Lecture 15: Register allocation
23 Mar 2010

Nate Nystrom
University of Texas at Arlington
PA 2 due Friday
Register allocation

We’ll discuss two algorithms today

- Graph-coloring algorithm
- Linear scan algorithm
Liveness analysis

Compute the set of temporaries *live* at each program point

A variable is *live* if it might be used in the future

Liveness analysis is an instance of a *dataflow* problem
Control-flow graphs

```c
int max(int x, int y) {
    int z;
    if (x > y)
        z = x;
    else
        z = y;
    return z;
}
```

For now: each node is a single instruction.
int max(int x, int y) {
    int z;
    if (x > y)
        z = x;
    else
        z = y;
    return z;
}

The live range of a variable extends from its last use to its definition.
int max(int x, int y) {
    int z;
    if (x > y)
        z = x;
    else
        z = y;
    return z;
}

Note: important to distinguish between LHS and RHS of MOVE instructions: z and x are not live at the same time!
Set-based algorithm

for each n:

\[ \text{in}(n) = \{\} \]
\[ \text{out}(n) = \{\} \]

repeat:

for each n:

\[ \text{in}'(n) = \text{in}(n) \]
\[ \text{out}'(n) = \text{out}(n) \]
\[ \text{in}(n) = \text{use}(n) \cup (\text{out}(n) - \text{def}(n)) \]
\[ \text{out}(n) = \bigcup_{s \in \text{succ}(n)} \text{in}(s) \]

until \[ \text{in}'(n) = \text{in}(n) \text{ and } \text{out}'(n) = \text{out}(n) \text{ for all } n \]
Outline

Control-flow analysis to build CFG

Liveness analysis to compute live ranges for each temporary
- liveness analysis is a *dataflow analysis*
- dataflow analyses are also used for optimization
Graph coloring register allocation

Use live ranges to construct an *interference graph*

- An undirected graph
  - Node for each variable
  - Edges connect variables whose live ranges overlap

Color the graph with 1 color/register

Adjacent nodes cannot have the same color
int max(int x, int y) {
    int z;
    if (x > y)
        z = x;
    else
        z = y;
    return z;
}

Interference graph
## Interference graph

### Live ranges

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
<th>t</th>
<th>u</th>
</tr>
</thead>
<tbody>
<tr>
<td>x ← 1</td>
<td>x</td>
<td>y</td>
<td></td>
<td>t</td>
<td>u</td>
</tr>
<tr>
<td>y ← 2</td>
<td></td>
<td>y</td>
<td>z</td>
<td></td>
<td>t</td>
</tr>
<tr>
<td>z ← x+y</td>
<td>z</td>
<td></td>
<td></td>
<td></td>
<td>t</td>
</tr>
<tr>
<td>t ← y</td>
<td>x</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>u ← x+t</td>
<td>u</td>
<td>x</td>
<td>y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>print z</td>
<td></td>
<td></td>
<td></td>
<td>z</td>
<td></td>
</tr>
<tr>
<td>print t</td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>print u</td>
<td></td>
<td></td>
<td></td>
<td>t</td>
<td>u</td>
</tr>
</tbody>
</table>

### Interference graph

```
\begin{tikzpicture}
    \node (x) at (0,0) {$x$};
    \node (y) at (1,0) {$y$};
    \node (z) at (1,1) {$z$};
    \node (t) at (0,1) {$t$};
    \node (u) at (2,0) {$u$};
    \draw (x) -- (y);
    \draw (x) -- (t);
    \draw (y) -- (z);
    \draw (z) -- (u);
\end{tikzpicture}
```
Register allocation can be viewed as a *graph coloring* problem.

Assign a register (color) to each node in the graph so that no two adjacent nodes have the same color.
4-color theorem

Theorem: any planar map (graph!) can be colored with 4 colors

Useful? Not really. Interference graphs aren’t necessarily planar
Can we always find a coloring for a given graph?

Can we efficiently find the optimal coloring?

Can we assign registers to avoid move instructions?

What to do if there aren’t enough colors?
Coloring a graph

Assume $K = \text{number of registers}$ (let $K = 3$)

Try to color graph with $K$ colors

Key heuristic = **Simplify**

- find some node with at most $K-1$ edges, and remove it from the graph
Once coloring is found for simplified graph, 
Color the removed node with a free color 

Algorithm: 
- simplify until there are no more nodes 
- unwind, adding nodes back and assigning colors
Recursive algorithm

\[
\text{color}(G) \{ \\
\quad \text{find node } v \text{ with } K-1 \text{ edges} \\
\quad \text{color}(G - v) \\
\quad \text{choose color } c \text{ such that for all } (v,w) \text{ in } G, \ w \text{ is not colored } c \\
\quad \text{color } v \text{ with } c \\
\}\]
Example
Example
Example
Example

a
Example

a

b
d
Example
Another example
Another example
Another example
Another example

Cannot simplify this graph when $K = 3$
Failure

If graph cannot be colored, it will simplify to a graph where all nodes have $\geq K$ neighbors

This can happen even if the graph is colorable with $K$ colors!

Finding $K$-coloring is NP-hard for $K \geq 3$
Spilling

Once all nodes have \( \geq K \) neighbors
Pick a node and mark it for spilling
Remove from the graph
Color the remainder of the graph

Try to spill variables that aren’t used much
- not in inner loop
Optimistic coloring

Spilled node may still be K-colorable

Try to color after coloring the simplified graph

If not colorable, **actual spill**: assign to a location on the stack
Accessing spilled variables

Need to generate additional instructions to move spilled variables to/from stack when used

Naive: always reserve extra registers for moving data back and forth

Better: rewrite code using a new temporary, rerun liveness, regalloc
Rewriting code

Example:

- add v1 = v1 + v2

Suppose v2 is selected for spilling to location [fp-24]

Add new variable t35 for **just this instruction**:

- load t35 = -24(fp)
- add v1 = v1 + t35

Rerun liveness and register allocation.

t35 has a short lifetime, and doesn’t interfere with other variables as much as v2 did
Putting it together

color(G) {
    find node v with <= K-1 edges
    if (v with K-1 edges not found)
        choose node v that isn’t used much // potential spill
    color(G - v)
    choose color c such that for all (v,w) in G, w is not colored c
    if (v not colorable)
        spill v
    else
        color v with c
}
Precolored nodes

Some variables need to be preassigned to registers

mul instruction on x86

- uses eax, defines both eax and edx

call instruction on x86

- defines eax, ecx, edx (callee-saves registers)

Treat these register uses as special temporaries

Precolor nodes in interference graph before coloring
Precolored nodes

Cannot simplify by removing a precolored node

Once simplified graph is all colored nodes, add temporaries back in
Optimizing moves

Code generation produces a lot of extra moves:

- `mov t5 = t9`

If we assign t5 and t9 to same register, can eliminate the mov

Idea: if t5 and t9 not connected in the interference graph, coalesce them into a single node—the mov becomes redundant
Coalescing

When coalescing nodes, take union of their edges

=> coalescing creates high-degree nodes
=> can make graph uncolorable
Coelescing example

\[
x \leftarrow 1 \\
y \leftarrow 2 \\
z \leftarrow x+y \\
t \leftarrow y \\
u \leftarrow x+t \\
\text{print } z \\
\text{print } t \\
\text{print } u
\]

coalescing of \(y\) and \(t\)

\[
x \leftarrow 1 \\
yt \leftarrow 2 \\
z \leftarrow x+yt \\
u \leftarrow x+yt \\
\text{print } z \\
\text{print } yt \\
\text{print } u
\]
Conservative coalescing

Ensure coalescing doesn’t make the graph uncolorable

Approach 1: coalesce only if resulting node has < K neighbors

Approach 2: coalesce a and b if every neighbor t of a, already interferes with b or has insignificant degree
Simplification + coalescing

Let $M =$ set of move-related nodes
Let $N =$ all other nodes

Simplify as much as possible with nodes in $N$

Coalese some pairs of nodes in $M$
  - gives more opportunities to simplify

If cannot either simplify or coalese, move node in $M$ to $N$
If $M$ is empty and cannot simplify, spill
Linear scan

Graph coloring algorithm can be slow
Not suitable for dynamic compilation, JIT compilers

Linear scan algorithm
- not based on graph coloring
- M. Poletto and V. Sarkar 1999
Definitions

Number instructions in some order, e.g., DFS

Live interval \([i,j]\) for variable \(v\) if \(v\) is not live at any instruction outside \([i,j]\)

\([1,N]\) trivial live interval for every variable

Note: live intervals may be larger than live ranges
Live ranges might be disjoint because of jumps
Preliminaries

Do liveness analysis

Compute live intervals in one pass

Interference between live intervals = overlap
Scan live intervals in order of increasing start point

- if $< K$ live variables at start point, assign to a register
- else spill variable whose interval has latest end point

Latest end point is a heuristic
- produces code with minimal possible number of spilled live ranges
Example

Live ranges

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Allocation

<table>
<thead>
<tr>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>b</td>
</tr>
<tr>
<td>d</td>
<td>e</td>
</tr>
<tr>
<td>d</td>
<td></td>
</tr>
</tbody>
</table>

c is spilled
Linear scan is very simple, very fast:

- About 2-3 times faster than graph coloring

But doesn’t do as good a job:

- Code runs within 10% of graph coloring code
Fast live intervals

Compute SCCs of the flow graph
  - one DF pass

Traverse SCCs
  - live interval of $v = [i,j]$ smallest and largest DFNs of any SCC that uses $v$

Resulting code is 2-8x slower than (good) dataflow algorithm
Questions?