Compiler Design

Lecture 17: Graph Colouring Register allocation (EaC§13)

Christophe Dubach
Winter 2021

Timestamp: 2021/03/18 09:30:57
Main idea

1. Build an *interference graph* (a.k.a. “conflict” graph)
   - Nodes = variables (virtual registers)
   - Edges = overlapping live ranges

2. Find a *k-colouring of the graph*
   - Colours = architectural registers
Interference graph

What is an interference graph? (also called conflict graph)

- Two values interfer if there exists a point in the program where both are simultaneously live
- If $x$ and $u$ interfer, they cannot occupy the same register

To compute interferences, we must know where values are live

- $\Rightarrow$ result of liveness analysis

Interference graph $G$

- Nodes in $G$ represents variables (or virtual registers)
- Edges in $G$ represents interference between two variables (or virtual registers)
A graph $G$ is *k-colourable* iff the nodes can be labelled (or colored) such that no edge in $G$ connects two nodes with the same label (or color).

**Examples:**

2-colourable

3-colourable

If we can find a $k$-colouring of the interference graph, then all the nodes (variables) with the same colour can share the same architectural register, assuming at least $k$ registers available.
Back to the main idea

1. Build an interference graph

2. Find a k-colouring of the graph
1. Building interference graph

Pseudo-assembly:

```
a = 0
L1: b = a + 1
c = c + b
a = b*2
if (a<9) goto L1
return c
```

Control flow graph:

Liveness:

<table>
<thead>
<tr>
<th>node</th>
<th>out</th>
<th>in</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>ac</td>
<td>ac</td>
</tr>
<tr>
<td>4</td>
<td>ac</td>
<td>bc</td>
</tr>
<tr>
<td>3</td>
<td>bc</td>
<td>bc</td>
</tr>
<tr>
<td>2</td>
<td>bc</td>
<td>ac</td>
</tr>
<tr>
<td>1</td>
<td>ac</td>
<td>c</td>
</tr>
</tbody>
</table>

Interference graph:
2. Graph colouring and register mapping

Graph colouring:

Virtual to architectural registers

Possible mapping:

- a → $t0$
- b → $t0$
- c → $t1$

(pseudo-)assembly final code:

```
$t0 = 0
L1: $t0 = $t0 + 1
 $t1 = $t1 + $t0
 $t0 = $t0*2
 if ($t0<9) goto L1
 return $t1
```

Job done! Or is it?
Challenges

- Graph colouring is NP-complete
  - Complexity is exponential
  - We don’t like such algorithms in our compilers!

- It might not be possible to colour a graph with k colours.
  - Need alternative strategy in these cases
Heuristic for Graph Colouring
Observations

Suppose we have $k$ architectural registers (or $k$ colours):

- Any vertex $n$ that has fewer than $k$ neighbours in the interference graph ($\text{degree}(n) < k$) can always be coloured!
- In such case, pick any colour not used by its neighbours — there must be one!
Sketch of an algorithm

- Pick any vertex $n$ such that $\text{degree}(n) < k$ and put it on the stack
- Remove that vertex $n$ and all connected edges from the graph
  - This may make some new nodes have fewer than $k$ neighbours
- In the end, if some vertex $n$ still has $k$ or more neighbours, then spill the variable associated with $n$ to memory
- Otherwise successively pop vertices off the stack and colour them in the lowest colour not used by some neighbour
Chaitin’s Algorithm (1982!)

1. While ∃ vertices with < k neighbours in G
   - Pick any vertex n such that degree(n) < k and put it on the stack
   - Remove that vertex and all connected edges from G
   - This will lower the degree of n’s neighbours

2. If G is non-empty (all vertices have k or more neighbours) then:
   - Pick a vertex n (using some heuristic) and spill the variable associated with n
   - Remove vertex n from G, along with all connected edges
   - If this causes some vertex in G to have fewer than k neighbours, then go to step 1; otherwise, repeat step 2

3. Successively pop vertices off the stack and colour them in a colour not used by the neighbours
Example with 3 registers
Example with 3 registers
Example with 3 registers

3 Registers

Stack

2
1

3
4
5
Example with 3 registers

3 Registers

Stack

4
2
1

3

5
Example with 3 registers

Stack

Colors:
1: 
2: 
3:  

5
3
4
2
1
Example with 3 registers

3 Registers

Stack

Colors:
1: ∅️
2: ∅️
3: ∅️
Example with 3 registers
Example with 3 registers
Example with 3 registers

3 Registers

Stack

Colors:
1: 
2: 
3: 

Diagram showing connections between 1, 2, 3, 4, and 5.
Example with 3 registers
Register Spilling
If it is not possible to find a k-colouring of the graph, we need to spill some variables in memory.

The idea is to map some variable to memory rather than register

- this is what our naive register allocator is doing (for all variables!)

(Other approaches are also possible (e.g. splitting live ranges) but this is the subject of a compiler optimization course.)
Choice of variable to spill

Choosing which variable to spill is critical for performance:

- extra load instructions for every use of the variable
- extra store instructions for every def of the variable.

The compiler should use a cost-benefit analysis to decide which variable to spill depending on:

- how often the variable is used/defined?
- how many other variables interfere with the variable?
- is the variable used in a loop?

For your project, simply pick the variable with highest connectivity as it is likely to increase the chances that the graph becomes k-colourable.
Spilling a variable requires a register

Original code with virtual registers:

```plaintext
... 
add v0, v1, v2 
... 
```

After register allocation, assuming we spill v1:

```plaintext
v1: .space 4 
...
lw $t0, v1 
add $t3, $t0,$t2 
... 
```

We have a bit of a 🐤 & 🐚 situation: spilling v1 uses a register!

However, the live range of the register used for spilling is very short!

⇒ it is not so bad.
Two possible solutions:

- **Naive approach**: reserve a set of registers just for spilling purpose (e.g. \{\$t0\}) and never use them for anything else
  - maximum number of such registers needed = maximum number of registers an instruction can use/def (three for MIPS)

- **Better approach**: every time a variable needs to be spilled, stop the register allocation process, and replace all the occurrences of the spilled variable with a load/store instruction that uses a virtual register. Then re-run everything:
  - liveness analysis
  - interference graph construction
  - register allocation

source: ShadowThrust at Deviant Art, CC BY-SA 3.0
Next lecture

- Instruction selection