Code Generation for Data Processing
Lecture 1: Introduction and Interpretation

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2022/23
Module “Code Generation for Data Processing”

Learning Goals

▶ Getting from an intermediate code representation to machine code
▶ Designing and implementing IRs and machine code generators
▶ Apply for: JIT compilation, query compilation, ISA emulation

Prerequisites

▶ Computer Architecture, Assembly
▶ Databases, Relational Algebra
▶ Beneficial: Compiler Construction, Modern DBs
Topic Overview

Introduction
- Introduction and Interpretation
- Compiler Front-end

Intermediate Representations
- IR Concepts and Design
- LLVM-IR
- LLVM Transforms and Analyses

Compiler Back-end
- Instruction Selection
- Register Allocation
- Linker, Loader, Debuginfo

Applications
- JIT- compilation + Sandboxing
- Query Compilation
- Binary Translation
Lecture Organization

- Lecturer: Dr. Alexis Engelke  engelke@in.tum.de
- Time slot: Thu 10-14, 02.11.018
- Material: https://db.in.tum.de/teaching/ws2223/codegen/

Exam

- Written exam, 90 minutes, no retake, date TBD
- (Might change to oral on very low registration count)
Exercises

- Weekly homework, often with programming exercise
- Submission via e-mail: engelke+cghomework@in.tum.de
  - Probably no explicit grading, feedback on best effort
- Exercise sessions to present and discuss solutions

Grade Bonus

- Requirement: \( N - 2 \) “sufficiently working” homework submissions and at least 2 presentations of homework in class
- Bonus: grades in \([1.3; 4.0]\) improved by 0.3
Why study compilers?

- Critical component of every system, functionality and performance
  - Compiler mostly *alone* responsible for using hardware well

- Brings together many aspects of CS:
  - Theory, algorithms, systems, architecture, software engineering, (ML)

- New developments/requirements pose new challenges
  - New architectures, environments, language concepts, ... 

- High complexity!
## Compiler Lectures @ TUM

### Compiler Construction
**IN2227, SS, THEO**
- Front-end, parsing, semantic analyses, types

### Program Optimization
**IN2053, WS, THEO**
- Analyses, transformations, abstract interpretation

### Virtual Machines
**IN2040, SS, THEO**
- Mapping programming paradigms to IR/bytecode

### Programming Languages
**CIT3230000, WS**
- Implementation of advanced language features

### Code Generation
**CIT3230001, WS**
- Back-end, machine code generation, JIT comp.
Why study code generation?

- Frameworks (LLVM, ...) exist and are comparably good, but often not good enough (performance, features)
  - Many systems with code gen. have their own back-end
  - E.g.: V8, WebKit FTL, .NET RyuJIT, GHC, Zig, QEMU, Umbra, ...

- Machine code is not the only target: bytecode
  - Often used for code execution
  - E.g.: V8, Java, .NET MSIL, BEAM (Erlang), Python, MonetDB, eBPF, ...
  - Allows for flexible design
  - But: efficient execution needs machine code generation
Proebsting’s Law

“Compiler advances double computing power every 18 years.”

– Todd Proebsting, 1998

Still optimistic; depends on number of abstractions

1http://proebsting.cs.arizona.edu/law.html
Motivational Example: Brainfuck

- Turing-complete esoteric programming language, 8 operations
  - Input/output: . ,
  - Moving pointer over infinite array: < >
  - Increment/decrement: + -
  - Jump to matching bracket if (not) zero: [ ]

```
++++++[->++++++<]>.
```

- Execution with pen/paper? 😐
Program Execution

Program → Hardware → Result

Programs
- High flexibility (possibly)
- Many abstractions (typically)
- Several paradigms

Hardware/ISA
- Low-level interface
- Few operations, imperative
- “Not easy” to write
Motivational Example: Brainfuck – Interpretation

▶ Write an interpreter!

```c
unsigned char state[10000];
unsigned ptr = 0, pc = 0;
while (prog[pc])
    switch (prog[pc++]) {
    case ’.’: putchar(state[ptr]); break;
    case ,,: state[ptr] = getchar(); break;
    case ’>’: ptr++; break;
    case ’<’: ptr--; break;
    case ’+’: state[ptr]++; break;
    case ’-’: state[ptr]--; break;
    case ’[’: state[ptr] || (pc = matchParen(pc, prog)); break;
    case ’]’: state[ptr] && (pc = matchParen(pc, prog)); break;
    }
```
Program Execution

Compiler

Program → Compiler → Program

- Translate program to other lang.
- Might optimize/improve program

- C, C++, Rust → machine code
- Python, Java → bytecode

Interpreter

Program → Interpreter → Result

- Directly execute program
- Computes program result

- Shell scripts, Python bytecode, machine code (conceptually)

- Multiple compilation steps can precede the “final interpretation”
Compilers

- Targets: machine code, bytecode, or other source language
- Typical goals: better language usability and performance
  - Make lang. usable at all, faster, use less resources, etc.

- Constraints: specs, resources (comp.-time, etc.), requirements (perf., etc.)

- Examples:
  - “Classic” compilers source → machine code
  - JIT compilation of JavaScript, WebAssembly, Java bytecode, ...
  - Database query compilation
  - ISA emulation/binary translation
Compiler Structure: Monolithic

- Inflexible architecture, hard to retarget
Compiler Structure: Two-phase architecture

- **Source Program** → **Front-end** → **IR** → **Back-end** → **Machine Code**
  - **Front-end**
    - Parses source code
    - Detects syntax/semantical errors
    - Emits *intermediate representation* to encode semantics/knowledge
    - Typically: $O(n)$ or $O(n \log n)$
  - **Back-end**
    - Translates IR to target architecture
    - Can assume valid IR (⇝ no errors)
    - Possibly one back-end per arch.
    - Contains $NP$-complete problems
Compiler Structure: Three-phase architecture

- Source Program
- Front-end → IR
- Optimizer → IR
- Back-end → Machine Code
- Errors

- **Optimizer:** analyze/transform/rewrite program inside IR

- **Conceptual architecture:** real compilers typically much more complex
  - Several IRs in front-end and back-end, optimizations on different IRs
  - Multiple front-ends for different languages
  - Multiple back-ends for different architectures
Compiler Front-end

1. Tokenizer: recognize words, numbers, operators, etc.
   - Example: $a+b*c \rightarrow \text{ID}(a) \text{ PLUS ID}(b) \text{ TIMES ID}(c)$

2. Parser: build (abstract) syntax tree, check for syntax errors
   - Syntax Tree: describe grammatical structure of complete program
     Example: $\text{expr}(\text{"a"}, \text{op}(\text{"+"}), \text{expr}(\text{"b"}, \text{op}(\text{"*"}), \text{expr}(\text{"c"}))$
   - Abstract Syntax Tree: only relevant information, more concise
     Example: $\text{plus}(\text{"a"}, \text{times}(\text{"b"}, \text{"c"}))$

3. Semantic Analysis: check types, variable existence, etc.

4. IR Generator: produce IR for next stage
   - This might be the AST itself
Compiler Back-end

1. Instruction Selection: map IR operations to target instructions
   - Use target features: special insts., addressing modes, ...
   - Still using virtual/unlimited registers

2. Instruction Scheduling: optimize order for target arch.
   - Start memory/high-latency earlier, etc.
   - Requires knowledge about micro-architecture

3. Register Allocation: map values to fixed register set/stack
   - Use available registers effectively, minimize stack usage
Motivational Example: Brainfuck – Front-end

- Need to skip comments
- Bracket searching is expensive/redundant
- Idea: “parse” program!
- Tokenizer: yield next operation, skipping comments
- Parser: find matching brackets, construct AST
Motivational Example: Brainfuck – AST Interpretation

- AST can be interpreted recursively

```c
struct node { char kind; int cldCnt; struct node* cld; };
struct state { unsigned char* arr; size_t ptr; };
void donode(struct node* n, struct state* s) {
    switch (n->kind) {
    case '+': s->arr[s->ptr]++; break;
    // ...
    case '[': while (s->arr[s->ptr]) children(n); break;
    case 0: children(n); break; // root
    }
}
void children(struct node* n, struct state* s) {
    for (int i = 0; i < n->cldCnt; i++) donode(n->cld + i, s);
}
```
Motivational Example: Brainfuck – Optimization

- Inefficient sequences of +/-/>< can be combined
  - Trivially done when generating IR

- Fold patterns into more high-level operations
  - [-] = set zero
  - [>] = find next zero (memchr)
  - [-->++<<] = add to next two siblings, set zero
  - [-->+++<] = add 3 times to next sibling, set zero
  - …
Motivational Example: Brainfuck – Optimization

- Fold offset into operation
  - right(2) add(1) = addoff(2, 1) right(2)
  - Also possible with loops

- Analysis: does loop move pointer?
  - Loops that keep position intact allow more optimizations
  - Maybe distinguish “regular loops” from arbitrary loops?
- Get rid of all “effect-less” pointer movements

- Combine arithmetic operations, disambiguate addresses, etc.
Motivational Example: Brainfuck – Bytecode

- Tree is nice, but rather inefficient $\leadsto$ flat and compact bytecode
- Avoid pointer dereferences/indirections; keep code size small

- Superinstructions: combine common sequences to one instruction
- Maybe dispatch two instructions at once?
  - `switch (ops[pc] | ops[pc] << 8)`
Motivational Example: Brainfuck – Threaded Interpretation

- Simple switch-case dispatch has lots of branch misses
- Threaded interpretation: at end of a handler, jump to next op

```c
struct op { char op; char data; };
struct state { unsigned char* arr; size_t ptr; };
void threadedInterp(struct op* ops, struct state* s) {
    static const void* table[] = { &&CASE_ADD, &&CASE_RIGHT, };
    #define DISPATCH do { goto *table[(++pc)->op]; } while (0)

    struct op* pc = ops;
    DISPATCH;

    CASE_ADD: s->arr[s->ptr] += pc->data; DISPATCH;
    CASE_RIGHT: s->arr += pc->data; DISPATCH;
}
```
Fast Interpretation

- Key technique to “avoid” compilation to machine code
- Preprocess program into efficiently executable bytecode
  - Easily identifiable opcode, homogeneous structure
  - Can be linear (fast to execute), but trees also work
- Perhaps optimize – if it’s worth the benefit
  - Fold constants, combine instructions, . . .
  - Consider superinstructions for common sequences
- For very cold code: avoid transformations at all
- Use threaded-interpretation to avoid branch misses
Compiler: Surrounding – Compile-time

- Typical environment for a C/C++ compiler:

- Calling Convention: interface with other objects/libraries
- Build systems, dependencies, debuggers, etc.
- Compilation target machine (hardware, VM, etc.)
Compiler: Surrounding – Run-time

- OS interface (I/O, ...)
- Memory management (allocation, GC, ...)
- Parallelization, threads, ...
- VM for execution of virtual assembly (JVM, ...)
- Run-time type checking
- Error handling: exception unwinding, assertions, ...
- Reflection, RTTI
Motivational Example: Brainfuck – Runtime Environment

- Needs I/O for . and ,
- Memory management: infinitely sized array
- Allocate on demand (easy?)
  - What if main memory or address space is insufficient?
- Deallocation of empty pages?
- Error handling: unmatched brackets
Compilation point: AoT vs. JIT

**Ahead-of-Time (AoT)**
- All code has to be compiled
- No dynamic optimizations
- Compilation-time secondary concern

**Just-in-Time (JIT)**
- Compilation-time is critical
- Code can be compiled on-demand
  - Incremental optimization, too
- Handle cold code fast
- Dynamic specializations possible
- Allows for `eval()`

Various hybrid combinations possible
Compiler Design: Effect of Languages – Imperative

- Step-by-step execution of program
  modification of state
- Close to hardware execution model
- Direct influence of result
- Tracking of state is complex
- Dynamic typing: more complexity
- Limits optimization possibilities

```c
void addvec(int* a, const int* b) {
    for (unsigned i = 0; i < 4; i++)
        a[i] += b[i]; // vectorizable?
}

func:
    mov [rdi], rsi
    mov [rdi+8], rdx
    mov [rdi], 0 // redundant?
    ret
```
Compiler Design: Effect of Languages – Declarative

- Describes execution target
- Compiler has to derive good mapping to imperative hardware
- Allows for more optimizations
- Mapping to hardware non-trivial
  - Might need more stages
  - Preserve semantic info for opt!
- Programmer has less “control”

```sql
select s.name
from studenten s
where exists (select 1
    from hoeren h
    where h.matrno=s.matrno)

let rec fac = function
    | 0 | 1 -> 1
    | n -> n * fac (n - 1)
```
Introduction and Interpretation – Summary

- Compilation vs. interpretation and combinations
- Compilers are key to usable/performant languages
- Target language typically machine code or bytecode
- Three-phase architecture widely used
- Interpretation techniques: bytecode, threaded interpretation, ...
- JIT compilation imposes different constraints
Introduction and Interpretation – Questions

▶ What is typically compiled and what is interpreted? Why?
  ▶ PostScript, C, JavaScript, HTML, SQL
▶ What are typical types of output languages of compilers?
▶ How does a compiler IR differ from the source input?
▶ What is the impact of the language paradigm on optimizations?
▶ What are important factors for an efficient interpreter?
▶ What are key differences between AoT and JIT compilation?