Chapter 9. Register Allocation

“Basics of Compiler Design”

Torben Ægidius Mogensen

Dr. Marco Valtorta, Professor
Computer Science and Engineering Dept.
University of South Carolina

Radu Vitoc, PhD candidate
CSCE 531
University of South Carolina
April 28, 2013

9.1 Introduction

The problem:

Processors have a limited number of registers that can be used for variable allocation during code generation.

The solution:

Mapping variables used in the program to a smaller number of registers via ***register allocation*** (sharing and spilling).

Where to apply: Intermediate code OR machine code?

Each method has its own pros and cons. We will study register allocation applied to the intermediate code.

9.2 Liveness - When can two variable share a register?

Definition:

A variable is ***live*** at some point in the program if the value it contains at that point might conceivably be used in future computations.

A variable is ***dead*** if there is no way its value can be used in the future.

Loose answer:

Two variables may share a register if there is no point in the program where they are both live.

9.2 Liveness (cont.)

Liveness Rules:

1. A variable is live at the start of an instruction that uses its content.
2. A variable is dead at the start of an instruction that assigns it a value and does not use its content.
3. If a variable is live at the end of an instruction and that instruction does not assign a value to it, then the variable is also live at the start of the instruction
4. A variable is live at the end of an instruction if it is live at the start of any of the immediately succeeding instr.

9.3 Liveness analysis

- $\boldsymbol{succ[i]}$ is the set of successors of instr. i
  1. $j \in succ[i]$ if $j = i + 1$ and $j \in \{'GOTO', \ 'IF\text{-}THEN\text{-}ELSE'\}$
  2. $l \in succ[i]$ if $i = '\mathrm{G}OTO\ l'$
  3. $l_t, l_f \in succ[i]$ if $i = 'IF\ p\ THEN\ l_t\ ELSE\ l_f'$
- $\boldsymbol{gen[i]}$ is the set of variables that i generates liveness for
- $\boldsymbol{kill[i]}$ is the set of variables that i kills
- $\boldsymbol{in[i]}$ is the set of variables that are live at the start of i
- $\boldsymbol{out[i]}$ is the set of variables that are live at the end of i

$$in[i] = gen[i] \cup (out[i] \setminus \mathrm{kill}[i])$$

$$out[i] = \bigcup_{j \in succ[i]} in[j]$$

9.3 Liveness analysis (cont.)

1: $a := 0$
2: $b := 1$
3: $z := 0$
4: LABEL $loop$
5: IF $n = z$ THEN end ELSE $body$
6: LABEL $body$
7: $t := a + b$
8: $a := b$
9: $b := t$
10: $n := n - 1$
11: $z := 0$
12: GOTO $loop$
13: LABEL end

Instruction i	$gen[i]$	$kill[i]$
LABEL l	$\emptyset$	$\emptyset$
$x := y$	$\{y\}$	$\{x\}$
$x := k$	$\emptyset$	$\{x\}$
$x :=$ **unop** y	$\{y\}$	$\{x\}$
$x :=$ **unop** k	$\emptyset$	$\{x\}$
$x := y$ **binop** z	$\{y, z\}$	$\{x\}$
$x := y$ **binop** k	$\{y\}$	$\{x\}$
$x := M[y]$	$\{y\}$	$\{x\}$
$x := M[k]$	$\emptyset$	$\{x\}$
$M[x] :=$ y	$\{x, y\}$	$\emptyset$
$M[k] :=$ y	$\{y\}$	$\emptyset$
GOTO l	$\emptyset$	$\emptyset$
IF x **relop** y THEN l_t ELSE l_f	$\{x, y\}$	$\emptyset$
$x :=$ CALL $f(args)$	$args$	$\{x\}$

Mogensen, T.Æ., “Basics of Compiler Design“, pp. 194-195

9.3 Liveness analysis (cont.)

i	$succ[i]$	$gen[i]$	$kill[i]$
1	2		a
2	3		b
3	4		z
4	5		
5	6, 13	n, z	
6	7		
7	8	a, b	t
8	9	b	a
9	10	t	b
10	11	n	n
11	12		z
12	4		
13			

	Initial		Iteration 1		Iteration 2		Iteration 3	
i	$out[i]$	$in[i]$	$out[i]$	$in[i]$	$out[i]$	$in[i]$	$out[i]$	$in[i]$
1			n, a	n	n, a	n	n, a	n
2			n, a, b	n, a	n, a, b	n, a	n, a, b	n, a
3			n, z, a, b	n, a, b	n, z, a, b	n, a, b	n, z, a, b	n, a, b
4			n, z, a, b	n, z, a, b	n, z, a, b	n, z, a, b	n, z, a, b	n, z, a, b
5			a, b, n	n, z, a, b	a, b, n	n, z, a, b	a, b, n	n, z, a, b
6			a, b, n	a, b, n	a, b, n	a, b, n	a, b, n	a, b, n
7			b, t, n	a, b, n	b, t, n	a, b, n	b, t, n	a, b, n
8			t, n	b, t, n	t, n, a	b, t, n	t, n, a	b, t, n
9			n	t, n	n, a, b	t, n, a	n, a, b	t, n, a
10				n	n, a, b	n, a, b	n, a, b	n, a, b
11					n, z, a, b	n, a, b	n, z, a, b	n, a, b
12					n, z, a, b	n, z, a, b	n, z, a, b	n, z, a, b
13			a	a	a	a	a	a

Mogensen, T.Æ., "Basics of Compiler Design", pp. 196-197

9.4 Interference - When can two variable share a register?

Definition:

A variable x ***interferes*** with variable y if x ≠ y and $\exists i$ such that $x \in kill[i], i \in out[i]$, and $i \neq 'x := y'$.

Precise answer:

Two variables can share a register if neither interferes with the other.

≠ “they should not be live at the same time”.

9.4 Interference (cont.)

Definition:

A variable x ***interferes*** with variable y if x ≠ y and $\exists i$ such that $x \in kill[i], i \in out[i]$, and $i \neq 'x := y'$.

Precise answer:

Two variables can share a register if neither interferes with the other.

≠ "they should not be live at the same time".

When may two live variables share a register?

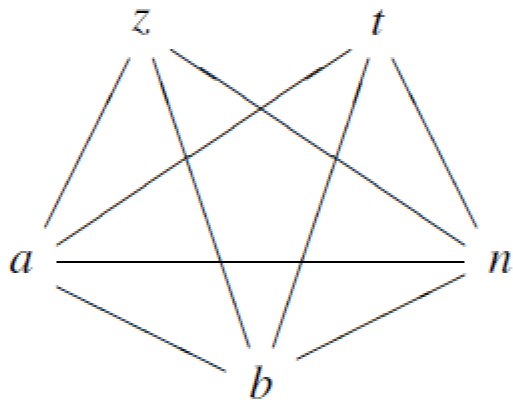
9.4 Interference (cont.)

Answer:

- After $'x := y'$, x and y may be live simultaneously, but as they contain the same value they can still share a register.
- If $x \notin out[i]$ even if $x \in kill[i]$ "we have assigned to x a value that is definitely not read from x later on)", x is not technically live after instruction i, but still interferes with any $y \in out[i]$. Can x be always eliminated?

  No. The instruction may set a flag, access memory, etc.

9.4 Interference (cont.)



Interference graph

Instruction	Left-hand side	Interferes with
1	a	n
2	b	n,a
3	z	n,a,b
7	t	b,n
8	a	t,n
9	b	n,a
10	n	a,b
11	z	n,a,b

Interference table

Mogensen, T.Æ., "Basics of Compiler Design", pp. 198

9.5. Register allocation by graph coloring

Precise answer alternative:

Two variables can share a register if they are not connected by an edge in the interference graph.

Graph coloring:

- Two nodes that share an edge have different register numbers (colors).
- The total number of different register numbers (colors) is no higher than the number of available registers.

9.5. Register allocation by graph coloring (cont.)

Coloring is NP-complete which means that no effective (polynomial-time) method for doing it optimally is known.

Heuristic methods (are used in practice):

1. If a node in the graph has < N edges (number of colors or registers available), it is set aside and color the rest of the graph. The remaining number of nodes ≤ N-1 connected by edges cannot use all N colors.

9.5. Register allocation by graph coloring (cont.)

2. If there is no node with < N edges look for a node of which the connected nodes have the same color. Together they have been given < N colors.
   2.1. If it exist, use one of the unused colors
   2.2. If it does not exist, mark the node for spill.

9.5. Register allocation by graph coloring (cont.)

Algorithm 9.3

initialise: *Start with an empty stack.*

simplify: *If there is a node with less than N edges, put this on the stack along with a list of the nodes it is connected to, and remove it and its edges from the graph.*

If there is no node with less than N edges, pick any node and do as above.

If there are more nodes left in the graph, continue with **simplify**, *otherwise go to* **select**.

select: *Take a node and its list of connected nodes from the stack. If possible, give the node a colour that is different from the colours of the connected nodes (which are all coloured at this point). If this is not possible, colouring fails and we mark the node for spilling (see below).*

If there are more nodes on the stack, continue with **select**.

Mogensen, T.Æ., "Basics of Compiler Design", pp. 200

9.6 Spilling

Definition:

The process of identifying some variables that will reside in memory instead of registers (at least for brief periods) and modifying code to move/retrieve them to/from memory.

After the rewrite of the program due to spilling, the register allocation process is repeated to identify any induced new variables for spilling, etc.

Example:

Apply algorithm 9.3 to the coloring of the interference graph with 3 colors.

Node	Neighbours	Colour
n		1
t	n	2
b	t,n	3
a	b,n,t	*spill*
z	a,b,n	2

Mogensen, T.Æ., "Basics of Compiler Design", pp. 204

9.7 Heuristics (Simplify step)

If the ***simplify*** step of algorithm 9.3 cannot find a node with less than N edges, the next node is chosen, in a way that makes coloring more likely or reduces the number of spilling variables, based on criteria like these:

- Chose a node with close to N neighbors
- Chose a node with many neighbors that have close to N neighbors of their own.
- Look at the program and select a variable that does not “cost” so much to spill (not used inside a loop).

Criteria can be combined using weighted average, etc.

9.7 Heuristics (Select step)

When picking a color during the ***select*** step of algorithm 9.3, it is more likely that the rest of the graph be colored if a color already used is chosen versus one that was not used. Preferably a color prevalently used by its rank 1 neighbors, rank 2, etc.

- **Removing redundant moves**

  'x := y' can be removed if x and y use the same register.

  Methods: Biased coloring, coalescing, live-range splitting.

- **Using explicit register numbers**

  Some instructions may require their arguments to be stored in specific registers. ***Pre-coloring*** constraint.

Discussions

Bibliography

Torben Ægidius Mogensen , “Basics of Compiler Design“, *online*, pp.191-208, 20 Aug. 2010, URL: http://www.diku.dk/~torbenm/Basics.