Mit dem richtigen Linker Zeit sparen

Unter Linux stehen mehrere Linker zur Auswahl. Sie unterscheiden sich nicht nur in ihren Funktionen, sondern auch bei Speicherverbrauch und Performance. Ein Vergleichstest gibt Aufschluss.

Von Dr. Christoph Erhardt

Linker können Module zusammenfügen, Sektionen verschmelzen, Symbolbezüge auflösen und dynamische Programmbibliotheken zur Laufzeit ladbar machen [1]. Doch das ist nicht alles: Sie verfügen über erweiterte Features wie Symbolversionierung, Identical COMDAT Folding, inkrementelles Linken, Skripting und Linkzeitoptimierung.

Die Symbolversionierung ermöglicht es in einer Bibliothek, mehrere Versionen desselben Symbols bereitzustellen, zum Beispiel eine ältere und eine neue inkompatible Implementierung einer Funktion. Wer die Bibliothek nutzt, kann dem Linker mitteilen, welche Version des Symbols er einbinden soll. Das führt dazu, dass Legacy-Software unmodifiziert auch mit neueren Releases der Bibliothek lauffähig bleibt. Ein versioniertes Symbol hat als Suffix einen durch das @-Zeichen abgetrennten Versionsbezeichner, die Standardversion wird durch @@ markiert. Die glibc nutzt Symbolversionierung exzessiv und definiert beispielsweise `glob@GLIBC_2.2.5` und `glob@@GLIBC_2.27`, die sich in Bezug auf Symlinks unterschiedlich verhalten.

Identical COMDAT Folding (ICF) ist eine Optimierung, die mehrfach vorkommende nur lesbare Sektionen mit identischem Inhalt (Code oder Daten) dedupliziert. Das spart Speicherplatz, macht das Programm aber auch schlechter debugbar, weil deduplizierte Symbole nicht mehr eindeutig zuordenbar sind. Implementiert wird ICF in der Regel als heuristischer Löser für eine Art Graphisomorphieproblem.

iX-TRACT

- Moderne Linker verfügen über erweiterte Features wie inkrementelles Linken, Linkzeitoptimierung, Linkerskripte und Symbolversionierung.
- Der Standardlinker lässt sich in C/C++ leicht austauschen, in anderen Sprachen wie Rust und Go ist es komplizierter.
- Die Linkzeiten von GNU BFD, gold, LLD und mold unterscheiden sich um bis zu Faktor 10.

Mit inkrementellem Linken schneller zum Ziel

Inkrementelles Linken ist eine Technik, bei der der Linker die Ausgabedatei nicht komplett neu schreibt, sondern nur diejenigen Teile, die sich seit dem letzten Linken geändert haben. Das ist vor allem für schnelle Debug-Editier-Kompilier-Zyklen während der Entwicklung interessant, wenn das Linken zum Flaschenhals wird.

Ein mächtiges Mittel sind Linkerskripte, mit denen sich diverse Aspekte der vom Linker erzeugten Ausgabedatei programmatisch festlegen lassen. Ein Linkerskript kann Sektionen an fixen Adressen platzieren, Zugriffsrechte setzen, Markersymbole generieren und vieles mehr. Relevant ist das bei Betriebssystemkernen und eingebetteten Systemen. Eingebettete Anwendungen können so gezielt bestimmte Daten im schnellen, aber kleinen internen Speicher platzieren und andere im großen externen Speicher.

Ein weiteres Feature ist Linkzeitoptimierung (LTO), eine Technik, die den Bauvorgang von Software quasi auf den Kopf stellt: Anstelle fertig kompilierter Objektdateien mit relozierbarem Maschinencode erhält der Linker eine Menge von Modulen mit vorverarbeitetem Compilerzwischencode als Eingabe. Er füttert diese Eingabemodule gebündelt in ein vom Compiler bereitgestelltes Plug-in, das sie weiterverarbeitet, Maschinencode daraus generiert und ihn an den Linker zurückgibt. So hat das Middle-End des Compilers das gesamte Programm im Blick und kann aggressive modulübergreifende Optimierungen fahren, die beim klassischen modularen Ansatz nicht möglich sind. Erkauft wird das mit teils deutlich längeren Bauzeiten.

GNU BFD: Altlinker und Eier legende Wollmilchsau

Open-Source-Ökosysteme sind bekanntlich gegen Monokulturen eher allergisch und tendieren zu Artenvielfalt. Wenig überraschend gilt das auch für Linker. Im Linkerzoo heutiger Linux-Distributionen spiegeln sich drei Jahrzehnte Softwareevolution wider (Abbildung 1). Vier geläufige Linker sind GNU BFD, gold, LLD und mold.

Die etablierte Standardsuite in der Open-Source-Welt für das Ver- und Bearbeiten von Binärprogrammen sind die fast dreißig Jahre alten GNU Binutils, veröffentlicht unter der GPLv3. Enthalten sind unter anderem ein Assembler, ein Profiler, ein Binäreditor, diverse Werkzeuge zum Analysieren von Binärdateien sowie ein Linker. Die Binutils wurden ursprünglich von der Firma Cygnus Solutions entwickelt, die um die Jahrtausendwende in Red Hat eingegliedert wurde.

Sämtliche Komponenten basieren auf der eigens für diesen Zweck entwickelten BFD-Bibliothek (Binary File Descriptor), die eine portable Basis für das Verarbeiten beliebiger Binärformate für Dutzende von Befehlssatzarchitekturen bereitstellt. Genau das ist sowohl die größte Stärke als auch die größte Schwäche des GNU-Linkers: Einerseits beherrscht er nahezu jede erdenkliche Zielplattform und jeden Anwendungsfall, andererseits erkauft er das mit allenfalls mittelmäßiger Performanz.

Die Struktur des GNU-Linkers wurde ursprünglich primär für den Umgang mit den Binärformaten a.out und COFF (Common Object File Format) ausgelegt und ELF erst nachträglich berücksichtigt. Daher führt der Linker beim Verarbeiten von ELF-Dateien redundante Arbeitsschritte mehrfach aus. Zusätzlich stellt BFD viele teure Indirektionen bereit, und die Art, wie die Abstraktionen geschnitten sind, erzwingt stellenweise exzessives Hin- und Herschaufeln von Daten zwischen Datenstrukturen.

Nicht zuletzt wegen seines enormen Funktionsumfangs und seiner allgemeinen Bewährtheit ist der BFD-basierte GNU-Linker jedoch weiterhin der Defacto-Standard in der Open-Source-Welt. Wer heute eine aktuelle Linux-Distribution installiert, wird nach wie vor ld.bfd als voreingestellten Standardlinker vorfinden.

Im Jahr 2006 unternahm Ian Lance Taylor – seinerzeit der Hauptautor des BFD-Linkers und nunmehr in Diensten von Google – den Vorstoß, einen neuen Linker nur für ELF zu entwickeln. Er sollte aufrufkompatibel zum bestehenden, aber in seinen internen Abstraktionen, Datenstrukturen und Verarbeitungsschritten speziell auf die Erfordernisse von ELF ausgelegt sein und damit deutliche Vorteile bei Verarbeitungsgeschwindigkeit und Speicherverbrauch bieten (siehe ix.de/zcdq).

Das Ergebnis dieser Arbeiten trägt den Namen gold und bringt verglichen mit BFD einen ordentlichen Geschwindigkeitsvorteil. Einzelne Aufgaben kann gold in mehreren Threads parallelisieren, was außer bei LTO aber nur wenig bringt. Der Quellcode steht unter der GPLv3 und ist in die Codebasis der GNU Binutils eingezogen, wo er als vollwertige Alternative neben BFD friedlich koexistiert. Bei Profis und Linux-Distributionen genoss gold wegen seiner Leistung lange Zeit große Wertschätzung, allerdings häufen sich allmählich die Anzeichen dafür, dass gold nicht mehr lange weitergepflegt werden wird.

Inhalt der Artikelreihe

Teil 1: Technische Grundlagen: Aufgaben eines Linkers, Relokation, Symbole, Funktionen, Bibliotheken (siehe iX 12/2022)

Teil 2: Linker-Auswahl unter Linux, Performanz, erweiterte Features: inkrementelles Linken, Skripte, Identical COMDAT Folding, Linkzeitoptimierung

Das Rad neu erfunden: LLD hat Leistung im Blick

Mit dem LLVM-Projekt existiert eine zweite große quelloffene Suite aus Compilern und Werkzeugen in Konkurrenz zu GNU. LLVM versucht sich vor allem durch eine moderne Architektur und eine freizügige Lizenz (NCSA/ASL 2.0) abzuheben; gerade der zweite Punkt macht es für Firmen wie Apple, Google, Intel oder AMD besonders interessant. Nachdem man mit Clang bereits einen vollwertigen C/C++-Compiler samt integriertem Assembler ins Leben gerufen hatte, verblieb als eine der letzten signifikanten externen Abhängigkeiten der Linker. Aus diesem Grund wurde 2015 mit den Arbeiten an LLD begonnen, einem format- und plattformübergreifenden Linker. Er bietet nicht nur für GNU/Linux, Windows und macOS ein zum jeweils nativen Linker kompatibles Frontend, sondern lässt sich – wie die meisten Komponenten von LLVM – auch als Bibliothek einbetten.

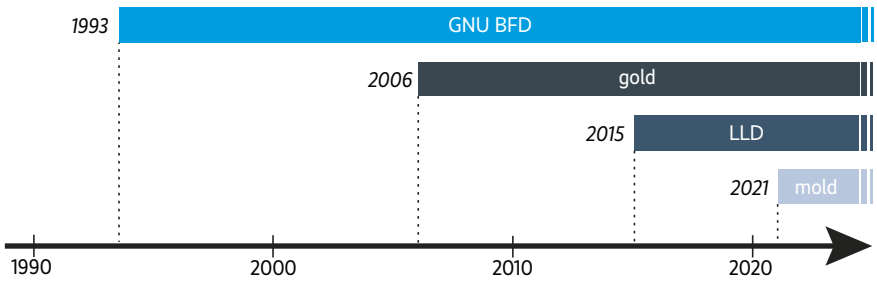
Ähnlich wie gold vermeidet LLD wo immer möglich kostspielige interne Abstraktionen; so existiert für das Verarbeiten jedes Binärformats (ELF, COFF und Mach-O) ein separater Codepfad. Das Team um Googles Softwareentwickler Rui Ueyama machte jedoch an diesem Punkt nicht halt, sondern beschloss, den internen Aufbau eines Linkers grundlegend zu überdenken und kompromisslos auf Performanz zu trimmen. Das schlägt sich unter anderem in den folgenden Aspekten nieder:

- Datenstrukturen sind darauf ausgelegt, ein und dieselbe Aufgabe nicht mehrfach auszuführen. So reduziert eine zweigeteilte Symboltabelle das teure Hashen von Strings auf ein einziges Mal pro Symbol.
- Operationen, die sich gut parallelisieren lassen, wie das Zusammenführen von String-Konstanten und das Kopieren aller Sektionen inklusive Relokationen, werden auf mehrere Threads aufgeteilt.
- In der Handhabung von Bibliotheken mit zyklischen Abhängigkeiten schneidet LLD alte Zöpfe ab und löst die Symbole auf, ohne dass die beteiligten Bibliotheken mehrfach anzugeben sind. Das geänderte Verhalten ist nicht nur schneller, sondern auch intuitiver.

All das sorgt im Vergleich zu gold für einen enormen Geschwindigkeitssprung. Unterm Strich ist LLD ein stabiler, ausgereifter und extrem performanter Linker und unbedingt empfehlenswert.

Ist mold die Zukunft?

Obwohl LLD die Messlatte bereits hoch legt, ruht sein Hauptautor sich nicht etwa auf seinen Lorbeeren aus, sondern hat sich eine neue Herausforderung gesucht: Eine leistungsfähige Maschine vorausgesetzt, soll es möglich sein, Chromium innerhalb einer Sekunde zu linken. Um das zu erreichen, hat sich Rui Ueyama zurück ans Reißbrett begeben und im Dezember



Stagnation? – Im Gegenteil: Das Innovationstempo von Linkern nimmt zu (Abb. 1).

2021 mold in Version 1.0.0 vorgestellt. mold legt seinen Fokus auf das Bauen von reinen Anwenderprogrammen und beherrscht nur die dafür relevanten Basisfeatures von Linkerskripten; eventuelle erweiterte Anwendungsfälle sollen sich stattdessen nachgelagert mit Werkzeugen wie objcopy erledigen lassen.

mold greift viele der Konzepte von LLD wieder auf, führt sie aber einen Schritt weiter. Anstatt nur die beiden bedeutendsten Arbeitsschritte zu parallelisieren, führt mold so viele Algorithmen wie möglich – vom Scannen der Relokationstabelle über die Garbage Collection unreferenzierter Sektionen bis zum Bilden des Build-ID-Hashes – mit dem Framework oneAPI Threading Building Blocks in parallelen Threads aus.

Zwecks weiterer Optimierung legt er zudem das Layout der Ausgabedatei so früh wie möglich fest, damit er unmittelbar mit dem Kopieren der Eingabedaten beginnen kann. So nutzt mold die Latenz I/O-lastiger Operationen, um verschränkt bereits mit rechenintensiven Aufgaben loslegen zu können – teils sogar spekulativ. Auch an den Algorithmen wurde nochmals geschraubt: So ist der ICF-Algorithmus fünfmal schneller.

Da der Kernel beim `exit()` eines Prozesses dessen gemappte Dateiinhalte wegräumen muss (was bei großen Programmen mehrere Hundert Millisekunden kosten kann), nutzt mold ein trickreiches Doppel-`fork()`-Schema: Die gesamte Arbeit ist in einen eigenen Kindprozess ausgelagert, der seinen Elternprozess via Pipe benachrichtigt, sobald er fertig ist. Der Elternprozess beendet sich daraufhin sofort und erlaubt dem Build-System weiterzuarbeiten, während der Kindprozess im Hintergrund noch in Ruhe aufräumen kann.

mold ist unter der AGPLv3 veröffentlicht. In seinem Funktionsumfang liegt er annähernd gleichauf mit seinen Konkurrenten und wird aktiv in hohem Tempo weiterentwickelt. Schon jetzt kann er für i686, x86_64, ARM, AArch64, ppcle64, RISC64, m68k und SPARC64 große Teile des Linux-Paketzoos durchbauen; Linkzeitoptimierung ist ebenfalls implementiert. Die Fähigkeit, macOS-Programme zu erzeugen, liegt in der aktuellen Version 1.7.1 im Alphastadium vor, Windows steht auf der Agenda. Nachdem alle gängigen Linux-Distributionen mold bereits in ihr Paket-Repository aufgenommen haben, ist es denkbar, dass mold die Zukunft gehören könnte. Allerdings gibt es derzeit ein paar Fragezeichen bezüglich der finanziellen Tragbarkeit dieses Ein-Mann-Vollzeitprojekts. Um von mehr als Spenden und Ersparnissen leben zu können, hat sein Entwickler unlängst eine kommerzielle Variante namens sold mit identischem Funktionsumfang ins Leben gerufen. Sie soll Firmenkunden an Land ziehen, die mit der AGPL fremdeln.

Linker in GCC und Clang

-fuse-ld=...	GCC	Clang
bfd	≥ 5.1	≥ 3.5
gold	≥ 5.1	≥ 3.5
lld	≥ 9.1	≥ 3.5
mold	≥ 12.1	≥ 3.5

Den Linker im System festlegen

In einem Unix-System ist der Standardlinker über den Pfad /usr/bin/ld zu finden; unter Linux verbirgt sich dahinter üblicherweise ein Symlink auf ld.bfd. Um systemweit einen anderen Linker zum Standard zu machen, muss man lediglich diesen Symlink umbiegen. Voreinstellungen mit globaler Tragweite sollte man allerdings nicht leichtfertig verändern.

Empfehlenswerter ist es deshalb, die Auswahl des Linkers pro Projekt über dessen Build-System zu treffen. Dafür gibt man dem Compiler über die LDFLAGS das Argument `-fuse-ld=...` mit. Die Tabelle zeigt für GCC und Clang, seit welchem Release sie offiziell welchen Linker ansteuern können.

Möchte man davon abweichend einen neuen Linker mit einer älteren Compilerversion nutzen, lässt sich das über folgenden Workaround bewerkstelligen: Man legt ein leeres Verzeichnis an (etwa tools), erstellt darin einen Symlink namens ld, der auf das gewünschte Linker-Binary verweist, und gibt in den LDFLAGS den Pfad zu diesem Verzeichnis über die Option `-B` an (im Beispiel `-Btools`).

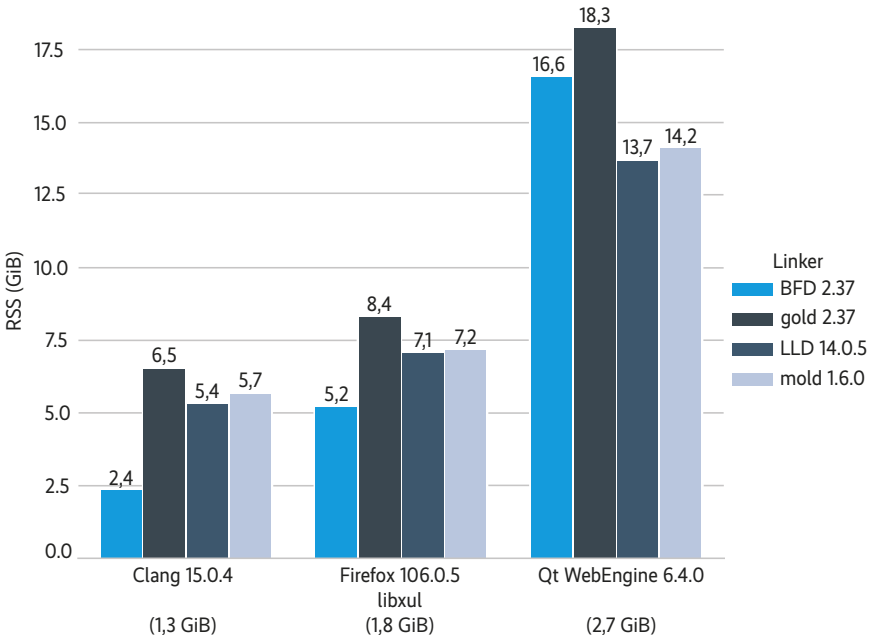
Bei vielen neueren Programmiersprachen ist der Aufruf des Linkers im Build-System weniger direkt exponiert als bei C/C++. Deshalb ist es dort aufwendiger, in die Wahl des Linkers einzugreifen. Rust delegiert das Linken standardmäßig an cc, das Frontend des systemweiten C-Compilers. Analog zur oben beschriebenen Vorgehensweise für C/C++ kann man eine eigene Linkerwahl durchschleusen, indem man rustc mit der Option `-C link-arg=-fuse-ld=...` füttert. Aber Vorsicht: Da cc üblicherweise ein Symlink auf gcc ist, gelten die oben aufgeführten Mindestversionen.

Im Zweifelsfall empfiehlt es sich, rustc mittels `-C linker=clang` zu instruieren, das Clang-Frontend zu verwenden. Möchte man beispielsweise mold benutzen, erstellt man eine Datei .cargo/config.toml mit folgendem Inhalt:

```
[build]
rustflags = ["-C", "linker=clang",↵
      "-C", "link-arg=-fuse-ld=mold"]
```

Der Swift-Compiler basiert auf LLVM und bietet die Option `-use-ld=...`. Um diese Option an `swiftc` durchzureichen, muss man den Build-Befehl folgendermaßen aufrufen:

```
swift build -Xswiftc -use-ld=mold
```



Wer große Programme mit Debugsymbolen bauen will, braucht viel Arbeitsspeicher – wie viel, unterscheidet sich von Linker zu Linker (Abb. 2).

Im Fall von Go ist die Situation etwas kompliziert, denn es existieren gleich zwei Werkzeugketten nebeneinander: Googles Referenzimplementierung Golang definiert ein komplett eigenes Binärobjektformat und liefert einen eigenen Assembler und Linker dafür mit. Die aus der C-Welt gewohnten Linker lassen sich nicht nutzen. Anders sieht es bei gccgo aus, einer Alternativimplementierung, die GCC als Backend nutzt. Hier kann man den gewünschten Linker einfach über die Befehlszeile mitgeben, was allerdings wiederum eine hinreichend aktuelle GCC-Version erfordert:

```
go build -gccgoflags=-fuse-ld=mold
```

Für Haskell erlaubt es der Standardcompiler GHC, per `-optl` Optionen an den Linker durchzuschleifen. Sofern man Cabal als Build-System nutzt, reicht es aus, zwei Zeilen in die .cabal-Datei einzutragen:

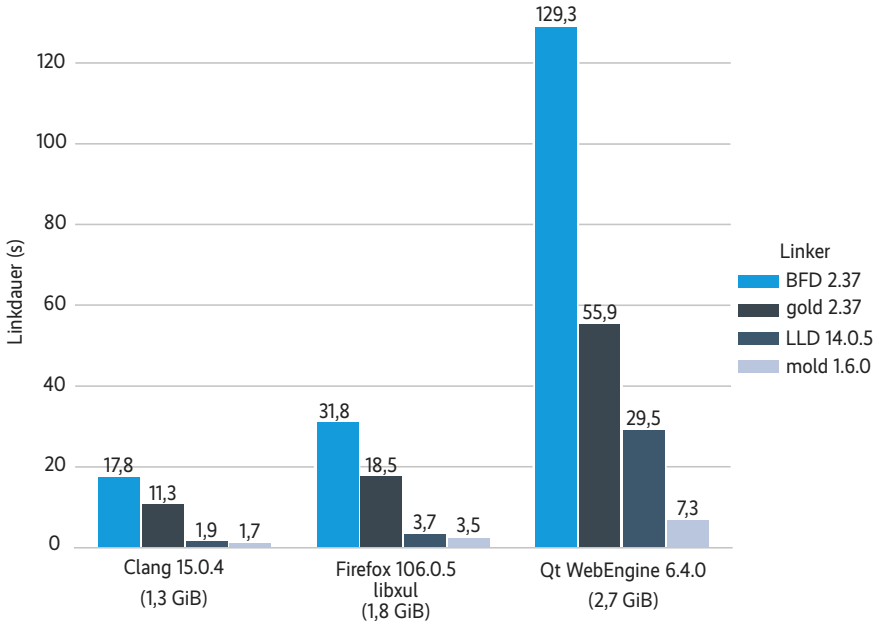
```
executable <name>
    ghc-options: -optl-fuse-ld=mold
     ld-options: -fuse-ld=mold
```

Um die Wette: schneller, besser, effizienter

Um herauszufinden, worin sich vier Generationen von Linkern in der Praxis unterscheiden, haben wir sie einem Vergleichstest hinsichtlich Speicherbedarf und Performanz unterzogen. Als Hardware kommt ein Entwicklerlaptop mit einem Intel Core i9-9880H (8 Kerne, 16 Fäden), 32 Gibibyte (GiB) DDR4-2666 und einer per PCIe 3.0 x4 angebundenen NVMe-SSD zum Einsatz. Die Softwareumgebung ist ein Podman-Container mit Fedora Linux 36. Der Testparcours besteht aus drei bekannten Open-Source-Großprojekten: dem C/C++-Compiler Clang in Version 15.0.4, dem Webbrowser Mozilla Firefox 106.0.5 und der Version 6.4.0 der auf Chromium basierenden Qt WebEngine.

Alle drei Projekte werden initial einmal vollständig mit Debuginformationen gebaut; anschließend misst man, wie viel Zeit es kostet, die Ausgabedatei zu erzeugen (clang-15, libxul.so, libQt6WebEngineCore.so.6.4.0). Damit die Zeitmessungen nur die reine Laufzeit des Linkers erfassen, ist Linkzeitoptimierung deaktiviert. Die Größe der Ausgabedateien liegt zwischen 1,3 und 2,7 GiB und ist in den nachfolgenden Diagrammen unterhalb des Programmnamens notiert.

Schon beim Betrachten des Speicherbedarfs (Resident Set Size, RSS) in Abbildung 2 fällt auf, dass es zwischen den Linkern signifikante Unterschiede gibt.



Die Wahl des Linkers kann den Unterschied ausmachen zwischen regelmäßiger Zwangskaffeepause und konzentriertem Durcharbeiten (Abb. 3).

Insbesondere gold sticht im Vergleich zum Rest des Teilnehmerfelds negativ heraus, während ausgerechnet BFD sich in zwei der drei Tests erstaunlich genügsam gibt. Jedenfalls ist ausreichend RAM für Entwicklerrechner eine lohnenswerte Investition.

Interessant wird es, wenn man die Zeiten fürs Linken miteinander vergleicht. Jede Zeitmessung findet zehnmal statt; die gemittelten Resultate sind in Abbildung 3 aufgelistet. Erwartungsgemäß ist BFD mit Abstand der langsamste Linker im Feld; gold schlägt ihn um den Faktor 1,5 bis 2,5. LLD legt nochmals eine Schippe drauf und steigert das gegenüber gold auf das Zwei- bis Sechseinhalbfache.

Doch auch damit ist das Ende der Fahnenstange noch nicht erreicht: Das Linken der Qt WebEngine ist mit mold nahezu viermal so schnell wie mit LLD. Unterm Strich arbeiten sowohl mold als auch LLD um ein Vielfaches schneller als der GNU-Standardlinker. Besonders beim Bauen der Qt WebEngine ergibt eine Gesamtdauer von weniger als acht Sekunden gegenüber mehr als zwei Minuten eine enorme Zeitersparnis.

Kritische Geister mögen anmerken, dass die Linkzeit im Vergleich zur gesamten Kompilierzeit in jedem Fall verschwindend gering ist, und fragen, ob sich Optimierungen an dieser Stelle lohnen. Dieser Einwand ist angebracht, solange man sich auf den Anwendungsfall „Bauen von null an“ beschränkt. Im Entwickleralltag findet man sich aber üblicherweise in einem Debug-Editier-Kompilier-Zyklus wieder. Häufig ändert man nur eine einzige Quelldatei, bevor man das Build-System anstößt. In einer solchen Situation fällt die Leistung des Linkers ins Gewicht und die erzielbare Zeitersparnis kann im Bereich einer Größenordnung liegen.

Fazit

Moderne Linker wie LLD und mold können den Linkprozess gegenüber dem GNU-Standardlinker erheblich beschleunigen. Gerade im Kontext eines größeren Projekts lohnt sich deshalb der Blick über den Tellerrand, denn wer den richtigen Linker einsetzt, kann im Softwareentwicklungsalltag viel Totzeit vermeiden. (nb@ix.de)

Quellen

[1] Christoph Erhardt; Ohne Linker kein Programm; iX 12/2022, S. 112

[2] Benchmarks und weiterführende Informationen zu den Linkern: ix.de/zcdq

DR. CHRISTOPH ERHARDT

entwickelt als Senior Software Engineer bei Method Park by UL Softwareplattformen für Medizingeräte. Er steuert Patches zum mold-Linker bei und betreut das mold-Paket als Maintainer für Fedora Linux.