Compilation Principle
编 译 原 理

第23讲：目标代码生成(3)

张献伟

xianweiz.github.io

DCS290, 6/16/2022

Machine Optimizations[机器相关优化]

- After performing IR optimizations
  - We need to further convert the optimized IR into the target language (e.g. assembly, machine code)
- Specific machines features are taken into account to produce code optimized for the particular architecture[考虑特定的架构特性]
  - E.g., specialized instructions, hardware pipeline abilities, register details
- Typical machine optimizations[典型的优化方案]
  - **Instruction selection and scheduling**: select insts to implement the operators in IR
  - **Register allocation**: map values to registers and manage
  - **Peephole optimization**: locally improve the target code

Instruction Selection[指令选取]

- To find an efficient mapping from the IR of a program to a target-specific assembly listing[IR到汇编的映射]
- Instruction selection is particularly important when targeting architectures with CISC (e.g., x86)
  - In these architectures there are typically several possible implementations of the same IR operation, each with different properties
  - e.g., on x86 an addition of one can be implemented by an *inc*, *add*, or *lea* instruction

x = y + z

```
MOV y,R0
ADD z,R0
MOV R0,x
```

a = a + 1

```
MOV a,R0
ADD #1,R0
MOV R0,a
```

```
MOV a,R0
INC R0
MOV R0,a
```

Instruction Cost[指令成本]

- Instruction cost = 1 + cost(source-mode) + cost(destination-mode)

Mode	Form	Address	Added Cost
Absolute	**M**	**M**	1
Register	**R**	**R**	0
Indexed	*c*(**R**)	*c*+*contents*(**R**)	1
Indirect register	***R**	*contents*(**R**)	0
Indirect indexed	***c*(**R**)	*contents*(*c*+*contents*(**R**))	1
Literal	**#*c***	N/A	1

- Examples

Instruction	Operation	Cost
`MOV R0,R1`	Store *content*(**R0**) into register **R1**	1
`MOV R0,M`	Store *content*(**R0**) into memory location **M**	2
`MOV M,R0`	Store *content*(**M**) into register **R0**	2
`MOV 4(R0),M`	Store *contents*(4+*contents*(**R0**)) into **M**	3
`MOV *4(R0),M`	Store *contents*(*contents*(4+*contents*(**R0**))) into **M**	3
`MOV #1,R0`	Store 1 into **R0**	2
`ADD 4(R0),*12(R1)`	Add *contents*(4+*contents*(**R0**)) to *contents*(12+*contents*(**R1**))	3

中山大學 SUN YAT-SEN UNIVERSITY

https://www.cs.fsu.edu/~engelen/courses/COP562107/Ch9a.pdf

NSCC GZ

Instruction Cost (cont.)

- Suppose we translate TAC x:=y+z to:
  - MOV *y*, R0
  - ADD *z*, R0
  - MOV R0, *x*

Mode	Form	Address	Added Cost
Absolute	**M**	**M**	1
Register	**R**	**R**	0
Indexed	*c*(**R**)	*c*+*contents*(**R**)	1
Indirect register	***R**	*contents*(**R**)	0
Indirect indexed	****c*(**R**)	*contents*(*c*+*contents*(**R**))	1
Literal	#*c*	N/A	1

- a := b + c

```
MOV b, R0
ADD c, R0
MOV R0, a
```

cost = 6

```
MOV b, a
ADD c, a
```

cost = 6

```
MOV *R1, *R0
ADD *R2, *R0
```

Assuming R0, R1 and R2 contain the addresses of a, b, and c

cost = 2

- a := a + 1

```
MOV a, R0
ADD #1, R0
MOV R0, a
```

cost = 6

```
ADD #1, a
```

cost = 3

```
INC a
```

cost = 2

Instruction Scheduling[指令调度]

- Some facts
  - Instructions take clock cycles to execute (latency)
  - Modern machines issue several operations per cycle (Out-of-Order execution)
  - Cannot use results until ready, can do something else
  - Execution time is order-dependent
- Goal: reorder the operations to minimize execution time
  - Minimize wasted cycles
  - Avoid spilling registers
  - Improve locality

```
A = x * y;
B = A + 1;
C = y;
```

→

```
A = x * y;
C = y;
B = A + 1;
```

(Now C=y; can execute while waiting for A=x*y;)

Register Allocation[寄存器分配]

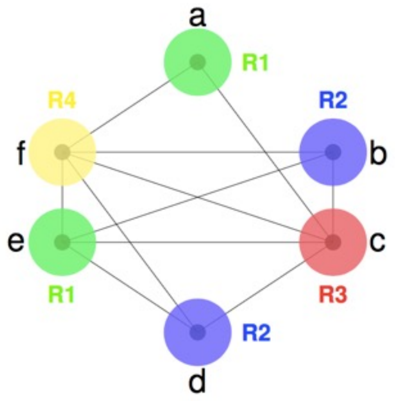
- In TAC, there are an unlimited number of variables
  - On a physical machine there are a small number of registers
- **Register allocation** is the process of assigning variables to registers and managing data transfer in and out of registers
  - How to assign variables to finitely many registers?
  - What to do when it can't be done?
  - How to do so efficiently?
- Using registers intelligently is a critical step in any compiler
  - Accesses to memory are costly, even with caches
  - A good register allocator can generate code orders of magnitude better than a bad register allocator

Register Allocation (cont.)

- Goals of register allocation
  - Keep frequently accessed variables in registers
  - Keep variables in registers only as long as they are live
- Local register allocation[局部]
  - Allocate registers basic block by basic block
  - Makes decisions on a per-block basis (hence ‘local’)
- Global register allocation[全局]
  - Makes global decisions about register allocation such that
    - Var to reg mappings remain consistent across blocks
    - Structure of CFG is taken into account on decisions
- Three well-known register allocation algorithms
  - Graph coloring allocator[图着色]
  - Linear scan allocator[线性扫描]
  - LP (Integer Linear Programming) allocator[整数线性规划]

Graph Coloring[图着色]

- Register interference graph (RIG)[相交图]
  - Each node represents a variable
  - An edge between two nodes V_1 and V_2 represents an interference in live ranges[活跃期/生存期]
- Based on RIG,
  - Two variables can be allocated in the same register if there is no edge between them[若无边相连，可使用同一寄存器]
  - Otherwise, they cannot be allocated in the same register
- Problem of register allocation maps to graph coloring
  - Once solved, *k* colors can be mapped back to *k* registers
  - If the graph is *k*-colorable, it's *k*-register-allocatable



Register Spilling[寄存器溢出]

- Determining whether a graph is *k*-colorable is NP-complete
  - Therefore, problem of *k*-register allocation is NP-complete
  - In practice: use heuristic polynomial algorithm that gives close to optimal allocations most of the time
  - Chaitin's graph coloring is a popular heuristic algorithm
    - E.g. most backends of GCC use Chaitin's algorithm
- What if *k*-register allocation does not exist?
  - Spill a variable to memory to reduce RIG and try again
  - Spilled var stays in memory and is not allocated a reg
- Spilling is slow
  - Placed into memory, loaded into register when needed, and written back to memory when no longer used

Peephole Optimization[窥孔优化]

- Optimization ways
  - Usual: produce good code through careful inst selection and register allocation
  - Alternative: generate naïve target code and then improve
- A simple but effective technique for locally improving the target code[很局部的优化，但可能带来性能的极大提升]
  - Done by examining a sliding window of target instructions (called **peephole**) and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever psbl
  - Can also be applied directly after IR generation to improve IR
- Example transformations
  - Redundant-instruction elimination
  - Flow-of-control optimizations
  - Algebraic simplifications
  - Use of machine idioms

```
    if a < b goto L1
    …
L1: goto L2
```

$\Downarrow$ Jump to jumps

```
    if a < b goto L2
    …
L1: goto L2
```

LLVM

- llc: LLVM static compiler
  - Input: .ll or .bc
  - Output: assembly language for a specified architecture
- End-user options

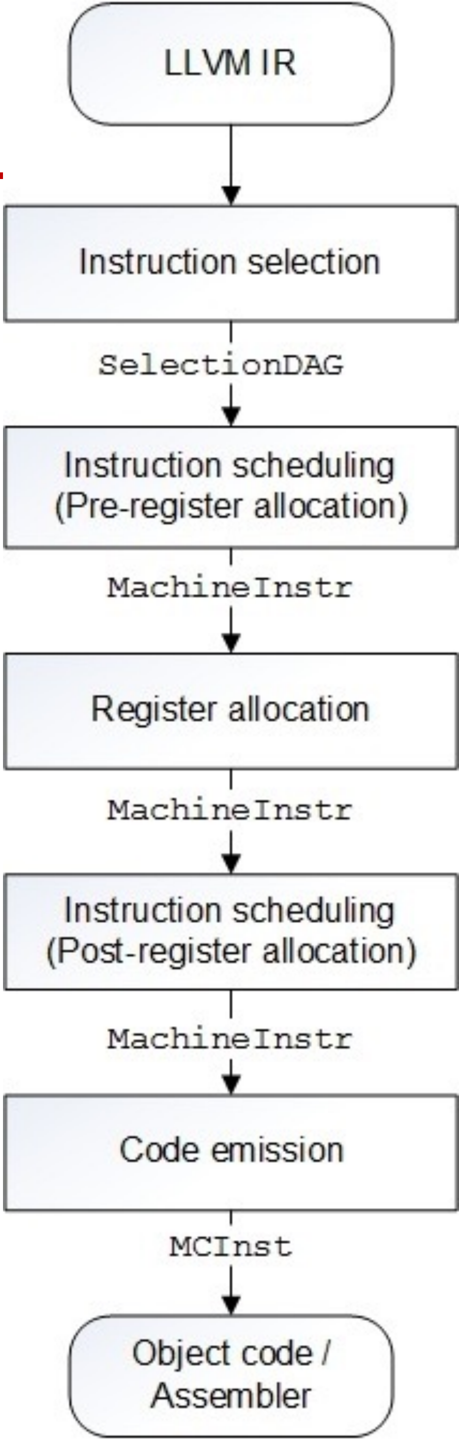
  -march=<arch>: e.g., x86

  -mcpu=<cpuname>: e.g., corei7-avx
- Tuning/Configuration Options

  --print-after-isel: print generated machine code after instruction selection (useful for debugging)

  --regalloc=<allocator>: specify the register allocator to use, basic/fast/greedy/pdqp

  --spiller=<spiller>: simple/local



中山大學 SUN YAT-SEN UNIVERSITY

https://www.llvm.org/docs/CommandGuide/llc.html

Optimizations[总结]

- Code can be optimized at different levels with various techniques
  - Peephole, local, loop, global
  - IR: local, global, common subexpression elimination, constant folding and propagation, …
  - Target: instruction, register, peephole, …
- Interactions between the various optimization techniques
  - Some transformations may expose possibilities for others
  - One opt. may obscure or remove possibilities for others
- Affect of compiler opts are intertwined and hard to separate
  - Finding optimal opt combinations is in itself research
  - Compilers package opts that typically go together into levels (e.g -O1, -O2, -O3)

Pahrump, Nevada 8/4/2017 5:37pm

The **END** is Near