Introduction to Optimization, Instruction Selection and Scheduling, and Register Allocation
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

- Analyzes the code to derive knowledge about run-time behavior
  - Data-flow analysis, pointer disambiguation, ...
  - 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

- Proofs of optimality assume restrictive & unrealistic conditions
Redundancy Elimination as an Example

An expression $x+y$ is redundant if and only if, along every path from the procedure’s entry, it has been evaluated, and its constituent subexpressions ($x$ & $y$) have not been re-defined.

If the compiler can prove that an expression is redundant
• It can preserve the results of earlier evaluations
• It can replace the current evaluation with a reference

Two pieces to the problem
• Proving that $x+y$ is redundant
• Rewriting the code to eliminate the redundant evaluation

One technique for accomplishing both is called value numbering
Handling Larger Scopes

Extended Basic Blocks

- Initialize table for $b_i$ with table from $b_{i-1}$
- With single-assignment naming, can use scoped hash table

The Plan:
- Process $b_1$, $b_2$, $b_4$
- Pop two levels
- Process $b_3$ relative to $b_1$
- Start clean with $b_5$
- Start clean with $b_6$

Using a scoped table makes doing the full tree of EBBs that share a common header efficient.

Otherwise, it is complex
Handling Larger Scopes

To go further, we must deal with merge points

• Our simple naming scheme falls apart in $b_4$
• We need more powerful analysis tools
• Naming scheme becomes SSA

This requires global data-flow analysis

“Compile-time reasoning about the run-time flow of values”

1 Build a model of control-flow
2 Pose questions as sets of simultaneous equations
3 Solve the equations
4 Use solution to transform the code

Examples: LIVE, REACHES, AVAIL
Traditional Three-pass Compiler

- Instruction Selection
- Register Allocation
- Instruction Scheduling
Writing a compiler is a lot of work

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

- Front end construction is largely automated
- Middle is largely hand crafted
- (Parts of) back end can be automated
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>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>addI ( r_i,0 \Rightarrow r_j )</td>
<td>subI ( r_i,0 \Rightarrow r_j )</td>
</tr>
<tr>
<td>multI ( r_i,1 \Rightarrow r_j )</td>
<td>divI ( r_i,1 \Rightarrow r_j )</td>
</tr>
<tr>
<td>orI ( r_i,0 \Rightarrow r_j )</td>
<td>xorI ( r_i,0 \Rightarrow r_j )</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

\( \text{(busy functional unit?)} \)
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
  
  \((\text{some metric for good})\)

- 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>IDENT &lt;a,ARP,4&gt;</td>
<td>loadI 4 (\Rightarrow r_5)</td>
<td>loadAI (r_{\text{arp}}, 4 \Rightarrow r_5)</td>
</tr>
<tr>
<td>IDENT &lt;b,ARP,8&gt;</td>
<td>loadAO (r_{\text{arp}}, r_5 \Rightarrow r_6)</td>
<td>loadAI (r_{\text{arp}}, 8 \Rightarrow r_6)</td>
</tr>
<tr>
<td>(\times)</td>
<td>loadI 8 (\Rightarrow r_7)</td>
<td>mult (r_5, r_6 \Rightarrow r_7)</td>
</tr>
<tr>
<td></td>
<td>loadAO (r_{\text{arp}}, r_7 \Rightarrow r_8)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mult (r_6, r_8 \Rightarrow r_9)</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?

Pretty easy to fix. See 1st digression in Ch. 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?

<table>
<thead>
<tr>
<th>Tree</th>
<th>Treewalk Code</th>
<th>Desired Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDENT &lt;a,ARP,4&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NUMBER &lt;2&gt;</td>
<td>loadI 4 \Rightarrow r_5</td>
<td>loadAI r_{arp},4 \Rightarrow r_5</td>
</tr>
<tr>
<td></td>
<td>loadAO r_{arp},r_5 \Rightarrow r_6</td>
<td>multI r_5,2 \Rightarrow r_7</td>
</tr>
<tr>
<td></td>
<td>loadI 2 \Rightarrow r_7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mult r_6,r_7 \Rightarrow r_8</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 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
  (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><code>×</code></td>
<td><code>loadI @G =&gt; r_5</code></td>
<td><code>loadI 4 =&gt; r_5</code></td>
</tr>
<tr>
<td><code>IDENT &lt;ɛ, @G, 4&gt;</code></td>
<td><code>loadI 4 =&gt; r_6</code></td>
<td><code>loadAI r_5, @G =&gt; r_6</code></td>
</tr>
<tr>
<td><code>IDENT &lt;d, @H, 4&gt;</code></td>
<td><code>loadAO r_5, r_6 =&gt; r_7</code></td>
<td><code>loadAI r_5, @H =&gt; r_7</code></td>
</tr>
<tr>
<td><code>loadI @H =&gt; r_7</code></td>
<td><code>loadAO r_8, r_9 =&gt; r_10</code></td>
<td><code>mult r_6, r_7 =&gt; r_8</code></td>
</tr>
<tr>
<td><code>loadI 4 =&gt; r_8</code></td>
<td><code>loadAO r_7, r_10 =&gt; r_11</code></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
<\epsilon, @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 (and has been since the 60’s)

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

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

<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>11</td>
<td>storeAl r1 (\Rightarrow r0,@w)</td>
</tr>
<tr>
<td>21</td>
<td>r1 is free</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

![Diagram showing the concept of instruction scheduling]

The task
- Produce correct code
- Minimize wasted cycles
- Avoid spilling registers
- Operate efficiently
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
```

```
the_diagram
```

The Code

The Dependence Graph
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 $L(S) \leq L(S_1)$, for all other schedules $S_1$.

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 \textit{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 \textit{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 $\leftarrow 1$
Ready $\leftarrow$ roots of $P$
Active $\leftarrow \emptyset$

while (Ready $\cup$ Active $\neq \emptyset$)
    if (Ready $\neq \emptyset$) then
        remove an op from Ready
        $S(op) \leftarrow$ Cycle
        Active $\leftarrow$ Active $\cup$ op
    Cycle $\leftarrow$ Cycle + 1
    for each op $\in$ Active
        if ($S(op) + \text{delay}(op) \leq \text{Cycle}$) then
            remove op from Active
            for each successor s of op in P
                if (s is ready) then
                    Ready $\leftarrow$ Ready $\cup$ 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

a → b → c → d → e → f → g → h → i
Scheduling Example

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

The Code

<table>
<thead>
<tr>
<th>a:</th>
<th>loadAl</th>
<th>r0,@w</th>
<th>➞ r1</th>
</tr>
</thead>
<tbody>
<tr>
<td>b:</td>
<td>add</td>
<td>r1,r1</td>
<td>➞ r1</td>
</tr>
<tr>
<td>c:</td>
<td>loadAl</td>
<td>r0,@x</td>
<td>➞ r2</td>
</tr>
<tr>
<td>d:</td>
<td>mult</td>
<td>r1,r2</td>
<td>➞ r1</td>
</tr>
<tr>
<td>e:</td>
<td>loadAl</td>
<td>r0,@y</td>
<td>➞ r2</td>
</tr>
<tr>
<td>f:</td>
<td>mult</td>
<td>r1,r2</td>
<td>➞ r1</td>
</tr>
<tr>
<td>g:</td>
<td>loadAl</td>
<td>r0,@z</td>
<td>➞ r2</td>
</tr>
<tr>
<td>h:</td>
<td>mult</td>
<td>r1,r2</td>
<td>➞ r1</td>
</tr>
<tr>
<td>i:</td>
<td>storeAl</td>
<td>r1</td>
<td>➞ r0,@w</td>
</tr>
</tbody>
</table>

The Dependence Graph
Scheduling Example

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

The Code

1) a: loadAl r0,@w ⇒ r1
2) c: loadAl r0,@x ⇒ r2
3) e: loadAl r0,@y ⇒ r3
4) b: add r1,r1 ⇒ r1
5) d: mult r1,r2 ⇒ r1
6) g: loadAl r0,@z ⇒ r2
7) f: mult r1,r3 ⇒ r1
9) h: mult r1,r2 ⇒ r1
11) i: storeAl r1 ⇒ r0,@w

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
  \[ O(n), O(n \log_2 n), \text{maybe } O(n^2), \text{but not } O(2^n) \]
Global Register Allocation

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
Global Register Allocation

What’s harder across multiple blocks?

• Could replace a load with a move
• Good assignment would obviate the move
• Must build a control-flow graph to understand inter-block flow
• Can spend an inordinate amount of time adjusting the allocation

This is an assignment problem, not an allocation problem!
Global Register Allocation

A more complex scenario

- Block with multiple predecessors in the control-flow graph
- Must get the “right” values in the “right” registers in each predecessor
- In a loop, a block can be its own predecessors

This adds tremendous complications
Global Register Allocation

Taking a global approach
• Abandon the distinction between local & global
• Make systematic use of registers or memory
• Adopt a general scheme to approximate a good allocation

Graph coloring paradigm \((Lavrov \text{ (later) Chaitin})\)
1. Build an interference graph \(G_I\) for the procedure
   → Computing LIVE is harder than in the local case
   → \(G_I\) is not an interval graph
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 $1\ldots k$ so that no edge in $G$ connects two nodes with the same label.

Examples

Each color can be mapped to a distinct physical register

2-colorable

3-colorable
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
  → For $x, y \in G_I$, $\langle x, y \rangle \in$ iff $x$ and $y$ 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 ($n^\circ < k$) can **always** be colored!
  → Pick any color not used by its neighbors — there must be one

• Ideas behind Chaitin’s algorithm:
  → 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
Chaitin’s Algorithm in Practice

3 Registers

Stack
Chaitin’s Algorithm in Practice

3 Registers

Stack
Chaitin’s Algorithm in Practice

3 Registers

Stack

2
1

3 4 5
Chaitin’s Algorithm in Practice

3 Registers

Stack

1
2
4
Chaitin’s Algorithm in Practice

3 Registers

---

Stack

5
3
4
2
1

Colors:

1: 
2: 
3: 

---
Chaitin’s Algorithm in Practice

3 Registers

Stack

Colors:
1: 
2: 
3: 

5
Chaitin’s Algorithm in Practice

3 Registers

Stack

Colors:
1: 
2: 
3: 

4
2
1

3
5

3

University of Delaware
Chaitin’s Algorithm in Practice

3 Registers

Stack

Colors:

1:  
2:  
3:  

Nodes:

3 - 4 - 5
Chaitin’s Algorithm in Practice

3 Registers

Stack

Colors:
1:
2:
3:
Chaitin’s Algorithm in Practice

3 Registers

Stack

Colors:
1: 
2: 
3: 

Diagram showing nodes 1, 2, 3, 4, and 5 connected by lines, with colors assigned to each node.
Picking a Spill Candidate

When $\forall \ n \in G_I, n^\circ \geq k$, simplify must pick a spill candidate

Chaitin’s heuristic

- Minimize spill cost $\div$ current degree
- If $LR_x$ has a negative spill cost, spill it pre-emptively
  $\rightarrow$ Cheaper to spill it than to keep it in a register
- If $LR_x$ has an infinite spill cost, it cannot be spilled
  $\rightarrow$ No value dies between its definition & its use
  $\rightarrow$ No more than $k$ definitions since last value died (safety valve)

Spill cost is weighted cost of loads & stores needed to spill $x$

Bernstein et al. Suggest repeating simplify, select, & spill with several different spill choice heuristics & keeping the best