Lecture 39: Register Allocation

[Adapted from notes by R. Bodik and G. Necula]

Topics:

• Memory Hierarchy Management
• Register Allocation:
  - Register interference graph
  - Graph coloring heuristics
  - Spilling
• Cache Management
The Memory Hierarchy

Computers employ a variety of memory devices, trading off capacity, persistence, and speed (some years ago):

<table>
<thead>
<tr>
<th>Device</th>
<th>Access time</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>1 cycle</td>
<td>256-2000 bytes</td>
</tr>
<tr>
<td>Cache</td>
<td>1-100 cycles</td>
<td>256KB-16MB</td>
</tr>
<tr>
<td>Main memory</td>
<td>150-1000 cycles</td>
<td>32MB — &gt;16GB</td>
</tr>
<tr>
<td>Disk</td>
<td>0.5-10M cycles</td>
<td>10GB — &gt; 1TB</td>
</tr>
<tr>
<td>Disk farm</td>
<td>...</td>
<td>&gt; 3PB</td>
</tr>
</tbody>
</table>

Last modified: Wed Apr 29 12:20:49 2015
Managing the Memory Hierarchy

- Programs are written as if there are only two kinds of memory: main memory and disk (variables and files).
- Programmer is responsible for moving data from disk to memory.
- Hardware is responsible for moving data between memory and caches.
- Compiler is responsible for moving data between memory and registers (which the programmer usually doesn’t see).
- Cache and register sizes are growing slowly: important to manage them well.
- Processor speed improves faster than memory speed and disk speed.
- The cost of a cache miss is growing, and the widening gap is bridged with more caches.
The Register Allocation Problem

- Our intermediate code style uses temporaries profligately, simplifying code generation and optimization, but complicating final translation to assembly

- Hence, the register allocation problem:
  Rewrite the intermediate code to use fewer temporaries than there are machine registers

- So we must assign more temporaries to a register, without changing the program behavior
An Example

Consider the program

\begin{align*}
a & := c + d \\
e & := a + b \\
f & := e - 1
\end{align*}

assuming that assumption that \texttt{a} and \texttt{e} die after use. Then,

- Can reuse \texttt{a} after \texttt{a + b}
- Same with temporary \texttt{e} after \texttt{e - 1}
- Can allocate \texttt{a}, \texttt{e}, and \texttt{f} all to one register (r1):

\begin{align*}
r1 & := c + d \\
r1 & := r1 + b \\
r1 & := r1 - 1
\end{align*}
Basic Register Allocation Idea

- So in general, since the value in a dead temporary is not needed for the rest of the computation,
  
  Any set of temporaries can share a single physical register if at most one is alive at any program point.

- This rule is easy to apply to basic blocks. General CFGs are considerably trickier.
First step is to compute live variables before each statement. In this example, assume that variable \( b \) is live at exit.
Allocation in CFGs (II): Register Interference Graphs

- The sets in the previous slide indicate sets of virtual registers that are simultaneously alive at all points in the program, and therefore cannot share a physical register.

- Can summarize all these sets by constructing an undirected graph with a node for each virtual register, and an edge between any two virtual registers that appear together in the same set somewhere in the program.

- Call this the register interference graph (RIG).

![Graph showing register interference graph (RIG)]

- The RIG extracts exactly the information needed to characterize legal register assignments
- Gives global (over the entire CFG) picture of the register requirements
Allocation in CFGs (III): Graph Coloring

• A coloring of a graph is an assignment of colors to nodes, such that nodes connected by an edge have different colors.

• A graph is \( k \)-colorable if it has a coloring with \( k \) colors.

• In our problem, \textit{colors} = \textit{registers}. That is,

  If we have \( k \) available machine registers and our register interference graph is \( k \)-colorable, then the coloring gives us a register assignment.
Graph Coloring: Example

Consider the sample RIG:

- There is no coloring with fewer than 4 colors
- There are 4-colorings of this graph

Before . After

A: \(a := b + c\) A: \(r_2 := r_3 + r_4\)
\(d := -a\) \(r_3 := -r_2\)
\(e := d + f\) \(r_2 := r_3 + r_1\)
if \(f \geq 0\) jump C if \(r_1 \geq 0\) jump C
B: \(f := 2 \times e\) B: \(r_1 := 2 \times r_2\)
jump D jump D
C: \(b := d + e\) C: \(r_3 := r_3 + r_2\)
\(e := e - 1\) \(r_2 := r_2 - 1\)
if \(e \leq 0\) jump E if \(r_2 \leq 0\) jump E
D: \(b := f + c\) D: \(r_3 := r_1 + r_4\)
if \(b \leq c\) jump A if \(r_3 \leq r_4\) jump A
E: E:
Allocation in CFGs (III): Computing Graph Colorings

- The remaining problem is to compute a coloring for the interference graph.
- Unfortunately, this problem is hard (NP-hard). No fast algorithms are known,
- And besides, a coloring might not exist for a given number of registers.
- For (1), we'll use heuristics.
Graph Coloring Heuristic: Motivation

• Observation:
  - Pick a node $t$ with $< k$ neighbors in RIG.
  - Eliminate $t$ and its edges from RIG.
  - If the resulting graph has a $k$-coloring then so does the original graph.

• Reason: whatever $n \leq k - 1$ colors $t$’s neighbors have, we know we’ll always be able to color $t$ (since there are $k$ colors). Therefore, eliminating $t$ cannot affect the colorability of the other nodes.
Graph Coloring Heuristic

The following works well in practice:

• Pick a node $t$ with $< k$ neighbors.
• Push $t$ on a stack and remove it from the RIG.
• Repeat until the graph has no nodes.
• Then start popping nodes from the stack and adding them back to the graph, assigning colors to each as we go (starting with the last node added).
• At each step, we know we can pick a color different from those assigned to already colored neighbors, by the observation on the last slide.
Example of Using the Heuristic (I)

Start with our sample RIG and with $k = 4$:

Now remove $a$ and then $d$, giving

Now all nodes have < 4 neighbors; remove. Stack is [f, e, b, c, d, a].
Graph Coloring Example (2)

- Now we assign colors ...er, ...registers to: $f$, $e$, $b$, $c$, $d$, $a$ in that order.
- At each step, guaranteed there's a free register.

\[ r_1 \quad f \]
Graph Coloring Example (2)

- Now we assign colors ...er, ...registers to: f, e, b, c, d, a in that order.
- At each step, guaranteed there's a free register.
Graph Coloring Example (2)

- Now we assign colors ...er, ...registers to: f, e, b, c, d, a in that order.

- At each step, guaranteed there's a free register.
Graph Coloring Example (2)

- Now we assign colors ... er, ... registers to: $f, e, b, c, d, a$ in that order.
- At each step, guaranteed there’s a free register.
Graph Coloring Example (2)

• Now we assign colors ...er, ...registers to: f, e, b, c, d, a in that order.

• At each step, guaranteed there's a free register.
Graph Coloring Example (2)

- Now we assign colors ...er, ...registers to: f, e, b, c, d, a in that order.
- At each step, guaranteed there's a free register.
**Spilling**

- What if during simplification we get to a state where all nodes have $k$ or more neighbors?
- Example: try to find a 3-coloring of the RIG we’ve been using. After removing $a$, we get

  ![Graph Diagram]

  ... and now we are stuck, since all nodes have $\geq 3$ neighbors.
- So, pick a node as a candidate for *spilling*, that is, to reside in memory.
Example of Spilling

- Assume that $f$ is picked as a candidate. When we remove it from the graph:

- Simplification now succeeds. We end up with the stack

  $[e, c, b, d, f, a]$
Example of Spilling (II)

- On the assignment phase we get to the point when we have to assign a color to $f$.

- Sometimes, it just happens that among the 4 neighbors of $f$ we use $< 3$ colors (optimistic coloring) …

\[ \begin{align*} 
  \text{r2} & \quad \text{e} \\
  f & \quad \text{b} \quad \text{r3} \\
  \text{c} & \quad \text{r4} \\
  \text{d} & \quad \text{r3} \\
\end{align*} \]

- … but not this time.
Example of Spilling (III)

- Since optimistic coloring failed we must spill register \( f \): Allocate a memory location call it \( fa \) as the home of \( f \) (typically in the current stack frame).

- Before each operation that uses \( f \), insert
  
  \[
  f := *fa
  \]

- After each operation that defines (assigns to) \( f \), insert
  
  \[
  *fa := f
  \]

- This gives us:

  \[
  \begin{align*}
  A: & \quad a := b + c \\
  & \quad d := -a \\
  & \quad f := *fa \\
  & \quad e := d + f \\
  & \quad \text{if } f \geq 0 \text{ jump C} \\
  B: & \quad f := 2 \times e \\
  & \quad *fa := f \\
  & \quad \text{jump D}
  \end{align*}
  \]

  \[
  \begin{align*}
  C: & \quad b := d + e \\
  & \quad e := e - 1 \\
  & \quad \text{if } e \leq 0 \text{ jump E} \\
  D: & \quad b := f + c \\
  & \quad \text{if } b \leq c \text{ jump A}
  \end{align*}
  \]

  \[
  \begin{align*}
  E: & \quad \text{jump D}
  \end{align*}
  \]
Recomputing Liveness Information

\[
\begin{align*}
a & := b + c \\
d & := -a \\
f & := *fa \\
e & := d + f \\
f < 0
\end{align*}
\]

\[
\begin{align*}
f & := 2 * e \\
*fa & := f
\end{align*}
\]

\[
\begin{align*}
b & := d + e \\
e & := e - 1 \\
e > 0
\end{align*}
\]

\[
\begin{align*}
b & := f + c \\
b > c
\end{align*}
\]
A New RIG

- The new liveness information is almost as before, except that that $f$ is live only
  - Between an $f := \ast fa$ and the next instruction, and
  - Between a store $f, fa$ and the preceding instruction.
- That is, spilling reduces the live range of $f$, and thus the registers it interferes with, giving us this RIG:

  ![Diagram](attachment:image.png)

- And this graph is 3-colorable (left to the reader).
What to Spill?

• In general, additional spills might be required to allow a coloring.

• The tricky part is deciding what to spill. Possible heuristics:
  - Spill temporaries with most conflicts
  - Spill temporaries with few definitions and uses
  - Avoid spilling in inner loops
Caches

- Compilers are very good at managing registers (much better than programmers: the C register declaration is really obsolete).
- Caches are another matter. The problem is still left to programmers, and it is still an open question whether compilers can do much in general to improve performance.
- But they can (and a few do) perform some simple cache optimization.
Cache Optimization

• Consider the loop
  
  \[
  \begin{align*}
  &\text{for}(j = 1; j < 10; j += 1) \\
  &\text{for}(i = 1; i < 1000000; i += 1) \\
  &\quad a[i] *= b[i]
  \end{align*}
  \]

• Why does this have terrible cache performance?

• On the other hand,
  
  \[
  \begin{align*}
  &\text{for}(i = 1; i < 1000000; i += 1) \\
  &\text{for}(j = 1; j < 10; j += 1) \\
  &\quad a[i] *= b[i]
  \end{align*}
  \]

  computes the same thing, but with much better (possibly 10x) performance [again why?].

• Compilers can do this: loop interchange.
Cache Optimization (II)

- Other kinds of memory layout decisions possible, such as *padding* rows of a matrix with extra bytes to avoid cache conflicts when traversing a column (or row in FORTRAN) of a matrix. [Why might that help?]

- *Prefetching* instructions on some hardware can inform cache of anticipated future memory fetches so that they can proceed in parallel. Again, it is possible for compilers to supply these to a limited extent.
Summary

- Both because it eases code generation, greatly improves performance, and because it is difficult for programmers to do it for themselves, register allocation is a “must have” optimization in production compilers for standard procedural languages.

- Graph coloring is a powerful register allocation scheme that compilers can apply automatically.

- Good cache management could give even larger payoffs, but so far is difficult.