Low Level Virtual Machine (Weitergeleitet von Clang)The LLVM Compiler Infrastructure Maintainer Chris Lattner[1] Entwickler The LLVM Team[2] Aktuelle Version 2.9 (6. April 2011) Betriebssystem Mac OS X, FreeBSD, Linux, Windows[3] Kategorie Compiler Lizenz BSD- und MIT-ähnlich (Freie Software) Deutschsprachig nein www.llvm.org Die Low Level Virtual Machine (LLVM) ist eine modulare Compiler-Unterbau-Architektur mit einem virtuellen Befehlssatz, einer virtuellen Maschine, die einen Hauptprozessor virtualisiert, und einem übergreifend optimierenden Übersetzungskonzept.[4] Kennzeichnend ist unter anderem, dass sämtliche Zeitphasen eines Programms (Laufzeit, Compilezeit, Linkzeit) inklusive der Leerlauf-Phase[5] zur Optimierung herangezogen werden können. 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][6][7] 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.Inhaltsverzeichnis 1 Arbeitsweise 2 Aufbau 2.1 Clang 2.1.1 Versionen 2.2 LLDB 2.3 DragonEgg 2.4 vmkit 2.5 KLEE 2.6 Unterstützte Architekturen 3 Geschichte 4 Aktuelle Entwicklung 5 Weblinks 6 Einzelnachweise Bearbeiten Arbeitsweise Herkömmliche Compilersysteme führen Optimierungsvorgänge meist beim Kompilieren durch und verbinden die kompilierten Module dann miteinander. Dieser zweite Vorgang wird Linken oder auch Binden 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.[4][8] 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 ("intermediate language") des Compilers (LLVM Assembly Language)[9] während der Ausführung für den Prozessor des aktuellen Systems zu übersetzen. Kennzeichnend ist hierbei, dass sie hocheffizient ist, was die Übersetzung auch in Echtzeit 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.[10] Bearbeiten Aufbau Der LLVM-Compiler verwendet derzeit primär die GNU Compiler Collection (GCC) oder Clang als Frontend. Dabei eignet sich LLVM dazu, Programme, die in frei wählbaren imperativen Programmiersprachen geschrieben wurden, zu kompilieren. Derzeit kann Programmcode unter anderem in den Programmiersprachen C, C++, Objective-C, Java, D, Ada, Fortran, Haskell, Dylan, Python, Ruby, ActionScript, OpenGL Shading Language 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. LLVM unterstützt weiterhin dynamische, interprozedurale Optimierung sowie statische Ahead-of-time- und Just-in-time-Compilierung.[10] Bearbeiten Clang Clang ist ein für auf „C“ basierende Sprachen optimiertes, natives LLVM-Frontend. Es ermöglicht gegenüber dem GCC-Frontend vor allem schnellere Compilierläufe mit geringerem Speicherverbrauch und als Ergebnis schnellere und kleinere Kompilate („executables“). Zudem verfügt es über umfangreichere und genauere statische Analysemethoden, welche dem Entwickler das Debugging erleichtern. Die Unterstützung von C++ gilt seit Version 2.7 auch als stabil und vollumfänglich getestet.[11] Seit September 2009 gilt Clang offiziell als stabil und produktiv verwendbar und findet sich mit LLVM Version 2.6 als fester Bestandteil im LLVM-Compiler-Paket.[12] Clang lässt sich aber auch ohne LLVM als rein statisches Codeanalyse- und Debug-Werkzeug, zum Beispiel beim Einsatz mit anderen Compilern, verwenden.[13] Clang ist zur statischen Code-Analyse in die Entwicklungsumgebung Xcode von Apple für die Programmiersprachen C, Objective-C und C++ integriert. Bearbeiten Versionen Um die Versionen von Clang herrscht zuweilen etwas Verwirrung. Das liegt darin begründet, dass die Versionsnummern bei LLVM im Normalfall etwa im halbjährlichen Rhythmus um einen Zähler ansteigen. Bei Clang wurde bei der letzten Veröffentlichung aber für eine Änderung davon abgewichen. Für zusätzliche Irritation sorgt der Umstand, dass diese Clang/LLVM-Veröffentlichungen in Apples integrierter Entwicklungsumgebung Xcode erneut andere Versionsnummern tragen. Die erste endgültige Version von Clang trug die Nummer 1.0 und war Teil des LLVM-Paketes der Version 2.6. Mit der nächsten offiziellen Veröffentlichung stieg die Versionsnummer von Clang wie üblich um einen Zähler auf 1.1 und war Bestandteil der Version 2.7 des LLVM-Paketes. Bei der nächsten LLVM-Veröffentlichung, der Version 2.8, setzte man aber die Versionsnummer von Clang mit der des übergeordneten LLVM-Projektes gleich, womit dieses seit dem ebenfalls die Nummer 2.8 trägt.[14] Dieselben Veröffentlichungen tragen in der Xcode-IDE analog hierzu aber die Versionsnummern 1.0, 1.5 und 1.6. Bearbeiten 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 Multi-Threading-Code.[15][16] Bearbeiten DragonEgg Bei DragonEgg handelt es sich um ein LLVM-Plugin für die GNU Compiler Collection (ab Version 4.5).[17] 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. Bearbeiten vmkit Hierbei handelt es sich um eine Modifikation der LLVM-VM, welche die Ausführung von Java- und CLI-Bytecode (.net/Mono) ermöglicht. Der Compiler beziehungsweise Linker kann das hochkompakte vmkit (~20 kB) vor den Java- oder CLI-Bytecode packen und ausführen, wodurch die Ausführung auf beliebigen Prozessorarchitekturen und auf System ohne vorherige Installation von Java oder .NET möglich ist.[18] Bearbeiten KLEE Mit KLEE wird ein Werkzeug zur Verfügung gestellt, das es erlaubt, Programme unüberwacht und automatisch auf Fehler zu untersuchen. Dabei wird das Programm Schritt für Schritt ausgeführt. Statt konkrete Werte werden als Eingabe und Zwischenresultate symbolische 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.[19] Bearbeiten 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.[10] x86, X86 64 Bit PowerPC, PowerPC 64Bit ARM, Thumb SPARC Alpha CellSPU PIC16 MIPS MSP430 System z XMOS Xcore Bearbeiten Geschichte Das Projekt startete 2000 an der University of Illinois at Urbana–Champaign unter der Leitung von Vikram Adve und Chris Lattner. 2005 heuerte die Firma Apple eine Gruppe Entwickler für die Weiterentwicklung der LLVM an und stellte Chris Lattner zu deren Führung ein.[20] Seit Version 2.6 vom Oktober 2009 bringt LLVM das eigene Compiler-Frontend Clang mit.[21][22] Bearbeiten 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[23] und Adobe

ActionScript.[7][24] Der LLVM-JIT-Compiler ist in der Lage, ungenutzte statische Zweige des Programms zur Laufzeit zu erkennen und anschließend zu entfernen. Dies optimiert Programme mit einem hohen Grad an Verzweigung. Aus diesem Grund nutzt Apple seit Mac OS X 10.5[25] LLVM im OpenGL-Stack, um einerseits sich selten ändernde Verzweigungspfade zu verkürzen und andererseits Vertex-Shader optimieren zu lassen.