TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik; Institut für Technische Informatik; Professur für VLSI-Entwurfssysteme, Diagnostik und Architektur

LLVM
Low Level Virtual Machine

Chris Iatrou
Chris_Paul.Iatrou@mailbox.tu-dresden.de
Dresden, den 16.01.13

Was ist „LLVM“?

LLVM ist die Bezeichnung eines Dachprojektes zur Entwicklung „modularer, wiederverwendbarer Compiler Technologien und Toolchains“[1]. Dazu gehören:

- Compiler Schnittstellen als *Application Programming Interface* (API).
- Werkzeugsammlung zur optimierungsorientierten Codegenerierung und Codeanalyse.

Was LLVM nicht ist:

- „LLVM“ ist kein Akronym (mehr).
- LLVM ist keine virtuelle Maschine.


[2]

1. Motivation

Traditionelle Compiler (GCC, ICC) verarbeiten Code „geschlossen“.

- Kein Zugang zur IR oder Zwischenergebnissen.
- Keine Möglichkeit, die Optimierungsschritte festzulegen.
- Codeanalyse beschränkt auf Programmcode und nativen Code.

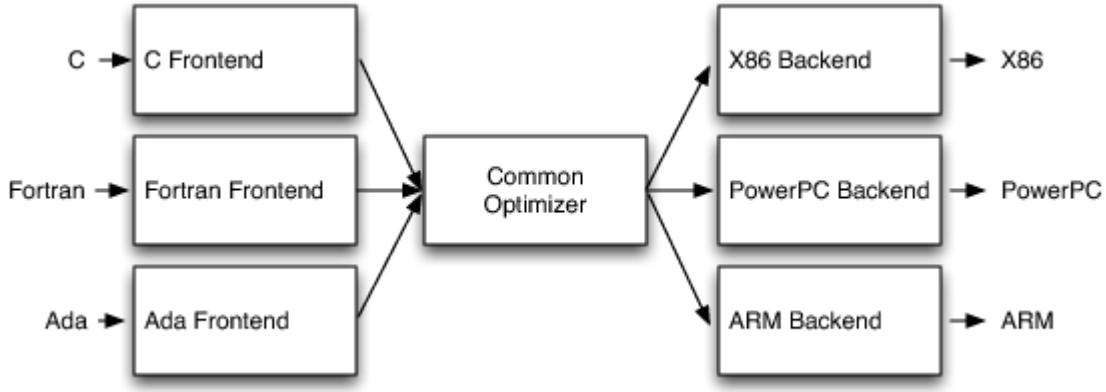
LLVM zerlegt den Kompilierungsprozess in unabhängige Teilschritte.

- Scanner/Parser generieren IR aus beliebigen Eingabedateien.
- Optimierer Teil der LLVM API
- Stellt modularen Linker bereit.

2.1. Grundlagen
Architektur klassischer Compiler

Optimale Modularität durch 3-Phasen-Design:

- Parser Frontends: Unterstützung mehrerer Eingabesprachen.
- IR Optimierer: Übliche Codeoptimierung der IR (AST/dAST).
- Printer Backends: Unterstützung mehrerer Zielarchitekturen.



2.2. Grundlagen
Häufige Architekturen existenter Compiler

„Saubere“ 3-Phasen-Architektur selten vollständig realisiert:

- Interpretierte Hochsprachen entwickeln ihre eigenen dedizierten Ausführungsformate und Compiler.
  → Python, Ruby, Java, Perl...
- Sprachspezifische Compiler unterstützen verschiedene Zielarchitekturen (Printer), aber nur eine Eingabesprache.
  → GHC, FreeBasic...
- „Große“ Compilerpakete sind historisch gewachsen, uneinheitlich geschrieben, schwer wiederverwendbar.
  → GCC
- Kompilierung nicht in Einzelschritte zerlegbar, atomarer Vorgang.

2.3 Grundlagen
Ziel: Kombination aller Vorteile in LLVM

- 3-Phasen-Compiler.
- Exportierbare architektur- und sprachunabhängige Darstellungsformate zwischen den Phasen.
- Modulare Optimierer und Linker in Form einer C++ API.
- Individuell aufrufbare Vorgangsfunktionen auf IR.
- Integration bereits existenter Parser und Printer.
- Integration neuer Optimierungserkenntnisse in Optimierer und Linker (Runtime Profiling, Bytecode Optimizer, etc.).
- Quelloffene Entwicklung.

3.1. LLVM
Konzept

- Compiler Workflow besteht aus frei kombinierbaren Einzelschritten.
- Zwischendarstellung zwischen Schritten ist LLVM IR.
- *LLVM Core* stellt Verarbeitungsbibliotheken für Einzelschritte (Parser/Scanner, Linker, Printer, Optimizer, JIT) bereit.

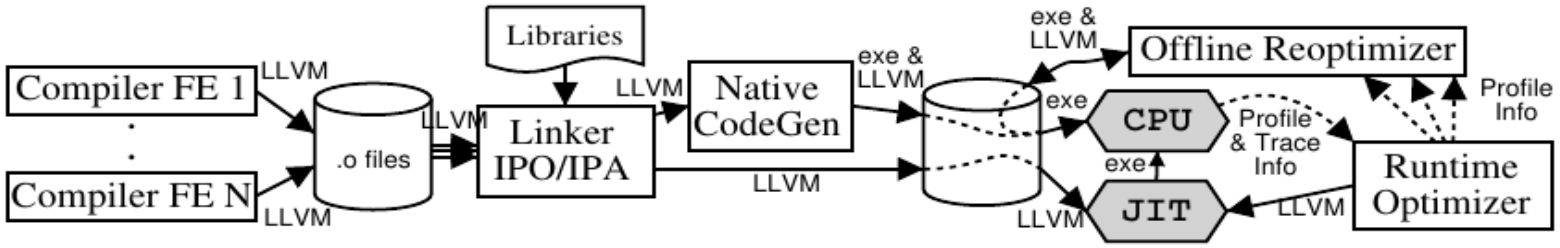


Figure 4: *LLVM system architecture diagram*

3.2. LLVM
Weitere Projektkomponenten

- *LLVM Clang* als Drop-In-Replacement für GCC (C, C++, Objective C).
  - Schnellerer Kompilierprozess als GCC.
  - Bessere Optimierungsergebnisse.
  - Statische Codeanalyse.
  - Wesentlich informativere Warn- und Fehlermeldungen.
- *DragonEgg* integriert LLVM Optimizer & Codegenerator in GCC.
  - Unterstützung für Fortran, Ada, C, C++, Obj-C, Java, Go.
  - Einbindung von OpenMP und OpenCL.
- *LLDB* als nativer LLVM IR Debugger.
  - Verbindet LLVM IR mit LLVM JIT Compiler und Disassembler.
  - Schnellere Symbolladezeit als GDB.

3.2. LLVM
Projektkomponenten (2)

- *LIBC++* als Ersatz für C++ Standard Bibliotheken.
- *LIBCLC* als Ersatz für OpenCL Bibliotheken.
- *VMKIT* als LLVM Umsetzung für virtuelle Java und .NET Maschinen.
- *SAFECode* als Ersatz für Memory Safety Debugging Tools wie Electric Fence.
- Compiler-RT bietet Hooks für low-level native Codegenerierungsfunktionen, die von der Hardware nicht unterstützt werden (Bsp.: Division, i32-zu-i64 Konvertierung, etc.)

3.3. LLVM
Intermediate Representation (IR)

- Die LLVM Intermediate Representation ist...
  - Ein Low Level RISC-like Instruction Set.
  - Starke typisiert.
  - Architekturunabhängig.
- Untestützt Typisierung von Bitbreite und Endianess.
- Unterstützt Casts und Typinformationen.
- Zwei grundlegende (austauschbare) Dateiformate für IR:
  - *.ll: Textueller Assembler (llvm-dis).
  - *.bc: Komprimierter Bytecode (llvm-as).
- „Perfekte Welt“: IR unterliegt weder den Eigenschaften der Quellsprache von den Eigenschaften der Zielarchitektur.

  → Optimal für Optimierung

3.3. LLVM
Intermediate Representation (IR): Beispiel

```
define i32 @add2(i32 %a, i32 %b) {
entry:
  %tmp1 = icmp eq i32 %a, 0
  br i1 %tmp1, label %done, label %recurse

recurse:
  %tmp2 = sub i32 %a, 1
  %tmp3 = add i32 %b, 1
  %tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
  ret i32 %tmp4

done:
  ret i32 %b
}
```

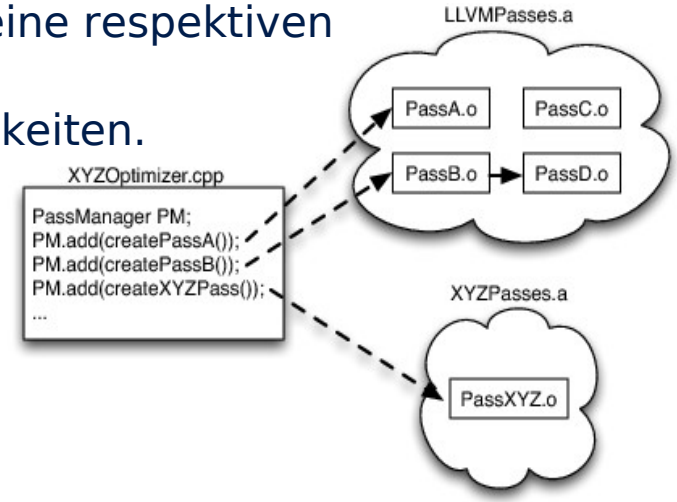
```
/* Perhaps not the most
   efficient way to add two
   Numbers.
*/
unsigned add2(unsigned a, \
              unsigned b) {
  if (a == 0) return b;
  return add2(a-1, b+1);
}
```

3.4. IR Optimierung

- Optimierung ist eine Form des Pattern Matchings auf IR:
  - Suche nach Muster.
  - Verifizieren der Korrektheit einer Ersetzung.
  - Transformation.
- Beispiele (LLVM):
  - inline: Function Integration/Inlining
  - internalize: Internalize Global Symbols
  - loop-unroll: Unroll loops
  - loop-simplify: Rotate Loops
- Problematik: Abfolge der Musterersetzung ist nicht trivial.
  - Anwendung einer Transformation kann Muster vernichten.
  - Anwendung einer Transformation kann neue Muster erzeugen.
  - Teilweise Abhängigkeiten zwischen Optimierungsvorgängen.

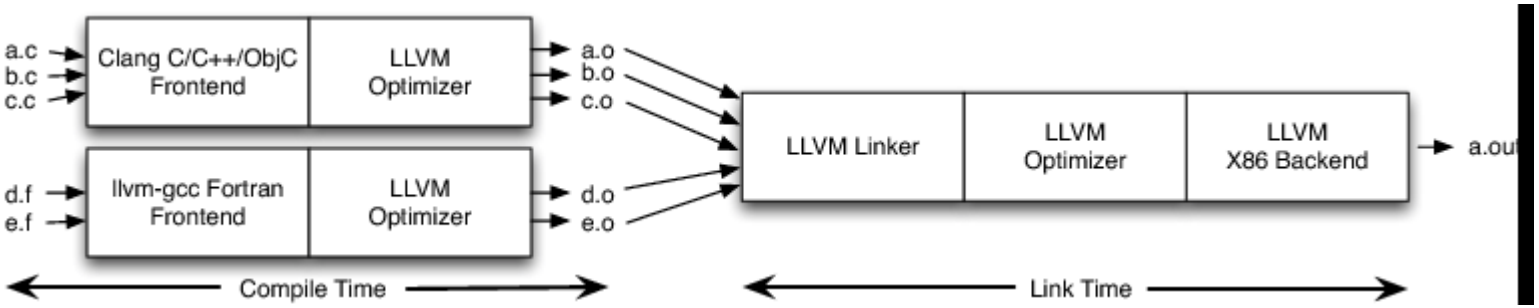
3.4. IR Optimierung (2)

- Lösungsstrategien in LLVM:
  - Optimierungsreihenfolge & -anzahl wird manuell festgelegt („Pass“).
    → Problemspezifische Optimierung (Bsp: Bildbearbeitung)
  - Optimierungsalgorithmen sind in Bibliotheken hinterlegt oder können vom LLVM Frontend aufgerufen werden.
  - Jeder Optimierungsalgorithmus ruft seine respektiven Abhängigkeiten selber auf.
    → Begrenzung auf minimale Abhängigkeiten.
  - Ein-/Ausgabe in ein Passes ist IR.



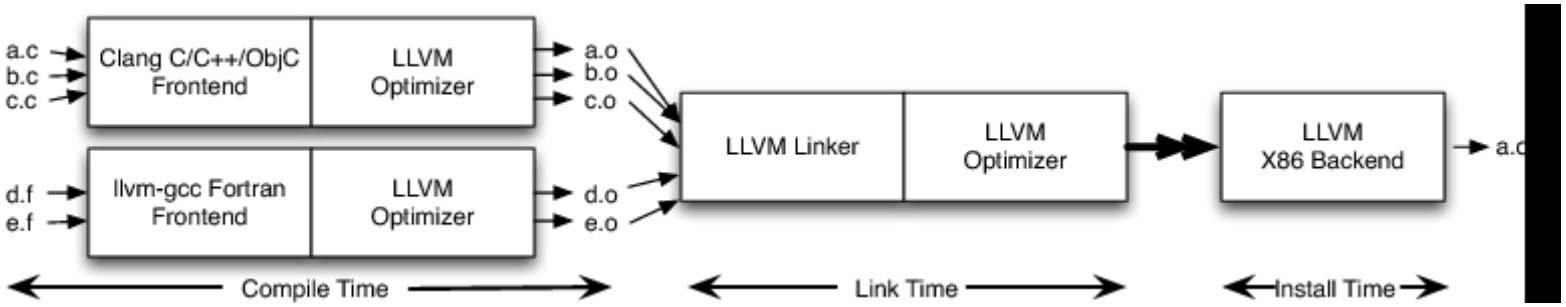
3.4. IR Optimierung (3)

- Gesamtes Programm erst beim Linking verfügbar.
- 3 Möglichkeiten für Optimierung:
  - Einzelne Dateien optimieren.
  - Includes und Quelldateien gemeinsam optimieren.
  - Linked Objects, Includes und Quelldatei gemeinsam optimieren.



3.4 IR Optimierung (4)

- Optimieren von Programmen ohne Kenntnis der Fähigkeiten der Zielarchitektur ist ineffektiv.
- Optimierte Codegenerierung kann erst auf der Zielplattform die Möglichkeiten der Architektur erkennen.
- Lösung: Install Time Compilation
  - Code Printer / Architektur Backen als Umfang eines Installers.

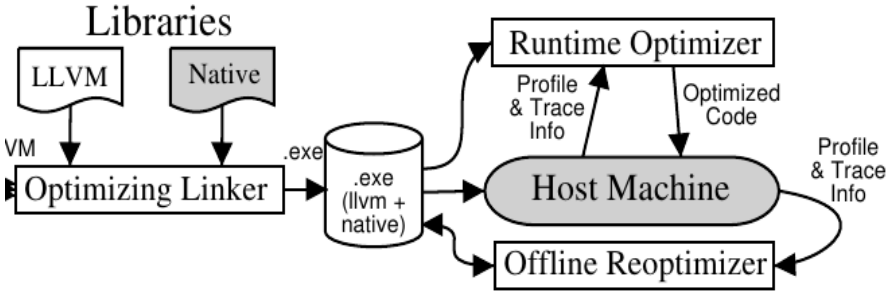


3.5 Runtime Optimization und JIT Compiler

- LLVM kann als
  - IR Interpreter oder
  - JIT Compiler genutzt werden.

- Runtime Profiling: JIT Compiler sammelt (*exportierbare*) Informationen über oft genutzte Codesegmente und Branches.
- Optimierung auf Grundlage des Profilings:
  - Runtime Optimization: Bytecode reordering zur Laufzeit des Programms.
  - Offline Optimization: Idle Zeit des Programms wird für aggressive Optimierung verwendet, die direkt zur Laufzeit nicht möglich ist.

3.5. Runtime Optimization und JIT Compiler (2)



- Anwendungsszenarien für JIT Compiler und Optimierung:
  - Auswertung von offline und Compile Time Optimierungen für algorithmusspezifische Compiler .
    → Beispiel: Bildbearbeitung, Frequenzanalyse, etc.
  - Erhöhen der Sicherheit durch Ausführung einer Zwischenschicht .
    → Beispiel: LLVM SafeCode.
  - Erhöhen der Ausführungsgeschwindigkeit für schwer analysierbaren/interaktiven Code.
  - Generieren von ISE Informationen für ASIP Prozessoren.
    → Beispiel: Woolcano PowerPC 405 Cores auf Xilinx Virtex 4 [10].

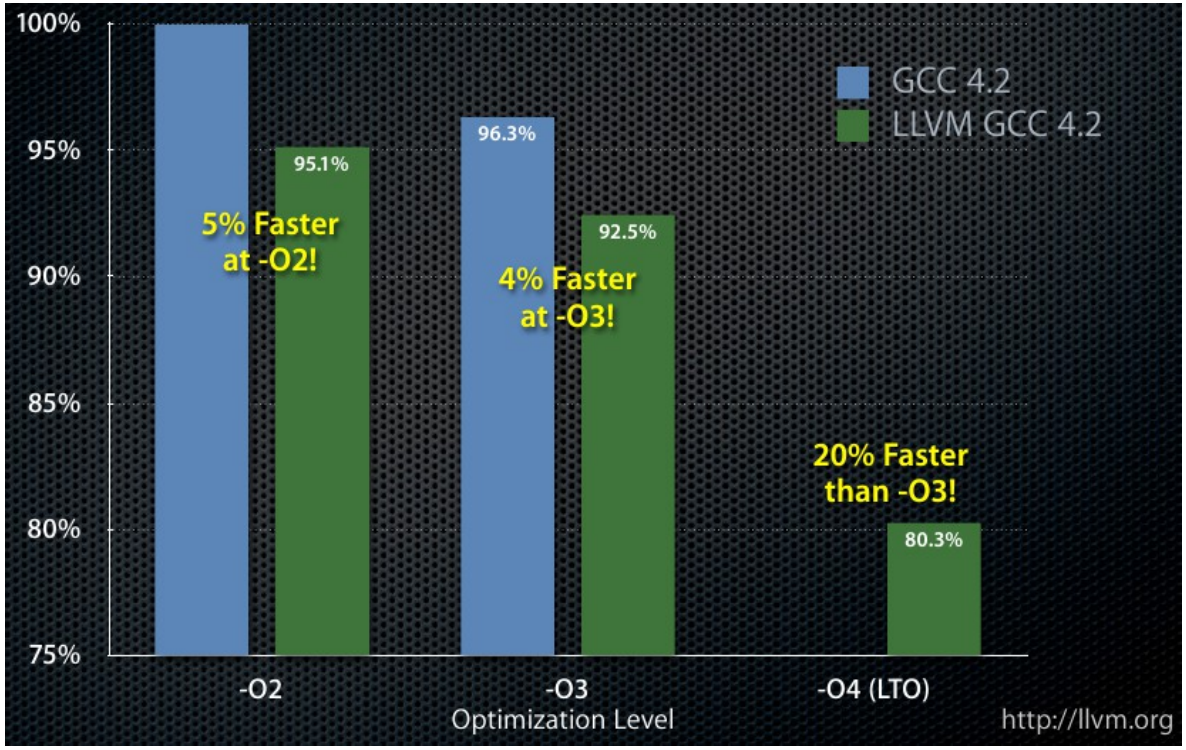
3.6. CLANG als Drop-in-Replacement für GCC

- Compiler für C, C++, ObjC, ObjC++.
- Verwendet ausschließlich LLVM Bibliotheken für Parser, Optimizer und Printer.
  → Komplett unabhängig von GCC.
- Kompilierungsvorgang ca. 20% schneller als GCC [9].
- Bessere native Codegenerierung als GCC [9].
- Wesentlich effizientere Optimierung der IR.
  → LLVM eigene (aggressive) -O4 Optimierungsstufe.

- Akzeptiert LLVM IR als Ein- und Ausgabeformat.
  → Nutzbar als offline Optimizer.
- GCC kompatible Ausführungsoptionen.
  → Drop-In-Replacement.

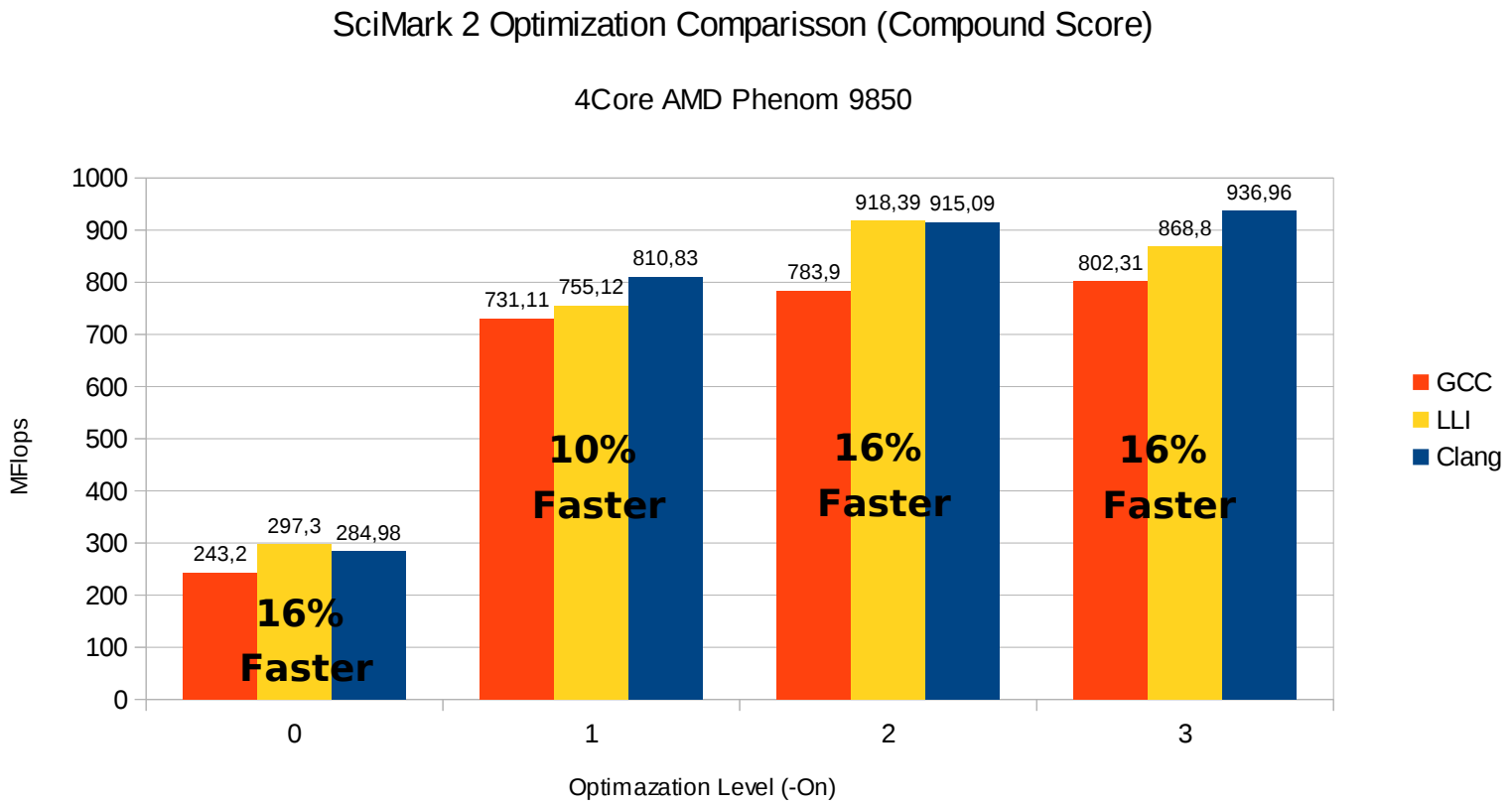
3.7 Benchmarks

- SPEC2000, gemessene Ausführungszeit relativ zu GCC -O2



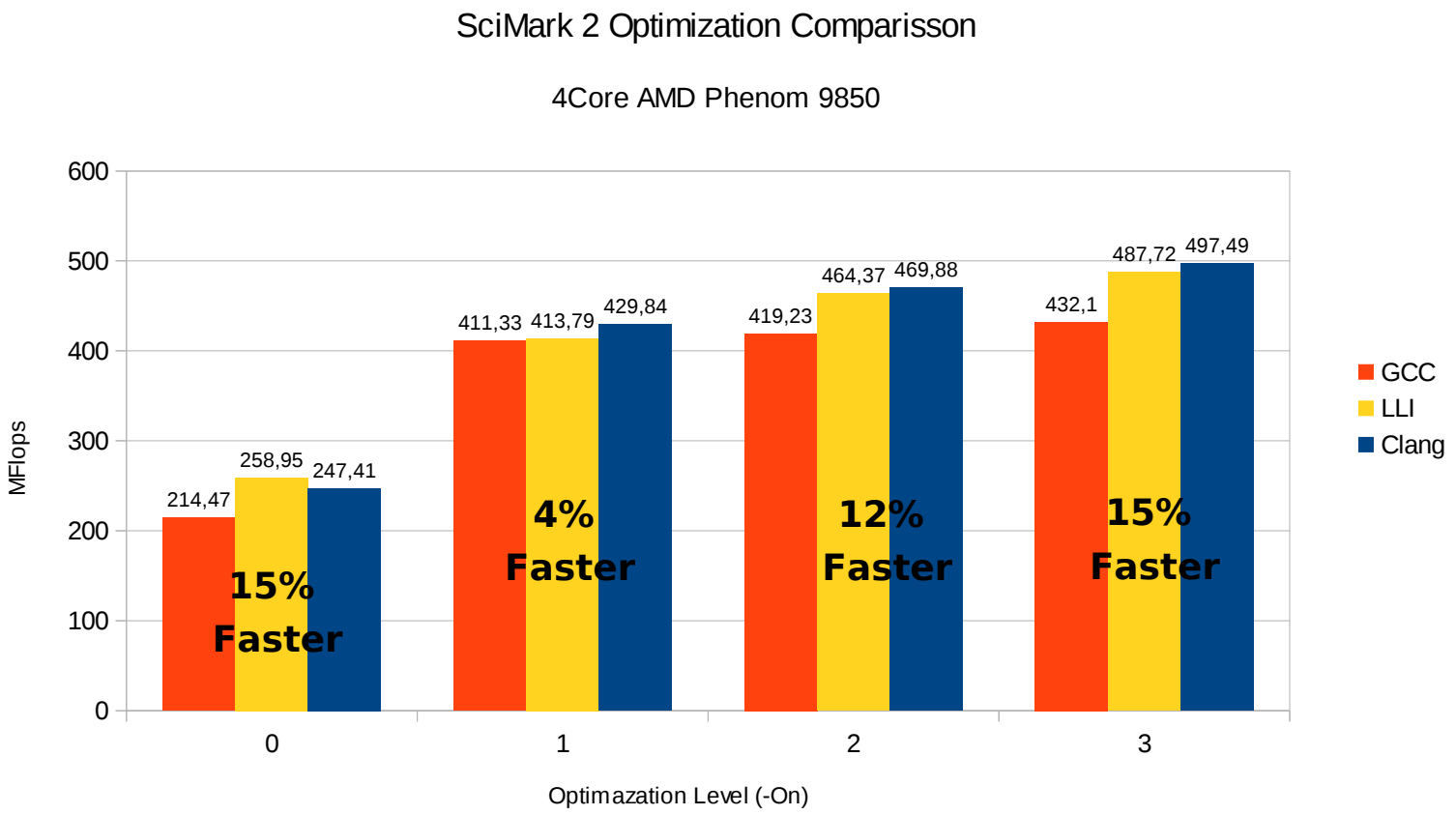
3.7 Benchmarks (2)

- SCIMARK2, absolute FP Leistung, „In Cache“



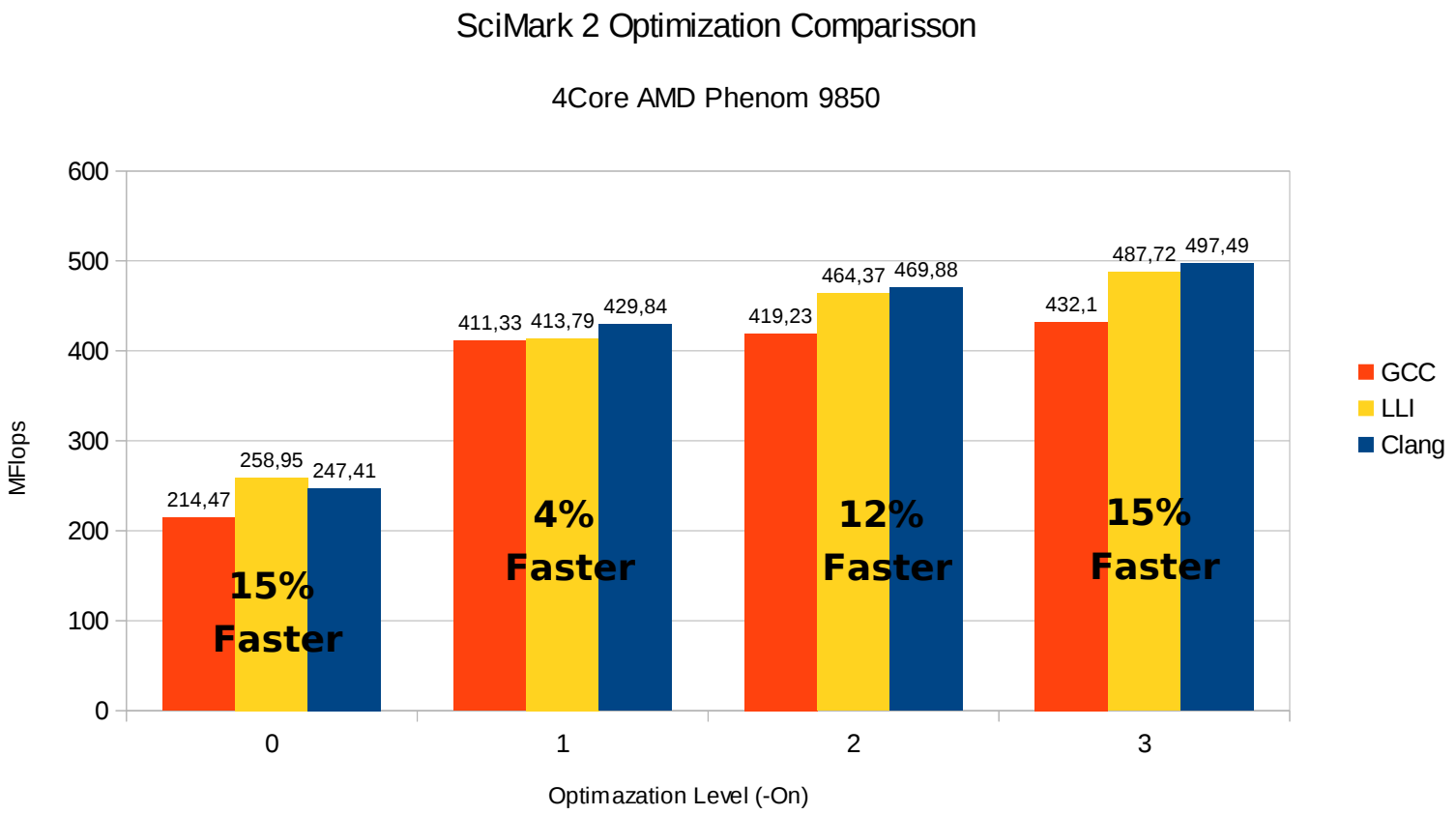
3.7 Benchmarks (3)

- SCIMARK2, absolute FP Leistung, „Large Data“



3.7 Benchmarks (3)

- SCIMARK2, absolute FP Leistung, „Large Data“



3.8 Profiling

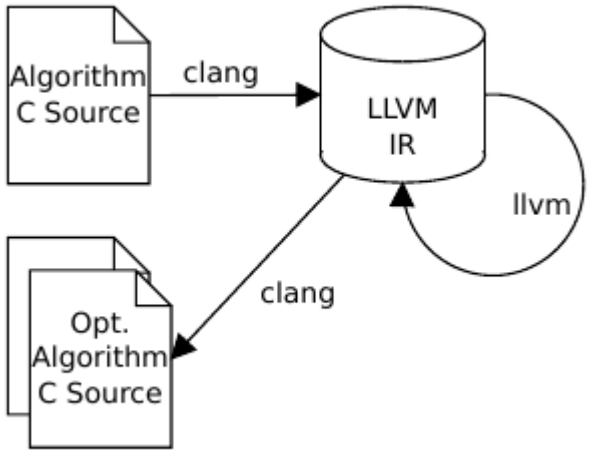
- Profiling Daten für beliebige Programmsegmente zur Analyse aus JIT Compiler extrahierbar.
  - Bsp: 20 häufigsten Blöcke aus Scimark2 ohne Optimierung

```
Top 20 most frequently executed basic blocks:

 ##      %%     Frequency
  1. 10.6273% 1364863500/1.2843e+10     LU_factor() - for.cond65
  2. 10.4694% 1344593250/1.2843e+10     LU_factor() - for.body67
  3. 10.4694% 1344593250/1.2843e+10     LU_factor() - for.inc74
  4. 6.12337% 786426000/1.2843e+10      SparseCompRow_matmult() - for.cond6
  5. 5.10281% 655355000/1.2843e+10      SparseCompRow_matmult() - for.inc
  6. 5.10281% 655355000/1.2843e+10      SparseCompRow_matmult() - for.body8
  7. 4.95071% 635820570/1.2843e+10      SOR_execute() - for.cond11
  8. 4.9007% 629398140/1.2843e+10       SOR_execute() - for.body13
  9. 4.9007% 629398140/1.2843e+10       SOR_execute() - for.inc
 10. 2.09035% 268463502/1.2843e+10      Random_nextDouble() - if.end9
 11. 2.09035% 268463502/1.2843e+10      Random_nextDouble() - if.end
 12. 2.09035% 268463502/1.2843e+10      Random_nextDouble() - if.end15
 13. 2.09035% 268463502/1.2843e+10      Random_nextDouble() - if.else20
 14. 2.09035% 268463502/1.2843e+10      Random_nextDouble() - return
 15. 2.09035% 268463502/1.2843e+10      Random_nextDouble() - entry
 16. 1.96738% 252671545/1.2843e+10      Random_nextDouble() - if.else13
 17. 1.96738% 252671528/1.2843e+10      Random_nextDouble() - if.else
 18. 1.3037% 167434260/1.2843e+10       FFT_transform_internal() - for.cond55
 19. 1.04526% 134242302/1.2843e+10      FFT_transform_internal() - for.body58
 20. 1.04526% 134242302/1.2843e+10      FFT_transform_internal() - for.inc98
NOTE: 3 functions were never executed!
```

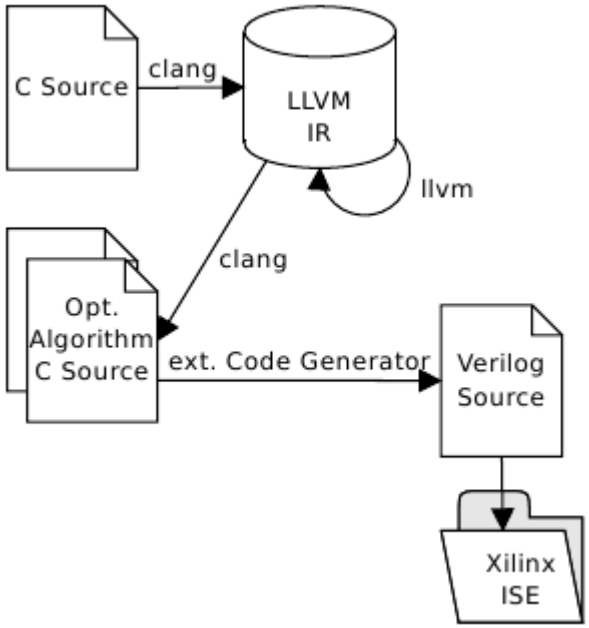
4.1. Anwendungsmöglichkeiten im Fachbereich VLSI Systementwurf

- Designbeschreibung in C, ObjC, CPP, ObjC++.
- Optimierung mit LLVM in IR:
- Optimierung von algorithmischem Komponenten:
  - Rückkonvertierung in C, C++



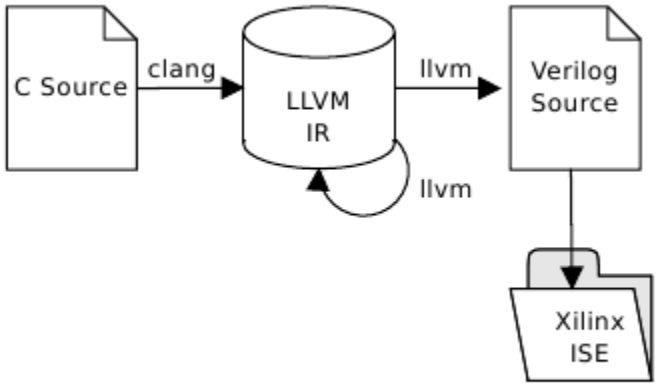
4.1. Anwendungsmöglichkeiten im Fachbereich VLSI Systementwurf (2)

- Code Optimierung mit LLVM.
- Verilog Code Generierung mit externem Transpiler.
  - FPGA-C [8].
  - C-To-Verilog [7].



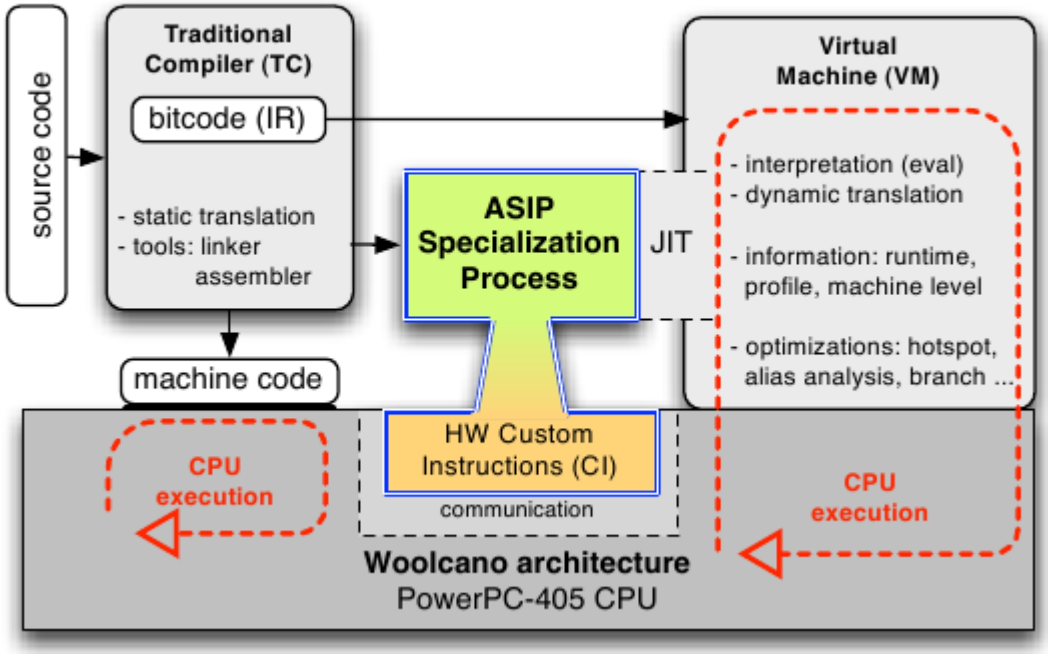
4.1. Anwendungsmöglichkeiten im Fachbereich VLSI Systementwurf (3)

- Code Generation mit Verilog Backend [6]
  - Experimentell, nicht offiziell in LLVM aufgenommen.
  - Nur zur Generierung von Pipeline Architekturen.



4.1. Anwendungsmöglichkeiten im Fachbereich VLSI Systementwurf (4)

- Übertragen von Application Level Profiling auf Echtzeit – Generierung von Instruction Set Extentions [10].
  - Mögliche Erweiterung: Generierung von Prozessorhardware on demand.

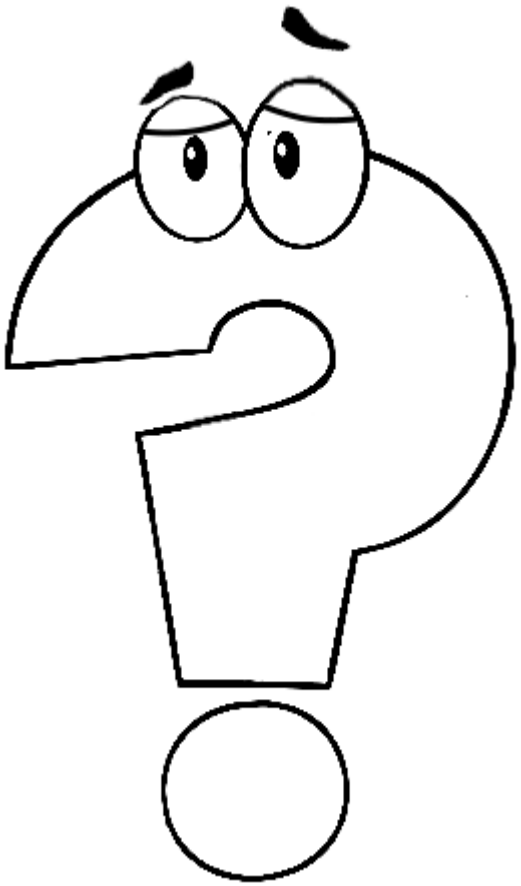


Zusammenfassung

- LLVM ist eine Werkzeugsammlung zur „Codeanalyse und Optimierung“.
- LLVM bietet einfachen und flexiblen Zugang zu den Ergebnissen jeder Stufe eines Kompilierungsvorgangs.
- LLVM Toolchains sind Alternativen zu etablierten Projekten wie GCC.

- Der Einsatz von LLVM im HDL Entwicklungsbereich ist noch experimentell, aber:
  - Vielfältig realisierbar (Transpiler, Optimizer).
  - Gut zugänglich (aufgeräumte & gut dokumentierte Schnittstellen).
  - Bietet neue Möglichkeiten bei der Entwicklung von dynamischen Prozessoren (Anpassung der Hardware an Softwareanforderungen).

Fragen?



»Wissen schafft Brücken.«

Quellen

[1] http://llvm.org/; LLVM Overview

[2] http://llvm.org/img/Dragon.ai; Logo ist Eigentum von Apple Inc. representativ für LLVM.

[3] http://www.aosabook.org/en/llvm.html; „Architecture of OpenSource Applications (Website)“, „LLVM“; Chris Lattner;

[4] „LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation“; Chris Lattner, Vikram Adve; University of Illinois at Urbana-Champaign; 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, Mar. 2004

[5] „Optimization Passes“, LLVM Core Documentation, „ http://llvm.org/docs/Passes.html“

[6] „Synthesis for variable pipelined function units“, Ben-Asher, Comput. Sci. Dep., Haifa Univ., Haif, International Symposium on System-on-Chip 2008 (SOC 2008)

Quellen (2)

[7] http://www.c-to-verilog.com/

[8] http://sourceforge.net/projects/fpgac/

[9] http://llvm.org/pubs/2008-10-04-ACAT-LLVM-Intro.html, „Introduction to the LLVM Compiler System“, Chris Lattner

[10] „Just-in-time Instruction Set Extension – Feasibility and Limitations for an FPGA-based Reconfigurable ASIP Architecture“; Mariusz Grad, Christian Plessl, Paderborn Center for Parallel Computing, University of Paderborn