WIKIPEDIA

LLVM

LLVM (früher **Low Level Virtual Machine**) ist eine modulare Compiler-Unterbau-Architektur mit einem virtuellen Befehlssatz, einer virtuellen Maschine, die einen Hauptprozessor virtualisiert, und einem übergreifend optimierenden Übersetzungskonzept.[6] Kennzeichnend ist unter anderem, dass sämtliche Zeitphasen eines Programms (Laufzeit, Compilezeit, Linkzeit) inklusive der Leerlauf-Phase[7] zur Optimierung herangezogen werden können. Der Zweck ist, einfach eigene Frontends für verschiedene Sprachen zu entwickeln, die die LLVM-Zwischensprache benutzen. Um die Erzeugung des Maschinen- bzw. VM-Codes kümmert sich die LLVM-Bibliothek, die vom Frontend aus aufgerufen wird.

Die Entwicklung von LLVM begann im Jahr 2000 unter der Leitung von Chris Lattner und Vikram Adve an der Universität von Illinois. Das Projekt wurde ursprünglich als Forschungsarbeit zur Untersuchung dynamischer Kompilierung und Optimierungen entwickelt. Heute beheimatet es eine Vielzahl an Unterprojekten und Erweiterungen aus der aktuellen Compilerforschung und -entwicklung.[1][8][9]

The LLVM Compiler Infrastructure

Maintainer	Chris Lattner[1]
Entwickler	The LLVM Team[2]
Aktuelle Version	4.0.1 (4. Juli 2017[3][4])
Betriebssystem	macOS, FreeBSD, Linux, Windows[5]
Programmiersprache	C++
Kategorie	Compiler
Lizenz	LLVM Release License (BSD- und MIT-ähnlich - Freie Software)
deutschsprachig	nein

www.llvm.org (http://www.llvm.org/)

LLVM ist als freie Software unter der University of Illinois/NCSA Open Source License verfügbar, die der 3-Klausel-BSD-Lizenz und der MIT-Lizenz ähnelt.

Der Name „Low Level Virtual Machine“ verursacht zuweilen Verwirrung, da LLVM *auch* für Virtualisierung genutzt werden kann. Mit der Zeit wurde LLVM ein Rahmenprojekt, in dem verschiedene Compiler- und Low-Level-Techniken enthalten sind. Inzwischen ist *LLVM* die Marke für das eigentliche Projekt, die LLVM-Zwischensprache (LLVM-IR), den LLVM-Debugger (LLDB), die LLVM-Standard-C++-Bibliothek (libc++) etc.

Inhaltsverzeichnis

Arbeitsweise

Herkömmliche Compilersysteme führen Optimierungsvorgänge meist beim Kompilieren durch und verbinden die kompilierten Module dann miteinander. Dieser zweite Vorgang wird Binden oder auch Linken genannt und bietet ebenfalls Optimierungsmöglichkeiten, die bisher wenig genutzt wurden, da der Linker nur die einzelnen Module sieht und nicht das gesamte Programm. Hier setzt LLVM an, indem es einen nach Vorbild der RISC-Befehlssätze gestalteten virtuellen Bytecode erstellt, der während des Linkens noch einmal optimiert werden kann.[6][10]

Bereits der Name LLVM verrät, dass ein Teil der Architektur auf einer virtuellen Maschine basiert, die einen Prozessor virtualisiert. Ein Prozessor kann dabei nicht nur ein Hauptprozessor (CPU) sein, sondern auch ein Grafikprozessor (GPU). Die virtuelle Maschine ist in der Lage, die intern generierte Sprache (sog. intermediate language) des Compilers (LLVM Assembly Language)[11] während der Ausführung für den Prozessor des aktuellen Systems zu übersetzen. Kennzeichnend ist hierbei, dass sie hocheffizient ist, was die Übersetzung auch Just-in-Time (also auf Anforderung, bei Bedarf) ermöglicht, und diese mit einer Größe von nur 20 kB extrem kompakt ist, wodurch die Ausführung auch auf einfachen Prozessoren, älteren Grafikprozessoren (GPUs) oder Embedded-CPUs, und insbesondere sogar direkt im Cache möglich ist. Über ein flexibles Backend-System ist es möglich, eine fast beliebige Vielzahl unterschiedlichster Prozessor-Architekturen zu unterstützen.[12]

Die LLVM-Bibliotheken können von Compilerprojekten, denen das Schreiben eines eigenen Codegenerators zu aufwendig wäre, eingesetzt werden, um auf einfache Weise Maschinencode (Bitcode, Microcode) zu erzeugen. Zum Beispiel muss Clang, welches Bestandteil von LLVM ist, nur den C- bzw. C++-Code parsen und in die LLVM-Zwischensprache (LLVM Intermediate Representation, LLVM IR) übersetzen. Das Erzeugen von effizientem Maschinencode kann dem LLVM-Backend überlassen werden. Bei diesem Beispiel kommt die virtuelle Maschine nicht zum Einsatz, da LLVM hier nur als Compiler-Backend für die jeweilige Architektur (x86, PowerPC, IA64, ...) agiert.[13]

Aufbau

Der LLVM-Compiler verwendet ab Version 2.9 primär Clang als Frontend. Dabei eignet sich LLVM dazu, Programme, die in frei wählbaren Programmiersprachen geschrieben wurden, zu kompilieren. Derzeit kann Programmcode unter anderem in den Programmiersprachen C, C++, Objective-C, Swift, Java, Julia, D, Ada, Fortran, Haskell, Dylan, Gambas, Python, Ruby, Rust, ActionScript, Vala, Genie und GLSL kompiliert werden.

Mit LLVM lassen sich virtuelle Maschinen für Sprachen wie Java, plattformspezifische Codegeneratoren und von Sprache und Plattform unabhängige Optimierer erstellen. Die LLVM-Zwischenschicht (IR) liegt zwischen sprachspezifischen Modulen und den jeweiligen Codegeneratoren und kann als eine Art plattformunabhängige Assemblersprache betrachtet werden. LLVM unterstützt weiterhin dynamische, interprozedurale Optimierung sowie statische Ahead-of-time- und Just-in-time-Compilierung.[12] Ein Beispiel für LLVM-IR:

```
; String-Konstante als globale Konstante deklarieren
@.str = private unnamed_addr constant [13 x i8] c"hello world\0A\00"

; Externe Deklaration der `puts()`-Funktion (enthalten in libc)
declare i32 @puts(i8* nocapture) nounwind

; Definition der `main()`-Funktion
define i32 @main() {  ; i32()*
  ; Konvertiere [13 x i8]* zu i8 *... (Pointer auf i8-Elemente)
  %cast210 = getelementptr [13 x i8]* @.str, i64 0, i64 0

  ; Rufe `puts()`-Funktion auf, um Text auszugeben
  call i32 @puts(i8* %cast210)
  ret i32 0
}

; Metadaten
!1 = metadata !{i32 42}
!foo = !{!1, null}
```

Clang

→ *Hauptartikel: Clang*

Clang ist ein für LLVM entwickeltes Frontend, das für C-ähnliche Sprachen optimiert ist. Es ermöglicht gegenüber dem GCC-Oberbau vor allem schnellere Übersetzungsläufe mit geringerem Speicherverbrauch und als Ergebnis oft kleinere ausführbare Programme. Zudem verfügt es über umfangreichere und genauere statische Analysemethoden, die dem Entwickler z. B. die Fehlersuche erleichtern. Die Unterstützung der Programmiersprache C++ gilt ab Version 2.7 als stabil.[14]

Seit September 2009 gilt Clang offiziell als stabil und produktiv verwendbar. Ab LLVM Version 2.6 befindet es sich als fester Bestandteil im LLVM-Compiler-Paket.[15] Clang lässt sich aber auch ohne LLVM als rein statisches Codeanalyse- und Debug-Werkzeug, zum Beispiel beim Einsatz mit anderen Compilern, verwenden.[16] Clang ist zur statischen Code-Analyse in die Entwicklungsumgebung *Xcode* von Apple für die Programmiersprachen C, Objective-C und C++ integriert.

LLDB

LLDB ist ein auf Techniken des LLVM-Projektes aufbauender und für C-basierte Sprachen optimierter, modularer und hochdynamischer Debugger. Er soll besonders speichereffizient und zugleich extrem leistungsfähig und schnell sein. Er verfügt über eine Plug-In-Schnittstelle zum Beispiel für die Unterstützung anderer Programmiersprachen. Zudem lassen sich Aufgaben mit Hilfe von Python automatisieren. Eine Besonderheit ist unter anderem die

spezielle Unterstützung für das Debuggen von Multithreading-Code.[17][18]

DragonEgg

Bei DragonEgg handelt es sich um ein LLVM-Plugin für die GNU Compiler Collection (ab Version 4.5).[19] Dieses ermöglicht es, LLVM optional als Compiler-Backend einer ungepatchten GCC-Installation zu nutzen. DragonEgg wurde zu Beginn der Entwicklung nur als „the gcc plugin“ bezeichnet. DragonEgg löst die bisher häufig verwendete LLVM-GCC-Mischkonfiguration ab.

vmkit

Hierbei handelt es sich um einen modifizierten Zweig (Fork) der LLVM-VM, welche die direkte Ausführung von Java- und CLI-Bytecode (.NET/Mono) ermöglicht. Der Compiler beziehungsweise Linker kann das hochkompakte vmkit (ca. 20 kB) vor den Java- oder CLI-Bytecode packen und ausführen, wodurch die Ausführung auf beliebigen Prozessorarchitekturen und auf Systemen ohne vorherige Installation von Java oder .NET möglich ist. Das Projekt wird allerdings derzeit nicht mehr offiziell unterstützt.[20]

KLEE

Mit KLEE kann man Programme unüberwacht und automatisch auf Programmfehler untersuchen lassen. Dabei wird das Programm Schritt für Schritt ausgeführt. Statt konkreter Werte werden Eingabe und Zwischenresultate symbolisch verwendet und jeweils gespeichert, welche Werte diese haben könnten. Dabei wird bei „gefährlichen Operationen“ (englisch: "dangerous operations", zum Beispiel Divisionen oder Speicherzugriffe per Zeiger) geprüft, ob sie einen Fehler erzeugen könnten, zum Beispiel eine Division durch null oder einen Zugriff auf nicht reservierten Speicher. KLEE gibt dann aus, bei welcher Eingabe das Programm den Fehler erzeugt und welcher Pfad durch den Quellcode dabei genommen wird.[21]

Unterstützte Architekturen

Zurzeit wird bereits eine große Anzahl von Prozessorarchitekturen unterstützt. Jedoch gibt es noch einige Einschränkungen, die besonders das Frontend (llvm-gcc) betreffen. Es ist noch nicht für alle Plattformen lauffähig. Dies kann jedoch umgangen werden, indem die LLVM als Cross-Compiler verwendet wird. Hier sollten aber eventuelle Abhängigkeiten, zum Beispiel die Programmbibliothek, berücksichtigt werden.[12]

- x86, AMD64
- PowerPC, PowerPC 64Bit
- ARM, Thumb
- SPARC
- Alpha
- CellSPU
- PIC16 MIPS
- MSP430
- System z
- XMOS Xcore

Geschichte

Das Projekt startete 2000 an der *University of Illinois at Urbana–Champaign* als Studienprojekt von Vikram Adve und Chris Lattner. Als die Firma Apple darauf aufmerksam wurde, stellte sie 2005 ein festes Entwicklerteam für die Weiterentwicklung von LLVM zusammen und stellte Chris Lattner als dessen Projektleiter an.[22] LLVM ist seit Juli 2008 Standardcompiler in Apples Entwicklungsumgebung Xcode.[23] Ab Version 2.6 vom Oktober 2009 ist das Compiler-Frontend *Clang* integraler Bestandteil von LLVM.[24][25]

Aktuelle Entwicklung

Viele weitere Komponenten befinden sich derzeit in intensiver Entwicklung, unter anderem Frontends für Java-Bytecode, OpenCL, Microsofts CIL, Python, Lua, PHP, Ruby, Mono[26] und Adobe ActionScript.[9][27] Der LLVM-JIT-Compiler kann ungenutzte statische Zweige des Programms zur Laufzeit erkennen und anschließend entfernen. Dies optimiert Programme mit einem hohen Grad an Verzweigung. Aus diesem Grund nutzt Apple seit macOS 10.5[28] LLVM im OpenGL-Stack, um einerseits sich selten ändernde Verzweigungspfade zu verkürzen und andererseits Vertex-Shader optimieren zu lassen.

Weblinks

- Offizielle Webpräsenz (http://llvm.org/)
  - Chris Lattner: *The LLVM Compiler System*. (http://llvm.org/pubs/2007-03-12-BossaLLVMIntro.pdf) (PDF; 416 kB) März 2007
  - Chris Lattner: *LLVM and Clang: Next Generation Compiler Technology*. (http://llvm.org/pubs/2008-05-17-BSDCan-LLVMIntro.pdf) (PDF; 5,8 MB) Mai 2008
- Chris Lattner: *The Design of LLVM*. (http://www.drdobbs.com/architecture-and-design/240001128) Mai 2012
- *LLVM – Die ‚Low Level Virtual Machine' Compiler-Infrastruktur*. (https://cre.fm/cre114-llvm) CRE Podcast

Einzelnachweise

1. The LLVM Compiler Infrastructure Project (http://www.llvm.org/) (englisch) – offizielle Webseite
2. llvm.org (http://www.llvm.org/docs/FAQ.html)
3. LLVM 4.0.1 Release (http://lists.llvm.org/pipermail/llvm-announce/2017-July/000074.html)
4. LLVM Download Page (http://releases.llvm.org/download.html)
5. llvm.org (http://www.llvm.org/docs/GettingStartedVS.html)
6. llvm.org (http://www.llvm.org/pubs/2002-12-LattnerMSThesis.html)
7. llvm.org (http://www.llvm.org/docs/LinkTimeOptimization.html)
8. llvm.org (http://www.llvm.org/ProjectsWithLLVM/)
9. *The LLVM Users*. (http://www.llvm.org/Users.html) llvm.org
10. Hans-Joachim Baader: *LLVM 1.5 freigegeben.* (http://www.pro-linux.de/news/1/8162/llvm-15-freigegeben.html) In: *Pro-Linux.* 20. Mai 2005, abgerufen am 5. Dezember 2010 (englisch).
11. llvm.org (http://llvm.org/docs/LangRef.html#introduction)
12. *LLVM Features*. (http://llvm.org/Features.html) llvm.org
13. LLVM. (http://www.aosabook.org/en/llvm.html) aosabook.org

14. *clang: a C language family frontend for LLVM*. (http://clang.llvm.org/) LLVM; Stand: 18. Oktober 2010
15. *LLVM 2.6 Release!*. (http://lists.cs.uiuc.edu/pipermail/llvm-announce/2009-October/000033.html) lists.cs.uiuc.edu, 23. Oktober 2009 (englisch)
16. *Expressive Diagnostics*. (http://clang.llvm.org/diagnostics.html) LLVM; Stand: 27. November 2009 (englisch)
17. *The LLDB Debugger Goals*. (http://lldb.llvm.org/goals.html) lldb.llvm.org
18. Chris Lattner: *New "lldb" Debugger*. (http://blog.llvm.org/2010/06/new-lldb-debugger.html) blog.llvm.org
19. dragonegg.llvm.org (http://dragonegg.llvm.org/)
20. vmkit.llvm.org (http://vmkit.llvm.org/)
21. Cristian Cadar, Daniel Dunbar, Dawson Engler: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs (http://llvm.org/pubs/2008-12-OSDI-KLEE.html). Universität Stanford, 2008.
22. Adam Treat: *mkspecs and patches for LLVM compile of Qt4.* (http://lists.trolltech.com/qt4-preview-feedback/2005-02/msg00691.html) In: *Qt4-preview-feedback-Mailingliste.* 19. Februar 2005, abgerufen am 5. Dezember 2010 (englisch).
23. developer.apple.com (https://developer.apple.com/library/mac/#documentation/CompilerTools/Conceptual/LLVMCompilerOverview/_index.html)
24. Chris Lattner: *LLVM 2.6 Release!* (http://lists.cs.uiuc.edu/pipermail/llvm-announce/2009-October/000033.html) In: *llvm-announce mailing list.* 23. Oktober 2009, abgerufen am 5. Dezember 2010 (englisch).
25. Hans-Joachim Baader: *LLVM 2.6 freigegeben.* (http://www.pro-linux.de/news/1/14864/llvm-26-freigegeben.html) In: *Pro-Linux.* 26. Oktober 2009, abgerufen am 5. Dezember 2010.
26. mono-project.com (http://www.mono-project.com/Mono_LLVM)
27. *Developing for the Apple iPhone using Flash*. (http://www.adobe.com/devnet/logged_in/abansod_iphone.html) adobe.com
28. *[LLVMdev] A cool use of LLVM at Apple: the OpenGL stack*. (http://lists.cs.uiuc.edu/pipermail/llvmdev/2006-August/006492.html)
