Introduction to Optimization, Instruction Selection and Scheduling, and Register Allocation
Traditional Three-pass Compiler

Code Improvement (or Optimization)

- Analyzes IR and rewrites (or transforms) IR
- Primary goal is to reduce running time of the compiled code
  - May also improve space, power consumption, ...
- Must preserve “meaning” of the code
  - Measured by values of named variables
  - A course (or two) unto itself
Modern optimizers are structured as a series of passes
The Optimizer (or Middle End)

Typical Transformations

• Discover & propagate some constant value
• Move a computation to a less frequently executed place
• Specialize some computation based on context
• Discover a redundant computation & remove it
• Remove useless or unreachable code
• Encode an idiom in some particularly efficient form
The Role of the Optimizer

- The compiler can implement a procedure in many ways.
- The optimizer tries to find an implementation that is "better"
  - Speed, code size, data space, ...

To accomplish this, it

- Analyze code to derive knowledge about run-time behavior
  - General term is "static analysis"
- Uses that knowledge in an attempt to improve the code
  - Literally hundreds of transformations have been proposed
  - Large amount of overlap between them

Nothing "optimal" about optimization
Redundancy Elimination as an Example

An expression $x+y$ is redundant iff

- along every path from the procedure’s entry, it has been evaluated and its constituent subexpressions ($x$ & $y$) have not been re-defined.
Traditional Three-pass Compiler

- Instruction Selection
- Register Allocation
- Instruction Scheduling
Instruction Selection: The Problem

Writing a compiler is a lot of work

- Would like to reuse components whenever possible
- Would like to automate construction of components

![Diagram showing Front End, Middle End, Back End, and Infrastructure with an arrow pointing to Instruction Selection]
Definitions

Instruction selection

- Mapping $IR$ into assembly code
- Assumes a fixed storage mapping & code shape
- Combining operations, using address modes

Instruction scheduling

- Reordering operations to hide latencies
- Assumes a fixed program (set of operations)
- Changes demand for registers

Register allocation

- Deciding which values will reside in registers
- Changes the storage mapping, may add false sharing
- Concerns about placement of data & memory operations
The Problem

Modern computers (still) have many ways to do anything

Consider register-to-register copy in ILOC

- Obvious operation is \( i2i \ r_i \Rightarrow r_j \)
- Many others exist

<table>
<thead>
<tr>
<th></th>
<th>( r_i,0 \Rightarrow r_j )</th>
<th>( r_i,1 \Rightarrow r_j )</th>
<th>( r_i,0 \Rightarrow r_j )</th>
</tr>
</thead>
<tbody>
<tr>
<td>addI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>subI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>multI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>divI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>orI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xorI</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Human would ignore all of these
- Algorithm must look at all of them & find low-cost encoding
  → Take context into account
The Goal

Want to automate generation of instruction selectors

Machine description can also help with scheduling & allocation
The Big Picture

Need pattern matching techniques
• Must produce good code  (some metric for good)
• Must run quickly

A treewalk code generator runs quickly
How good was the code?

Tree Treewalk Code Desired Code

 IDENT \( <a, \text{ARP}, 4> \) loadI \( 4 \Rightarrow r_5 \)

 IDENT \( <b, \text{ARP}, 8> \) loadAO \( r_{\text{arp}}, r_5 \Rightarrow r_6 \)

loadI \( 8 \Rightarrow r_7 \)

loadAO \( r_{\text{arp}}, r_7 \Rightarrow r_8 \)

mult \( r_6, r_8 \Rightarrow r_9 \)

loadAI \( r_{\text{arp}}, 4 \Rightarrow r_5 \)

loadAI \( r_{\text{arp}}, 8 \Rightarrow r_6 \)

mult \( r_5, r_6 \Rightarrow r_7 \)
The Big Picture

Need pattern matching techniques
- Must produce good code
- Must run quickly

A treewalk code generator runs quickly
How good was the code?

Tree

 IDENT <a,ARP,4>

 IDENT <b,ARP,8>

 Treewalk Code

 loadI 4 \Rightarrow r_5
 loadAO r_{arp},r_5 \Rightarrow r_6
 loadI 8 \Rightarrow r_7
 loadAO r_{arp},r_7 \Rightarrow r_8
 mult r_6,r_8 \Rightarrow r_9

 Desired Code

 loadAI r_{arp},4 \Rightarrow r_5
 loadAI r_{arp},8 \Rightarrow r_6
 mult r_5,r_6 \Rightarrow r_7
The Big Picture

Need pattern matching techniques

- Must produce good code (some metric for good)
- Must run quickly

A treewalk code generator runs quickly

How good was the code?

<table>
<thead>
<tr>
<th>Tree</th>
<th>Treewalk Code</th>
<th>Desired Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>×</td>
<td>loadI 4  ⇒ r₅</td>
<td>loadAI r_{arp},4  ⇒ r₅</td>
</tr>
<tr>
<td></td>
<td>loadAO r_{arp},r₅ ⇒ r₆</td>
<td>multI r₅,2  ⇒ r₇</td>
</tr>
<tr>
<td>IDENT</td>
<td>loadI 2  ⇒ r₇</td>
<td></td>
</tr>
<tr>
<td>&lt;a,ARP,4&gt;</td>
<td>mult r₆,r₇  ⇒ r₈</td>
<td></td>
</tr>
<tr>
<td>NUMBER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;2&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Big Picture

Need pattern matching techniques

• Must produce good code
• Must run quickly

A treewalk code generator runs quickly

How good was the code?

Tree

Treewalk Code

Desired Code

Must combine these

This is a nonlocal problem
The Big Picture

Need pattern matching techniques

- Must produce good code
- Must run quickly

A treewalk code generator runs quickly

How good was the code?

<table>
<thead>
<tr>
<th>Tree</th>
<th>Treewalk Code</th>
<th>Desired Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\times$</td>
<td>$\text{loadI} \quad \text{@G} \Rightarrow r_5$</td>
<td>$\text{loadI} \quad 4 \Rightarrow r_5$</td>
</tr>
<tr>
<td></td>
<td>$\text{loadI} \quad 4 \Rightarrow r_6$</td>
<td>$\text{loadAI} \quad r_5,\text{@G} \Rightarrow r_6$</td>
</tr>
<tr>
<td>IDENT $&lt;c,\text{@G},4&gt;$</td>
<td>$\text{loadAO} \quad r_5,r_6 \Rightarrow r_7$</td>
<td>$\text{loadAI} \quad r_5,\text{@H} \Rightarrow r_7$</td>
</tr>
<tr>
<td>IDENT $&lt;d,\text{@H},4&gt;$</td>
<td>$\text{loadI} \quad \text{@H} \Rightarrow r_7$</td>
<td>$\text{loadAI} \quad r_5,\text{@H} \Rightarrow r_7$</td>
</tr>
<tr>
<td></td>
<td>$\text{loadI} \quad 4 \Rightarrow r_8$</td>
<td>$\text{mult} \quad r_6,r_7 \Rightarrow r_8$</td>
</tr>
<tr>
<td></td>
<td>$\text{loadAO} \quad r_8,r_9 \Rightarrow r_{10}$</td>
<td></td>
</tr>
</tbody>
</table>
The Big Picture

Need pattern matching techniques

- Must produce good code *(some metric for good)*
- Must run quickly

A treewalk code generator can meet the second criteria

How did it do on the first?

Tree

- IDENT <c, @G, 4>
- IDENT <d, @H, 4>
- Common offset

Treewalk Code

- loadI @G ⇒ r_5
- loadI 4 ⇒ r_6
- loadAO r_5, r_6 ⇒ r_7
- loadI @H ⇒ r_7
- loadI 4 ⇒ r_8
- loadAO r_8, r_9 ⇒ r_10
- mult r_7, r_10 ⇒ r_11

Desired Code

- loadI 4 ⇒ r_5
- loadAI r_5, @G ⇒ r_6
- loadAI r_5, @H ⇒ r_7
- mult r_6, r_7 ⇒ r_8

Again, a nonlocal problem
How do we perform this kind of matching?

Tree-oriented IR suggests pattern matching on trees
• Tree-patterns as input, matcher as output
• Each pattern maps to a target-machine instruction sequence
• Use dynamic programming or bottom-up rewrite systems

Linear IR suggests using some sort of string matching
• Strings as input, matcher as output
• Each string maps to a target-machine instruction sequence
• Use text matching or peephole matching

In practice, both work well; matchers are quite different
Definitions

Instruction selection
- Mapping IR into assembly code
- Assumes a fixed storage mapping & code shape
- Combining operations, using address modes

Instruction scheduling
- Reordering operations to hide latencies
- Assumes a fixed program (set of operations)
- Changes demand for registers

Register allocation
- Deciding which values will reside in registers
- Changes the storage mapping, may add false sharing
- Concerns about placement of data & memory operations
What Makes Code Run Fast?

• Many operations have non-zero latencies
• Modern machines can issue several operations per cycle
• Execution time is order-dependent

Assumed latencies (conservative)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>load</td>
<td>3</td>
</tr>
<tr>
<td>store</td>
<td>3</td>
</tr>
<tr>
<td>loadI</td>
<td>1</td>
</tr>
<tr>
<td>add</td>
<td>1</td>
</tr>
<tr>
<td>mult</td>
<td>2</td>
</tr>
<tr>
<td>fadd</td>
<td>1</td>
</tr>
<tr>
<td>fmult</td>
<td>2</td>
</tr>
<tr>
<td>shift</td>
<td>1</td>
</tr>
<tr>
<td>branch</td>
<td>0 to 8</td>
</tr>
</tbody>
</table>

• Loads & stores may or may not block
  > Non-blocking ⇒ fill those issue slots
• Branch costs vary with path taken
• Scheduler should hide the latencies
Example

\[ w \leftarrow w \cdot 2 \cdot x \cdot y \cdot z \]

<table>
<thead>
<tr>
<th>Cycles</th>
<th>Simple schedule</th>
<th>Cycles</th>
<th>Schedule loads early</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>loadAl r0,@w (\Rightarrow) r1</td>
<td>1</td>
<td>loadAl r0,@w (\Rightarrow) r1</td>
</tr>
<tr>
<td>4</td>
<td>add r1,r1 (\Rightarrow) r1</td>
<td>2</td>
<td>loadAl r0,@x (\Rightarrow) r2</td>
</tr>
<tr>
<td>5</td>
<td>loadAl r0,@x (\Rightarrow) r2</td>
<td>3</td>
<td>loadAl r0,@y (\Rightarrow) r3</td>
</tr>
<tr>
<td>8</td>
<td>mult r1,r2 (\Rightarrow) r1</td>
<td>4</td>
<td>add r1,r1 (\Rightarrow) r1</td>
</tr>
<tr>
<td>9</td>
<td>loadAl r0,@y (\Rightarrow) r2</td>
<td>5</td>
<td>mult r1,r2 (\Rightarrow) r1</td>
</tr>
<tr>
<td>12</td>
<td>mult r1,r2 (\Rightarrow) r1</td>
<td>6</td>
<td>loadAl r0,@z (\Rightarrow) r2</td>
</tr>
<tr>
<td>13</td>
<td>loadAl r0,@z (\Rightarrow) r2</td>
<td>7</td>
<td>mult r1,r3 (\Rightarrow) r1</td>
</tr>
<tr>
<td>16</td>
<td>mult r1,r2 (\Rightarrow) r1</td>
<td>9</td>
<td>mult r1,r2 (\Rightarrow) r1</td>
</tr>
<tr>
<td>18</td>
<td>storeAl r1 (\Rightarrow) r0,@w</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>r1 is free</td>
<td>11</td>
<td>storeAl r1 (\Rightarrow) r0,@w</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14</td>
<td>r1 is free</td>
</tr>
</tbody>
</table>

2 registers, 20 cycles

3 registers, 13 cycles

Reordering operations for speed is called instruction scheduling
Instruction Scheduling (Engineer’s View)

The Problem
Given a code fragment for some target machine and the latencies for each individual operation, reorder the operations to minimize execution time.

The Concept
Machine description

slow code

Scheduler

fast code

The task
- Produce correct code
- Minimize wasted cycles
- Avoid spilling registers
- Operate efficiently
Instruction Scheduling (The Abstract View)

To capture properties of the code, build a dependence graph $G$

- Nodes $n \in G$ are operations with $\text{type}(n)$ and $\text{delay}(n)$
- An edge $e = (n_1, n_2) \in G$ if & only if $n_2$ uses the result of $n_1$

The Code

<table>
<thead>
<tr>
<th>Operation</th>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadAl</td>
<td>r0,@w</td>
<td>r1</td>
</tr>
<tr>
<td>add</td>
<td>r1,r1</td>
<td>r1</td>
</tr>
<tr>
<td>loadAl</td>
<td>r0,@x</td>
<td>r2</td>
</tr>
<tr>
<td>mult</td>
<td>r1,r2</td>
<td>r1</td>
</tr>
<tr>
<td>loadAl</td>
<td>r0,@y</td>
<td>r2</td>
</tr>
<tr>
<td>mult</td>
<td>r1,r2</td>
<td>r1</td>
</tr>
<tr>
<td>loadAl</td>
<td>r0,@z</td>
<td>r2</td>
</tr>
<tr>
<td>mult</td>
<td>r1,r2</td>
<td>r1</td>
</tr>
<tr>
<td>storeAl</td>
<td>r1</td>
<td>r0,@w</td>
</tr>
</tbody>
</table>

The Dependence Graph
Instruction Scheduling  (Definitions)

A **correct schedule** $S$ maps each $n \in N$ into a non-negative integer representing its cycle number, and

1. $S(n) \geq 0$, for all $n \in N$, obviously
2. If $(n_1, n_2) \in E$, $S(n_1) + \text{delay}(n_1) \leq S(n_2)$
3. For each type $t$, there are no more operations of type $t$ in any cycle than the target machine can issue

The **length** of a schedule $S$, denoted $L(S)$, is

$$L(S) = \max_{n \in N} (S(n) + \text{delay}(n))$$

The goal is to find the shortest possible correct schedule. $S$ is **time-optimal** if $L(S) \leq L(S_j)$, for all other schedules $S_j$

A schedule might also be optimal in terms of registers, power, or space....
Instruction Scheduling  (What’s so difficult?)

Critical Points
- All operands must be available
- Multiple operations can be ready
- Moving operations can lengthen register lifetimes
- Placing uses near definitions can shorten register lifetimes
- Operands can have multiple predecessors

Together, these issues make scheduling hard (NP-Complete)

Local scheduling is the simple case
- Restricted to straight-line code
- Consistent and predictable latencies
Instruction Scheduling

The big picture
1. Build a dependence graph, $P$
2. Compute a *priority function* over the nodes in $P$
3. Use list scheduling to construct a schedule, one cycle at a time
   a. Use a queue of operations that are ready
   b. At each cycle
      I. Choose a ready operation and schedule it
      II. Update the ready queue

Local list scheduling
• The dominant algorithm for twenty years
• A greedy, heuristic, local technique
Local List Scheduling

Cycle ← 1
Ready ← roots of P
Active ← Ø

while (Ready ∪ Active ≠ Ø)
  if (Ready ≠ Ø) then
    remove an op from Ready
    S(op) ← Cycle
    Active ← Active ∪ op
  Cycle ← Cycle + 1

for each op ∈ Active
  if (S(op) + delay(op) ≤ Cycle) then
    remove op from Active
    for each successor s of op in P
      if (s is ready) then
        Ready ← Ready ∪ s

Removal in priority order
op has completed execution
If successor’s operands are ready, put it on Ready
Scheduling Example

1. Build the dependence graph

The Code

a: loadAI r0,@w ⇒ r1
b: add r1,r1 ⇒ r1
c: loadAI r0,@x ⇒ r2
d: mult r1,r2 ⇒ r1
e: loadAI r0,@y ⇒ r2
f: mult r1,r2 ⇒ r1
g: loadAI r0,@z ⇒ r2
h: mult r1,r2 ⇒ r1
i: storeAI r1 ⇒ r0,@w

The Dependence Graph
Scheduling Example

1. Build the dependence graph
2. Determine priorities: longest latency-weighted path

The Code

- a: loadAI r0,@w ↔ r1
- b: add r1,r1 ↔ r1
- c: loadAI r0,@x ↔ r2
- d: mult r1,r2 ↔ r1
- e: loadAI r0,@y ↔ r2
- f: mult r1,r2 ↔ r1
- g: loadAI r0,@z ↔ r2
- h: mult r1,r2 ↔ r1
- i: storeAI r1 ↔ r0,@w

The Dependence Graph
Scheduling Example

1. Build the dependence graph
2. Determine priorities: longest latency-weighted path
3. Perform list scheduling

a: loadAl r0,@w \(\Rightarrow r1\)
b: add r1,r1 \(\Rightarrow r1\)
c: loadAl r0,@x \(\Rightarrow r2\)
d: mult r1,r2 \(\Rightarrow r1\)
e: loadAl r0,@y \(\Rightarrow r3\)
f: mult r1,r3 \(\Rightarrow r1\)
g: loadAl r0,@z \(\Rightarrow r2\)
h: mult r1,r2 \(\Rightarrow r1\)
i: storeAl r1 \(\Rightarrow r0,@w\)

The Code

The Dependence Graph

New register name used
Register Allocation

Part of the compiler’s back end

Critical properties

- Produce **correct** code that uses \( k \) (or fewer) registers
- Minimize added loads and stores
- Minimize space used to hold *spilled values*
- Operate efficiently
  \[ O(n), O(n \log_2 n), \text{maybe } O(n^2), \text{but not } O(2^n) \]
Register Allocation using Graph-Coloring

The big picture

At each point in the code

1. Determine which values will reside in registers
2. Select a register for each such value

The goal is an allocation that “minimizes” running time

Most modern, global allocators use a graph-coloring paradigm

- Build a “conflict graph” or “interference graph”
- Find a $k$-coloring for the graph, or change the code to a nearby problem that it can $k$-color

Optimal global allocation is NP-Complete, under almost any assumptions.
Register Allocation using Graph Coloring

Graph coloring paradigm \( (Chaitin)\)

1. Build an interference graph \( G_I \) for the procedure
2. (try to) construct a \( k \)-coloring
   - Minimal coloring is NP-Complete
   - Spill placement becomes a critical issue
3. Map colors onto physical registers
Graph Coloring (A Background Digression)

The problem

A graph $G$ is said to be $k$-colorable iff the nodes can be labeled with integers 1... $k$ so that no edge in $G$ connects two nodes with the same label.

Examples

Each color can be mapped to a distinct physical register.
Building the Interference Graph

What is an “interference”? (or conflict)

• Two values *interfere* if there exists an operation where both are simultaneously live

• If $x$ and $y$ interfere, they cannot occupy the same register

To compute interferences, we must know where values are “live”

The interference graph, $G_I$

• Nodes in $G_I$ represent values, or live ranges

• Edges in $G_I$ represent individual interferences
  \[ \Rightarrow \text{For } x, y \in G_I, \langle x, y \rangle \in \text{iff } x \text{ and } y \text{ interfere} \]

• A $k$-coloring of $G_I$ can be mapped into an allocation to $k$ registers
Observation on Coloring for Register Allocation

• Suppose you have \( k \) registers—look for a \( k \) coloring

• Any vertex \( n \) that has fewer than \( k \) neighbors in the interference graph \((n^o < k)\) can always be colored!
  → Pick any color not used by its neighbors — there must be one
Observation on Coloring for Register Allocation

• Pick any vertex \( n \) such that \( n^\circ < k \) and put it on the stack

• Remove that vertex and all edges incident from the interference graph
  → This may make some new nodes have fewer than \( k \) neighbors

• At the end, if some vertex \( n \) still has \( k \) or more neighbors, then spill the live range associated with \( n \)

• Otherwise successively pop vertices off the stack and color them in the lowest color not used by some neighbor
Graph Coloring in Practice

3 Registers

Stack
Graph Coloring in Practice

3 Registers

Stack

\begin{itemize}
\item 1
\end{itemize}
Graph Coloring in Practice

3 Registers

Stack

2
1

3
4
5
Graph Coloring in Practice

3 Registers

Stack

4
2
1

3 5
Graph Coloring in Practice

3 Registers

<table>
<thead>
<tr>
<th>Stack</th>
<th>Colors:</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1:</td>
</tr>
<tr>
<td>3</td>
<td>2:</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3:</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Graph Coloring in Practice

3 Registers

Stack

Colors:
1:  
2:  
3:  

5
Graph Coloring in Practice

3 Registers

Stack

Colors:
1: ○
2: ●
3: ■
Graph Coloring in Practice

3 Registers

Stack

Colors:
1:
2:
3:
Graph Coloring in Practice

3 Registers

Stack

Colors:
1: ✑
2: 🔴
3: 🔵
Graph Coloring in Practice

3 Registers

Stack

Colors:
1:
2:
3: