Syllabus

Front End
- Scanning
- Top-down Parsing (LL)
- Bottom-up Parsing (LR)

Intermediate Representation
- Operational Semantics
- Lowering

Optimizations
- Dataflow Analysis
- Loop Optimizations

Code Generation
- Register Allocation

mid-term exam
• Dataflow framework
  – Loop optimizations
  – Strong liveness analysis → dead code elimination
AGENDA

• An simple assembly language
• The need for register allocation
• Register allocation algorithms
  – Graph coloring
  – Linear scan (tentative)
ASSEMBLY LANGUAGE:

ASM
Syntax

\[ n \in \textbf{Num} \quad \text{Numerals} \]
\[ l \in \textbf{Num} \quad \text{Labels} \]
\[ r \in \textbf{Reg} \quad \text{Registers} \]

\[ V \rightarrow n \mid r \]
\[ R \rightarrow V \ Op \ V \]
\[ Op \rightarrow - \mid + \mid * \mid / \mid = \mid \leq \mid \ll \mid \gg \mid \ldots \]
\[ C \rightarrow l: \text{skip} \]
\[ [ l: r := R \mid l: r := M[V] \mid l: M[V] := V \]
\[ l: \text{Goto } l' \]
\[ l: \text{IfZ } x \text{ Goto } l' \mid l: \text{IfNZ } x \text{ Goto } l' \]
\[ l: \text{Push } V \mid l: \text{Pop } r \mid \text{Call } \text{foo} \]
\[ \text{ASM} \rightarrow C^+ \]
REGISTER ALLOCATION
MOTIVATION
Registers

• Most machines have a set of registers, dedicated memory locations that
  – can be accessed quickly,
  – can have computations performed on them, and
  – exist in small quantity

• Using registers intelligently is a critical step in any compiler
  – A good register allocator can generate code orders of magnitude better than a bad register allocator
Register allocation

- In IL, there are an unlimited number of variables
- On a physical machine there are a small number of registers:
  - x86 has four general-purpose registers and a number of specialized registers
  - MIPS has twenty-four general-purpose registers and eight special-purpose registers
- **Register allocation** is the process of assigning variables to registers and managing data transfer in and out of registers
Challenges in register allocation

• Registers are scarce
  – Often substantially more IR variables than registers
  – Need to find a way to reuse registers whenever possible

• Registers are complicated
  – x86: Each register made of several smaller registers; can't use a register and its constituent registers at the same time
  – x86: Certain instructions must store their results in specific registers; can't store values there if you want to use those instructions
  – MIPS: Some registers reserved for the assembler or operating system
  – Most architectures: Some registers must be preserved across function calls
NAIVE REGISTER ALLOCATION
Simple approach

- **Straightforward solution:**
  - Allocate each variable in activation record
  - At each instruction:
    - bring values needed into registers (**rematerialization**),
    - perform operation,
    - then store result to memory

```plaintext
x := y + z
```

```plaintext
mov 16(%ebp), %eax
mov 20(%ebp), %ebx
add %ebx, %eax
mov %eax, 24(%ebp)
```

- **Problem:** program execution very inefficient – moving data back and forth between memory and registers
REUSING REGISTERS
Find a register allocation

<table>
<thead>
<tr>
<th>variable</th>
<th>register</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>?</td>
</tr>
<tr>
<td>b</td>
<td>?</td>
</tr>
<tr>
<td>c</td>
<td>?</td>
</tr>
</tbody>
</table>

register

eax

EBX

b := a + 2

c := b * b

b := c + 1

Ret b * a
Is this a valid allocation?

<table>
<thead>
<tr>
<th>variable</th>
<th>register</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>eax</td>
</tr>
<tr>
<td>b</td>
<td>ebx</td>
</tr>
<tr>
<td>c</td>
<td>eax</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
b & := a + 2 \\
c & := b \times b \\
b & := c + 1 \\
\text{Ret } b & \times a
\end{align*}
\]

\[
\begin{align*}
ebx & := eax + 2 \\
eax & := ebx \times ebx \\
ebx & := eax + 1 \\
\text{Ret } ebx & \times eax
\end{align*}
\]

Overwrites previous value of ‘a’ also stored in eax
Is this a valid allocation?

<table>
<thead>
<tr>
<th>variable</th>
<th>register</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>ebx</td>
</tr>
<tr>
<td>b</td>
<td>eax</td>
</tr>
<tr>
<td>c</td>
<td>eax</td>
</tr>
</tbody>
</table>

```
b := a + 2

earning := ebx + 2

c := b * b

earning := earning * earning

b := c + 1

earning := earning + 1

Ret b * a

Ret earning * ebx
```

Value of ‘c’ stored in eax is not needed anymore so reuse it for ‘b’
REGISTER INTERERENCE GRAPH
Main idea

• For every command $c$, we have $\text{live}[c]$
  – Set of temporaries live out of $c$

• Two variables *interfere* if they appear in the same $\text{live}[c]$ of any command $c$
  – *Cannot be allocated to the same register*

• Conversely, if two variables do not interfere with each other, they can be assigned the same register
  – We say they have disjoint live ranges

• How to assign registers to variables?
Interference graph

• **Nodes** of the graph = variables
• **Edges** connect variables that interfere with one another
• Nodes will be assigned a **color** corresponding to the register assigned to the variable
• Two colors can’t be next to one another in the graph
Liveness analysis

\[
b := a + 2\\
c := b \times b\\
b := c + 1\\
Ret b \times a
\]
Liveness analysis

\[
\begin{align*}
b &:= a + 2 \\
c &:= b \times b \\
b &:= c + 1 \\
\{b, a\} \\
\text{Ret } b \times a
\end{align*}
\]
Liveness analysis

\begin{align*}
b & := a + 2 \\
c & := b \ast b \\
b & := c + 1 \\
\text{Ret } b \ast a
\end{align*}
Liveness analysis

\[
\begin{align*}
  b &:= a + 2 \\
  c &:= b \times b \\
  b &:= c + 1 \\
  \text{Ret } b \times a
\end{align*}
\]
Liveness analysis

\[ b := a + 2 \]
\[ c := b \times b \]
\[ b := c + 1 \]
\[ \text{Ret } b \times a \]
Interference graph

b := a + 2
{a}

c := b * b
{b, a}

b := c + 1
{a, c}

Ret b * a
{b, a}
b := a + 2
{a}
c := b * b
{b, a}
b := c + 1
{a, c}
Ret b * a
{b, a}
What's New in GWT 2.5?

The latest release of GWT, version 2.5, includes the new features and functionality listed below. See the 2.5 Release Notes for bug fixes and other changes.

New Features

- Super Dev Mode (experimental)
- Elemental (experimental)
- New compiler optimizations
- Updated ARIA support
- UIBinder Enhancements
- Validation Enhancements

Getting Started

New Features

Super Dev Mode (experimental)

Super Dev Mode is an experimental replacement for Development Mode. For more, see Introducing Super Dev Mode.

Elemental (experimental)

Elemental is an experimental new library for fast, lightweight, and “to the metal” web programming in GWT. It’s intended for developers who are comfortable working with the browser API’s that JavaScript programmers use. For more information, please see the Introducing Elemental article.

New compiler optimizations

- The GWT compiler can optionally use the Closure compiler to provide additional JavaScript optimizations. The Closure compiler has a collection of Javascript optimizations that can benefit code size, including a graph-coloring-based variable allocator, comprehensive JavaScript function and variable inlining, cross-module code motion, statement fusing, name shadowing and many more. However, this makes the GWT compiler slower, so it’s not enabled by default. Simply add the -XenableClosureCompiler to the list of compiler flags to enable optimization.
- Large projects that use Code Splitting and have many split points can take advantage of Fragment Merging. The GWT compiler can automatically merge fragments to reduce the size of the “leftover” fragment. For more, please see the article, Fragment Merging in the GWT Compiler.
New compiler optimizations

- The GWT compiler can optionally use the Closure compiler to provide additional JavaScript optimizations that can benefit code size, including a graph-coloring-based variable allocator, comprehension, name shadowing and many more. However, this makes the GWT compiler slow if compiler flags to enable optimization.
GRAPH COLORING
Graph coloring

• Register allocation is equivalent to graph-coloring, which is NP-hard if there are at least three registers

• No good polynomial-time algorithms (or even good approximations!) are known for this problem

• We have to be content with a heuristic that is good enough for RIGs that arise in practice
Coloring by simplification [Kempe 1879]

• How to find a $K$-coloring of a graph
• Observation:
  – Suppose we are trying to $K$-color a graph and find a node with fewer than $K$ edges
  – If we delete this node from the graph and color what remains, we can find a color for this node if we add it back in
  – Reason: fewer than $K$ neighbors $\Rightarrow$ some color must be left over
Coloring by simplification [Kempe 1879]

• How to find a $K$-coloring of a graph
• Phase 1: Simplify
  – Repeatedly simplify graph
  – When a variable (i.e., graph node) is removed, push it on a stack
• Phase 2: Select
  – Unwind stack and reconstruct the graph as follows:
    • Pop variable from the stack
    • Add it back to the graph
    • Color the node for that variable with a color that it doesn’t interfere with
COLORING EXAMPLE
Coloring for $K=2$

Stack:

```
  a
 /\  \\
 b / \c
|  \  |
|   \ |
|    d|
|    |
|    e|
```
Coloring for $K=2$

stack:

- c
Coloring for $K=2$

<table>
<thead>
<tr>
<th>color</th>
<th>register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>eax</td>
</tr>
<tr>
<td></td>
<td>ebx</td>
</tr>
</tbody>
</table>

stack:
- e
- c
Coloring for $K=2$

stack:
- a
- e
- c
Coloring for $K=2$

- **color**
  - eax: blue
  - ebx: green

- **register**
  - eax
  - ebx

- **stack:**
  - b
  - a
  - e
  - c
Coloring for $K=2$

<table>
<thead>
<tr>
<th>color</th>
<th>register</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>eax</td>
</tr>
<tr>
<td></td>
<td>ebx</td>
</tr>
</tbody>
</table>

stack:
- d
- b
- a
- e
- c
Coloring for $K=2$

stack:
  b
  a
e  c

color  register

eax

ebx
Coloring for $K=2$

stack:

a
e
c

d
b

e

a

eax

ebx
Coloring for $K=2$

Stack:

- $e$
- $c$
Coloring for $K=2$

stack:

$c$
Coloring for $K=2$

- eax
- ebx
SPILLING
Failure of heuristic

• If the graph cannot be colored, it will eventually be simplified to graph in which every node has at least $K$ neighbors

• Sometimes, the graph is still $K$-colorable!

• Finding a $K$-coloring in all situations is an NP-complete problem
  – We will have to approximate to make register allocators fast enough
Example where Kempe’s approach fails

color     register

stack:
Example where Kempe’s approach fails

color | register
--- | ---
eax

stack:
d
But RIG is actually 2-colorable

color  register
 eax
 ebx
Example of non 2-colorable graph

Some graphs can’t be colored in $K$ colors:

A 3-clique
Chaitin’s algorithm [CC’82]

• Choose and remove an arbitrary node, marking it “troublesome”
  – Use heuristics to choose which one:
    – When adding node back in, it may be possible to find a valid color
    – Otherwise, we have to spill that node
Chaitin’s algorithm [CC’82]

• Choose and remove an arbitrary node, marking it “troublesome”
  – Use heuristics to choose which one:
    - spill priority = \( \frac{uo + 10 \, ui}{\text{deg}} \)
    - \( uo = \#\text{use+defs outside of loop} \)
    - \( ui = \#\text{use+defs inside of loop} \)
  – When adding node back in, it may be possible to find a valid color
  – Otherwise, we have to spill that node (lowest priority)
Spilling

• Phase 3: Spill
  – once all nodes have $K$ or more neighbors, pick a node for spilling
    • There are many heuristics that can be used to pick a node
    • Try to pick node not used much, not in inner loop
    • Assign its storage to activation record
      – Remove it from graph and mark it as potential spill (technically, push on a spill stack)
  
• We can now repeat phases 1-2 without this node
POTENTIAL SPILL EXAMPLE
Potential spill example

• Color the following graph

<table>
<thead>
<tr>
<th>color</th>
<th>register</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td></td>
</tr>
<tr>
<td>BX</td>
<td></td>
</tr>
<tr>
<td>CX</td>
<td></td>
</tr>
</tbody>
</table>

potential spill:  
(color) stack:
Potential spill example

• Pick a for potential spill
Potential spill example

• Simplify d
Potential spill example

• Simplify e

<table>
<thead>
<tr>
<th>color</th>
<th>register</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td></td>
</tr>
<tr>
<td>BX</td>
<td></td>
</tr>
<tr>
<td>CX</td>
<td></td>
</tr>
</tbody>
</table>

Potential spill:
- a

(stack):
- d

(color):
- AX
- BX
- CX
Potential spill example

• Simplify b
Potential spill example

• Simplify c
Potential spill example

• Simplify f
Potential spill example

• Select color for f
Potential spill example

• Select color for c
Potential spill example

• Select color for b
Potential spill example

• Select color for e
Potential spill example

- Select color for d
Potential spill example

• Select color for a
Potential spill example

• Done – no actual spill
Chaitin’s algorithm [CC’82]

- **Phase 1:** Simplify
- **Phase 2:** Select
  - Unwind stack and reconstruct the graph as follows:
    - Pop (non-spill) variable from the stack
    - Add it back to the graph
    - Color the node for that variable with a color that it doesn’t interfere with its neighbors
  - Unwind spill stack
    - Pop spill variable
    - If there is an available color add back to graph
    - Otherwise mark variable as actual spill
- **Phase 3:** Spill
  - If all nodes have $K$ or more neighbors, pick a “heavy” node for spilling and add to potential spill stack
- **Phase 4:** Rewrite
  - Rewrite code for actual spill variables
  - Recompute liveness information
  - Repeat phases 1-3
Chaitin’s algorithm [CC’82]

- Simplify
- Mark potential spills
- Select colors and detect actual spills
- Liveness analysis
- Rewrite code to implement actual spills

Any potential spill done → Simplify

Found light node → Mark potential spills

Don’t spill same variable twice

Any actual spill done → Rewrite code to implement actual spills
ACTUAL SPILL EXAMPLE
Actual spill example

• Apply Chaitin’s algorithm for the following program
  – The set of registers is AX, BX, CX
  – Upon any non-deterministic choice, choose by lexicographic order

```plaintext
c := e;
a := a + e;
d := a + c;
d := a + b;
d := d + b;
// live = {d}
```
Step 1: compute liveness

• Let’s compute liveness information

```plaintext
{a, b, e}
c := e;
{a, b, c, e}
a := a + e;
{a, b, c}
d := a + c;
{a, b}
d := a + b;
{b, d}
d := d + b;
// live = {d}
```
Step 2: draw RIG

- Simplify d

### Diagram:

#### Color-Register Mapping:
- **Color:** AX, BX, CX
- **Register:** AX, BX, CX

#### Graph:
-Nodes: a, b, c, d, e
- Edges: a-b, a-c, b-d, c-e

#### Potential Spill:
- Stack:

### Table:

<table>
<thead>
<tr>
<th>Potential Spill</th>
<th>(Color) Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Potentially spill b

- #use+def(a)=4, #use+def(b)=2, #use+def(c)=2, #use+def(d)=4, #use+def(e)=2
- Priorities: p(a)=4/4, p(b)=2/3, p(c)=2/3, p(e)=2/3
- (potentially) spill b

```c
c := e;
a := a + e;
d := a + c;
d := a + b;
d := d + b;
// live = {d}
```

Color register

- AX
- BX
- CX

Potential spill: d

Stack: d
Simplify

• Simplify a
Simplify

• Simplify c
Simplify

• Simplify e
Attempt to color nodes on stack

• Select color for e
Attempt to color nodes on stack

• Select color for c

<table>
<thead>
<tr>
<th>color</th>
<th>register</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td></td>
</tr>
<tr>
<td>BX</td>
<td></td>
</tr>
<tr>
<td>CX</td>
<td></td>
</tr>
</tbody>
</table>

Potential spill:
- b

(color) stack:
- c
- a
- d
Attempt to color nodes on stack

• Select color for a
Attempt to color nodes on stack

- Select color for d
b cannot be colored

- Cannot color b – actual spill
Actual spill: rich assembly

• Suppose assembly allows commands of the form \( x := y + M[\text{offset}] \)

• Rewrite program for spilled variable \( b \)
  – Choose location on frame: \( b_{\text{loc}} \)

\[
\begin{align*}
c & := e; \\
a & := a + e; \\
d & := a + c; \\
d & := a + b; \\
d & := d + b; \\
\text{// live} & = \{d\}
\end{align*}
\]

\[
\begin{align*}
c & := e; \\
a & := a + e; \\
d & := a + c; \\
d & := a + M[b_{\text{loc}}]; \\
d & := d + M[b_{\text{loc}}]; \\
\text{// live} & = \{d\}
\end{align*}
\]
Actual spill: basic assembly

- Rewrite program for spilled variable b
  - Choose location on frame: b_loc
  - Use temporaries for reading from frame: b1, b2
  - Now attempt to color all variables, including temporaries
    - If unable don’t spill temporaries, choose other variables to spill, otherwise can go into infinite spill-color loop

```plaintext
const

\[
c := e;
\]
\[
a := a + e;
\]
\[
d := a + c;
\]
\[
d := a + b;
\]
\[
d := d + b;
\]
\[
// live = \{d\}
```

```plaintext
const

\[
c := e;
\]
\[
a := a + e;
\]
\[
d := a + c;
\]
\[
b1 := M[b_loc];
\]
\[
d := a + b1;
\]
\[
b2 := M[b_loc];
\]
\[
d := d + b2;
\]
\[
// live = \{d\}
```
Rewriting rules

• Assume we want to spill variable x

• Case 1: a command that reads x:
  
  \[ \text{c = l: z := a + x} \]
  
  \[ \Rightarrow x_1 := M[x_{\text{loc}}]; z := a + x_1 \]

• Case 2: a command that writes to x:
  
  \[ \text{c = l: x := a + b} \]
  
  \[ \Rightarrow x_1 := a + b; M[x_{\text{loc}}] := x_1 \]
Compute liveness for new program

```plaintext
{a, e}
c := e;
{a, c, e}
a := a + e;
{a, c}
d := a + c;
{a}
b1 := M[b_loc];
{a, b1}
d := a + b1;
{d}
b2 := M[b_loc];
{d, b2}
d := d + b2;
// live = {d}
```
Attempts to color

- Simplify b₁, b₂, d, a, c, e
- Select colors

<table>
<thead>
<tr>
<th>color</th>
<th>register</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td></td>
</tr>
<tr>
<td>BX</td>
<td></td>
</tr>
<tr>
<td>CX</td>
<td></td>
</tr>
</tbody>
</table>
Attempt to color

- Simplify $b_1$, $b_2$, $d$, $a$, $c$, $e$
- Select colors

<table>
<thead>
<tr>
<th>color</th>
<th>register</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td></td>
</tr>
<tr>
<td>BX</td>
<td></td>
</tr>
<tr>
<td>CX</td>
<td></td>
</tr>
</tbody>
</table>

Diagram:
- $b_1$ connected to $a$
- $b_2$ connected to $d$
- $c$ connected to $e$
Emit code based on colors

c := e;
a := a + e;
d := a + c;
b1 := M[b_loc];
d := a + b1;
b2 := M[b_loc];
d := d + b2;

BX := AX;
CX := CX + AX;
AX := CX + BX;
BX := M[b_loc];
AX := AX + BX;
Pathological cases

• In general, spilling a variable may not help
  – E.g., if there is only one use
• On next iteration choose a different variable to spill
REGISTER CONSTRAINTS
Handling precolored nodes

• Some variables are pre-assigned to registers
  – e.g.: mul on x86/pentium
    • Uses eax; defines eax, edx
  – e.g.: call on x86/pentium
    • Defines (trashes) caller-save registers eax, ecx, edx

• To properly allocate registers, treat these register uses as special temporary variables and enter into interference graph as precolored nodes
Handling precolored nodes

• **Simplify.** Never remove a pre-colored node – it already has a color, i.e., it is a given register

• **Select.** Once simplified graph has all colored nodes, add other nodes back in and color them using precolored nodes as starting point
OPTIMIZING MOVES
Optimizing move instructions

- Code generation produces a lot of extra MOVE instructions
  \[ t5 := t9 \]
- If we can assign \( t5 \) and \( t9 \) to same register, we can get rid of the MOVE
  – Effectively, copy propagation at the register allocation level
- **Idea:** if \( t5 \) and \( t9 \) are not connected in inference graph, **coalesce** them into a single variable; the move will be redundant
RIG with MOVE edges

• Add a second type of edges – MOVE edges
Coalescing

- **Problem**: coalescing nodes can make a graph un-colorable
  - Conservative coalescing heuristic

- What should we do?
Conservative coalescing

• Coalesce MOVE edge \((a,b)\) if it does not affect colorability

• **Definition:** A node is heavy if its degree is \(\geq K\) and light otherwise
Briggs criterion

- Coalesce only if the merged node $ab$ has $< K$ heavy neighbors
- Reason:
  - Simplify will first remove all light neighbors
  - $ab$ will then be adjacent to $< K$ neighbors
  - Simplify can remove $ab$
George criterion

• Coalesce only if all heavy neighbors of $a$ are also neighbors of $b$

• Reason:
  – Simplify can remove all light neighbors of $a$
  – Remaining heavy neighbors of $a$ are neighbors of $b$
    so if $b$ is colorable then so is $a$
  – The light neighbors of $a$ are light $\Rightarrow$ colorable
Coalescing criterion example

• By the Briggs criterion?
  – #(heavy-neighbors(a,e) < K

• By the George criterion?
  – All heavy neighbors of a are also neighbors of e
Coalescing criterion example

• By the Briggs criterion? NO
  – \( \#(\text{heavy-neighbors}(a,e)) < K \)

• By the George criterion? YES
  – All heavy neighbors of a are also neighbors of e
Simplify, coalesce and freeze

• **Phase 1: Simplify**
  – **Step 1 (simplify):** simplify as much as possible without removing nodes that are the source or destination of a move (MOVE-related nodes)
  – **Step 2 (coalesce):** coalesce a MOVE-related edge provided low-degree node results
  – **Step 3 (freeze):** if neither steps 1 or 2 apply, freeze a MOVE instruction: low-degree nodes involved are marked not MOVE-related (remove MOVE edge) and try step 1 again
  – **Step 4 (spill):** if all nodes are heavy select a candidate for spilling using a priority function and move to potential spill stack

• **Phase 2: Select**
  – Unwind stack and reconstruct the graph as follows:
    • Pop variable from the stack
    • Add it back to the graph
    • Color the node node with a color that it doesn’t interfere with
  – Unwind potential spill stack and attempt to color node – if unable mark corresponding variable as actual spill

• **Phase 4: Rewrite**
  – Allocate position in frame for spilled variable \( v \)
  – On each usage of \( v \) load to \( v_i \) from frame
Simplify, coalesce and freeze

**Simplify:** recursively remove non-MOVE nodes with $<K$ neighbors, pushing them onto stack

**Coalesce:** conservatively merge unconstrained MOVE-related nodes via Briggs/George Criterion

**Freeze:** give up coalescing on some low-degree MOVE-related node
Overall algorithm

1. Simplify, freeze and coalesce
2. Mark potential spills
3. Select colors and detect actual spills
4. Liveness analysis
5. Rewrite code to implement actual spills
COALESCE EXAMPLE
Coalesce example

• Apply Chaitin’s algorithm for the following program
  – The set of registers is AX, BX, CX
  – Use the Briggs criterion for conservative coalescing
  – Upon any non-deterministic choice, choose by lexicographic order

```
a := e;
d := a + c;
d := d + b;
// live = {d}
```
Step 1: compute liveness

\[
\begin{align*}
\{b, c, e\} \\
a &:= e; \\
\{a, b, c\} \\
d &:= a + c; \\
\{b, d\} \\
d &:= d + b; \\
// \text{live} = \{d\}
\end{align*}
\]
Attempt to color

- Simplify d
Coalesce step

- Cannot simplify a or e because they are MOVE-related
- Coalesce (a,e)
Simplify

• Simplify ae
Simplify

• Simplify b
Simplify

• Simplify c
Select

• Color c
Select

- Color b

<table>
<thead>
<tr>
<th>color</th>
<th>register</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td></td>
</tr>
<tr>
<td>BX</td>
<td></td>
</tr>
<tr>
<td>CX</td>
<td></td>
</tr>
</tbody>
</table>

potential spill:
stack:
(color)
b
ae
d
Select

• Color ae
Select

• Color d
Select

color    register
AX
BX
CX

potential spill:
(stack):

(ax)
c
(b)
 Emit code

```
a := e;
d := a + c;
d := d + b;
```

```
CX := CX;
AX := CX + AX;
AX := AX + BX;
```
FREEZE EXAMPLE
Step 1: compute liveness

```plaintext
{b, e}
c := e + b;
{a, c}
e := a;
{a, c}
b := a + c;
{b}
d := b;
{b, d}
d := b + d;
// live = {d}
```
Attempt to color

- Simplify d
Attempt to color

• Can we coalesce a and e according to Briggs?
• Let’s try
Unsuccessful attempt to color

- No, node becomes heavy
- Undo coalesce and try something else
Freeze a
Simplify a stack:

d
potential spill:

color

AX

BX

(stack):

d
Simplify c

potential spill:
stack: a d

AX
BX
color
register
Simplify b

potential spill:

stack:
c
a
d

color

register

AX

BX
Simplify e

color | register
---|---
 AX | BX

potential spill:
b
c
a
d
(color)
Select color for e

<table>
<thead>
<tr>
<th>color</th>
<th>register</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td></td>
</tr>
<tr>
<td>BX</td>
<td></td>
</tr>
</tbody>
</table>

potential spill:
stack:
  e
  b
  c
  a
  d

(color)
Select color for b

color | register
--- | ---
 AX | BX

potential spill:
(stack):
 b
 c
 a
 d

color
Select color for c

<table>
<thead>
<tr>
<th>color</th>
<th>register</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td></td>
</tr>
<tr>
<td>BX</td>
<td></td>
</tr>
</tbody>
</table>

potential spill:
(stack: c
  a
  d)
Select color for a

<table>
<thead>
<tr>
<th>color</th>
<th>register</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td></td>
</tr>
<tr>
<td>BX</td>
<td></td>
</tr>
</tbody>
</table>

potential spill: (color) stack: a d
Select color for d

<table>
<thead>
<tr>
<th>color</th>
<th>register</th>
</tr>
</thead>
<tbody>
<tr>
<td>AX</td>
<td></td>
</tr>
<tr>
<td>BX</td>
<td></td>
</tr>
</tbody>
</table>

potential spill: (color) stack: d
Select color for d

(color) stack:

potential spill:

AX
BX

color     register

d

b

e
c

a
LINEAR SCAN REGISTER

ALLOCATION
Live ranges and live intervals

- The **live range** for a variable is the set of program points at which the variable is alive.
- The **live interval** for a variable is the smallest sub-range of the IR code containing all of a variable’s live ranges.
  - A property of the IR code, **not** the CFG.
  - Less precise than live ranges, but simpler to work with.
Live ranges and live intervals

```
    d  e  f  g
    a  b  c
e := d + a
f := b + c
f := f + b
IfZ e Goto _L0
d := e + f
d := e - f
Goto _L1
_L0:
d := e - f
_L1:
g := d
Ret g
```
Register allocation with live ranges

• Given the live intervals for all the variables in the program, we can allocate registers using a simple greedy algorithm
• Idea: Track which registers are free at each point
• When a live interval begins, give that variable a free register
• When a live interval ends, the register is once again free
• We can't always fit everything into a register (spill)
Example

Free registers

R₀  R₁  R₂  R₃

a  b  c  d  e  f  g
Example

Free registers

R₀  R₁  R₂  R₃
Example

Free registers

\[ R_0 \quad R_1 \quad R_2 \quad R_3 \]
Example

Free registers

R₀  R₁  R₂  R₃
Example

Free registers

\[
\begin{array}{cccc}
R_0 & R_1 & R_2 & R_3 \\
\end{array}
\]
Example

Free registers

\[ \begin{array}{cccc}
R_0 & R_1 & R_2 & R_3 \\
\end{array} \]
Example

Free registers

R₀  R₁  R₂  R₃
Example

Free registers

<table>
<thead>
<tr>
<th>R_0</th>
<th>R_1</th>
<th>R_2</th>
<th>R_3</th>
</tr>
</thead>
</table>

a b c d e f g
Example

Free registers

\[ \begin{array}{cccc}
R_0 & R_1 & R_2 & R_3 \\
\end{array} \]
Example

Free registers

R₀  R₁  R₂  R₃
Example

Free registers
Example

Free registers

R₀  R₁  R₂  R₃
Another example

Free registers

R₀  R₁  R₂
Another example

Free registers

R₀  R₁  R₂
Another example

Free registers

R₀  R₁  R₂
Another example

Free registers

R₀ | R₁ | R₂
Another example

Which register should we use?
Another example

Which variable should we spill?

One with longer interval
Another example

Free registers

R₀  R₁  R₂
Another example

Free registers

R₀  R₁  R₂
Another example

Free registers

R₀  R₁  R₂
Another example

Free registers

R_0  R_1  R_2
Another example

Free registers

<table>
<thead>
<tr>
<th>R₀</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
</table>

abcdefg
Another example

Free registers

\[ \begin{array}{c}
R_0 & R_1 & R_2 \\
\end{array} \]
Another example

Free registers

R₀  R₁  R₂
Another example

Free registers

<table>
<thead>
<tr>
<th>R₀</th>
<th>R₁</th>
<th>R₂</th>
</tr>
</thead>
</table>

abcdefg
Another example

Free registers

\[ \begin{array}{c|c|c}
R_0 & R_1 & R_2 \\
\end{array} \]
Linear scan register allocation

• A relatively new algorithm (Polleto and Sarkar)

• Advantages:
  – Very efficient (after computing live intervals, runs in linear time)
  – Produces good code in many instances
  – Allocation step works in one pass; can generate code during iteration
  – Often used in JIT compilers like Java HotSpot

• Disadvantage
  – Imprecise due to use of live *intervals* rather than live *ranges*
Comparison to RA by coloring

Fig. 5. Overhead of graph coloring and linear scan as a function of the number of simultaneously live variables for programs of type (a).
RA recap

• Graph-coloring
• Linear scan
Program Analysis and Verification course next semester

- How to verify programs do what they are supposed to do
- How to verify they don’t loop forever
- Automatically, without ever running the program
- Possibly teach code synthesis
Are you looking to do some cool research? I am looking for enthusiastic students for M.Sc.

Come talk to me
Good luck with the exam and the final project!