Register Allocation

## Register Allocation

- **Register Allocation**: Determines values (variables, temporaries, constants) to be kept in registers.
- **Register Assignment**: Determine which physical register such a value should reside in.
- **Essential for Load-Store Architectures**
- **Reduce memory traffic**: \( \text{\#memory / cache latency, energy} \)
- **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 \([\text{Ershov'62}] [\text{Cocke'71}]\).

### 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.

### Local Register Allocation

For variable \( v \) and basic block \( B_i \):
\[
\text{netavail}(v, i) = \text{netuse}(v) - \#\text{defined}_i(v)
\]
- \( l - \text{load} \) cost \( (l = 1 \text{ if Load}(v) \text{ needed at beg. of } B_i, 0 \text{ otherwise}) \)
- \( s - \text{store} \) cost \( (s = 1 \text{ if Store}(v) \text{ needed at end of } B_i, 0 \text{ otherwise}) \)

For loop \( L \) estimate benefit of keeping \( v \) in a register:
\[
\text{benefit}(v, L) = 1^{\text{pref}(v)} \sum \text{netavail}(v, l)
\]
with \( R \) registers available:
- allocate the \( R \) objects \( v \) with greatest benefit in \( L \)
- moves may be necessary instead of \( \text{Load}(v) / \text{Store}(v) \)
- if \( v \) could reside in \( R \) different registers in \( \text{Pref}(v) \), \( \text{Store}(v) \)
- add worst-case terms \( |\text{Pref}(v)| \cdot \text{mcost}, |\text{Store}(v)| \cdot \text{mcost} \)

---

### Global Register Assignment by Graph Coloring

\([\text{Ershov'62}] [\text{Cocke'71}] [\text{Chaitin et al'81}] [\text{Chaitin'82}] [\text{Chow/Hennessey'84, 90}] [\text{Briggs et al'39}] [\text{Briggs'92}]\)

1. allocate objects that can be assigned to registers to distinct symbolic registers \( a_1, a_2, ... \)
2. determine candidates for allocation to registers (\( a / i \) 'webs')
3. build interference graph
   - nodes: allocatable objects, target machine registers
   - edges: (undir.) \( (a_i, a_j) \) if 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 \( R = \#\text{available registers} \)
   - such that any two adjacent nodes have different colors
5. allocate each object to a register that has the same color.
Allocatable objects: Web (Live ranges)

- Live ranges instead of variable names => less constraints, less interference
- Each web is equivalent to a symbolic register
- Easy to determine from SSA form: each SSA-form variable is head of a DU-chain

Register Allocation by Graph Coloring

1. Given a program with symbolic registers $s_1, s_2, ...$
   - Determine live ranges of all variables

   ```
   i = c+4; load 8(fp), $s_1$ ! c
   nop
   addl $s_1$, $s_2$, $s_3$
   store $s_2$, 4(fp) ! i
   
   d = c-2; subl $s_1$, $s_2$, $s_3$
   store $s_2$, 12(fp) ! d
   
   c = c+1; muli $s_1$, $s_2$, $s_4$
   store $s_4$, 8(fp) ! c
   ```

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 $s_i$

3. Color the register interference graph with $k$ colors where $k = \#available registers$
   - 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...

   ```
   i = c+4; load 8(fp), $s_1$ ! c
   nop
   addl $s_1$, $s_4$, $s_2$
   store $s_2$, 4(fp) ! i
   
   d = c-2; subl $s_1$, $s_2$, $s_3$
   store $s_2$, 12(fp) ! d
   
   c = c+1; muli $s_1$, $s_2$, $s_4$
   store $s_4$, 8(fp) ! c
   ```

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):
  - If $s$ has less than $k$ neighbors in the graph
    - There will be some color left for $s$
  - Else modify the graph (spill, coalesce... nodes) and restart.

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.

```
  s2 \leftarrow ...
  ...
  s3 \leftarrow s2
  ...
  s3 \leftarrow ...
  ...
  r1 \leftarrow ...
```
Conservative Coalescing

Coalescing two live ranges \( u \) and \( v \) can increase the node degree:
\[ d(u \& v) > \max(d(u), d(v)) \]
is possible
\[ \rightarrow \] may make \( u \& v \) harder to color

Conservative coalescing:
\[ \text{coalesce only if } d(u \& v) < R \]
\[ \rightarrow \text{can color } u \& v \text{ by the naive degree } < R \text{ heuristic} \]

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]

\[ \text{minimize ratio } \frac{\text{spillcost}(w)}{\text{degree}(w)} \text{ etc.} \]

\[ \text{spillcost}(w) = c_{\text{store}} \sum_{\text{def}(w)} \text{store} + c_{\text{load}} \sum_{\text{use}(w)} \text{load} - c_{\text{reg-use}} \sum_{\text{reg-use}(w)} \text{reg-use} \]

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

- spill value once per block if possible
  \[ \rightarrow \text{avoids redundant loads and stores} \]
- 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.

\[ \begin{array}{c}
  \text{l} \quad \text{l} \quad \text{l} \quad \text{l} \\
  \text{l} \quad \text{l} \quad \text{l} \quad \text{l} \\
  \text{l} \quad \text{l} \quad \text{l} \quad \text{l} \\
  \text{l} \quad \text{l} \quad \text{l} \quad \text{l} \\
  \text{l} \quad \text{l} \quad \text{l} \quad \text{l} \\
  \text{l} \quad \text{l} \quad \text{l} \quad \text{l} \\
  \end{array} \\
\text{if a spilled value is used several times} \]

\[ \text{and the restored value remains live for several adjacent uses,} \]

\[ \text{a Load/Rematerialize is necessary only before the first of them.} \]

\[ (= \text{ live range splitting}) \quad \text{[Chow/Hennessy84]} \]

Live Range Splitting

- Long live ranges tend to interfere with many others
  \[ \rightarrow \text{harder to color} \]
- Idea: Split up long live ranges to avoid some spilling
  \[ \text{reg-to-reg copy is much cheaper than memory spill} \]
- Live range splitting = the reverse of coalescing
Chaitin's Register Allocator (1981)

find live ranges; systematically remove them

build interference graph \( G \)

coalesce copies

estimate cost of spill for each live range

While \( G \) nonempty:
- if node \( n \) with degree \( k \) in \( G \), remove \( n \) and push \( n \) to the stack
- else pick a node \( n \) to spill and remove \( n \) from \( G \)

simplify (changes \( G \))

select

while stack non-empty
- pop \( n \)
- insert \( n \) into \( G \); assign a color to \( n \)

Improvement: Optimistic Graph Coloring

\( G \) may be colorable even if \( \chi \geq R \) 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 \(< \beta\)-rule, color remaining graph
- reinstall the pushed node and try to color now

Allocator with Briggs’ improvement

find live ranges; systemically remove them

build interference graph \( G \)

coalesce copies

estimate cost of spill for each live range

While \( G \) nonempty:
- if node \( n \) with degree \( k \) in \( G \), remove \( n \) and push \( n \) to the stack
- else pick a node \( n \) to spill and remove \( n \) from \( G \) and push it on the stack

simplify (changes \( G \))

select

While stack non-empty
- pop \( n \)
- insert \( n \) into \( G \); try to color \( n \)
- if no color available for \( n \), leave \( n \) uncolored

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)

Interdependences

Register Allocation \(\iff\) 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 [Bradley et al '91]
    - integrated approaches, space-aware scheduling [Goodman/Hsu '88, Freudenberger/Rutenberg '92, Pinter '93, Brasier et al '95, Motwani et al '95, Kastner 97.00, ...[K. Bednarski '01]