Register Allocation

Register Allocation

Register Allocation: Determines values (variables, temporaries, constants) to be kept in registers.

Register Assignment: Determine in which physical register such a value should reside.

Essential for Load-Store Architectures
Reduce memory traffic (→ 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].

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.

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]

y = x3 + 42
All variables interfere with each other — need 4 regs?

Global Register Assignment by Graph Coloring

[Ershov'62][Cocke'71][Chatin et al.'81][Chatin'82]
[Chow/Hennessy'84, 90][Briggs et al.'89][Briggs'92]...
1. allocate objects that can be assigned to registers
to distinct symbolic registers a1, a2, ...
2. determine candidates for allocation to registers (ai/aweb)
3. build interference graph
   nodes: allocatable objects, target machine registers
   edges: (undir.) (ai,aj) iff allocatable objects ai, aj simultaneously live
   (ai, aj) iff ai should not reside in register aj
4. color nodes with k colors
   (k = Reassignable registers, such that any two adjacent nodes have different colors)
5. allocate each object to a register that has the same color.
Register Allocation by Graph Coloring

- **Step 1:** Given a program with symbolic registers s1, s2, ...
  - Determine live ranges of all variables

  i = c+4; load %fp, s1 ! c
  nop
  addi s1, $4, s2
  store s2,4(%fp) ! i

  d = c-2; subi s1, $2, s3
  store s3,12(%fp) ! d

  c = c+1; muli s1, s2, s4
  store s4,8(%fp) ! c

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

  i = c+4; load %fp, s1 ! c
  nop
  addi s1, $4, s2
  store s2,4(%fp) ! i

  d = c-2; subi s1, $2, s3
  store s3,12(%fp) ! d

  c = c+1; muli s1, s2, s4
  store s4,8(%fp) ! c

- **Step 3:** Color the register interference graph with k colors, where k = #available registers.
  - If not possible: pick a victim si to spill, generate spill code (store after def., reload before use)

  This may remove some interferences.
  Rebuild the register interference graph + repeat Step 3...

  i = c+4; load %fp, s1 ! c
  nop
  addi s1, $4, s2
  store s2,4(%fp) ! i

  d = c-2; subi s1, $2, s3
  store s3,12(%fp) ! d

  c = c+1; muli s1, s2, s4
  store s4,8(%fp) ! c

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

Coloring a graph with k colors

- NP-complete for k > 3
- **Chromatic number** χ(G) = minimum number of colors to color a graph G

  χ(G) ≥ 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) do:
    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.

Live range coalescing = fusion of webs

- For a copy instruction sj ← si
  - where si and sj do not interfere
  - and si and sj are not rewritten after the copy operation

- Merge si and sj:
  - patch (rename) all occurrences of si to sj
  - update the register interference graph
  - and remove the copy operation.

s2 ← ... s3 ← s2 ... s3 ← ... r1 ← ...
Conservative Coalescing

Coalescing two live ranges \( u \) and \( v \) can increase the node degree:
\[
d(u\&v) > \max\{d(u), d(v)\}
\]
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 degree-\( R \) heuristic

Spilling (1)

- **Spilling** a (physical) register \( r \)
  = spilling the live range \( w \) contained in \( r \)
  → uses some memory location \( w\).tmp
  (on stack, scratchpad memory, or \( w \)'s home memory location)
  → insert a Store \( r, w\).tmp
  immediately after each definition of \( w\).var
  → insert a Load \( r, w\).tmp
  immediately before each use of \( w\).var
- Some interferences disappear,
  the interference graph must be updated.

Spilling (2)

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

Minimize ratio  
\[
\frac{\text{spillcost}(w)}{\text{degree}(w)^2}
\]
etc.

\[
\text{spillcost}(w) = c_{\text{store}} \sum_{d(w)} 10^{\text{depth}(d(w))} + c_{\text{reg}} \sum_{d(w)} 10^{\text{depth}(d(w))} - c_{\text{copy}} \sum_{r \in \text{copy}} 10^{\text{depth}(r)}
\]

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

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

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]

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
Chaitin's Register Allocator (1981)

Improvement: Optimistic Graph Coloring

Hierarchical Register Allocation

Two-Step Approach
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)

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

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

Interdependences

Register Allocation ↔ 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 [Dredge et al. '01]
    - integrated approaches, space-aware scheduling
      [Goodman/Hsu '88, [Fuenzalida/Rutenberg '92, [Pinter '93], [Brasier et al. '95], [Motwani et al. '95], [Kastner '97, '00], ..., [K.Bednarski '01]