Fast, Simple and Portable Linker

Rui Ueyama <ruiu@google.com>
LLVM Developers' Meeting 2017

# Talk overview

1. Implementation status
2. Design goals
3. Comparisons with other linkers
4. Concurrency
5. Semantic differences
6. Miscellaneous features

# Implementation status

# Implementation status

lld supports ELF (Unix), COFF (Windows) and Mach-O (macOS)

- lld/ELF is production-ready. It can build the entire FreeBSD/AMD64 system including the kernel (It is /usr/bin/ld in FreeBSD-CURRENT)
- lld/COFF is complete including PDB debug info support
- lld/Mach-O is unfinished

(I'll be talking about ELF in the rest of this presentation.)

# Design goals

# Design goals

1. Simple
   - lld is significantly simpler than GNU linkers
   - Simple = easy to understand, easy to add features, easy to experiment new ideas, etc.
2. Fast
   - lld is significantly faster than GNU linkers
   - This is the *only* thing that matters to most users
3. Easy to use
   - lld takes the command line options and linker scripts
   - lld can be used just by replacing /usr/bin/ld or lld-link.exe

# How to use lld

- Replace /usr/bin/ld with lld, or
- Pass -fuse-ld=lld to clang

# Speed comparisons

# Two GNU linkers

GNU binutils have two linkers, bfd and gold

- bfd linker got ELF support in 1993
- gold started in 2006 as a ELF-only, faster replacement for bfd

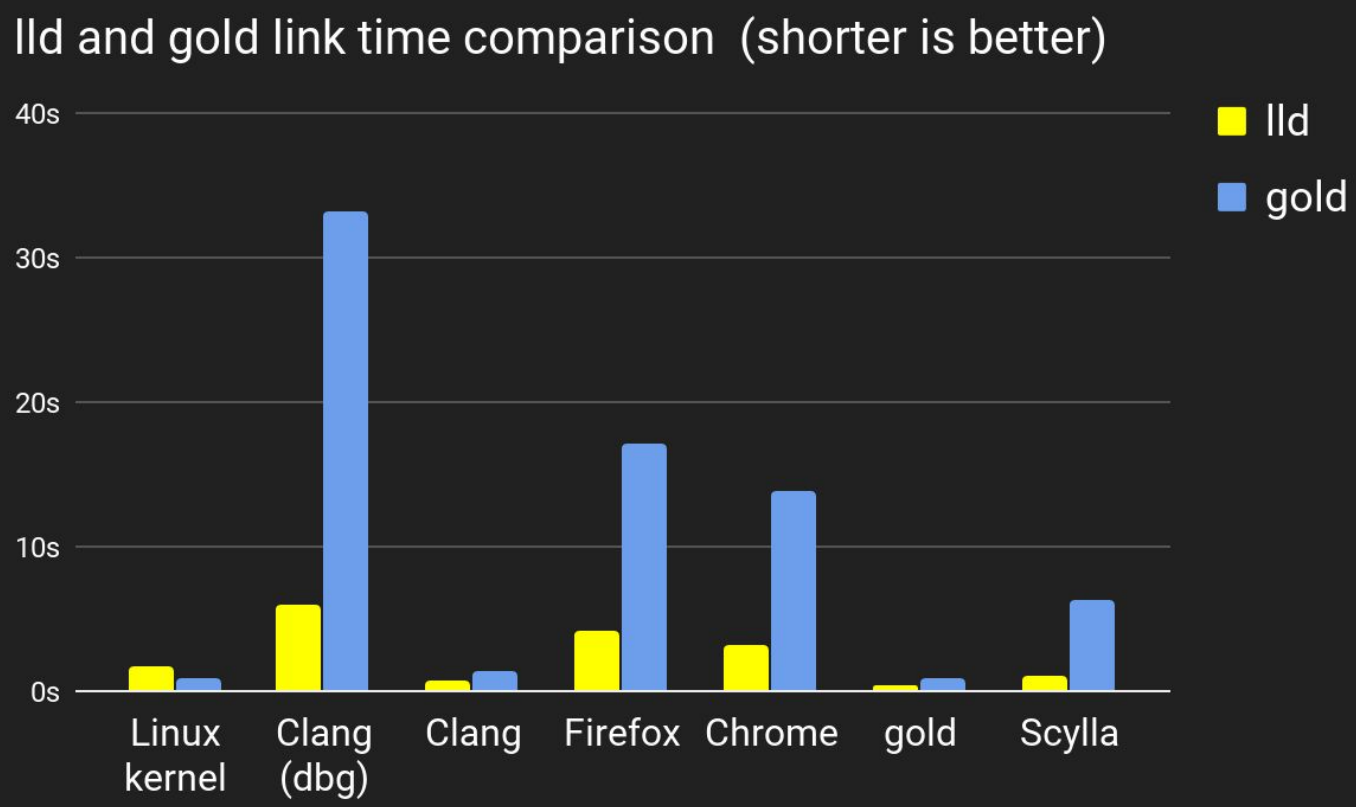
bfd linker is written on top of the BFD library. gold is written to completely remove that abstraction layer, and that's why gold is much faster than bfd.

lld is written from scratch just like gold, and it is significantly faster than gold.

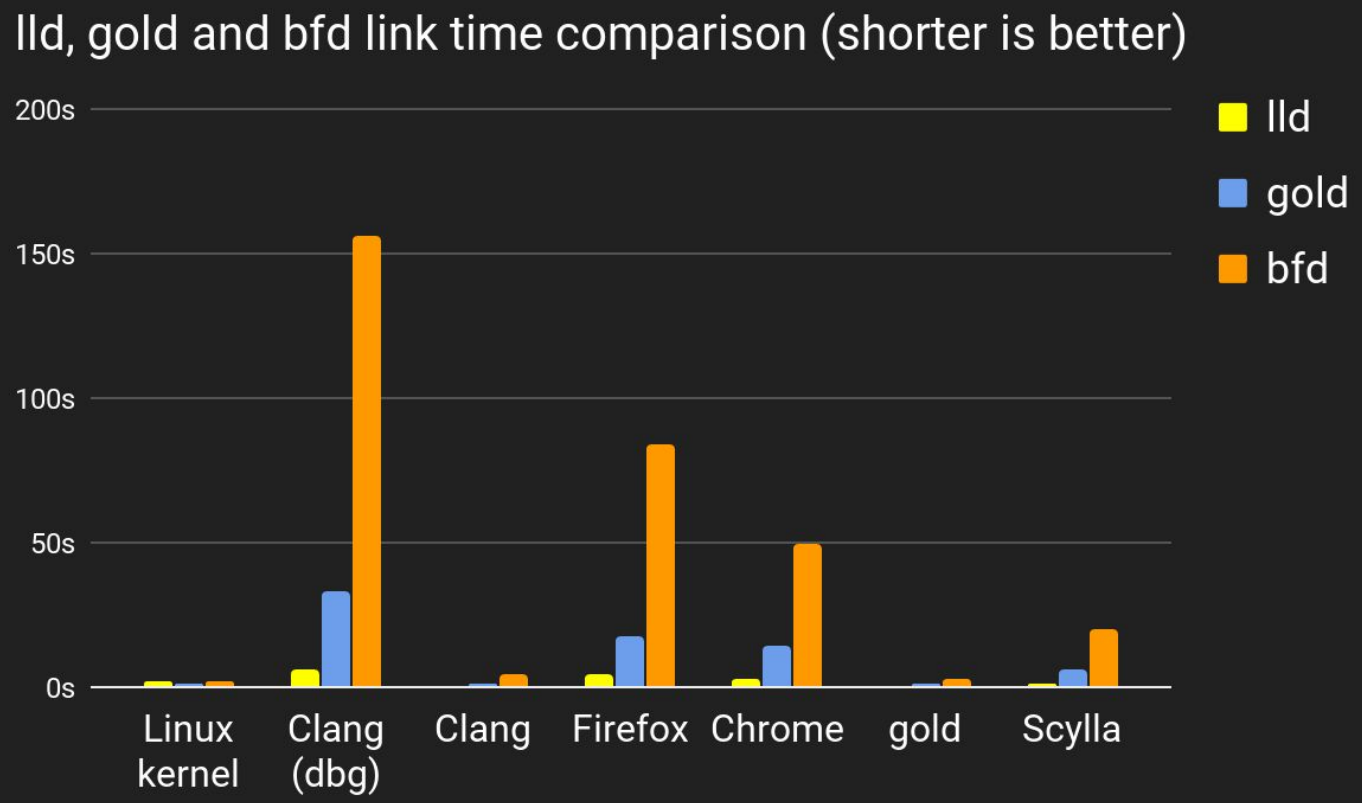
# How much faster?

- In general testing, lld ranges from two to four times as fast as gold
- lld is better at large programs, which is when the link time matters most

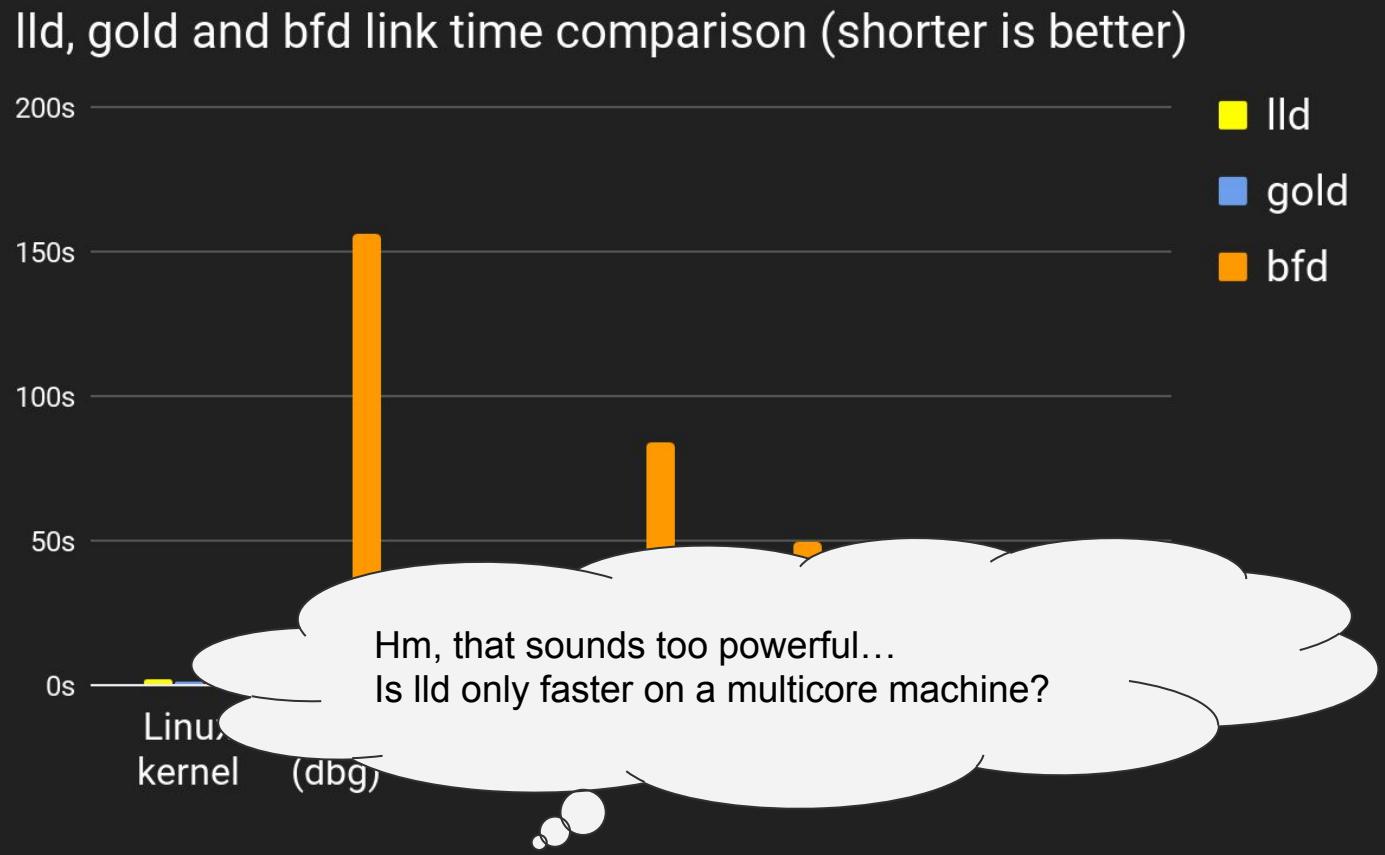
† It depends on target programs, number of available cores, and command line options




(Measured on a 2-socket 20-core 40-thread Xeon E5-2680 2.80 GHz machine with an SSD drive)

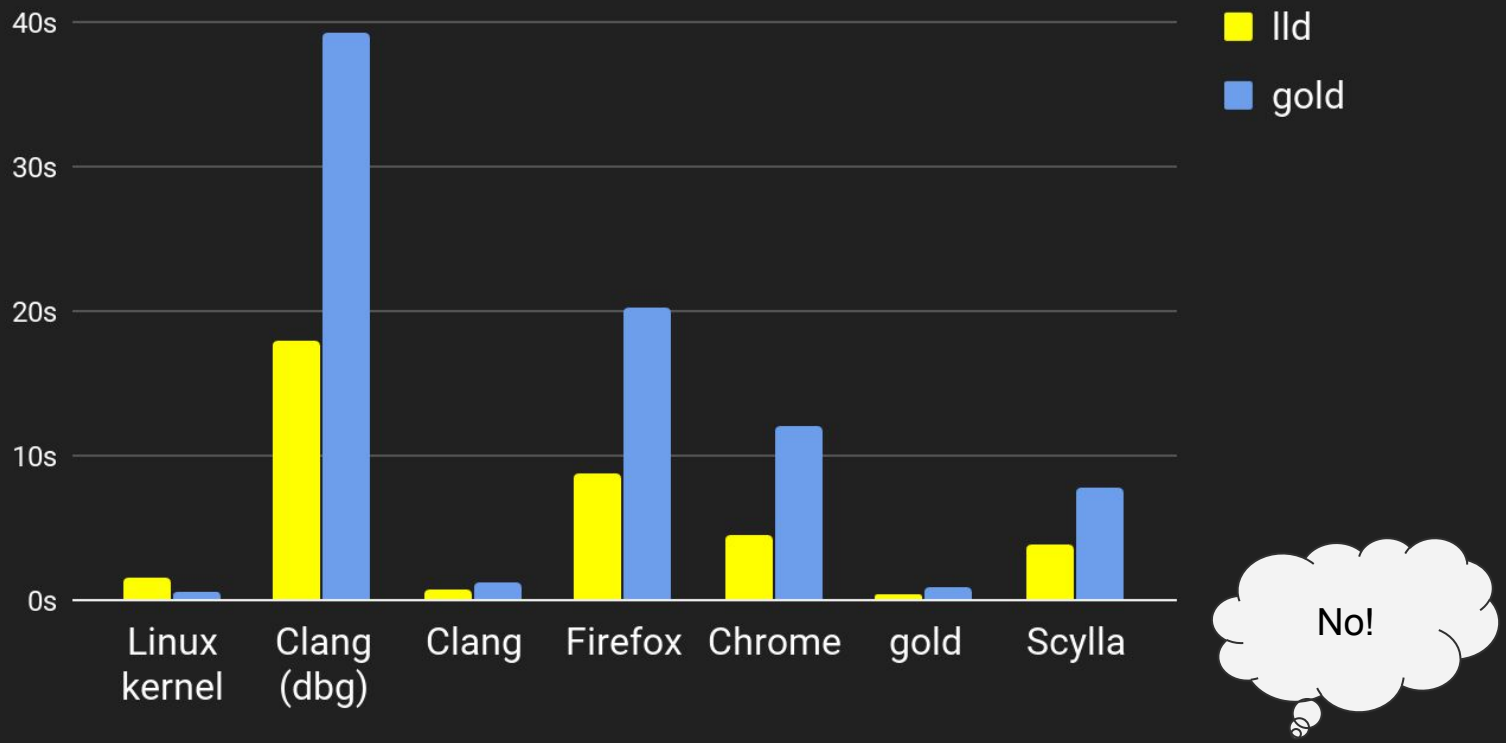



(Measured on a 2-socket 20-core 40-thread Xeon E5-2680 2.80 GHz machine with an SSD drive)




(Measured on a 2-socket 20-core 40-thread Xeon E5-2680 2.80 GHz machine with an SSD drive)

## lld and gold link time comparison (shorter is better)




Measured with the --no-threads option. lld is still much faster than gold.

# Optimization

We do *not* want to optimize our linker. We want to make it *naturally* faster because:

- *"Premature optimization is the root of all evil"* — Donald Knuth
- *"Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming."* — Rob Pike's 5 Rules of Programming

# The scale of our problems

Chrome with debug info is almost 2 GiB in output size. In order to produce the executable, the linker reads and processes

- 17,000 files,
- 1,800,000 sections,
- 6,300,000 symbols, and
- 13,000,000 relocations.

E.g. If you add 1 µs overhead for each symbol, it adds up to 6 seconds.

# String operations you can do in 1 µs

The average symbol length is 58 bytes for Chromium. In 1 µs, you can

- instantiate 80 symbol StringRefs
  - Symbol table is a sequence of null-terminated strings, so you have to call strlen() on each string
- compute hash values for 50 symbols
- insert 4 symbols into a hash table

Not that many you can do, particularly because C++ mangled symbols are long and inefficient. In lld, we designed the internal data structure so that we minimize hash table lookups (exactly once for each symbol).

# Concurrency

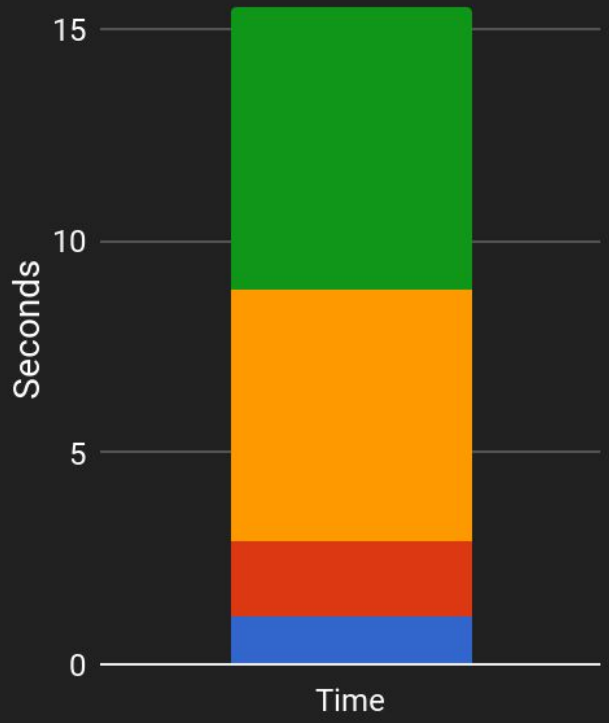
# Concurrency

When we made lld concurrent, we kept in mind:

- As always, *"Premature optimization is the root of all evil"*
- Debugging a concurrency bug is hard, so you want to keep it simple
- Amdahl's law — you want to extract as much parallelism as you can
- Adding more cores doesn't make slow programs magically fast, thus your program must be fast without threads
- Parallel algorithms shouldn't degrade single-core performance
- Output must be deterministic — "correct, but different" results are not acceptable because of build reproducibility.

Don't guess, but measure!

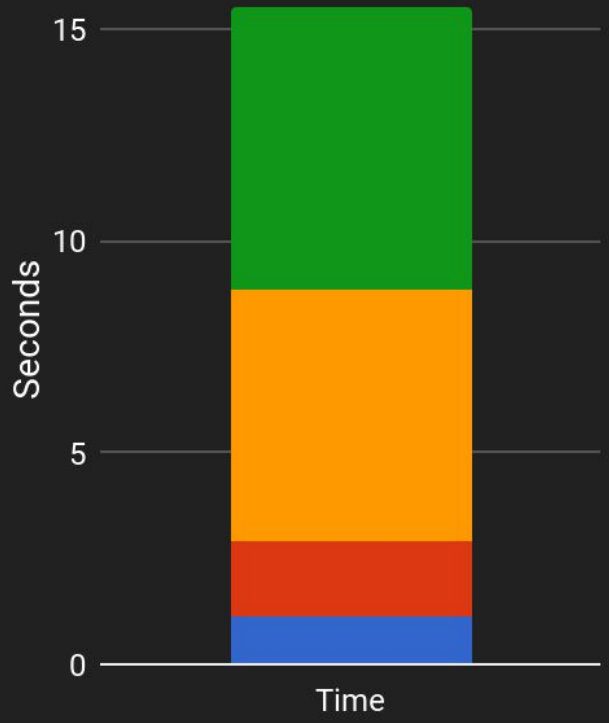
# Breakdown of lld executiontime (when linking clang dbg)




- Copy input files to output files and apply relocations
- Merge string constants
- Insert symbol to symbol tables
- Others

Only three small passes dominate most of the execution time! Can we parallelize them?

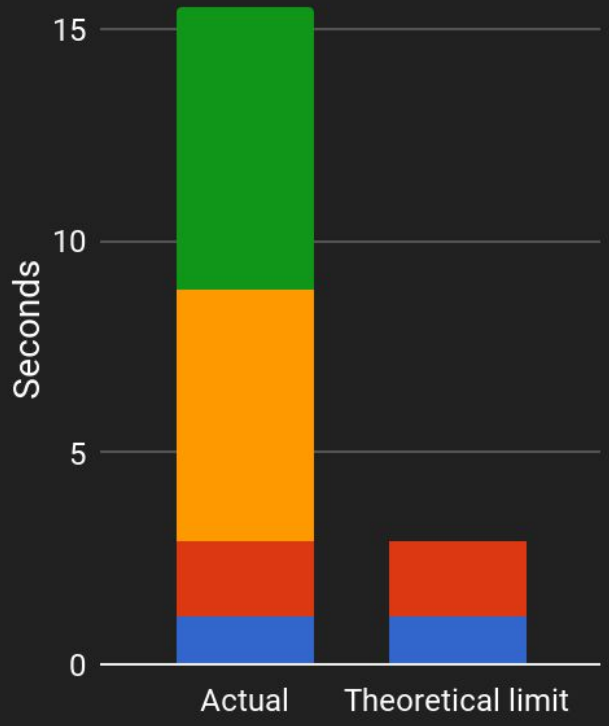
# Breakdown of lld executiontime (when linking clang dbg)




- Copy input files to output files and apply relocations — parallelizable
- Merge string constants — parallelizable
- Insert symbol to symbol tables — hard to parallelize
- Others

That means, by Amdahl's law, theoretical speedup limitation is...

# Breakdown of lld executiontime (when linking clang dbg)




- Copy input files to output files and apply relocations — parallelizable
- Merge string constants — parallelizable
- Insert symbol to symbol tables — hard to parallelize
- Others

That means, by Amdahl's law, theoretical speedup limitation is… this (if we have an infinite number of CPUs and there's zero multithreading overhead.)

# How to parallelize

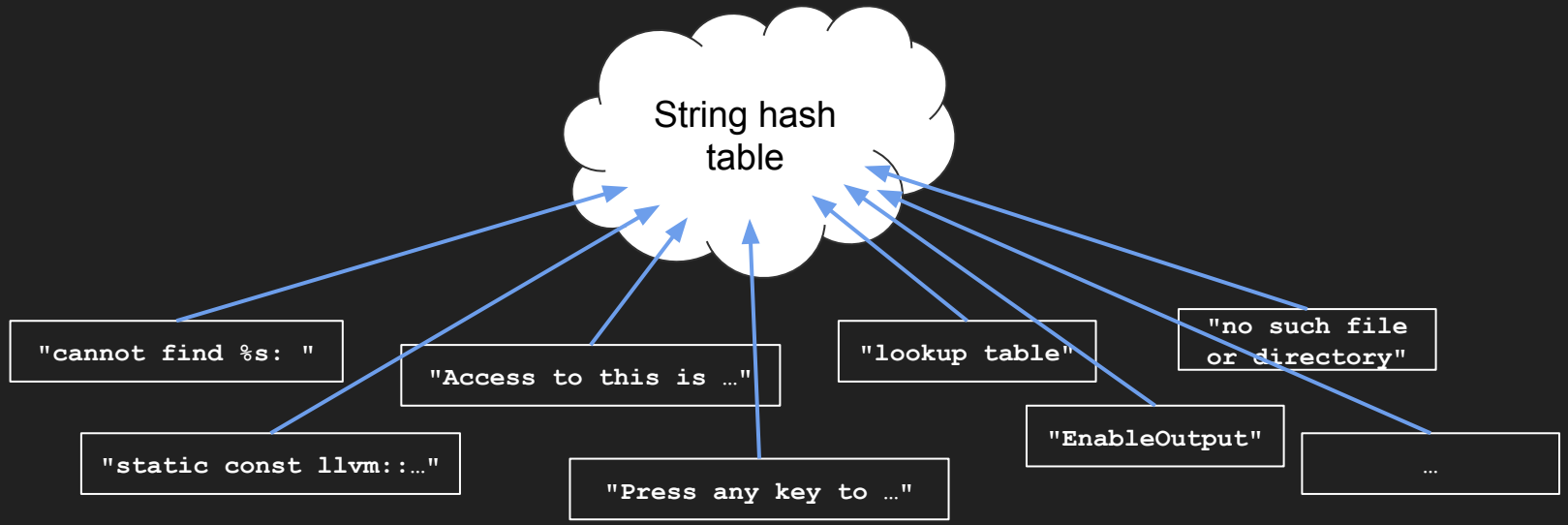
# There's no single recipe for concurrency

Making section copying & relocation application concurrent is easy because sections don’t share any state.

1. Assign non-overlapping file offset to sections
2. memcpy them to the output buffer in parallel
3. Make them apply relocations in the output buffer in parallel

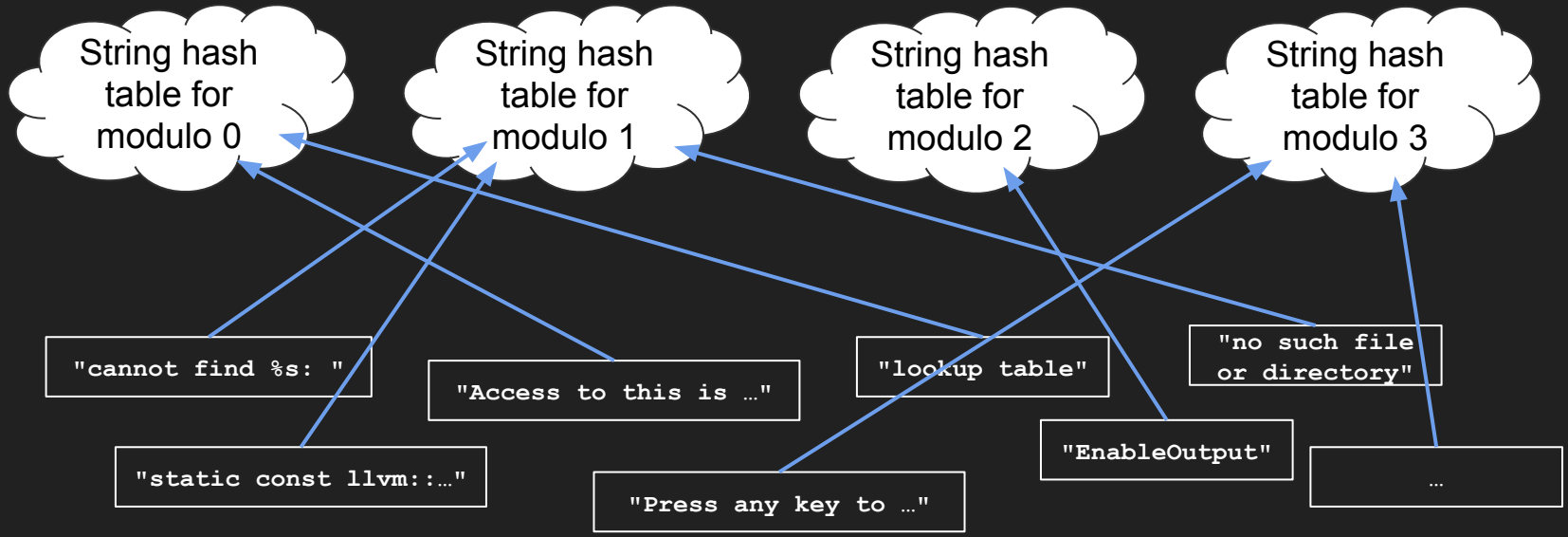
# String merging (before)

String merging is tricky. You want to uniquify strings, but how do you do that concurrently? Originally, we used a hash table and inserted all strings to it.



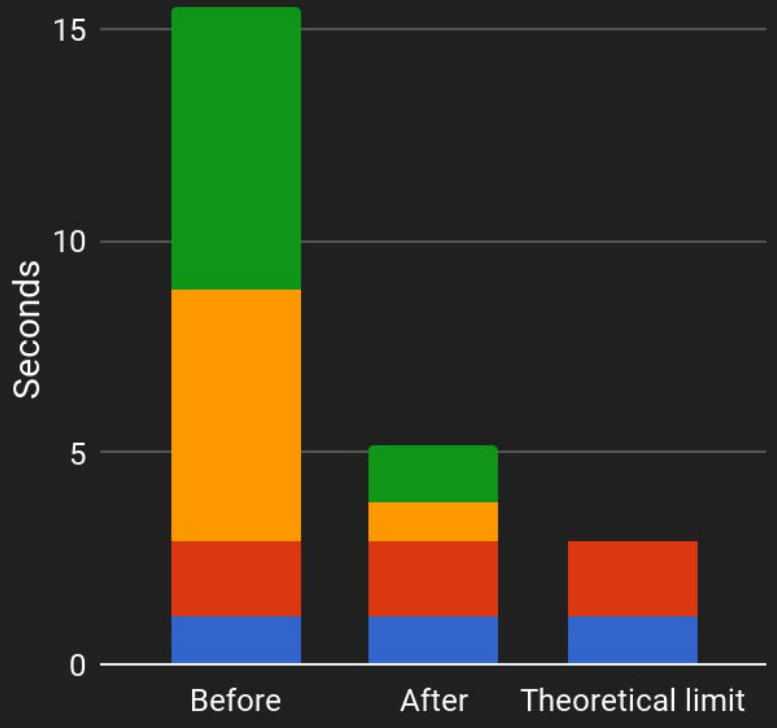
# String merging (after)

We split the hash table into multiple shards, and distribute workloads to the shards by modulus of string hash values. This doesn't require any locking, thus is fast.



# Optimization results

# Breakdown of lld executiontime (when linking clang dbg)




- Copy input files to output files and apply relocations — parallelizable
- Merge string constants — parallelizable
- Insert symbol to symbol tables — hard to parallelize
- Others

The concurrent version takes about 1/3 of the time compared to non-concurrent version.

# Good points of lld's concurrency model

- Most parts are still single-threaded, and we have no interest in making it concurrent.
- A few, significantly time-consuming passes are parallelized by custom concurrent algorithms. The complexity of multi-threading is hidden from the outside.

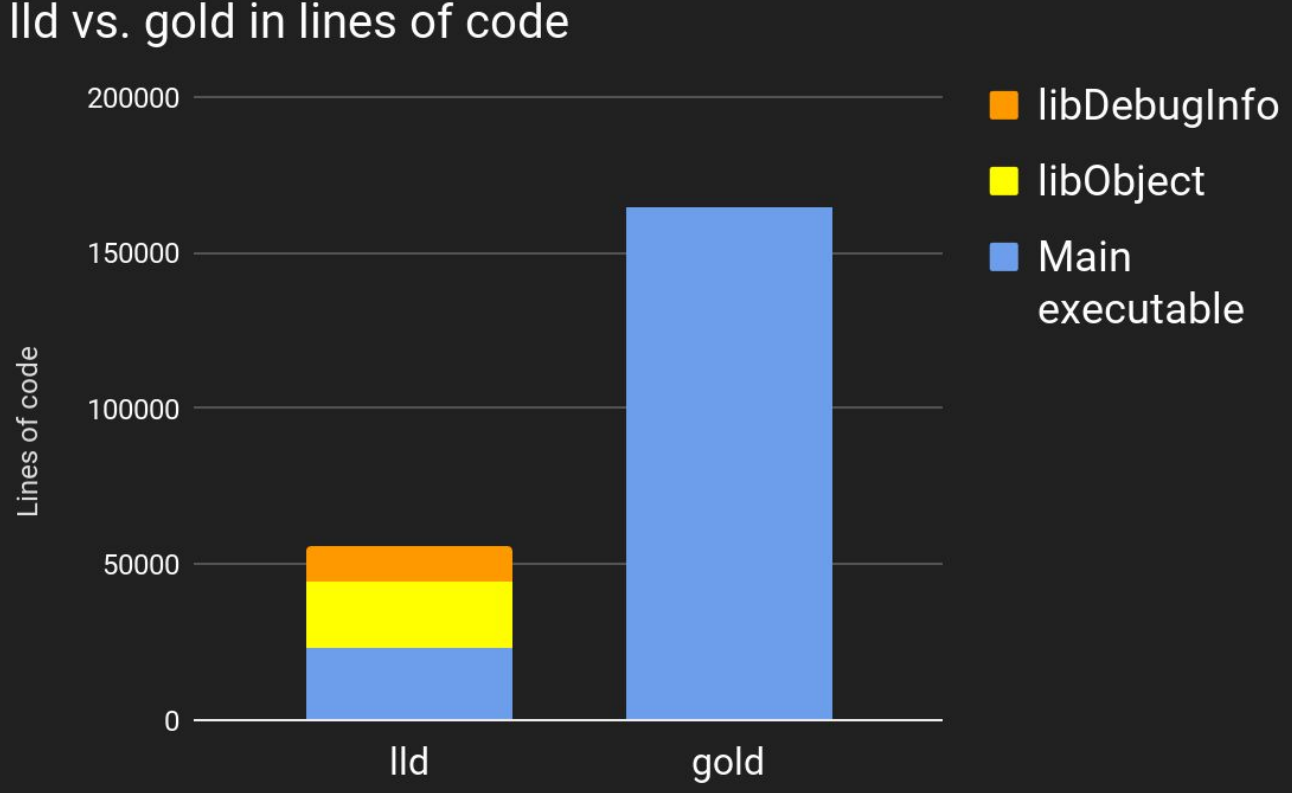
# Simplicity

# How much simpler?

One metric is lines of code, but it is not an apple-to-apple comparison

- gold supports more targets than lld (s390, nacl, etc.)
- lld depends on LLVM libObjects and libDebugInfo to read object files and debug info
- libObjects and libDebugInfo have more features than lld needs

However, you can get some idea by counting the number of lines.


lld vs. gold in lines of code
200000
150000
100000
50000
0
Lines of code
lld
gold
libDebugInfo
libObject
Main executable

# Better diagnostics

# Linker's error messages

Just like Clang significantly improved C++ diagnostics, we wanted to do the same thing for the linker. lld uses

- color in diagnostics
- more vertical space to print out structured error messages

# Examples of error messages

## lld

```
ld.lld: error: undefined symbol: lld::elf::demangle(llvm::StringRef)
>>> referenced by SymbolTable.cpp:669 (/src/lld/ELF/SymbolTable.cpp:669)
>>>               SymbolTable.cpp.o:(lld::elf::SymbolTable::getDemangledSyms())
in archive lib/liblldELF.a
```

## gold

```
/src/lld/ELF/SymbolTable.cpp:669: error: undefined reference to
'lld::elf::demangle(llvm::StringRef)'
/src/lld/ELF/Symbols.cpp:375: error: undefined reference to
'lld::elf::demangle(llvm::StringRef)'
```

# Examples of error messages

## lld

```
ld.lld: error: duplicate symbol: lld::elf::log(llvm::Twine const&)
>>> defined at Driver.cpp:67 (/ssd/llvm-project/lld/ELF/Driver.cpp:67)
>>>            Driver.cpp.o:(lld::elf::log(llvm::Twine const&)) in archive
lib/liblldELF.a
>>> defined at Error.cpp:73 (/ssd/llvm-project/lld/ELF/Error.cpp:73)
>>>            Error.cpp.o:(.text+0x120) in archive lib/liblldELF.a
```

## gold

```
ld.gold: error: lib/liblldELF.a(Error.cpp.o): multiple definition of
'lld::elf::log(llvm::Twine const&)'
ld.gold: lib/liblldELF.a(Driver.cpp.o): previous definition here
```

# Semantic differences

# Semantic differences between lld and GNU linkers

lld's symbol resolution semantics is different from traditional Unix linkers.

How traditional Unix linkers work:

- Maintains a set S of undefined symbols
- Visits files in the order they appeared in the command line, which adds or removes (resolves) symbols to/from S
- When visiting an archive, it pulls out object files to resolve as many undefined symbols as possible

# Semantic differences between lld and GNU linkers

File order is important in GNU linkers. Assume that `object.o` contains undefined symbols that `archive.a` can resolve.

- Works: `ld object.o archive.a`
- Does not work: `ld archive.a object.o`

# lld's semantics

In lld, archive files don't have to appear before object files.

- Works: `ld object.o archive.a`
- Also work: `ld archive.a object.o`

This is (in my opinion) intuitive and efficient but could result in a different symbol resolution result, if two or more archives provide the same symbols.

No need to worry too much; in FreeBSD, there were only a few programs that didn't work because of the difference, but you want to keep it in mind.

# Other features

# Link-Time Optimization

lld has built-in LTO (link-time optimization) support

- Unlike gold, you don't need a linker plugin
- To use LTO, all you need to do is to use clang as a compiler, and add `C{,XX}FLAGS=-flto` and `LDFLAGS=-fuse=lld`

# Cross-linking

- It always supports all targets
  - In fact, we do not provide a ./configure-time option to enable/disable targets. All lld executables can handle all targets
  - It should make it easier to use lld as part of a cross toolchain
- It is agnostic on the host operating system
  - There's no implicit default setting
  - It works fine because all options we need are passed explicitly by compilers
  - Again, it is cross toolchain-friendly

# Please use lld!

Instructions on how to check out, build and use lld are available at:

https://lld.llvm.org

Any questions?