Code Generation: Intro

Sebastian Hack
Saarland University

Compiler Construction
W2015

Code Generation

Consists (roughly) of three parts:

1. Instruction Selection
   Select processor instructions for IR instructions

2. Instruction Scheduling
   Linearize data-dependence graph of each basic block.

3. Register Allocation
   For each program point, decide which IR variable resides in what register or in memory.

Properties:

- All three are influence each other (phase ordering problem)
- For reasonably realistic scenarios,
  each one is a NP-hard optimization problem
- Compilers usually attack them heuristically
  (which works ok, often well)

Target Properties that Compilers have to care about

- Instruction set architecture (ISA) of the CPU
  - How to “talk” to the processor
  - Affects several optimizations and transformations

- Aspects of the CPU’s implementation
  - Organization of instruction execution (pipeline)
  - Memory hierarchy topology
    (cache sizes, associativity, sharing among cores)
  - Core topology (for automatic parallelization)

- Conventions of the runtime / operating system
  - parameter passing of subroutines in libraries
  - how to address global data
  - interface to garbage collector
  - ...

Instruction Set Architectures

- RISC
  - Many registers, typically 32
  - Few simple address modes
  - Load-/store-architecture
  - three-address code: $Rz \leftarrow Rx \oplus Ry$
  - constant-length instruction encoding, typically 4 bytes
  - VLIW like RISC but compiler packs insns into bundles and manages parallel exec of instructions

- CISC
  - Fewer registers, 8–16
  - Complex address modes
  - Memory operands
  - two-address code: $Rx \leftarrow Rx \oplus Ry$
  - variable-length instruction encoding (x86: from 1 to 15 bytes)

Beware of the classical RISC / CISC debate! Today, most CPUs are RISC inside but might have CISC ISA. The processor translates CISC instructions into RISC instructions internally

ISA Examples: MIPS

- prototypical RISC ISA
- 32 registers
- minimal core instruction set

```
int *A;
...
A[i+2] += 100
```

```
# $a0 = A, $a1 = i
sal    $t0 $a1 2
addu   $t0 $a0 $t0
lw     $t1 8($t0)
addiu  $t1 $t1 100
sw     $t1 8($t0)
```

= 20 Bytes

ISA Examples: x86

- CISC ISA
- 8 Registers (64-bit mode 16 registers)
- Powerful addressing modes:
  base register + (1,2,4) * index register + constant
- For many instructions, one operand can be a memory cell (instead of reg)
- Inhomogeneous register usage:
  some registers only work with some instructions
- Hundreds of instructions in vector extensions

```
int *A;
...
A[i+2] += 100
```

```
# ebx = A, ecx = i
mov   eax, 100
add   [ebx + ecx*4 + 8], eax
```

= 5 Byte

ISA Examples: ARM

- RISC-style: load/store, fixed-size insns, three-adress code
- CISC-style: addressing modes (barrel shifter, pre/post increment/decrement)
- 15 Registers (Reg 15 is PC)
- Every instruction can be predicated (effect only on certain condition)

Addressing Modes:

```
RSB r9, r5, r5, LSL #3     ; r9 = r5 * 8 - r5 or r9 = r5 * 7
SUB r3, r9, r8, LSR #4     ; r3 = r9 - r8 / 16
ADD r9, r5, r5, LSL #3     ; r9 = r5 + r5 * 8 or r9 = r5 * 9
LDR r2, [r0, r1, LSL #2]   ; r2 = M[r0 + 4 * r1]
LDR r2, [r1], #4           ; r2 = M[r1], r1 = r1 + 4
```

Predication:

```
       CMP   r3,#0
       BEQ   skip               CMP   r3,#0
       ADD   r0,r1,r2           ADDNE r0,r1,r2
  skip:
```

Hardware Properties relevant to the Compiler

- In-order execution:
  - Compiler has to manage instruction level parallelism
  - Instruction scheduling very important
    direct influence on code latency
  - Cores have different functional units / pipes
    Not every instruction can go into each pipe
  - VLIW processors allow to pack instructions into bundles

- Out-of-order execution:
  - Processor schedules instructions to functional units dynamically
    Analyzes data dependences of instruction stream
  - Resolves false dependencies by register renaming:
    Internally, processor has way more regs than the ISA has
  - Instruction scheduling less important because done by CPU
  - List of instruction merely a "data structure" to communicate the data dependence graph to the processor
  - Avoiding spill code is more important (critical)

Out-of-order vs. In-order

- OOO costs more energy
- OOO allows for worse compilers
- OOO goes well along with speculation
- Modern OOO processors speculate over several loop iterations to keep the FUs busy
- Hard to imagine that something similar can be done statically
- Itanium (high-performance Intel VLIW CPU from the 2000s) is considered a failure
- Unclear, if same performance for less energy can be achieved with in-order arch and better compilers