Intro to Global Register Allocation

Problem:
- Allocation of variables (pseudo-registers) to hardware registers in a procedure

One of the most important optimizations
- Memory accesses are more costly than register accesses
  - True even with caches
  - True even with CISC architectures
- Important for other optimizations
  - E.g., CSE assumes old values are kept in registers
  - When it does not work well, the performance impact is noticeable.

Terminology

Allocation
- decision to keep a pseudo-register in a hardware register
- prior to register allocation, we assume an infinite set of registers
  - (aka “temps” or “pseudo-registers” or (bad) “variables”).

Spilling
- when allocation fails...
- a pseudo-register is spilled to memory, if not kept in a hardware register

Assignment
- decision to keep a pseudo-register in a specific hardware register

What are the Problems?

- For this example:
  - What is the minimum number of registers needed to avoid spilling?
  - Given n registers in a machine, is spilling necessary?
  - Find an assignment for all pseudo-registers, if possible.
  - If there are not enough registers in the machine, how do we spill to memory?
Abstraction for Reg Alloc & Assignment

Intuitively:
• Two pseudo-registers interfere if at some point in the program they cannot both occupy the same register.

Interference graph: an undirected graph, where
• nodes = pseudo-registers
• there is an edge between two nodes if their corresponding pseudo-registers interfere

Register Allocation and Coloring

• A graph is n-colorable if every node in the graph can be colored with one of n colors such that two adjacent nodes do not have the same color.
• Assigning n registers (without spilling) = Coloring with n colors
  -- assign a node to a register (color) such that no two adjacent nodes are assigned same registers(colors)
• Is spilling necessary? = is the graph n-colorable?
• To determine if a graph is n-colorable is NP-complete, for n>2

Simple Algorithm

Build an interference graph
• refining notion of a node
• finding the edges

Coloring
• use heuristics to try to find an n-coloring
  -- Success ⇒ coloring and we have an assignment
  -- Failure ⇒ graph not colorable, or graph is colorable, but we couldn’t find a coloring

Nodes in an Interference Graph
Live Ranges & Merged Live Ranges

- **Motivation:** to create an interference graph that is easier to color
  - Eliminate interference in a variable’s “dead” zones.
  - Increase flexibility in allocation: can allocate same variable to different registers

- A **live range** consists of a definition and all the points in a program (e.g. end of an instruction) in which that definition is live.
  - How to compute a live range?

- Two overlapping live ranges for the same variable must be merged

---

Merging Live Ranges

**Merging definitions into equivalence classes:**
- Start by putting each definition in a different equivalence class
- For each point in a program
  - if variable is live,
    - and there are multiple reaching definitions for the variable
      - merge the equivalence classes of all such definitions into a one equivalence class

```
A = 2 \{ A_2 \}
```

From now on, refer to merged live ranges simply as live range
- Merged live ranges are also known as “webs”

---

Example

Live Variables

```
A = \ldots \{ A_1 \}
B = \ldots \{ B_1 \}
C = \ldots \{ C_1 \}
D = \ldots \{ D_1 \}
```

Reaching Definitions

```
[ A ] = \{ A_1 \}
[ A,B ] \{ A_1,B_1 \}
[ B ] = \{ B_1 \}
[ D ] = \{ D_1 \}
```

```
A = 2 \{ A_2 \}
```

If A goto L1

```
\begin{align*}
L1: & \quad \text{C = \ldots \{ C_1 \}} \\
& \quad \text{D = \ldots \{ D_1 \}} \\
& \quad \text{A = 2 \{ A_2 \}} \\
& \quad \text{\{ A_1,B,D \}} \quad \text{Merge}
\end{align*}
```

Example: Merged Live Ranges

```
A = \ldots \{ A_1 \}
B = \ldots \{ B_1 \}
C = \ldots \{ C_1 \}
D = \ldots \{ D_1 \}
```

```
\begin{align*}
A = 2 \{ A_2 \} \\
\{ A_1,B,D \} & \quad \text{Merge}
\end{align*}
```
Edges of Interference Graph

Intuitively:
- Two live ranges (necessarily of different variables) may interfere if they overlap at some point in the program.
- Algorithm:
  - At each point in program, enter an edge for every pair of live ranges at that point

An optimized definition & algorithm for edges:
- For each defining inst $i$
  - Let $x$ be live range of definition at inst $i$
  - For each live range $y$ present at end of inst $i$
    - insert an edge between $x$ and $y$

• Faster
• Better quality?

Example: Interference Graph

So was it worth it to split the live ranges?

Example 2

If $Q$ goto L1

A = L1: B = L2:

• Reminder: coloring for $n > 2$ is NP-complete*
• Observations
  - a node with degree < $n$ ⇒ can always color it successfully, given its neighbors’ colors
  - a node with degree = $n$ ⇒
  - a node with degree > $n$ ⇒

Coloring Algorithm

Algorithm
- Iterate until stuck or done
  - Pick any node with degree < n
  - Remove the node and its edges from the graph
- If done (no nodes left)
  - reverse process and add colors

Example (n = 3)

Example:

Note: degree of a node may drop in iteration
- Avoids making arbitrary decisions that make coloring fail

What Does Coloring Accomplish?

Done:
- colorable
- also obtained an assignment (colors correspond to registers)

Stuck (n = 2):
- colorable or not?

Example:

One solution: optimistically remove nodes and hope we get lucky...

Checkpoint

Problems:
- Given n registers in a machine, is spilling avoided?
- Find an assignment for all pseudo-registers, whenever possible.

Solution:
- Abstraction: an interference graph
  - nodes: (merged) live ranges
  - edges: presence of live range at time of definition
- Register Allocation and Assignment problems
  - n-colorability of interference graph
  - NP-complete
- Heuristics to find an assignment for n colors
  - successful: colorable, and finds assignment
  - unsuccessful: colorability unknown & no assignment

Discussion

What about when we can't k-color?
- spill to memory

Is the minimum coloring always what we want?
- Hint: no

What about architecture strangeness?
- subword registers (x86, 68k, ColdFire...)
- register pairing (HP PA-RISC, SPARC, x86)
- register classes (x86, 68k, ColdFire...)

What about when we can't k-color?
- spill to memory
An Improvement: Move Coalescing

Basic idea:
- eliminate moves by assigning the src and dest to the same register
- copy propagation and dead code elimination can’t eliminate all unnecessary moves

If we allocate X and Y to the same register we can eliminate X = Y (copy prop couldn’t)

How can we modify our interference graph to do this?

An Exciting New Example

First compute live ranges...
...then construct interference graph

An Exciting New Example cont.

Want u and v to be assigned same color...
...merge u and v to form a single node

That is, if the src and dest don’t interfere

u and v are special:
A move whose source is not live-out of the move is a candidate for coalescing

Is Coalescing Always Good?

2 colorable
And the winner is? 3 colorable
When should we coalesce?

Always
- If we run into trouble start un-coalescing
  - no nodes with degree < k, see if breaking up coalesced nodes fixes
- yuck

Only if we can prove it won't cause problems
- Briggs: Conservative Coalescing
- George: Iterated Coalescing

Briggs: Conservative Coalescing
- Can coalesce u and v if:
  - (# of neighbors of uv with degree ≥ k) < k
- Why?
  - Simplify pass removes all nodes with degree < k
  - # of remaining nodes < k
  - Thus, uv can be simplified

George: Iterated Coalescing
- Can coalesce u and v if
  - foreach neighbor t of u
    - t interferes with v, or, doesn't change degree
    - degree of t < k
  - doesn't change degree removed by simplification

Why?
- Let S be set of neighbors of u with degree < k
- If no coalescing, simplify removes all nodes in S, call that graph G¹
- If we coalesce we can still remove all nodes in S, call that graph G²
- G² is a subgraph of G¹

When we simplify the graph, we remove nodes of degree < k... want to make sure we will still be able to simplify coalesced node, uv
Why Two Methods?

- Why not?
- With Briggs, one needs to look at all neighbors of a & b
- With George, only need to look at neighbors of a.

So:
- Use George if one of a & b has very large degree
- Use Briggs otherwise

Where We Are

Build
Simplify
Coalesce

Where We’re Going

Build
Simplify
Coalesce
Potential Spill
Select
Actual Spill

plus a bunch of important details...

Review: Build

First compute live ranges:
- use both reach defs and liveness
- live range defined by definition point
- ends when variable dies
- merge overlapping ranges of same var

v <- 1
w <- v + 3
x <- w + v
u <- v
t <- u + x
<- w
<- t
<- u
Review: Build

Construct interference graph:
- each node represents a live range
- edges represent live ranges that overlap
- put in move edges between move operands

Review: Simplify

Reduce the graph:
- remove non-move related, easy to color, nodes
- easy to color: degree < k
- place on stack

Review: Coalesce

Coalesce moves:
- conservatively combine operands of a move
- Briggs, George heuristics for being conservative

Repeat Simplify
- Detail: If both Simplify and Coalesce get stuck, start simplifying move related nodes

Transition Slide!

Build
Simplify
Coalesce
Potential Spill
Select
Actual Spill
What if we can’t simplify?

Now what?
Be optimistic:
- Put a node with degree ≥ k on stack
- Lose guarantee that anything we put on stack is colorable
- If we’re lucky this node will still be colorable when popped from stack
Be realistic:
- If unlucky, this node will have to be spilled (allocated to memory)
- Mark as potential spill to avoid recomputation later

Select

Pop a node from the stack
Assign it a color that does not conflict with neighbors in interference graph
This will always be possible, unless the node is a potential spill
If it is not possible must spill

Spilling to Memory

RISC Architectures
- Only load and store can access memory
  - every use requires load
  - every def requires store
  - create new temporary for each location

CISC Architectures
- can operate on data in memory directly
  - makes writing compiler easier(?), but isn’t necessarily faster
- pseudo-registers inside memory operands still have to be handled

Spilling

Allocate w to memory location $M_w$
Spilled variables are allocated to the stack in an area completely controlled by the compiler. These memory locations are special in that they can be optimized without concern for memory aliasing issues.

Now Start Over...
...compute live ranges...
**Spilling**

We have to start from scratch every time we spill

- Suggestions?
  - Fewer iterations?
  - Faster iterations?

**What to Spill?**

When choosing potential spill node want:

- A node that makes graph easier to color
  - Fewer spills later
- A node that isn’t “expensive” to spill
  - First nodes pushed on stack are last to be colored
    - more likely to be spilled
  - An expensive node would slow down the program if spilled
- We can apply heuristics both when choosing potential spill nodes and when choosing actual spill nodes
  - not required to spill node that we popped off stack and can’t color
A Spill Heuristic

Pick node (live range) \( n \) that minimizes:

\[
\sum_{\text{def} \in n} 10^{\text{depth(def)}} + \sum_{\text{use} \in n} 10^{\text{depth(use)}}
\]

This heuristic prefers nodes that:

- Are used infrequently
- Aren’t used inside of loops
- Have a large degree

Could use any one of several other heuristics as well...

Reducing Stack Frame Size

- How do you allocate spilled live ranges?
  - every live range gets its own location on the stack frame
  - or we can be smarter...
- What about `mov a, b` where both \( a \) & \( b \) have been spilled?
  - Use graph-coloring with aggressive coalescing!
- Use liveness info to create an interference graph of the spilled nodes
- Always coalesce
- Simplify/Select
- Colors map to frame locations

Is it worth it?

Rematerialization

An alternative to spilling

- Recompute value of variable instead of store/load to memory

Example:

\[
\begin{align*}
\text{v} & \leftarrow 1 \\
\text{w} & \leftarrow \text{v} + 3 \\
\text{x} & \leftarrow \text{w} + \text{v} \\
\text{u} & \leftarrow \text{v} \\
\text{t} & \leftarrow \text{u} + \text{x} \\
\text{w} & \leftarrow 4 \\
\text{t} & \leftarrow \text{w} \\
\text{u} & \leftarrow \text{t} \\
\end{align*}
\]
Special Registers

Which registers can be used?

- Some registers have special uses.
  - Register 0 or 31 is often hardwired to contain 0.
  - Special registers to hold return address, stack pointer, frame pointer, etc.
- Reserved registers for operating system.
- Typically, leaves about 20 or so registers for other general uses.

Impact on register allocation:

- Temps should be assigned only to the non-reserved registers (allocable).
- Hard registers are pre-colored in the interference graph.

Register Usage Conventions

Certain registers are used for specific purposes defined by the standard calling convention.

- 4-6 argument registers.
  - The first 4-6 arguments to procedures/functions are always passed in these registers.
- ~8 callee-save registers.
  - These registers must be preserved across procedure calls. Thus, if a procedure wants to use a callee-save register, it must first save the old value and then restore it before returning.
  - The remainder are caller-save registers.
  - These are not preserved across procedure calls. Thus, a procedure is free to use them without saving first.
  - Includes the argument registers.

How do we support these?

- neat trick for handling callee save
- call instruction

Allocating Callee-Save Registers

Move callee-save reg to temp at start of procedure
Move it back at end of procedure
What happens if there is no register pressure?
What happens if there is a lot of register pressure?

entry: define r

temp <- r
...
exit: r <- temp
use r

Allocating to callee-save registers

CALL instruction “modifies” all caller-save regs

entry: define r

t1 <- r
x <-
...
call
\[(r_1, r_2, r_3 <-)
\;
\]
\[
( <- r_1, r_2, r_3)
\]
...
<- x
exit: r_e <- t_1
use r_e
Reducing Register Pressure

Recall: Split pseudo-registers into live ranges to create an interference graph that is easier to color
- Eliminate interference in a variable’s “dead” zones.
- Increase flexibility in allocation: can allocate same variable to different registers

Insight

Split a live range into smaller regions (by paying a small cost) to create an interference graph that is easier to color
- Eliminate interference in a variable’s “nearly dead” zones.
  - Cost: Memory loads and stores
  - Load and store at boundaries of regions with no activity
  - # active live ranges at a program point can be > # registers
- Can allocate same variable to different registers
  - Cost: Register operations
    - a register copy between regions of different assignments
  - # active live ranges cannot be > # registers

Examples

Example 1:

```
FOR i = 0 TO 10
  A = A + ...
  (does not use B)
FOR j = 0 TO 10000
  B = B + ...
  (does not use A)
```

Example 2:

```
a = b
b = b + a
b = c
```
One Algorithm

Observation: Spilling is absolutely necessary if
* number of live ranges active at a program point > n not degree in graph

Apply live-range spilling before coloring
* Identify a point where number of live ranges > n
* For each live range active around that point
  - find the outermost "block construct" that does not access the variable
* Choose a live range with the largest inactive region
* Split the inactive region from the live range

Alternative Allocators

Graph allocator, as described, has issues
* What are they?

Alternative: Single pass graph coloring
* Build, Simplify, Coalesce as before
  - Keep going
* Requires second, reload phase
  - "fixes" spilled variables
  - Requires that we reserve a register
  - Can get messy

Claim: Does a pretty good job
* Why?
  - Key is order nodes are colored...

Advantages? Disadvantages?

Alternative Allocators

Local/Global Allocation
* Allocate "local" pseudo-registers
  - Lifetime contained within basic block
  - Register sufficiency no longer NP-Complete!
* Allocate global pseudo-registers
  - Single pass global coloring
  - Reload pass to fix spills (allocator does not generate spill code)
* Can also do global then local (Morgan)
* Advantages? Disadvantages?

How good is it in practice?

Percent of functions with no spills

<table>
<thead>
<tr>
<th></th>
<th>PPC (32)</th>
<th>68k (16)</th>
<th>x86 (8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Percent</td>
<td>97.3</td>
<td>99</td>
<td>54.35</td>
</tr>
</tbody>
</table>

*Used gcc -fnew-ra to compile >10,000 functions from Mediabench, Spec95, Spec2000, and micro-benchmarks
*Recorded for which functions graph coloring had to spill
Register Allocation

**PPC (32 registers)**

Increase in Spills as Number of Variables in Function Grows

**68k (16 registers)**

Increase in Spills as Number of Variables in Function Grows

**x86 (8 registers)**

Increase in Spills as Number of Variables in Function Grows

**What’s Next**

Good Question