Code generation for modern processors

Refs: AS&U, Chapter 9 + Notes.

Optional: Muchnick, 16.3 & 17.1

What are the dominant performance issues for a superscalar RISC processor?

Strategy

$$\xrightarrow[\infty\ \textit{regs}]{\textit{il}} \boxed{\textit{select}} \xrightarrow[\infty\ \textit{regs}]{\textit{il}} \boxed{\textit{schedule}} \xrightarrow[\infty\ \textit{regs}]{\textit{il}} \boxed{\textit{allocate}} \xrightarrow[k\ \textit{regs}]{\textit{il}} \boxed{\textit{schedule}} \xrightarrow[k\ \textit{regs}]{\textit{asm}}$$

select is fairly simple (*problem of the 80's*)

allocate and *schedule* are complex

Definitions (1 of 2)

Instruction selection

- the process of mapping *il* into assembly code
- assumes a *fixed* storage mapping (*code shape*)
- combining instructions, using address modes

Register allocation

- the process of deciding which values reside in registers
- changes storage mapping (*and the code*)
- concern about placement of data

Definitions (2 of 2)

Instruction scheduling

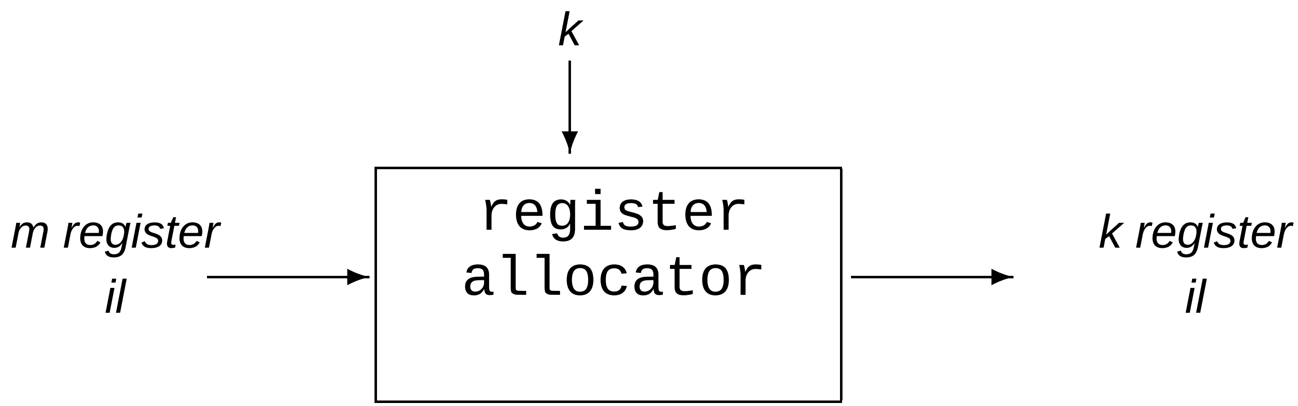
- the process of reordering instructions to hide latencies
- assumes a *fixed* program
- changes demand for registers

Each problem is NP-complete for a non-trivial target processor.

The problems are tightly intertwined, but conventional wisdom says we can (and should) *attack each one separately.*

Register allocation

Concept:



Assumptions

- Load-store RISC architecture
- Three-address IL
- Previous analysis identifies values that are *illegal* to hold in registers. *Which?*
  - Load into register before use
  - Store back after def

Goals

- Produce correct k register code
- Minimize added `loads` and `stores`
- Minimize memory space needed to hold spills
- Allocator must be efficient ($\Rightarrow$ *no backtracking*)

Register classes

Some architectures have multiple register classes:

- *commonly*: general-purpose (GP) and floating-point (FP) (and double-precision may use two FPs)
- PowerPC: condition code registers
- IA 64: predicate registers, branch target registers

Problem: Interactions between classes

- If all classes were used independently, could allocate separately
- Arithmetic ops may produce condition codes, predicates, etc. to be set
- FP and other register spills may cause address arithmetic!

Our Approach

1. *Assume separate allocation for each class except where noted*.
2. *Allocate GPRs last*.

Allocation versus assignment

The distinction:

- *allocation* : choosing what to keep in registers at each point
- *assignment* : choosing specific registers for values

Complexity

	Allocation	Assignment
Local	• optimal, linear time methods for simplest case • almost everything else is NP-complete	• uniform regs, no spilling $\Rightarrow$ linear time • adjacent register pairs $\Rightarrow$ NP-complete
Global	• NP-complete for 1 register machine • NP-complete for k register machine • most subproblems are NP-complete	NP-complete

Register Allocation Preliminaries

Definitions and observations

1. *Spill* virtual register (aka value) $V \equiv$
   - Assign to a memory location; not a physical register
     $\Rightarrow$ Load just before each use
     $\Rightarrow$ Store just after each def
   - Usually assign to a stack slot
   - Some algorithms may spill a value in one interval and allocate to a register in another
2. Reserve registers to ensure feasibility *default*
   (a) must be able to compute addresses, load, & store
   (b) requires a minimal number of registers, $\mathcal{F}$: *feasible set*
   (c) $\mathcal{F}$ depends on architecture
3. MAXLIVE $\equiv$ Maximum number of values live at any instruction (in some region of code, e.g., a basic block)

Two Approaches for Single Basic Block

Common features of single-basic block allocation

- All values live in memory between basic blocks
- Therefore, even for values allocated a physical register:
  $\Rightarrow$ Load before first use in block
  $\Rightarrow$ Store after last def in block
- if MAXLIVE $\leq k$, allocation is trivial; $\mathcal{F}$ irrelevant
- if MAXLIVE $> k$, must spill some values to memory

(1): Top-down allocation : Usage Counts

- a register can hold only a single value in entire block
- sort variables by ($|$*uses*$|$)
- assign first $k - \mathcal{F}$ names to registers
- spill all other values (load before each use; store after each def)

Two Approaches for Single Basic Block

(2) Bottom-up allocation : Linear scan using live ranges

- a register can hold different values at different statements
- keep a stack of free registers
- for each statement ($v_3 = v_1$ op v_2)

  assign free registers to v_1, v_2, v_3 (if not in register already)

  free a register at the end of a live range
- if no register available, spill some busy register

- *Key question: Which register to spill*?
  - spill register used farthest in the future (*Sheldon Best 1955*)
  - on a tie, favor value that need not be stored back to memory

Global Register Allocation: An Early Approach

Global extension of *Usage Counts*

- Extend usage counts to account for loops, branches
- Insert load at block entry; store at block exit (*live values only*)
- Some cross-block analysis:
  try to avoid load, store at block boundaries

A few extra spills in the wrong places can be extremely expensive

Global Register Allocation: The Modern Approach

A fundamentally global approach: Graph Coloring

- *Lavrov (1961), Cocke (1971), Chaitin (1981)*
- abandon the distinction between local and global
- model live ranges of entire procedure in a single graph
- reduce allocation problem to coloring nodes in the graph

  minimal coloring is NP-Complete

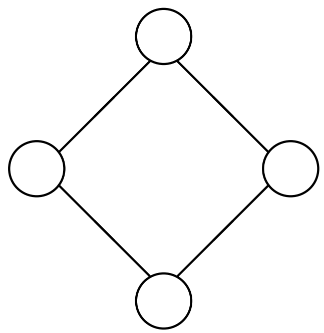
  $\Rightarrow$ use heuristics to choose coloring
- map colors onto physical registers

Graph coloring

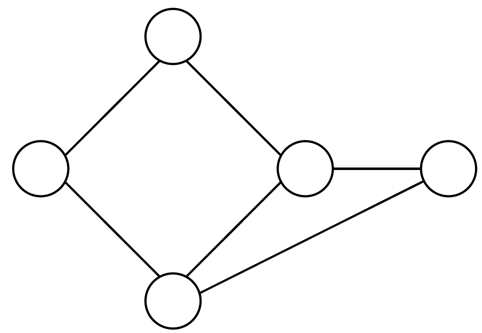
The problem

A graph $G = (N, E)$ is said to be *k-colorable* if and only if the nodes can be labeled with integers 1,. . . ,k so that no edge in G connects nodes with the same label.

Examples



"diamond" graph
2-colorable



a more complex graph
3-colorable

Application to allocation & assignment

- graphical representation of conflicts
- coloring corresponds to feasible assignment
- model machine constraints in graph

Graph Coloring Register Allocation

4 major aspects:

1. Constructing *global live ranges*
   - *not* the same as intuitive live range in straight-line code
2. Building *interference graph* for a procedure
   - captures information about overlapping live ranges
3. Estimating spill costs
   - important to consider when k-coloring fails
4. (*Try to*) construct a k-coloring
   - if unsuccesful, choose values to spill and repeat
   - spill placement becomes critical issue

Let's take these one by one.

Global Live Ranges

Definition: Live Range *Also called a web*

A *live range* is a set of references (definitions and uses) s.t.

- for any use in the set, all defs that reach it are in the set too
- for any def in the set, all uses it reaches are in the set too

Fundamental Invariant

All references in a live range are allocated to the same physical register.

Information needed to compute live ranges:

- need all definitions that reach a single use, and vice versa
- SSA form provides exactly this information:
  - each SSA variable has a single def and zero or more uses
  - ϕ-functions show multiple defs that reach a use:

$$x_3 \leftarrow \phi(x_1, \ldots x_n)$$

Computing Global Live Ranges

Idea:
Partition the SSA references into disjoint sets:

- all references to a variable belong in the same set
- all arguments to a ϕ-function belong in the same set as the output variable of the function

Algorithm:
Use the disjoint set *union-find* algorithm:

1. initially: every name is a separate set
2. repeatedly merge sets that meet at a ϕ-function:

   $X = \phi(Y, Z)$: Merge the sets currently holding Y and Z into the set holding X
3. Finally, treat all references in the same live range as a single virtual register

The concept of interference

Definition of Interference

- Idea: Two values cannot be in one register if "*used in overlapping intervals*"
- One way to define "interfere":
  n_i and n_j interfere if they are simultaneously *live*
  - *Problem*: What if both are not used in some basic block (where both are live)?
- Better way: n_i and n_j interfere if n_i is defined at a point where n_j is live (or vice versa)

Representing the *interference* property

- model the problem with an *interference graph*, I
  - nodes represent values
  - any two values that interfere are connected by an edge
- a k-coloring for $I \Rightarrow$ fits in k registers

Building the interference graph

Algorithm:

1. Identify global live ranges
2. Build LIVE_IN, LIVE_OUT sets for each block
3. Walk backwards through each block separately:
   (a) Initialize live set: $LiveNow \leftarrow$ LIVE_OUT
   (b) For each instruction: $v_1 \; op \; v_2 \rightarrow v_3$
      (i) v_3 interferes with every value in $LiveNow$
      (ii) remove v_3 from $LiveNow$
      (ii) add v_1, v_2 to $LiveNow$

Use two representations:

1. adjacency matrix: lower-diagonal bit matrix
   - allows test for interference in $O(1)$ time
   - hash-table has same benefit with lower memory usage for large graphs
2. adjacency list:
   - allows efficient iteration over neighbors of a node

Copy Coalescing

An important optimization folded into register allocation

When to coalesce

$$\texttt{mov}\ v_i \ \rightarrow \ v_j$$

Coalesce if :

1. v_i and v_j do not interfere, OR
2. v_i and v_j are not modified after the copy
   i.e., values remain equal always

Coalesce means . . .

1. Replace v_j with v_i
2. Remove copy instruction
3. Combine nodes in the interference graph

Estimating spill costs

Components of cost (per reference) for a spill :

Load / store:

1. address computation
2. memory operation
3. estimated execution frequency

Or "Rematerialization:

1. recomputing the value
2. estimated execution frequency

Address computation:

- minimize by keeping spilled values in activation record
- can load / store with (fp + offset) address

Execution frequencies:

- Static estimation:
  - weight by 10^d for loop depth d
  - weight by branching probability if enclosed in branches
- Profiling:
  - measure execution frequencies for representative inputs

Coloring by Graph Pruning: Observations

Two Key Observations

1. **The $degree < k$ rule**:
   A graph having a node n with $degree < k$ is k-colorable
   iff the graph with node n removed is k-colorable
   Proof:
   - $\Rightarrow$ obvious
   - $\Leftarrow$ given k-coloring of graph without node n, all neighbors of N use fewer than k colors. Pick a remaining color for n.

2. **Consider a node n with $degree >= k$**:
   - Neighbors of N may still use fewer than k distinct colors
   - $\Rightarrow$ defer spilling until you are sure it is needed

Idea: Simplify graph by removing nodes

Repeat until graph is empty:

- Repeatedly remove a node with $degree < k$ from the graph
- When no such node exists: choose a candidate to spill and remove it

Coloring by Graph Pruning: Algorithm

Algorithm *Use stack of nodes*

while N is non-empty
- if $\exists$ node n with $n^\circ < k$, push n on stack
- else, pick n as possible spill candidate and push n on stack
- remove n from I (along with its incident edges)

while stack is non-empty
- pop n, insert n into I, try to color n
- if fail to color n, mark n for spilling

Spill all marked nodes (insert code) *Need registers!*

Rewrite code to use assigned physical registers

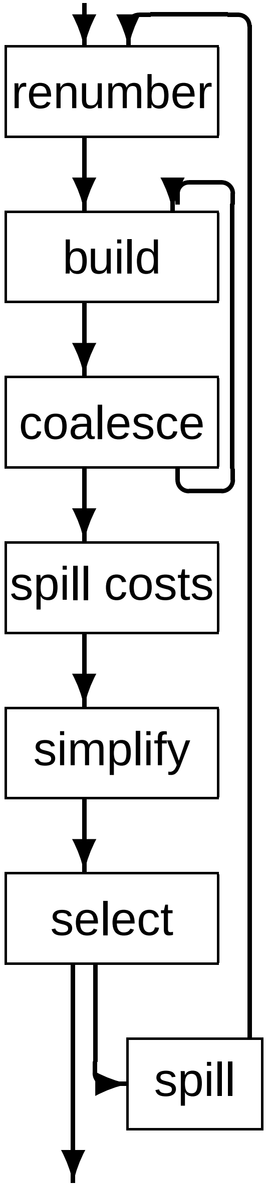
Heuristic for choosing spill candidates is key

Observations about Reserving Registers

Observations

- If reserved registers for executing spill code:
  - after spilling all nodes, we are done
- If did not reserve registers:
  - use virtual registers for spilling
  - repeat entire allocation algorithm
  - more expensive but could give fewer spills

Chaitin-Briggs register allocators



find live ranges and rename them

build the interference graph, $I = (N, E)$

fold unneeded copies:
$l_i \rightarrow l_j$ & $\langle l_i, l_j \rangle \notin E \;\Rightarrow$ *combine* l_i & l_j

estimate cost for spilling each live range

while N *is non-empty*
if $\exists\, n$ *with* $n^\circ < k$*, push* n *on stack*
else, pick n *for possible spilling & push it*
remove n *from* I

while stack is non-empty
pop n*, insert* n *into* I*, & try to color it*

spill uncolored definitions and uses

Incorporates deferred spilling (Briggs 1989)

Picking a spill candidate

When $\forall\, n \in N, n^\circ \geq k$*, it must pick a spill candidate*

Chaitin's heuristic

Chaitin says "minimize $\frac{\textit{spill cost}}{\textit{current degree}}$" where

- *current degree* is the number of remaining neighbors
- *spill cost* of l_i is defined as

$$\sum_{j \in \textit{refs}(l_i)} 10^{d(j)} - \sum_{\texttt{loads} \in \textit{defs}(l_i)} 2 \cdot 10^{d(j)} - \sum_{\texttt{stores} \in \textit{uses}(l_i)} 2 \cdot 10^{d(j)}$$

where

- *refs*(l_i) is *defs*$(l_i) \cup$ *uses*(l_i)
- $d(j)$ is the nesting depth of j

Bernstein *et al.* suggested repeating simplify, select, & spill with different spill choice heuristics

Improvements to Chaitin: Best of 3 spilling

The idea

- when allocator blocks, it chooses a value to spill
- determines which value spills & where code is inserted
- spill choice is the critical issue
- let $area(lr) = \sum_{i \in refs(lr)} \textit{num\_live}(i) \times 5^{d(i)}$

Author	*Ratio to minimize*
Chaitin	$\frac{\textit{cost}}{\textit{current degree}}$
Bernstein	$\frac{\textit{cost}}{\textit{current degree}^2}$
Bernstein	$\frac{\textit{cost}}{\textit{area}}$
Bernstein	$\frac{\textit{cost}}{\textit{area}^2}$
—	*cost*

The implementation

- no metric dominates others (NP-noise)
- actual coloring is inexpensive; coalescing takes time
- run multiple colorings & use best result (20%)

How does Chaitin-style do?

Strengths and Weaknesses

- ⇑ precise interference graph
- ⇑ strong coalescing mechanism
- ⇑ handles register assignment well
- ⇑ runs relatively quickly

- ⇓ known to overspill in tight cases
- ⇓ graph has no geography
- ⇓ spills live range everywhere
- ⇓ long blocks become spilling by use counts

Is further improvement possible?

- rising spill costs
- aggressive transformations
- live range splitting
- better allocation for long blocks

Remaining problems

1. spill code
2. spill code
3. spill code

⇒ *still room for improvement on this problem*

Improvements

Situations that we see in practice

1. *Pass-through live ranges*

   value live but not referenced in block or region

2. *Spill the "wrong" value*

   some other value might lead to better code

3. *Excessive demand for registers*

   register pressure simply too high

Improvements are both possible and desirable
Allocator is final "filter" through which code must pass

Pass-through live ranges A regional effect

- should be lowest priority for register
- can be heavily weighted by Chaitin
- want “fair competition” inside each loop

This problem motivated Briggs’s work on live range splitting

```
do i ← 1 to n
  ...
  do j ← 1 to m
    do k ← 1 to o
      x ←
      ...
  do j ← 1 to m
    do k ← 1 to o
      no reference to x
      ...
  do j ← 1 to m
    do k ← 1 to o
      ← x
      ...
```

Improvements to Chaitin (1 of 2) Overview

Experience suggests that there is no silver bullet
Difficulties are not *symptom of a single effect*

Global:

deferred spilling	Briggs *et al.*	20%
best of three	Bernstein *et al.*	20%
rematerialization	Briggs *et al.*	20%
optimal coloring		expensive

$\rightarrow$ *these are good things to do*
$\rightarrow$ *do not change underlying problem*

Improvements to Chaitin (2 of 2) Overview

Regional:

live range splitting	Briggs *et al.*	$\pm 4\times$
	Koblenz & Callahan	(?)
scalar replacement	Carr	20% to $3\times$

$\rightarrow$ *move to near-by problem with a "better" solution*

Local:

Best's method	Sheldon Best (1955)	naively "optimal"
	(*et al.* in 65,75,88,95)	

$\rightarrow$ *good spill decisions*

$\rightarrow$ *ignore surrounding context*