Introduction

Sebastian Hack

http://compilers.cs.uni-saarland.de

Compiler Construction Core Course 2017
Saarland University
Why take a compiler course?

- Compilers are everywhere!
  - web browsers, graphics drivers, databases, phones, etc.
- Compilers are interesting!
  - We’ll use concepts from automata, graph theory and algorithms, linear programming, lattice theory, etc.
- Learn the foundations of Syntax Analysis
  - Helpful when you design your own language
- Learn the foundations of Program Analysis
  - Helpful beyond compilers (security, verification, PL)
- Improve your software engineering skills
  - Compilers are sophisticated artifacts that are hard to test and debug
- Very good job market for compiler experts!
- Get some easy 9 CP ;)

2
Compilers

• Compilers translate a program from language \( S \) to language \( T \) and thereby implement \( S \) in \( T \).
• Typically, \( S \) is some “higher” programming language and \( T \) some “low-level” language, like assembly.
• Programming languages provide abstractions to make the life of the programmer easier.
• Their straight-forward implementation in \( T \) typically incurs some overhead (in space and time).
• It is the purpose of the compiler to reduce (remove) this overhead.
• More convenient languages need more powerful compilers.
## Abstractions

<table>
<thead>
<tr>
<th>High level</th>
<th>Low level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control flow (for, while, functions)</td>
<td>Instruction pointer, jumps</td>
</tr>
<tr>
<td>Variable</td>
<td>Memory address, register name</td>
</tr>
<tr>
<td>Objects(^1)</td>
<td>Memory, registers</td>
</tr>
<tr>
<td>Lifetimes</td>
<td>Garbage collection, memory management stack frames</td>
</tr>
<tr>
<td>Basic data types (int, float)</td>
<td>Different instruction sets</td>
</tr>
<tr>
<td>Compound data types (structs, arrays, etc.)</td>
<td>Addresses, index arithmetic</td>
</tr>
<tr>
<td>Parallelism (task, data)</td>
<td>Threads, SIMD instructions</td>
</tr>
</tbody>
</table>

\(^1\)C Standard terminology: means a container that holds a value
sort.c
Challenges

• Compile-time has to be sub-quadratic for the common case
• Most relevant code generation problems are at least NP-hard
• Mainly because target machines have resource constraints (finite amount of memory, registers, parallelism, etc.)

• Target machines become less “standard” (aka heterogeneous) Think of GPUs, accelerators or even FPGAs
• Target machines become more complex Memory hierarchy, out-of-order execution, speculation, etc.
• Hence it is often impossible to give a precise notion of “optimality” with respect to the quality of the code.

• End of Dennard Scaling:
  \[
  \text{Performance} \cong \frac{\text{Performance}}{\text{Watt}}
  \]
Example

matmul.c
Compiler Structure: Back in the day

- **AST** = abstract syntax tree
- **Frontend**: Dependent on input language
- **Backend**: Dependent on target language

![Diagram of compiler structure]

- Text → Syntax Ana → AST → Type Chk → AST
- AST → Opt → AST → Code Gen → ASM

- Syntax Ana: Parses input text into an abstract syntax tree (AST)
- Type Chk: Checks the type correctness of the AST
- Opt: Optimizes the AST
- Code Gen: Generates code from the optimized AST
- ASM: Assembles the generated code
• intermediate representation (IR) decouples language-specific AST from code generation
Course Structure

Text → Syntax Ana → AST → Type Chk → AST

(4) Lexing, LL/LR Parsing

IR Cons → IR → Opt → IR

(1) (2) (8)

Control Flow, LLVM

Abstract Interpr., Lattice Theory, Scalar Analyses (SCCP, GVN, Intervals, Pentagons), Liveness, PRE, SSA

Polyhedral Loop Optimization

Code Gen → ASM

(4)

Instruction Selection, Register Allocation

(9) Number of lectures
Course Organization

- Course website (all materials, dates, etc.):
  http://compilers.cs.uni-saarland.de/teaching/cc/2017
- Online discussion forum at:
  https://discourse.cdl.uni-saarland.de
- Please register for the forum until 18 Oct 2017 18:00
- Read and follow the First Steps post
- (voluntary) exercise sheets
- Exam and Re-exam (20 Feb 2018, 20 Mar 2018)
- Grade: 50% exam, 50% project (have to pass both)
Project

- Programming project done in groups of 2–3 students (present to us end of February 2018 the latest)
- Compiler for a subset of C using LLVM
- Own frontend, own optimizations, use LLVM as IR and code generator
- Project is organized in milestones Assignments handed out periodically
- You can test your compiler against our test suite by pushing to your repo (tests run once a day)
- You can track the progress of all groups on the course web site
- Competition: There will be a prize for the fastest compiler and for the compiler that produces the fastest code.
  Precondition: Pass all tests.