CSC2/455 Software Analysis and Improvement
Instruction Scheduling/Register Allocation

Sreepathi Pai

URCS

February 27, 2019
Outline

Review

Instruction Scheduling

Global Register Allocation

Postscript
Outline

Review

Instruction Scheduling

Global Register Allocation

Postscript
Optimizations

- Part I: Analysis
  - Iterative Dataflow Analysis
  - SSA Form
- Part II-A: Optimization
  - Dead Code Elimination
  - Partial Redundancy Elimination
- Part III: Code Generation
  - Instruction Selection (not today)
  - Instruction Scheduling
  - Register Allocation
Outline

Review

Instruction Scheduling

Global Register Allocation

Postscript
The need for instruction scheduling

- Code is in 3-address form
- Instructions have been selected
  - e.g. $a / 4$ becomes $\text{shr } a, 2$ (shift-right) or $\text{sar } a, 2$ (shift-arithmetic right)
  - usually through tree rewrites (see Chapter 11 in Cooper and Torczon)
- What (linear) order should we output instructions?
A modern CPU pipeline

Source: Intel 64 and IA-32 Architectures Optimization Manual
Pipeline Details Known to Compiler

- Number of functional units and their types
  - Which instructions can be issued together
- Latencies of operations
  - Note some operations can have variable latencies (e.g. memory operations)
- Forwarding latencies
  - Delays when sending values from one functional unit to another
  - Or from one instruction to another
- And other sundry architecture details
  - See the architecture manuals for processor of interest
Two metrics of interest

- Throughput: Instructions completed per cycle
  - Higher is better
- Latency: Total cycles for execution
  - Lower is better
Basic block to data dependence graph

\[ a = x + y \]
\[ b = v / w \]
\[ c = a * b \]
\[ d = z << 2 \]
\[ e = c - d \]
Schedule

Assume the following delays:

- Add, Shifts, Subtraction: 1 cycle
- Multiplication: 2 cycles
- Division: 3 cycles

<table>
<thead>
<tr>
<th>Cycle</th>
<th>ALU1</th>
<th>ALU2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(a = x + y)</td>
<td>(b = v / w)</td>
</tr>
<tr>
<td>1</td>
<td>(d = z \ll 1)</td>
<td>(busy)</td>
</tr>
<tr>
<td>2</td>
<td>(idle)</td>
<td>(busy)</td>
</tr>
<tr>
<td>3</td>
<td>(c = a \ast b)</td>
<td>(idle)</td>
</tr>
<tr>
<td>4</td>
<td>(busy)</td>
<td>(idle)</td>
</tr>
<tr>
<td>5</td>
<td>(e = c - d)</td>
<td>(idle)</td>
</tr>
</tbody>
</table>

What determines the total time of the path?
Critical Path

- The path that takes the longest to execute
  - Equivalently, has zero *slack*
- Delaying operations on critical path will increase total time
List scheduling

Goal: Schedule instructions on every cycle (i.e. build a table like in previous figure)

- Overall structure:
  - Mark operations whose predecessors have completed as ready
  - Pick operations that are ready
  - Schedule them **in some order** if resources available
  - Proceed to next cycle
  - Repeat until all operations are scheduled

(Figure 12.3 in Cooper and Torczon)
Always prioritize instructions on critical path.

- Can be hard to achieve
- Variable delays
- Multiple functional units
- Other architecture-specific constraints
Other Complications

- Small basic block sizes
  - Trace scheduling (in Part II-B)
- Long dependence chains with little parallelism
  - Software Pipelining (in Part II-B)
Outline

Review

Instruction Scheduling

Global Register Allocation

Postscript
The Memory Hierarchy (on CPUs)

- Registers (on chip, few tens to low hundreds)
- L1 cache (tens of KB)
- L2 cache (tens of MB)
- DRAM (tens of GB)
  - off-chip
Numbers every programmer should know

- Register access: 1 cycle
- L1 cache: less than 10 cycles
- L2 cache: less than 100 cycles
- DRAM: hundreds of cycles

Bottomline: Placing frequently used variables in registers can improve performance
The Problem of Global Register Allocation

- Code is in SSA form (with $\phi$-functions removed)
- SSA form uses infinite registers (each temporary is a “register”)
- Must map these logical registers to physical machine registers
Problem and Setup

Must answer two main questions:

- How many physical registers are needed? \((k\) is the number of physical registers\)
  - \(< k\), fewer than available – everybody gets a register!
  - \(> k\), more than available – results in spill code
  - Spilled registers are stored in memory and reloaded later

- Which physical register is assigned to which variable?
Live Ranges

A live range is a set of all definitions and uses of the same variable such that:

- If a use $u$ for variable $i$ is in $LR_i$, then all definitions $d$ that reach $u$ are also in $LR_i$.
- If a definition $d$ for variable $i$ is in $LR_i$, then all uses $u$ reached by $d$ are also in $LR_i$. 
Live range example

- **Live ranges**
  - $LR_a = \{a_0\}$
  - $LR_b = \{b_0\}$
  - $LR_c = \{c_0\}$
  - $LR_d = \{d_0, d_1, d_2\}$
Interference Graphs

- Two live ranges "interfere" if they are both live at an operation
  - Implies that they must be in different registers
- Represented by an interference graph
  - Nodes are live ranges
  - An (undirected) edge between nodes $n$ and $m$ indicates interference
Register Allocation as Graph Coloring

Given an interference graph, the minimal number of physical registers required is equal to the *chromatic number* of the graph (due to Chaitin).

- Chromatic number is the minimum number of colors required such adjacent nodes have different colors
- NP-complete problem
Four Color Example
Parting thoughts

What order should register allocation and instruction scheduling be performed?
Outline

Review

Instruction Scheduling

Global Register Allocation

Postscript
References

- Chapter 12 of Cooper and Torczon (Instruction Scheduling)
- Chapter 13 of Cooper and Torczon (Register Allocation)