Register Allocation

Introduction

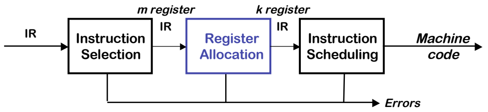
Local Register Allocators

Outline

- What is Register Allocation and Its Importance
- Simple Register Allocators
- Webs
- Interference Graphs
- Graph Coloring
- Splitting
- More Transformations

What is Register Allocation?

Part of the Compiler's Back End



Critical Properties

- Produce <u>Correct</u> 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), maybe O(n^2), but not O(2^n)

Register Allocation & Assignment

Optimal global allocation is NP-Complete, under almost any assumptions.

- At Each Point in the Code
  1. Allocation: Determine which Values will reside in Registers
  2. Assignment: Select a Register for each such value

The Goal is an Allocation that "Minimizes" Running Time

Importance of Register Allocation

- *Optimally* Use of one of the Most Critical Processor Resources
  - Affects almost every statement of the program
  - Register accesses are much faster than memory accesses
    - Eliminates expensive memory instructions
    - wider gap in faster newer processors
    - Number of instructions goes down due to direct manipulation of registers (no need for load and store instructions)
- *Probably* is the optimization with the most impact!
- Common Trade-Off:
  - Registers: Fast Storage with Small Capacity (say 32, 64, 128)
  - Main Memory: Slow Storage with High Capacity (say Giga Bytes)

Importance of Register Allocation

- What Can Be Put in Registers?
  - Scalar Variables
  - Big Constants
  - Some Array Elements and Record Fields
  - Register set depending on the data-type
    - Floating-point in fp registers
    - Fixed-point in integer registers
- Allocation of Variables (including temporaries) up-to-now stored in Memory to Hardware Registers
  - *Pseudo* or *Virtual* Registers
    - unlimited number of registers
    - space is typically allocated on the stack with the stack frame
  - Hard Registers
    - Set of Registers Available in the Processor
    - Usually need to Obey some Usage Convention

Register Usage Convention in MIPS

Name	Number	Use	Preserved Across a Function Call?
$zero	0	The Constant Value 0	Yes
$at	1	Assembler Temporary	No
$v0, $v1	2,3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Function Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8,$t9	24,25	Temporaries	No
$k0,$k1	26,27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

Register Allocation Approaches

- Local Allocators: use instruction-level knowledge
  - Top-Down: Use Frequency of Variables Use for Allocation
  - Bottom-Up: Evaluate Instructions Needs and Reuse Registers
- 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
- Common Algorithmic Trade-Off
  - Local Allocators are Fast
  - Some Problems with the Generated Code as they lack more Global Knowledge

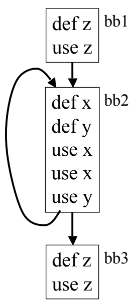
Local Register Allocation

- In General Hard Problem (*still*)
  - Code Generation for more than a single Register is NP-Complete
- Use Simple Strategies:
  - Top-Down: Just put in Register Names that Occur more Often
  - Bottom-Up: Evaluate each Instruction and Keep Track of When Values are Needed Later On.
- Extension to Multiple Basic Blocks
  - Using Profile Data to Determine Frequently Executed Paths
  - Use Nesting Depth of Code

Top-Down Local Register Allocator

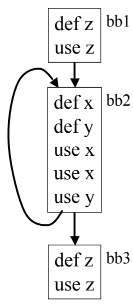
- Estimate the Benefits of Putting each Variable in a Register in a Particular Basic Block
  - cost(V, B) = Number of *uses* and *defs* of the var V in basic block B
- Estimate the Overall Benefit
  - TotCost(V) = cost(V, B)*freq(B) for all basic block B
  - If freq(B) is not known, use 10^{depth} where *depth* represents the nesting depth of B in the CFG of the code.
- Assign the (R-*feasible*) Highest-payoff Variables to Registers
  - Reserve *feasible* registers for basic calculations and evaluation.
  - Rewrite the code inserting load/store operation where appropriate.

Example



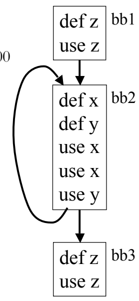
Example

- Cost for basic blocks
  - cost(x, bb1) = 0
  - cost(x, bb2) = 3
  - cost(x, bb3) = 0
  - cost(y, bb1) = 0
  - cost(y, bb2) = 2
  - cost(y, bb3) = 0
  - cost(z, bb1) = 2
  - cost(z, bb2) = 0
  - cost(z, bb3) = 2



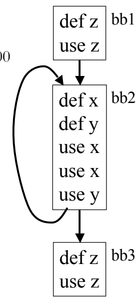
Example

- Cost for basic blocks
  - cost(x, bb1) = 0
  - cost(x, bb2) = 3
  - cost(x, bb3) = 0
  - cost(y, bb1) = 0
  - cost(y, bb2) = 2
  - cost(y, bb3) = 0
  - cost(z, bb1) = 2
  - cost(z, bb2) = 0
  - cost(z, bb3) = 2
- Frequency
  - freq(bb1) = 1
  - freq(bb2) = 100
  - freq(bb3) = 1



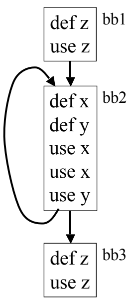
Example

- Cost for basic blocks
  - cost(x, bb1) = 0
  - cost(x, bb2) = 3
  - cost(x, bb3) = 0
  - cost(y, bb1) = 0
  - cost(y, bb2) = 2
  - cost(y, bb3) = 0
  - cost(z, bb1) = 2
  - cost(z, bb2) = 0
  - cost(z, bb3) = 2
- Frequency
  - freq(bb1) = 1
  - freq(bb2) = 100
  - freq(bb3) = 1
- Total cost
  - TotCost(x) = 0*1 + 3*100 + 0*1 = 300
  - TotCost(y) = 0*1 + 2*100 + 0*1 = 200
  - TotCost(z) = 2*1 + 0*100 + 2*1 = 4



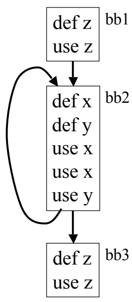
Example

- Total cost
  - TotCost(x) = 300
  - TotCost(y) = 200
  - TotCost(z) = 4
- Assume 2 Registers are Available
  - Assign x and y to Registers



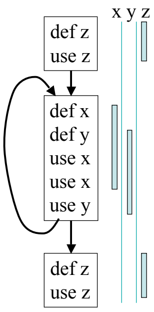
Problem 1

- Allocation is same as above
  - x, and y get registers, but not z
- The variables need to occupy the registers even when it does not need it
- All x, y and z can have registers



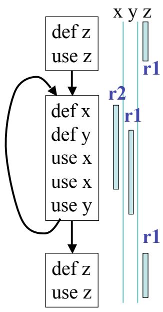
Problem 1

- Allocation is same as above
  - x, and y get registers, but not z
- The variables need to occupy the registers even when it does not need it
- All x, y and z can have registers



Problem 1

- Allocation is same as above
  - x, and y get registers, but not z
- The variables need to occupy the registers even when it does not need it
- All x, y and z can have registers

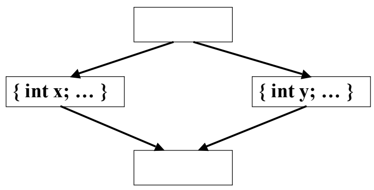


Problem 2

- Even non-interfering variables don't share registers

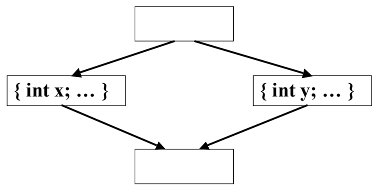
Problem 2

- Even non-interfering variables don't share registers



Problem 2

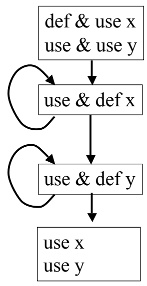
- Even non-interfering variables don't share registers



- x and y can use the same registers

Problem 3

- Different phases of the program behave differently
- One register available
  - Register for x in the first loop
  - Register for y in the second loop
  - Don't care too much about the rest
- Need to spill
  - Top-Down "All or Nothing" will not work



Bottom-Up Local Allocator

- Basic Ideas:
  - Focus on the Needs of Each Instructions in a Basic Block
    - Ensure Each Instruction Can Execute
    - Instruction Operands and Results in Registers
  - Transitions Between Instructions
    - Observe Which Values are Used Next; in the Future
- On-Demand Allocation:
  - Iterate Through the Instructions of a Basic Block
  - Allocate the Value of the Operand, if Not Already in a Register
  - Allocate Register for Result
  - When Out of Registers:
    - Release Register Whose Value is to be Used *Farthest* into the Future
    - Dirty Register Value Requires Memory Operation to Update Storage

Bottom-Up Local Allocator

- Details:
  - Instructions in format: vr_x op $vr_y \Rightarrow vr_z$ using virtual registers
  - Data Structures: A Class of Registers
    - The Number of Registers in Each Class
    - The Virtual Name for Each Register in the Class
    - For Each Virtual Name a Distance to the Next Use in the Basic Block
    - A Flag Indicating if the Corresponding Physical Register is in Use
    - A Stack of Free Physical Registers with a Stack Pointer (Integer Index)

```
initialize(class,size)
  class.Size ← size;
  for i ← size-1 to 0 do
    class.Name[i] ← -1;
    class.Next[i] ← ∞;
    class.Free[i] ← true;
    push(i,class);
  class.StackTop = size-1;
```

  - Functions:
    - `class(vr`$_x$`)` defines the set of Registers the Value in vr_x can be Stored into
    - `ensure(vr`$_x$`)`, `free(vr`$_x$`)` and `allocate(vr`$_x$`)` functions.
    - `dist(vr`$_x$`)` returns the distance to the next reference to vr_x

Bottom-Up Local Allocator

<u>Bottom-Up Local Allocation Algorithm</u>

Input: Basic Block B
Output: Rewritten Instruction in B

```
foreach instr i: vr_i3 ← vr_i1 op vr_i2 ∈ B do

  r_x ← ensure(vr_i1,class(vr_i1));
  r_y ← ensure(vr_i2,class(vr_i2));
  if(vr_i1 is not needed after i)then
    free(vr_i1,class(vr_i1));
  if(vr_i2 is not needed after i)then
    free(vr_i2,class(vr_i2));
  r_z ← allocate(vr_i3,class(vr_i3));

  rewrite i as r_z ← r_x op r_y

  if (vr_i1 is needed after i) then
    class.Next[r_x] = dist(vr_i1);
  else
    class.Next[r_x] = ∞;
  if (vr_i2 is needed after i) then
    class.Next[r_y] = dist(vr_i2);
  else
    class.Next[r_y] = ∞;
  class.Next[r_z] = dist(vr_i3);

end
```

```
void free(i,class)
  if(class.Free[i]≠ true) then
    push(i,class);
    class.Name[i] ← -1;
    class.Next[i] ← ∞;
    class.Free[i] ← true;
  end

reg ensure(vr,class)
  r ← find(vr,class);
  if(r exists) then
    result ← r;
  else
    result ← allocate(vr,class);
    emit code to move vr into r;
  end
  return result;

reg allocate(vr,class)
  if(class.StackTop ≥ 0) then
    r ← pop(class);
  else
    r ← findMaxNext(class);
    if(r is dirty) then
      save r in memory;
  class.Name[r] ← vr;
  class.Next[r] ← -1;
  class.Free[r] ← false;
  return r;
```

Bottom-Up Local Allocator

<u>Bottom-Up Local Allocation Algorithm</u>

Input: Basic Block B
Output: Rewritten Instruction in B

```
foreach instr i: vr_i3 ← vr_i1 op vr_i2 ∈ B do

  r_x ← ensure(vr_i1,class(vr_i1));
  r_y ← ensure(vr_i2,class(vr_i2));
  if(vr_i1 is not needed after i)then
    free(vr_i1,class(vr_i1));
  if(vr_i2 is not needed after i)then
    free(vr_i2,class(vr_i2));
  r_z ← allocate(vr_i3,class(vr_i3));

  rewrite i as r_z ← r_x op r_y

  if (vr_i1 is needed after i) then
    class.Next[r_x] = dist(vr_i1);
  else
    class.Next[r_x] = ∞;
  if (vr_i2 is needed after i) then
    class.Next[r_y] = dist(vr_i2);
  else
    class.Next[r_y] = ∞;
  class.Next[r_z] = dist(vr_i3);

end
```

```
void free(i,class)
  if(class.Free[i]≠ true) then
    push(i,class);
    class.Name[i] ← -1;
    class.Next[i] ← ∞;
    class.Free[i] ← true;
  end

reg ensure(vr,class)
  r ← find(vr,class);
  if(r exists) then
    result ← r;
  else
    result ← allocate(vr,class);
    emit code to move vr into r;
  end
  return result;

reg allocate(vr,class)
  if(class.StackTop ≥ 0) then
    r ← pop(class);
  else
    r ← findMaxNext(class);
    if(r is dirty) then
      save r in memory;
  class.Name[r] ← vr;
  class.Next[r] ← -1;
  class.Free[r] ← false;
  return r;
```


`Next` field is temporarily set to -1 so that a possible second `allocate` will not bump this register on the same instruction

Bottom-Up Allocator Example

```
vr_3 ← vr_1 op vr_2

vr_5 ← vr_4 op vr_1

vr_6 ← vr_5 op vr_6

vr_7 ← vr_3 op vr_2
```

Bottom-Up Allocator Example


```
vr_3 ← vr_1 op vr_2

vr_5 ← vr_4 op vr_1

vr_6 ← vr_5 op vr_6

vr_7 ← vr_3 op vr_2
```


Size = 3

	0	1	2
Name	-1	-1	-1
Next	∞	∞	∞
Free	T	T	T

Stack
2
1
0

Top = 2

Bottom-Up Allocator Example

$vr_3 \leftarrow r_0$ op vr_2

$vr_5 \leftarrow vr_4$ op vr_1

$vr_6 \leftarrow vr_5$ op vr_6

$vr_7 \leftarrow vr_3$ op vr_2

Size = 3

	0	1	2
Name	vr_1	-1	-1
Next	1	∞	∞
Free	F	T	T
Stack	2		
	1		

Top = 1

Bottom-Up Allocator Example

$vr_3 \leftarrow r_0$ op r_1

$vr_5 \leftarrow vr_4$ op vr_1

$vr_6 \leftarrow vr_5$ op vr_6

$vr_7 \leftarrow vr_3$ op vr_2

Size = 3

	0	1	2
Name	vr_1	vr_2	-1
Next	1	3	∞
Free	F	F	T
Stack	2		

Top = 0

Bottom-Up Allocator Example

$r_2 \leftarrow r_0$ op r_1

$vr_5 \leftarrow vr_4$ op vr_1

$vr_6 \leftarrow vr_5$ op vr_6

$vr_7 \leftarrow vr_3$ op vr_2

Size = 3

	0	1	2
Name	vr_1	vr_2	vr_3
Next	1	3	3
Free	F	F	F
Stack			

Top = -1

Bottom-Up Allocator Example

$r_2 \leftarrow r_0$ op r_1

$vr_5 \leftarrow vr_4$ op vr_1

$vr_6 \leftarrow vr_5$ op vr_6

$vr_7 \leftarrow vr_3$ op vr_2

Run Out of Registers for vr_4

`Allocate` will find register with maximum value of Next to Spill to Memory, *e.g.*, vr_2

Size = 3

	0	1	2
Name	vr_1	vr_2	vr_3
Next	1	3	3
Free	F	F	F
Stack			

Top = -1

Bottom-Up Allocator Example

$r_2 \leftarrow r_0$ op r_1

$r_1 \rightarrow$ mem

$vr_5 \leftarrow r_1$ op vr_1

$vr_6 \leftarrow vr_5$ op vr_6

$vr_7 \leftarrow vr_3$ op vr_2

Size = 3

	0	1	2
Name	vr_1	vr_4	vr_3
Next	1	-1	3
Free	F	F	F
Stack			

Top = -1

Bottom-Up Allocator Example

$r_2 \leftarrow r_0$ op r_1

$r_1 \rightarrow$ mem

$vr_5 \leftarrow r_1$ op vr_1

$vr_6 \leftarrow vr_5$ op vr_6

$vr_7 \leftarrow vr_3$ op vr_2

Size = 3

	0	1	2
Name	vr_1	-1	vr_3
Next	1	∞	3
Free	F	T	F
Stack	1		

Top = 0

Bottom-Up Allocator Example

$r_2 \leftarrow r_0$ op r_1

$r_1 \rightarrow$ mem

$vr_5 \leftarrow r_1$ op r_0

$vr_6 \leftarrow vr_5$ op vr_6

$vr_7 \leftarrow vr_3$ op vr_2

Size = 3

	0	1	2
Name	-1	-1	vr_3
Next	∞	∞	3
Free	T	T	F
Stack	1		
	0		

Top = 1

Bottom-Up Allocator Example

$r_2 \leftarrow r_0$ op r_1

$r_1 \rightarrow$ mem

$r_0 \leftarrow r_1$ op r_0

$vr_6 \leftarrow vr_5$ op vr_6

$vr_7 \leftarrow vr_3$ op vr_2

Size = 3

	0	1	2
Name	vr_5	-1	vr_3
Next	1	∞	3
Free	F	T	F
Stack	1		

Top = 0

Bottom-Up Allocator Example

Size = 3

$r_2 \leftarrow r_0$ op r_1
$r_1 \rightarrow$ mem
$r_0 \leftarrow r_1$ op r_0
$r_0 \rightarrow$ mem
$vr_6 \leftarrow r_0$ op vr_6
$vr_7 \leftarrow vr_3$ op vr_2

	0	1	2
Name	vr_5	-1	vr_3
Next	-1	∞	3
Free	F	T	F
Stack	1		

Top = 0

Bottom-Up Allocator Example

Size = 3

$r_2 \leftarrow r_0$ op r_1
$r_1 \rightarrow$ mem
$r_0 \leftarrow r_1$ op r_0
$r_0 \rightarrow$ mem
$vr_6 \leftarrow r_0$ op r_1
$vr_7 \leftarrow vr_3$ op vr_2

	0	1	2
Name	vr_5	vr_6	vr_3
Next	-1	-1	3
Free	F	F	F
Stack			

Top = -1

Bottom-Up Allocator Example

Size = 3

$r_2 \leftarrow r_0$ op r_1
$r_1 \rightarrow$ mem
$r_0 \leftarrow r_1$ op r_0
$r_0 \rightarrow$ mem
$r_1 \leftarrow r_0$ op r_1
$vr_7 \leftarrow vr_3$ op vr_2

	0	1	2
Name	vr_5	vr_6	vr_3
Next	-1	-1	3
Free	F	F	F
Stack			

Top = -1

Bottom-Up Allocator Example

Size = 3

$r_2 \leftarrow r_0$ op r_1
$r_1 \rightarrow$ mem
$r_0 \leftarrow r_1$ op r_0
$r_0 \rightarrow$ mem
$r_1 \leftarrow r_0$ op r_1
$vr_7 \leftarrow vr_3$ op vr_2

	0	1	2
Name	-1	-1	vr_3
Next	∞	∞	3
Free	T	T	F
Stack	0		
	1		

Top = 1

Bottom-Up Allocator Example

$r_2 \leftarrow r_0$ op r_1

$r_1 \rightarrow$ mem

$r_0 \leftarrow r_1$ op r_0

$r_0 \rightarrow$ mem

$r_1 \leftarrow r_0$ op r_1

$r_1 \rightarrow$ mem

$r_0 \leftarrow r_2$ op r_1

Size = 3

	0	1	2
Name	vr_7	vr_6	vr_3
Next	-1	-1	-1
Free	F	F	F

Stack Top = -1

Dirty and Clean Registers

$r_2 \leftarrow r_0$ op r_1

$r_1 \rightarrow$ mem // not needed

$r_0 \leftarrow r_1$ op r_0

$r_0 \rightarrow$ mem

$r_1 \leftarrow r_0$ op r_1

$r_1 \rightarrow$ mem

$r_0 \leftarrow r_2$ op r_1

- Choosing which Register to Reuse:
  - Dirty - Need to Update Memory
  - Clean - Just Reuse the Physical Register
- Idea: Give Preference to Clean Registers
  - No Need to Save Contents to Memory
- Not Always the Best Approach:
  - When Dirty Value is Reused Far Way
  - Better to Restore it to Memory
  - Rather than Holding on to the Register

What a Smart Allocator Needs to Do

- Determine ranges for each variable can benefit from using a register (webs)
- Determine which of these ranges overlap (interference)
- Find the benefit of keeping each web in a register (spill cost)
- Decide which webs gets a register (allocation)
- Split webs if needed (spilling and splitting)
- Assign hard registers to webs (assignment)
- Generate code including spills (code generation)

Summary

- Register Allocation and Assignment
  - Very Important Transformations and Optimization
  - In General Hard Problem (NP-Complete)
- Many Approaches
  - Local Methods: Top-Down and Bottom-Up
  - Quick but not Necessarily Very Good