Compiling Techniques

Lecture 18: Introduction to Code Generation
Overview

**Frontend:** Lexer/Parser & AST Builder & Semantic Analyzer (CW 1 & 2)

**Middle End:** Optimizations

**Backend:** ChocoPy -> FrontEnd -> Middle End -> Backend -> ASM (RISC-V)
The Backend

- Translate IR into target machine code
- Choose instructions to implement each IR operation
- Decide which value to keep in registers
- Ensure conformance with system interfaces
- Automation has been less successful in the back end
Mapping the IR into assembly code (in our case AST to RISC-V-V assembly)
- Assumes a fixed storage mapping & code shape
- Combining operations, using address modes
Register Allocation

- Deciding which value reside in a register
- Minimise amount of spilling
Instruction Scheduling

- Avoid hardware stalls and interlocks
- Reordering operations to hide latencies
- Use all functional units productively
The Big Picture

How hard are these problems?

- **Instruction selection**
  - Can make locally optimal choices, with automated tool
  - Global optimality is NP-Complete

- **Instruction scheduling**
  - Single basic block ⇒ heuristic work quickly
  - General problem, with control flow ⇒ NP-Complete

- **Register allocation**
  - Single basic block, no spilling ⇒ linear time
  - Whole procedure is NP-Complete (graph colouring algorithm)

*These problems are tightly coupled*

However, conventional wisdom says we lose little by solving these problems independently.
How to solve these problems?

- **Instruction selection**
  - Use some form of pattern matching
  - Assume enough registers or target “important” values

- **Instruction scheduling**
  - Within a block, list scheduling is “close” to optimal
  - Across blocks, build framework to apply list scheduling

- **Register allocation**
  - Start from virtual registers & map “enough” into k
  - With targeting, focus on “good” priority heuristic

---

**Approximate Solutions**

Will be important to define good metrics for “close”, “good”, “enough”, . . . .
Generating Code for a Register-Based Machine

The key code quality issue is holding values in registers

- when can a value be safely allocated to a register?
  - When only 1 name can reference its value
  - Pointers, parameters, aggregates & arrays all cause trouble

- when should a value be allocated to a register?
  - when it is both safe & profitable

Encoding this knowledge into the IR:

- assign a virtual register to anything that go into one
- load or store the others at each reference
Representing Values in Memory/Registers

Statically Typed Languages (C/C++)

Dynamically Typed Languages (Python)

Plain (Unboxed) Values

- a : int
- b : bool
- c : int

Boxed Values

- a : ?
- b : ?
- c : ?

ChocoPy

Plain (Unboxed) Values:
- a : int
  - b : bool
  - c : int
  - 36
  - T
  - 12

Boxed Values:
- a : ?
  - b : ?
  - c : ?
  - int
  - 36
  - bool
  - T
  - int
  - 12
Properties of Boxed Values

Advantages

● Better for
  ○ Dynamically Typed Languages
  ○ Dynamic Dispatch
● Can be Treated More Uniformly in The Compiler

Disadvantages

● Reduced Performance (Runtime)
● Higher Memory Cost
● Requires Complex Runtime System
Instruction Selection: SSA Rewrites Across Dialects

%0 : !choco.ir.named_type<"int"> = choco.ir.literal() ["value" = 42 : !i32]

%0 : !riscv_ssa.reg = riscv_ssa.li() ["immediate" = 42 : !i32]
Rewriting Simple Literals

def rewrite_literal(self, op: Literal, rewriter: PatternRewriter):
    value = op.value

    if isinstance(value, IntegerAttr):
        constant = RISCVSSA.LIOP(op.value)

    if isinstance(value, BoolAttr):
        if value.data == True:
            constant = RISCVSSA.LIOP(1)
        elif value.data == False:
            constant = RISCVSSA.LIOP(0)

    rewriter.replace_op(op, [constant])
PatternRewriter: insert|erase_op - the low-level interface

def insert_op_before_matched_op(self, op: (Operation | Sequence[Operation]))
def insert_op_after_matched_op(self, op: (Operation | Sequence[Operation]))
def insert_op_at_start(self, op: Operation | Sequence[Operation], block: Block)
def insert_op_at_end(self, op: Operation | Sequence[Operation], block: Block)
def insert_op_before(self, op: Operation | Sequence[Operation], target_op: Operation)
def insert_op_after(self, op: Operation | Sequence[Operation], target_op: Operation)
def erase_op(self, op: Operation, safe_erase: bool = True)
Replace an operation with a list of new operations

The users of the old operation are automatically connected to use the new operation.
What about Control Flow

```python
if 1 < 0:
    print(1)
else:
    print(2)
```

```
%0 = "riscv_ssa.li"() <{"immediate" = 1 : i32}> : () -> !riscv_ssa.reg
%1 = "riscv_ssa.li"() <{"immediate" = 0 : i32}> : () -> !riscv_ssa.reg
%2 = "riscv_ssa.slt"(%0, %1) : (!riscv_ssa.reg, !riscv_ssa.reg) -> !riscv_ssa.reg
%3 = "riscv_ssa.li"() <{"immediate" = 0 : i32}> : () -> !riscv_ssa.reg
"riscv_ssa.beq"(%2, %3) <{"offset" = #riscv.label<if_else_1>}> : ...

"riscv_ssa.label"() <{"label" = #riscv.label<if_then_1}> : () -> ()
%4 = "riscv_ssa.li"() <{"immediate" = 1 : i32}> : () -> !riscv_ssa.reg
"riscv_ssa.call"(%4) <{"func_name" = "_print_int"}> : (!riscv_ssa.reg) -> ()
"riscv_ssa.j"() <{"offset" = #riscv.label<if_after_1}> : () -> ()

else:
    print(2)

"riscv_ssa.label"() <{"label" = #riscv.label<if_else_1}> : () -> ()
%5 = "riscv_ssa.li"() <{"immediate" = 2 : i32}> : () -> !riscv_ssa.reg
"riscv_ssa.call"(%5) <{"func_name" = "_print_int"}> : (!riscv_ssa.reg) -> ()

...
Basic Blocks: Linear Sequences of Operations

```plaintext
if i < 0
  fallthrough
  if_then_1
  print(2)
  if_after_1
  ...
  beq (%2, %3)
  if_else_1
  print(1)
```
Do we need basic blocks

**Advantages**
- Facilitate reordering of control flow

**Disadvantages**
- Pattern rewrites become more difficult
- Not as easy to reason about
PatternRewriter: Moving Blocks

def inline_block_before_matched_op(self, block: Block)

def inline_block_before(self, block: Block, op: Operation)

def inline_block_after(self, block: Block, op: Operation):