Code Generation for Data Processing (CIT3230001)

Prof. Dr. Thomas Neumann   Friday, February 17, 2023 16:30 – 18:00

Problem 1: Code Representations and Formats (12)

a) *(3 pts)* Name the three typical phases of a compiler front-end and the three phases of a back-end.

b) *(1 pts)* Why do compilers often use several intermediate representations (IRs) instead of going from source code to machine code in a single step?

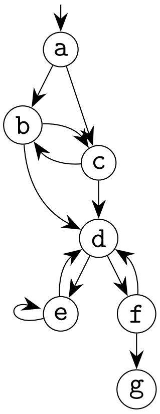
c) *(2 pts)* What are advantages of compiling to bytecode as opposed to machine code?

d) *(2 pts)* Name two widespread bytecode languages that implement a stack machine. What is the key advantage of stack machines over register machines?

e) *(2 pts)* Give name and purpose of the two ELF sections that typically contain unwind information.

f) *(2 pts)* Which components are responsible for resolving static and dynamic relocations?

Problem 2: Control Flow Graphs (20)



a) *(2 pts)* What is a basic block and what are its properties regarding control flow?

b) *(3 pts)* Is a Control Flow Graph (CFG) *always* a connected graph during compilation? If not, what does it mean and how can this happen?

c) *(2 pts)* When does some basic block A *strictly dominate* a basic block B?

d) *(5 pts)* Give the dominator tree for the CFG on the right.

e) *(6 pts)* Identify all loops in the CFG on the right. For each loop, give its basic blocks, its entry block(s), and its header. When ambiguous, give all possibilities.

f) *(2 pts)* What is an irreducible loop? Is the CFG on the right reducible?

Problem 3: Single Static Assignment (20)

a) *(1 pts)* Name an important advantage of using SSA.

b) *(19 pts)* Derive an LLVM-IR module for the C code on the right, include function definitions and declarations. An `int` is 32 bits wide. Construct pruned SSA and avoid `alloca`.

Note: your LLVM-IR does not need to be 100% syntactically correct, but important concepts should be shown clearly.

```c
int f(int);
void magic(int a, int b) {
  while (a != b || f(a) < 7) {
    if (a > b) {
      a = f(b);
    } else {
      b = 5;
      a = f(a);
    }
    b = f(b);
  }
}
```

Problem 4: Generating Executable Code (12)

a) *(4 pts)* For achieving low compile times, which instruction selection strategy is beneficial? Briefly describe the approach and a related simple optimization to improve code quality.

b) *(2 pts)* What is a critical edge and why is it problematic for SSA destruction?

c) *(4 pts)* Describe two approaches and their respective advantage and disadvantage to handle critical edges during SSA destruction.

d) *(2 pts)* Briefly outline a common strategy of JIT compilers to combine low-latency with high-performance code execution.

Problem 5: Unwinding (18)

a) *(3 pts)* C++ exceptions are often implemented with stack unwinding ("zero-overhead exception"). When and why is this beneficial?

b) *(2 pts)* How do dynamic shared objects loaded by the kernel or runtime linker and JIT-compiled code differ in exposing/registering their unwind information?

c) *(1 pts)* What is the canonical frame address (CFA) of a stack frame?

d) *(8 pts)* Consider the assembly code and the DWARF CFI program below. Construct the call frame information table with a row for *every instruction* and columns for the instruction address, the CFA, the return address, and `rbp`. *Note:* the DWARF code is erroneous, see below.

Disassembly

```
15f0: cmp rdi, 0x5
15f4: ja 1610
15f6: push rbp
15f7: shl rdi, 0x4
15fb: mov rbp, rsp
15fe: sub rsp, rdi
1601: mov rdi, rsp
1604: call 1660
1609: leave
160a: ret
160b: nop [rax+rax*1+0x0]
1610: lea eax, [rdi+0x1]
1613: ret
```

Decoded DWARF CIE

```
CIE
  Format: DWARF32
  Version: 1
  Augmentation: "zR"
  Code alignment factor: 1
  Data alignment factor: -8
  Return address column: 16
  Augmentation data: 1B

  DW_CFA_def_cfa: RSP +8
  DW_CFA_offset: RIP -8
  DW_CFA_nop:
  DW_CFA_nop:
```

Decoded DWARF FDE

```
FDE cie=0000 pc=15f0...1614
  Format: DWARF32
  DW_CFA_advance_loc: 7
  DW_CFA_def_cfa_offset: +16
  DW_CFA_offset: RBP -16
  DW_CFA_advance_loc: 7
  DW_CFA_def_cfa_register: RBP
  DW_CFA_advance_loc: 12
  DW_CFA_def_cfa: RSP +8
```

e) *(4 pts)* Where exactly is a mismatch between the CFI program and the assembly code? Accurately describe a fix for the FDE to match the assembly code. You may only use the following instructions:

- `DW_CFA_advance_loc` *delta*
- `DW_CFA_def_cfa` *reg off*, `DW_CFA_def_cfa_register` *reg*, `DW_CFA_def_cfa_offset` *off*
- `DW_CFA_offset` *reg off*, `DW_CFA_restore` *reg*
- `DW_CFA_remember_state`, `DW_CFA_restore_state`

Problem 6: Query Compilation (8)

a) *(1 pts)* What does vectorization refer to in the context of query execution?

b) *(2 pts)* What is the key benefit of vectorized processing for unpredictable selection predicates?

c) *(5 pts)* Derive the query plan for the SQLite3 bytecode below.

```
addr  opcode         p1  p2  p3  p4        p5  comment
----  -------------  --  --  --  --------  --  -------------
0     Init           0   13  0             0   Start at 13
1     OpenRead       0   2   0   1         0   root=2 iDb=0; "customer"
2     OpenRead       1   2   0   2         0   root=2 iDb=0; "customer"
3     Rewind         0   12  0             0
4       Rewind       1   12  0             0
5         Column     0   0   1             0   r[1]= cursor 0 column 0
6         Column     1   1   2             0   r[2]= cursor 1 column 1
7         Eq         2   10  1   BINARY-8  83  if r[1]==r[2] goto 10
8         Column     0   0   3             0   r[3]= cursor 0 column 0
9         ResultRow  3   1   0             0   output=r[3]
10      Next         1   5   0             1
11    Next           0   4   0             1
12    Halt           0   0   0             0
13    Transaction    0   0   1   0         1   usesStmtJournal=0
14    Goto           0   1   0             0
```