Register Allocation III

Announcements
- Recommend have interference graph construction working by Monday

Last lecture
- Register allocation across function calls

Today
- Register allocation options

Interference Graph Allocators

Chaitin
- build
- coarsegrain
- split costs
- simplify
- select

Briggs
- PROVIDE
- build
- coarsegrain
- split costs
- simplify
- select

Granularity of Allocation (Renumber step in Briggs)

What is allocated to registers?
- Variables/Temporaries
- Live ranges/Webs (i.e., du-chains with common uses)
- Values (i.e., definitions; same as variables with SSA)

Variables: 2 (x & y)
Live Ranges/Webs: 3 (t1 → t2, t3; t2 → t3)
Values: 4 (t3, t2, t1, φ (t3,t1))

What are the tradeoffs?
Each allocation unit is given a symbolic register name (e.g., s1, s2, etc.)

Computing the Interference Graph (in MiniJava compiler)

Use results of live variable analysis

for each flow graph node n do
  for each def in def(n) do
    for each temp in liveout(n) do
      if (not stim(n) isa MOVE or def != temp) then
        E ← E ∪ (def, temp)
Coalescing

Move instructions
- Code generation can produce unnecessary move instructions
  MV t1, t2
- If we can assign t1 and t2 to the same register, we can eliminate the move

Idea
- If t1 and t2 are not connected in the interference graph, coalesce them into a single variable

Problem
- Coalescing can increase the number of edges and make a graph uncolorable
- Limit coalescing to avoid uncolorable graphs

Coalescing Logistics

Rule
- When building the interference graph, do NOT make virtual registers interfere due to copies.
- If the virtual registers s1 and s2 do not interfere and there is a copy statement s1 = s2 then s1 and s2 can be coalesced.
- Example

Coalescing in MiniJava compiler

Currently the InterferenceGraph only has one Temp.Temp associated with each node
- represent each merged node with just one of the temps
- keep a separate map of representatives mapped to sets of temps
- also keep a map of temps mapped to their representative
- when rewriting the code use the representative instead of the original temp

Register Allocation: Spilling

If we can’t find a k-coloring of the interference graph
- Spill variables (nodes) until the graph is colorable

Choosing variables to spill
- Choose least frequently accessed variables
- Break ties by choosing nodes with the most conflicts in the interference graph
- Yes, these are heuristics!
Weighted Interference Graph

Goal
- Weight(s) = \sum_{r \in \text{schedule}, s \in r} f(r), f(r) is execution frequency of \( r \)

Static approximation
- Use some reasonable scheme to rank variables
  - One possibility
    - Weight(s) = 1
    - Nodes after branch: \( \frac{1}{2} \) weight of branch
    - Nodes in loop: \( 10 \times \) weight of nodes outside loop

The Problem: Worst Case Assumptions

Is the following graph 2-colorable?

Clearly 2-colorable
- But Chaitin’s algorithm leads to an immediate block and spill
- The algorithm assumes the worst case, namely, that all neighbors will be assigned a different color

Improvement #1: Simplification Phase [Chaitin 81]

Idea
- Nodes with < \( k \) neighbors are guaranteed colorable
  - Improvement over simple greedy coloring algorithm

Remove them from the graph first
- Reduces the degree of the remaining nodes

Must spill only when all remaining nodes have degree \( \geq k \)

Referred to as pessimistic spilling

Improvement #2: Optimistic Spilling [Briggs 89]

Idea
- Some neighbors might get the same color
  - Nodes with \( k \) neighbors might be colorable
- Blocking does not imply that spilling is necessary
  - Push blocked nodes on stack (rather than place in spill set)
  - Check colorability upon popping the stack, when more information is available

Defer decision
**Algorithm** [Briggs et al. 89]

\[
\text{while } \text{interference graph not empty do} \\
\text{while } \exists \text{ a node } n \text{ with } < k \text{ neighbors do} \\
\text{Remove } n \text{ from the graph} \\
\text{Push } n \text{ on a stack} \\
\text{if any nodes remain in the graph then} \\
\text{Pick a node } n \text{ to spill} \\
\text{Push } n \text{ on stack} \\
\text{Remove } n \text{ from the graph} \\
\text{while stack not empty do} \\
\text{Pop node } n \text{ from stack} \\
\text{if } n \text{ is colorable then} \\
\text{Allocate } n \text{ to a register} \\
\text{else} \\
\text{Insert spill code for } n \text{ } \text{ } \{ \text{Store after def; load before use } \} \\
\text{Reconstruct interference graph & start over}
\]

---

**Possible Register Allocation Design**

**Overall algorithm:** graph coloring with simplification

**Interference graph:** two temps interfere if

- one is defined in a stmt and the other is live out of the same stmt
- exception is a MOVE statement where the temps are the source and dest

**Coalesce:** Briggs strategy

- coalesce if new node will have fewer than K neighbors of significant degree (>= K)

**Spill heuristic:**

- spill the node with the lowest weight and break ties by spilling the node with the most adjacent edges

**Simplification:** optimistic

**Select:**

- pop everything off the stack before generating spill code

---

**Example**

Attempt to 2-color this graph (   ,   )

![Graph with nodes a, b, c, d, e, f and edges between them]

- Stack: d c b* f* a* e*}
- Weighted order: e a f b c d

* blocked node

---

**Improvement #3: Live Range Splitting** [Chow & Hennessy 84]

**Idea**

- Start with variables as our allocation unit
- When a variable can’t be allocated, split it into multiple subranges for separate allocation
- Selective spilling: put some subranges in registers, some in memory
- Insert memory operations at boundaries

**Why is this a good idea?**
Improvement #4: Rematerialization [Chaitin 82] & [Briggs 84]

Idea
- Selectively re-compute values rather than loading from memory
- "Reverse CSE"

Easy case
- Value can be computed in single instruction, and
- All operands are available

Examples
- Constants
- Addresses of global variables
- Addresses of local variables (on stack)

Next Time

Lecture
- Instruction scheduling