Register Allocation

March 19, 2001

CS 132: Compiler Design

Review: Liveness

- A variable is said to be *live* at a program point if its contents may affect the result of the program after that point.
  - And said to be *dead* otherwise.
  - True liveness is an undecidable property
- Approximation: variable is (statically) dead if its current value can never be read
  - And said to be (statically) live otherwise.

Review: Liveness Analysis

- For each instruction s:
  - Define $def(s)$ to be variables potentially modified by s.
  - Define $use(s)$ to be the variables accessed by s.
  - Solve for $livein(s)$ and $liveout(s)$ (e.g., by iteration)

$$livein(s) = use(s) \cup (liveout(s) \setminus def(s))$$

$$liveout(s) = \bigcup_{p \in succ(s)} livein(p)$$

$$liveout(terminal\ node) = \varnothing$$

Register Allocation

- Assigns temporaries to machine locations
  - Input: Assembly code using arbitrary number of *temporaries.*
  - Output: Equivalent code using only machine registers.
- Note: For the rest of the day I will refer to machine registers as temporaries as well
  - These are assigned to themselves by the allocator
- Note: Sometimes a distinction is made between register allocation (which temporaries are stored in registers) and register assignment (which registers these are).
  - Today's algorithm does these simultaneously

Interference Graph

- Temporaries are said to *interfere* if they cannot be stored in the same machine location.
- The *interference graph* for a piece of code is defined as follows:
  - Nodes: the temporaries and machine registers used by the code
  - Edges: there is an edge between every pair of nodes that interfere.

Graph Coloring

- Definitions:
  - A *coloring* of an undirected graph is an assignment of "colors" to nodes such that no two adjacent nodes have the same color.
  - A k-*coloring* of a graph is a coloring that uses at most k colors.
- Given a graph and k, is there a k-coloring of this graph? (And if so, what is it?)
  - Called the *graph coloring problem*

Graph Coloring Register Allocation

- Elegant observation (Chaitin):
  - The result of a correct register allocation would be an assignment of registers to temporaries such that no interfering temporaries are assigned the same register.
  - This is exactly the graph coloring problem applied to the interference graph
    - Each color corresponds to a machine register!

Building a Graph-Coloring Register Allocator

- Step 1: Build the interference graph.
- Step 2: Find a k-coloring, where k is the number of available machine registers.
- Problems:
  - The graph coloring problem is NP-complete ($k>2$)
  - What if the graph isn't k-colorable?

Solutions

- Although graph coloring is NP-complete, there are reasonable heuristics.
- If the heuristic fails to k-color the graph, we can try finding equivalent code with an easier interference graph.

Spilling

- A temporary that is stored in memory (e.g., the stack frame) instead of a register is said to have been *spilled*.
- Spilling transformation:
  - Choose a problematic temporary `t` and a memory location M for it.
  - Rewrite each use of `t` to first load from M into a fresh temporary, and to use this temporary instead.
  - Rewrite each definition of `t` to write to a fresh temporary and to store it into M.
- Gets rid of the temporary `t`, at the cost of adding new temporaries (with very short live ranges).

Spilling Example

```
a ← x+1
b ← x+2
x ← a+b
```

spill x

```
t1 ← *(%fp-8)
a  ← t1+1
t2 ← *(%fp-8)
b  ← t2+2
t3 ← a+b
*(%fp-8) ← t3
```

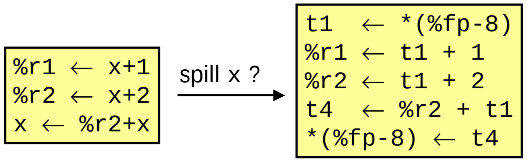
Spilling Example

```
%r1 ← x + 1
%r2 ← x + 2
x ← %r2 + x
```

spill x

```
t1  ← *(%fp-8)
%r1 ← t1 + 1
t2  ← *(%fp-8)
%r2 ← t2 + 2
t3  ← *(%fp-8)
t4  ← %r2 + t3
*(%fp-8) ← t4
```

Spilling Example



```
%r1 ← x+1
%r2 ← x+2
x ← %r2+x
```

spill x ?

```
t1  ← *(%fp-8)
%r1 ← t1 + 1
%r2 ← t1 + 2
t4  ← %r2 + t1
*(%fp-8) ← t4
```

Interference

- Which temporaries interfere?

```
a ← x+1
b ← a+x
return b
```

Interference Graph Construction

- Algorithm (first try).
  - Put edges between any pair of temporaries that appear simultaneously in a *livein* (or *liveout*) set for some program point.
- Problem:
  - This is insufficient (and inefficient if implemented naively)

Interference

- Which temporaries interfere?

```
a ← x+1
b ← a+x
return a
```

Interference

- Which temporaries interfere?

```
a ← x+y
b ← f(x,3)
c ← a+1
```

Interference Graph Construction

- Algorithm (second try).
  - For each instruction *s*, put edges between every node in *def*(*s*) and every temporary in *liveout*(*s*)
    - Ignore self-loops.
- Problem:
  - Correct, but now overly conservative.

Interference

- Which temporaries interfere?

```
a ← x+y
b ← a
c ← a+x
d ← b+c
return d
```

Interference

- Which temporaries interfere?

```
a ← x+y
b ← a
c ← a+x
b ← b+1
d ← a+c
return d
```

Interference Graph Construction

- Final algorithm: For each instruction *s*:
  - If *s* is a move instruction c ← a then add edges between c and every member of *liveout*(*s*) except a.
  - Otherwise, add an edge between every node in *def*(*s*) and every temporary in *liveout*(*s*).

Graph Coloring Algorithms

- Naive algorithm for *k*-coloring a graph
  - Put the graph nodes into a sequence
  - Iterate through this sequence, assigning each node a color that does not introduce an immediate conflict in the graph.
    - i.e., choose a color different from that of any neighbors who have already been assigned colors.
  - Backtrack when we reach a node that cannot be assigned a color
    - i.e., because it has neighbors of every possible color.

Graph Coloring Algorithms

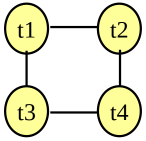
- Observation
  - Let G be the graph to be k-colored.
  - Assume that node n has degree $< k$.
  - Let G' be the graph G after n and adjacent edges are removed
  - Then G' is k-colorable if and only if G is k-colorable! (Why?)
- *Simplification*
  - Removal of low-degree ($< k$) nodes from interference graph.
  - Note: Removing a node decreases the degrees of its neighbors, possibly creating more low-degree nodes.

Chaitin's Heuristic

- Given the graph G:
  - If G contains a node n of degree < k:
    - Remove it to get G'
    - Recursively color G'
    - Find a non-conflicting color for n (always possible).
  - If G contains only high-degree nodes:
    - Pick high-degree node n to be spilled
    - Remove n from the graph and recurse
    - At end, rewrite code for all spilled temporaries, and re-run register allocation (Why?)
    - (Alternative: Abort allocation as soon as first temporary is spilled; rewrite code and re-run register allocation)

Briggs' Optimistic Coloring

- The following graph can be 2-colored, but Chaitin's algorithm will spill one of the temporaries.



- Observation: Just because a node has many neighbors doesn't mean it can't be colored.
  - If several neighbors turn out to have the same color, the node doesn't need to be spilled.

Briggs' Optimistic Coloring

- *Simplify* routine builds a stack of nodes from graph G.
  - If G contains a node n of degree < k:
    - Put n onto the stack, remove it from the graph, and recurse
  - If G contains only high-degree nodes:
    - Pick high-degree node n (spill candidate)
    - Put n onto the stack, remove it from the graph, and recurse
- *Select* routine pops the nodes of the stack and assigns them colors in order.
  - Only mark node to be spilled if its neighbors do use all the available colors.
  - At end, if any nodes must be be spilled, rewrite graph and re-run the register allocator.

Spilling Heuristics

- What are good temporaries to spill?
  - Temporaries that don't appear much in the code
    - To minimize size penalty of spill-code
  - Temporaries that aren't used much at run-time
    - To minimize time penalty of spill-code
  - Temporaries that interfere with many nodes
    - To maximize the effect of their removal on the interference graph
  - NB: Never want to spill temporaries introduced by spill code!