



Register Allocation

Christoph Kessler, IDA,
Linköpings universitet, 2014.

Register Allocation



- **Register Allocation:** Determines values (variables, temporaries, constants) to be kept when in registers
- **Register Assignment:** Determine in which physical register such a value should reside.
- Essential for Load-Store Architectures
- Reduce memory traffic (\rightarrow memory / cache latency, energy)
- Limited resource
- Values that are alive simultaneously cannot be kept in the same register
- Strong interdependence with instruction scheduling
 - scheduling determines live ranges
 - spill code needs to be scheduled
- **Local register allocation** (for a single basic block) can be done in linear time (see function `getreg()` in the Dragon Book).
- **Global register allocation** (with minimal spill code) is NP-complete. Can be modeled as a graph coloring problem [Ershov'62] [Cocke'71].

C. Kessler, IDA, Linköpings universitet.

2

Live range



(Here, variable = program variable or temporary)

- A variable is being **defined** at a program point if it is written (given a value) there.
- A variable is **used** at a program point if it is read (referenced in an expression) there.
- A variable is **alive** at a point if it is referenced there or at some following point that has not (may not have) been preceded by any definition.
- A variable is **reaching** a point if an (arbitrary) definition of it, or usage (because a variable can be used before it is defined) reaches the point.
- A variable's **live range** is the area of code (set of instructions) where the variable is both alive and reaching.
 - does not need to be consecutive in program text.

C. Kessler, IDA, Linköpings universitet.

3

Local Register Allocation



For variable v and basic block B_i :

$$\begin{aligned} netsave(v, i) &= \#uses_i \cdot usesave + \#defs_i \cdot defsave \\ &= l \cdot ldcost \quad (l = 1 \text{ if Load}(v) \text{ needed at beg. of } B_i, 0 \text{ otherw.}) \\ &= s \cdot stcost \quad (s = 1 \text{ if Store}(v) \text{ needed at end of } B_i, 0 \text{ otherw.}) \end{aligned}$$

For loop L estimate benefit of keeping v in a register:

$$benefit(v, L) = l \cdot depth(L) \cdot \sum_{i \in blocks(L)} netsave(v, i)$$

with R registers available:

allocate the R objects v with greatest benefit in L

moves may be necessary instead of `Load(v)` / `Store(v)`

if v could reside in (different) registers in $Pred(B_i)$, B_i , $Succ(B_i)$

add worst-case terms $|Pred(v)| \cdot mvcost$, $|Succ(v)| \cdot mvcost$

C. Kessler, IDA, Linköpings universitet.

4

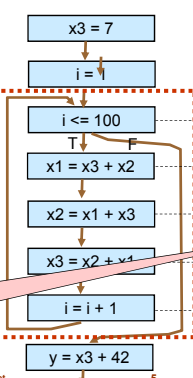
Register Allocation for Loops



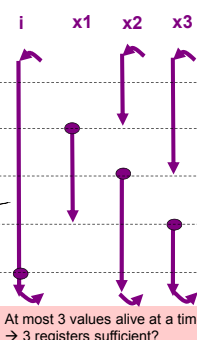
Example:

```
x3 = 7
for i = 1 to 100 {
  x1 = x3 + x2
  x2 = x1 + x3
  x3 = x2 + x1
}
y = x3 + 42
```

Control flow graph



Live ranges (loop only):
cyclic intervals
e.g. for i : $[0, 6)$, $[6, 7)$



All variables interfere with each other – need 4 regs?

At most 3 values alive at a time
 \rightarrow 3 registers sufficient?

C. Kessler, IDA, Linköpings universitet.

5

Global Register Assignment by Graph Coloring



[Ershov'62] [Cocke'71] [Chaitin et al.'81] [Chaitin'82]
[Chow/Hennessy'84,'90] [Briggs et al.'89] [Briggs'92] ...

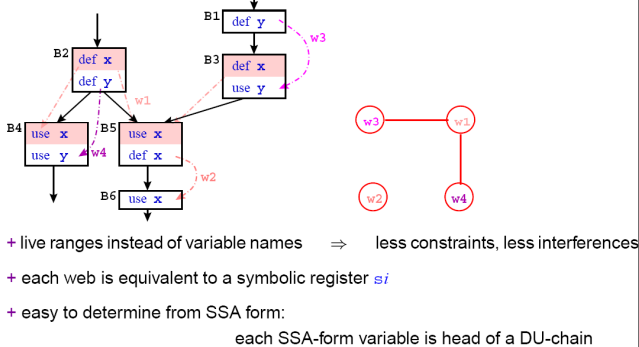
1. allocate objects that can be assigned to registers to **distinct symbolic registers** s_1, s_2, \dots
2. determine candidates for allocation to registers (s_i / webs)
3. build **interference graph**
 - nodes:** allocatable objects, target machine registers
 - edges:** (undir.) $\{a_i, a_j\}$ iff allocatable objects a_i, a_j simultaneously live $\{a_i, r_j\}$ iff a_i should not reside in register r_j
4. color nodes with R colors ($R = \#$ available registers) such that any two adjacent nodes have different colors
5. allocate each object to a register that has the same color.

C. Kessler, IDA, Linköpings universitet.

6

Allocatable objects: Webs (Live ranges)

web = max. union of DU-chains $\langle d, u_1, \dots, u_n \rangle$ that overlap in at least one use



C. Kessler, IDA, Linköping universitet.

7

Register Allocation by Graph Coloring

- Step 1: Given a program with symbolic registers s_1, s_2, \dots
 - Determine live ranges of all variables

```

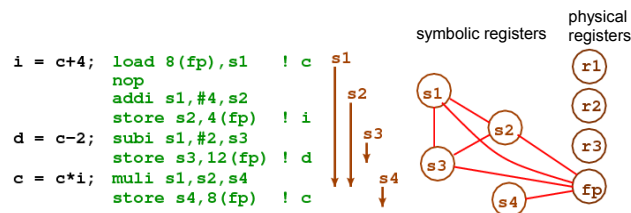
i = c+4;  load 8(fp), s1  ! c  s1
          nop
          addi s1, #4, s2
          store s2, 4(fp) ! i  s2
d = c-2;  subi s1, #2, s3
          store s3, 12(fp) ! d  s3
c = c*i;  muli s1, s2, s4
          store s4, 8(fp) ! c  s4
    
```

C. Kessler, IDA, Linköping universitet.

8

Register Allocation by Graph Coloring

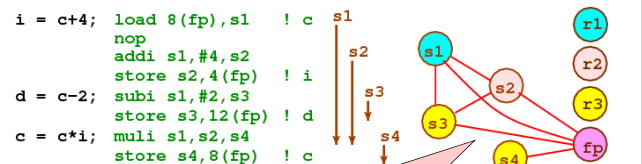
- Step 2: Build the Register Interference Graph
 - Undirected edge connects two symbolic registers (s_i, s_j) if live ranges of s_i and s_j overlap in time
 - Reserved registers (e.g. fp) interfere with all s_i



C. Kessler, IDA, Linköping universitet.

9

- Step 3: Color the register interference graph with k colors, where $k = \text{\#available registers}$.
 - If not possible: pick a victim s_i to spill, generate spill code (store after def., reload before use)
 - This may remove some interferences. Rebuild the register interference graph + repeat Step 3...



This register interference graph cannot be colored with less than 4 colors, as it contains a 4-clique

C. Kessler, IDA, Linköping universitet.

10

Coloring a graph with k colors

- NP-complete for $k > 3$
- Chromatic number $\gamma(G)$ = minimum number of colors to color a graph G
- $\gamma(G) \geq c$ if the graph contains a c -clique
 - A c -clique is a completely connected subgraph of c nodes
- Chaitin's heuristic (1981):


```

S ← { s1, s2, ... } // set of spill candidates
while ( S not empty )
  choose some s in S
  if s has less than k neighbors in the graph
    then // there will be some color left for s:
      delete s (and incident edges) from the graph
    else modify the graph (spill, split, coalesce ... nodes) → changes IR and restart.
// once we arrive here, the graph is empty:
color the nodes greedily in reverse order of removal.
            
```

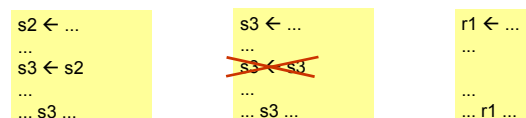
degree < k rule

C. Kessler, IDA, Linköping universitet.

11

Live range coalescing = fusion of webs

- For a copy instruction $s_j \leftarrow s_i$
 - where s_i and s_j do not interfere
 - and s_i and s_j are not rewritten after the copy operation
- Merge s_i and s_j :
 - patch (rename) all occurrences of s_i to s_j
 - update the register interference graph
- and remove the copy operation.



C. Kessler, IDA, Linköping universitet.

12

Conservative Coalescing



Coalescing two live ranges u and v can increase the node degree:

$$d(u \& v) > \max(d(u), d(v)) \text{ is possible}$$

→ may make $u \& v$ harder to color

Conservative coalescing:

coalesce only if $d(u \& v) < R$

→ can color $u \& v$ by the naive $\text{degree} < R$ heuristic

C. Kessler, IDA, Linköping universitet.

13

Spilling (1)



■ Spilling a (physical) register r

= spilling the live range w contained in r

- uses some memory location $w.\text{tmp}$ (on stack, scratchpad memory, or w 's home memory location)
- insert a Store $r, w.\text{tmp}$ immediately after each definition of $w.\text{var}$
- insert a Load $r, w.\text{tmp}$ immediately before each use of $w.\text{var}$

■ Some interferences disappear, the interference graph must be updated.

C. Kessler, IDA, Linköping universitet.

14

Spilling (2)



Heuristic choice of the best spill candidate [Bernstein et al.'89]

minimize ratio $\text{spillcost}(w) / \text{degree}(w)^2$ etc.

$$\text{spillcost}(w) = c_{\text{store}} \cdot \sum_{\text{def} \in w} 10^{\text{depth}(\text{def})} + c_{\text{load}} \cdot \sum_{\text{use} \in w} 10^{\text{depth}(\text{use})} - c_{\text{move}} \cdot \sum_{\text{copy} \in w} 10^{\text{depth}(\text{copy})}$$

A copy instruction whose source or target is spilled can be removed.

- spill value once per block if possible
→ avoids redundant loads and stores
- consider rematerialization as alternative to spilling

C. Kessler, IDA, Linköping universitet.

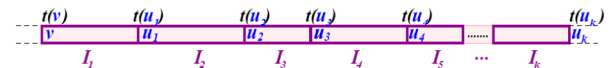
15

Rematerialization



Recomputing a value to a register (rematerialization) may be cheaper than storing and reloading it, e.g. for loading constants to a register.

Modify $\text{spillcost}(w)$ accordingly.



If a spilled value is used several times and the restored value remains live for several adjacent uses, a Load/Rematerialize is necessary only before the first of them.

(= live range splitting)

[Chow/Hennessy'84,'90]

C. Kessler, IDA, Linköping universitet.

16

Spilling (3)

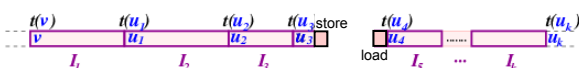
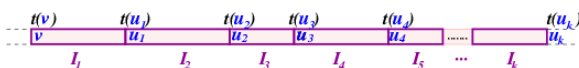


■ Total spilling eliminates a live range completely

- store after *each* definition, reload before *each* use

■ Partial spilling splits a live range into several ones

- Some reduction in interference, some spill code



C. Kessler, IDA, Linköping universitet.

17

Live Range Splitting



- Long live ranges tend to interfere with many others
→ harder to color.

■ Idea: Split up long live ranges to avoid some spilling

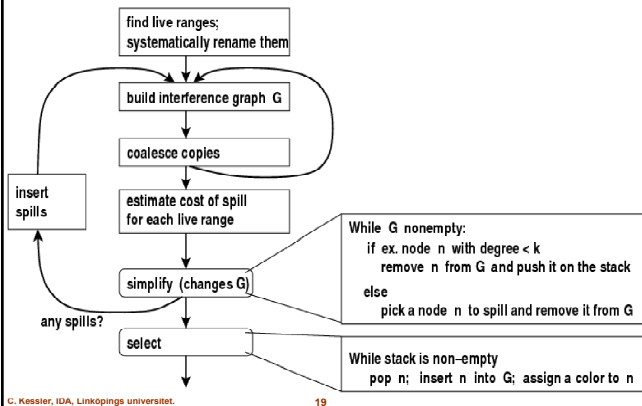
© reg-to-reg copy is much cheaper than memory spill

- Live range splitting = the reverse of coalescing

C. Kessler, IDA, Linköping universitet.

18

Chaitin's Register Allocator (1981)

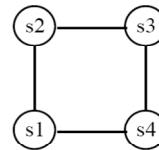


C. Kessler, IDA, Linköping universitet.

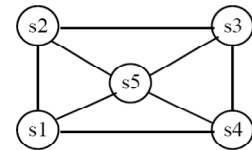
19

Improvement: Optimistic Graph Coloring

G may be colorable even if v has $\geq R$ neighbors



2-colorable
 but degree < 2 -rule creates a spill



3-colorable
 but degree < 3 -rule creates a spill

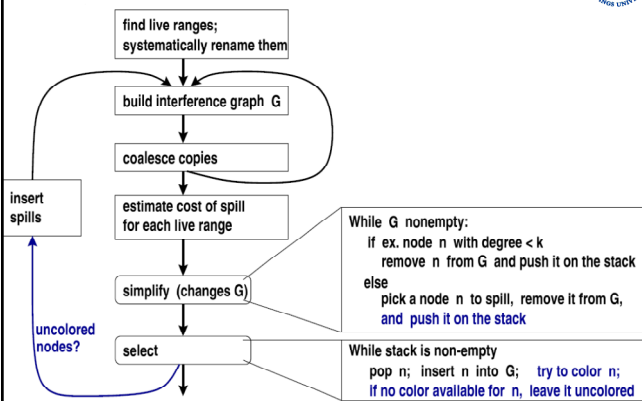
Optimistic coloring [Briggs'92]

- pick a node to spill and push it on the stack
- (postponing spilling decisions);
- proceed with degree $< R$ -rule, color remaining graph;
- reinsert the pushed node and try to color now.

C. Kessler, IDA, Linköping universitet.

20

Allocator with Briggs' improvement



C. Kessler, IDA, Linköping universitet.

21

Hierarchical Register Allocation

- find hierarchical structure (e.g., regions)
- color intervals bottom-up with Chaitin-style allocator, using local and global interference information
- propagate summary information from children to parents
- top-down pass assigns the registers and places spill code

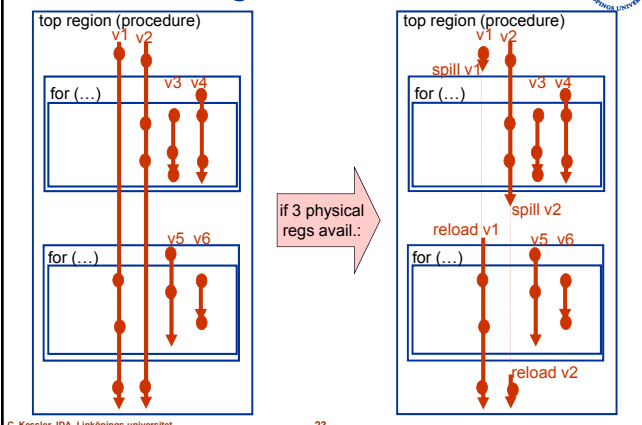
- + allocator is more sensitive to program structure
- + better placement of spill code (always placed outside a loop if possible)
- + smaller interference graphs considered at each step (the global interference graph is never built)

→ [Example]

C. Kessler, IDA, Linköping universitet.

22

Hierarchical Register Allocation



C. Kessler, IDA, Linköping universitet.

23

Two-Step Approach

Pre-Spilling phase

- Limit the remaining register pressure at any program point to the available number of physical registers
- Can attempt for optimal spilling →

Graph-Coloring phase

- Now easier to K -color

- Appel, George PLDI'01
- Ebner 2009

C. Kessler, IDA, Linköping universitet.

24

Optimal Spilling?



- Select those live ranges for spilling whose accumulated spill cost is minimal
- Optimal (pre-)spilling and a-posteriori insertion of spill code for given instruction schedule is NP-complete even for basic blocks
 - Dynamic programming
 - ▶ e.g. Horwitz *et al.* 1966
 - Integer Linear Programming
 - ▶ e.g. Appel, George PLDI 2001
 - Most compilers use (greedy) heuristics (see above)

C. Kessler, IDA, Linköpings universitet.

25

SSA-Based Register Allocation



- For SSA programs, the register interference graph is **chordal**
 - Can be K -colored in quadratic time!
 - ▶ Hack, Goos 2006
 - ▶ Bouchez *et al.* 2006
 - ▶ Brisk *et al.* 2009: Optimistic chordal coloring
 - Optimal coalescing in spill-free SSA programs
 - ▶ Brisk *et al.* 2009: heuristic
 - ▶ Grund, Hack 2007: Integer Linear Programming
 - Optimal pre-spilling in SSA programs
 - ▶ Ebner 2009: heuristic

C. Kessler, IDA, Linköpings universitet.

26

Fast Register Allocation



- For JIT compilers:
 - Compilation time critical (trade-off with code quality)
 - Linear-Scan Register allocators
 - ▶ Poletto, Sarkar TOPLAS 1999
 - ▶ Traub, Holloway, Smith PLDI 1998

C. Kessler, IDA, Linköpings universitet.

27

Interdependences Register Allocation \leftrightarrow Instruction Scheduling



- Determining live ranges requires a linear sequence of instructions (pre-scheduled MIR, LIR, or target code with symbolic registers)
- Spill code must be scheduled as well
 - may destroy quality of a beforehand good schedule
- ⇒ Integration of register allocation and instruction scheduling
 - quantitative evaluation [Bradlee *et al.*'91]
 - integrated approaches, space-aware scheduling
 - [Goodman/Hsu'88], [Freudenberger/Ruttenberg'92], [Pinter'93]
 - [Brasier *et al.*'95], [Motwani *et al.*'95], [Kästner'97,'00], ..., [K./Bednarski'01]

C. Kessler, IDA, Linköpings universitet.

28