Compiling Techniques

Lecture 19: Register Allocation
Overview

ChocoPy → FrontEnd → Middle End → Backend → RISC-V

AST → IR → IR → ASM

Errors
Instruction Selection

ChocoPy → Instruction Selection → Register Allocation → Instruction Scheduling → ASM → RISC-V

AST → IR → IR

Errors
Introduction

This lecture:

- Local Allocation - spill code
- Global Allocation based on graph colouring
- Techniques to reduce spill code
Register Allocation

- Physical machines have limited number of registers
- Scheduling and selection typically assume infinite registers
- Register allocation and assignment from infinite to $k$ registers

- Produce correct code that uses $k$ (or fewer) registers
- Minimise added loads and stores
- Minimise space used to hold spilled values
- Operate efficiently:
  - $O(n)$, $O(n^2)$, but not $O(2^n)$
Register Allocation: Definitions

**Allocation vs Assignment:**
- *Allocation* is deciding which values to keep in registers
- *Assignment* is choosing specific registers for values

**Liveness**
A value is live from its definition to its last use.

**Inference**
Two values cannot be mapped to the same register wherever they are both live. Such values are said to **interfere**.

**Live Range**
The live range of a value is the set of statements at which it is live. A live range may be conservatively overestimated (e.g., just begin → end)
Register Allocation: Definitions

**Spilling**
Spilling saves a value from a register to memory. That register is then free - Another value often loaded.
Requires $F$ registers to be reserved.

**Clean and dirty values**
A previously spilled value is **clean** if not changed since last spill. Otherwise it is **dirty**.
A clean value can be spilled without a new store instruction.
Local Register Allocation

Register allocation only on basic block.

Let MAXLIVE be the maximum, over each instruction i in the block, of the number of values (pseudo-registers) live at i.

- If MAXLIVE \( \leq k \), allocation should be easy
- If MAXLIVE \( \leq k \), no need to reserve F registers for spilling
- If MAXLIVE \( > k \), some values must be spilled to memory
- If MAXLIVE \( > k \), need to reserve F registers for spilling

Two main forms:
- Top down
- Bottom up
Local Register Allocation: MAXLIVE

loadI 1028 → rₐ // rₐ ← 1028
load rₐ → rₐ // rₐ ← MEM(rₐ)
mult rₐ, rₐ → rₙ // rₙ ← 1028 · y
load x → rₙ // rₙ ← x
sub rₙ, rₐ → rₚ // rₚ ← x–y
load z → rₚ // rₚ ← z
mult rₚ, rₚ → rₗ // rₗ ← z · (x–y)
sub rₗ, rₚ → rₗ // rₗ ← z · (x–y)–1028 · y
store rₗ → rₐ // MEM(rₐ) ← z · (x–y)–1028 · y
Local Register Allocation: MAXLIVE

Example MAXLIVE computation

Live registers

```
loadI 1028 => ra  // ra
load ra => rb   // ra rb
mult ra, rb => rc // ra rb rc
load x => rd   // ra rb rc rd
sub rd, rb => re // ra rc re
load z => rf   // ra rc re rf
mult re, rf => rg // ra rc re rf rg
sub rg, rc => rh // ra rc re rf rg rh
store rh => ra  //
```
Local Register Allocation: MAXLIVE

Example MAXLIVE computation
MAXLIVE is 4

```
loadI 1028  => r_a  // r_a
load r_a  => r_b  // r_a r_b
mult r_a, r_b  => r_c  // r_a r_b r_c
load x  => r_d  // r_a r_b r_c r_d
sub r_d, r_b  => r_e  // r_a r_b r_c r_d r_e
load z  => r_f  // r_a r_b r_c r_d r_e r_f
mult r_e, r_f  => r_g  // r_a r_b r_c r_d r_e r_f r_g
sub r_g, r_c  => r_h  // r_a r_b r_c r_d r_e r_f r_g r_h
store r_h  => r_a  // r_a
```
Local register allocation: Top Down

Algorithm:
• If number of values > k
  • Rank values by occurrences
  • Allocate first k - F values to registers
  • Spill other values
Local register allocation: top down

Example top down

Usage counts

```plaintext
loadI 1028 => r_a  // r_a
load r_a => r_b  // r_a r_b
mult r_a, r_b => r_c  // r_a r_b r_c
load x => r_d  // r_a r_b r_c r_d
sub r_d, r_b => r_e  // r_a r_c r_e
load z => r_f  // r_a r_c r_e r_f
mult r_e, r_f => r_g  // r_a r_c r_e r_f r_g
sub r_g, r_c => r_h  // r_a r_c
store r_h => r_a  //
```

Counts:

- r_a = 4
- r_b = 3
- r_c = 2
- r_d = 2
- r_e = 2
- r_f = 2
- r_g = 2
- r_h = 2
Local register allocation: Top Down

loadI 1028 => r_a
load r_a => r_b
mult r_a, r_b => r_c
load x => r_d
sub r_d, r_b => r_e
load z => r_f
mult r_e, r_f => r_g
sub r_g, r_c => r_h
store r_h => r_a

// r_a
// r_a r_b
// r_a r_b r_c
// r_a r_b r_c r_d
// r_a r_c r_e
// r_a r_c r_e r_f
// r_a r_c
// r_a r_c r_g
// r_a r_c
//

Must have r_d
r_c < r_a, r_b
Counts
ra=4
rb=3
rc=2
rd=2
e=2
rf=2
rg=2
rh=2

Spill r_c
Restore r_c
rg
rh
Local Register Allocation: Top down

Example top down
Spill code inserted

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>loadI 1028</td>
<td>r_a</td>
</tr>
<tr>
<td>load r_a</td>
<td>r_b</td>
</tr>
<tr>
<td>mult r_a, r_b</td>
<td>r_c</td>
</tr>
<tr>
<td>store r_c</td>
<td>r_{arp}, spill_c</td>
</tr>
<tr>
<td>load x</td>
<td>r_d</td>
</tr>
<tr>
<td>sub r_d, r_b</td>
<td>r_e</td>
</tr>
<tr>
<td>load z</td>
<td>r_f</td>
</tr>
<tr>
<td>mult r_e, r_f</td>
<td>r_g</td>
</tr>
<tr>
<td>load r_{arp}, spill_c</td>
<td>r_c</td>
</tr>
<tr>
<td>sub r_f, r_c</td>
<td>r_{h}</td>
</tr>
<tr>
<td>store r_{h}</td>
<td>r_a</td>
</tr>
</tbody>
</table>
Local register allocation: Top Down

Example top down
Register assignment straightforward

```
loadI 1028      r1
load  r1        r2
mult  r1, r2    => r3
store r3        r_{arp, spillc}
load  x         r3
sub   r3, r2    r2
load  z         r3
mult  r2, r3    r2
load  r_{arp, spillc}  r3
sub   r2, r3    r2
store r2        r1
```
Local register allocation: Bottom Up

Algorithm:
- Start with empty register set
- Load on demand
- When no register is available, free one

Replacement:
- Spill the value whose next use is farthest in the future
- Prefer clean value to dirty value
Local register allocation: Bottom Up

```
loadI 1028 ⇒ ra
load ra ⇒ rb
mult ra, rb ⇒ rc
load x ⇒ rd
sub rd, rb ⇒ re
load z ⇒ rf
mult re, rf ⇒ rg
sub rg, rc ⇒ rh
store rh ⇒ ra
```

// ra used
// rb
// rc
// rd
// re
// rf
// rg
// rh

Spill ra

Restore ra
Local register allocation: Bottom Up

Example bottom up

Spill code inserted

```
loadI 1028
load  r_a
mult  r_a, r_b  =>  r_c
store r_a        ->  r_{arp}, spill_a
load  x
sub   r_d, r_b
load  z
mult  r_e, r_f
sub   r_f, r_c
load  r_{arp}, spill_a
store r_h        ->  r_a
```
Global register allocation

Local allocation does not capture reuse of values across multiple blocks
Most modern, global allocators use a graph-colouring paradigm
Build a “conflict graph” or “interference graph”
   Data flow based liveness analysis for interference
   Find a k-colouring for the graph, or change the code to a nearby problem that it can k-colour
NP-complete under nearly all assumptions

1 Local allocation is NP-complete with dirty vs clean
Global register allocation: algorithm sketch

- From live ranges construct an interference graph
- Colour interference graph so that no two neighbouring nodes have same colour
- If graph needs more than k colours - transform code
  - Coalesce merge-able copies
  - Split live ranges
  - Spill
- Colouring is NP-complete so we will need heuristics
- Map colours onto physical registers
Global register allocation: Graph Coloring

A graph $G$ is said to be **k-colourable** if the nodes can be labeled with integers $1 \ldots k$ so that no edge in $G$ connects two nodes with the same label.

- **2-colourable**
- **3-colourable**
Global register allocation: Interference Graph

The interference graph, $G = (N, E)$
- Nodes in $G$ represent values, or live ranges
- Edges in $G$ represent individual interferences
- $\forall x, y \in N, x \rightarrow y \in E$ iff $x$ and $y$ interfere

A $k$-colouring of $G$ can be mapped into an allocation to $k$ registers

Two values interfere wherever they are both live
Two live ranges interfere if their values interfere at any point
Global register allocation: Coloring the Register Graph

- Degree³ of a node (n°) is a loose upper bound on colourability
- Any node, n, such that n° < k is always trivially k-colourable
  - Trivially colourable nodes cannot adversely affect the colourability of neighbours
  - Can remove them from graph
  - Reduces degree of neighbours - may be trivially colourable
- If left with any nodes such that n° ≥ k spill one
  - Reduces degree of neighbours - may be trivially colourable
Global register allocation: Chaitin’s Algorithm

1. While $\exists$ vertices with $< k$ neighbours in $G$
   o Pick any vertex $n$ such that $n^\circ < k$ and put it on the stack
   o Remove $n$ and all edges incident to it from $G$

2. If $G$ is non-empty ($n^\circ \geq k$, $\exists \ n \ni G$) then:
   o Pick vertex $n$ (heuristic), spill live range of $n$
   o Remove vertex $n$ and edges from $G$, put $n$ on “spill list”
   o Goto step 1

3. If the spill list is not empty, insert spill code, then rebuild the interference graph and try to allocate, again

4. Otherwise, successively pop vertices of the stack and colour them in the lowest colour not used by some neighbour
Global Register Allocation: Chaitin’s Algorithm

Colour with k = 3 colours

Stack

Colours

r1
r2
r3
Global Register Allocation: Chaitin’s Algorithm

\[ a^o = 2 < k \text{ Choose } a \]
Global Register Allocation: Chaitin’s Algorithm

Stack

Colours

Push a and remove from graph

r1
r2
r3
Global Register Allocation: Chaitin’s Algorithm

\[ b^\circ = 2 < k \quad \text{and} \quad c^\circ = 2 < k \]

Choose \( b \)
Global Register Allocation: Chaitin’s Algorithm

Push b and remove from graph

e – d

Stack

Colours

r1

r2

r3
Global Register Allocation: Chaitin’s Algorithm

c° = 2 < k,
d° = 2 < k, and
e° = 2 < k
Choose c

Stack

Colours

r1
r2
r3
Global Register Allocation: Chaitin’s Algorithm

Push c and remove from graph

Stack

Colours

r1
r2
r3
Global Register Allocation: Chaitin’s Algorithm

- Choose $d$

- $d^* = 1 < k$ and $e^* = 1 < k$

Stack:

- $a$
- $b$
- $c$

Colours:

- $r1$
- $r2$
- $r3$
Global Register Allocation: Chaitin’s Algorithm

Push d and remove from graph

Stack

Colours

r1
r2
r3
Global Register Allocation: Chaitin’s Algorithm

\[ e^\circ = 0 < k \text{ Choose } e \]
Global Register Allocation: Chaitin’s Algorithm

Push e and remove from graph

Stack

Colours

r1

r2

r3
Global Register Allocation: Chaitin’s Algorithm

Pop e, neighbours use no colours, choose red
Global Register Allocation: Chaitin’s Algorithm

Pop d, neighbours use red, choose blue
Global Register Allocation: Chaitin’s Algorithm

Pop c, neighbours use red and blue choose green
Global Register Allocation: Chaitin’s Algorithm

Pop c, neighbours
use red and blue
choose green
Global Register Allocation: Chaitin’s Algorithm

- Pop a, neighbours
- use blue choose red

Diagram: Nodes a, b, c, d, e connected by edges. Stack and colours (r1, r2, r3) are also shown.
Global Register Allocation: Optimistic Colouring

If Chaitin's algorithm reaches a state where every node has $k$ or more neighbours, it chooses a node to spill.

*Example of Chaitin overzealous spilling*

$k = 2$

Graph is 2-colourable
Chaitin must immediately spill one of these nodes

Briggs said, take that same node and push it on the stack! When you pop it off, a colour might be available for it! Chaitin-Briggs algorithm uses this to colour that graph
Global register allocation: Chaitin-Briggs algorithm

- While \( \exists \) vertices with < k neighbours in \( G \)
  - Pick any vertex \( n \) such that \( n^{\circ} < k \) and put it on the stack
  - Remove \( n \) and all edges incident to it from \( G_I \)

- If \( G \) is non-empty (\( n^{\circ} \geq k, \forall n \in G \)) then:
  - Pick vertex \( n \) (heuristic) (Do not spill)
  - Remove vertex \( n \) from \( G_I \), put \( n \) on stack (Not spill list)
  - Goto step 1

- Otherwise, successively pop vertices off the stack and colour them in the lowest colour not used by some neighbour
  - If some vertex cannot be coloured, then pick an uncoloured vertex to spill, spill it, and restart at step 1
Global Register Allocation: Chaitin-Briggs Algorithm
Global Register Allocation: Chaitin-Briggs Algorithm

\[ a^\circ = 2 \geq k \]
Don’t Spill, Choose a!
Global Register Allocation: Chaitin-Briggs Algorithm

Push a and remove the graph!
Global Register Allocation: Chaitin-Briggs Algorithm

\[ b^\circ = 1 < k \text{ and } c^\circ = 1 < k \]
Choose \( b \)
Global Register Allocation: Chaitin-Briggs Algorithm

\[ c^\circ = 1 < k, \]
\[ \text{and } d^\circ = 1 < k \]
Choose \( c \)

\( \text{Stack} \)

\( \text{Colours} \)

\( r1 \)

\( r2 \)
Global Register Allocation: Chaitin-Briggs Algorithm

Stack

Colours

r1
r2

d

Push c and remove from graph

a
b
c
Global Register Allocation: Chaitin-Briggs Algorithm

\[ d^\circ = 1 < k \text{ Choose } d \]
Global Register Allocation: Chaitin-Briggs Algorithm

Stack

a
b
c
d

Colours

r1
r2

Push d and remove from graph
Global Register Allocation: Chaitin-Briggs Algorithm

Stack

d
c
b
a

Colours

r1
r2

Pop d, neighbours use no colours, choose blue
Global Register Allocation: Chaitin-Briggs Algorithm

Pop c, neighbours use blue choose green
Global Register Allocation: Chaitin-Briggs Algorithm

Pop b, neighbours use blue choose green
Global Register Allocation: Chaitin-Briggs Algorithm

Pop a, neighbours use green choose blue
Global register allocation: Spill Candidates

- Minimise spill cost/degree
- Spill cost is the loads and stores needed. Weighted by scope - i.e. avoid inner loops
- The higher the degree of a node to spill the greater the chance that it will help colouring
- Negative spill cost load and store to same memory location with no other uses
- Infinite cost - definition immediately followed by use. Spilling does not decrease live range
Global Register Allocation: Alternative Spilling

- Splitting live ranges
- Coalesce
Global Register Allocation: Live Range Splitting

- A whole live range may have many interferences, but perhaps not all at the same time
- Split live range into two variables connected by copy
- Can reduce degree of interference graph
- Smart splitting allows spilling to occur in “cheap” regions
Global register allocation

Splitting example: Non contiguous live ranges - cannot be 2 coloured
Global Register Allocation: Live Range Splitting

Splitting example: Non contiguous live ranges - can be 2 coloured
Global register allocation: Coalescing

If two ranges don’t interfere and are connected by a copy coalesce into one – opposite of splitting. Reduces degree of nodes that interfered with both

If $x := y$ and $x \rightarrow y \in G_i$ then can combine $LR_x$ and $LR_y$

- Eliminates the copy operation
- Reduces degree of LRs that interfere with both $x$ and $y$
- If a node interfered with both both before, coalescing helps
- As it reduces degree, often applied before colouring takes place
Global register allocation: Coalescing

Coalescing can make the graph harder to color

- Typically, \( LR_{xy} > \max(LR_x, LR_y) \)
- If \( \max(LR_x, LR_y) < k \) and \( k < LR_{xy} \) then \( LR_{xy} \) might spill, while \( LR_x \) and \( LR_y \) would not spill

\[ a^\circ = b^\circ = 3 \quad ab^\circ = 4 \]
Global register allocation: Coalescing

Observation led to conservative coalescing

1. Conceptually, coalesce \( x \) and \( y \) iff \( x \rightarrow y \in G_I \) and \( LR_{xy}^o < k \)

2. We can do better
   - Coalesce \( LR_x \) and \( LR_y \) iff \( LR_{xy} \) has < \( k \) neighbours with degree > \( k \)
   - Only neighbours of “significant degree” can force \( LR_{xy} \) to spill

3. Always safe to perform coalesce
   - Cannot introduce a node of non-trivial degree
   - Cannot introduce a new spill
Global register allocation: Other Approaches

- Top-down uses high level priorities to decide on colouring
- Hierarchical approaches - use control flow structure to guide allocation
- Exhaustive allocation - go through combinatorial options - very expensive but occasional improvement
- Re-materialisation - if easy to recreate a value do so rather than spill
- Passive splitting using a containment graph to make spills effective
- Linear scan - fast but weak; useful for JITs
Global register allocation: Ongoing work

- Eisenbeis et al examining optimality of combined reg alloc and scheduling. Difficulty with general control-flow
- Partitioned register sets complicate matters. Allocation can require insertion of code which in turn affects allocation.
- Leupers investigated use of genetic algs for TM series partitioned reg sets.
- New work by Fabrice Rastello and others. Chordal graphs reduce complexity
- As latency increases see work in combined code generation, instruction scheduling and register allocation
Summary

- Local Allocation - spill code
- Global Allocation based on graph colouring
- Techniques to reduce spill code