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>

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

Last modified: Wed Apr 29 12:20:49 2015
An Example

Consider the program
\[
\begin{align*}
a &:= c + d \\
e &:= a + b \\
f &:= e - 1
\end{align*}
\]
assuming that assumption that \(a\) and \(e\) die after use. Then,

- Can reuse \(a\) after \(a + b\)
- Same with temporary \(e\) after \(e - 1\)
- Can allocate \(a\), \(e\), and \(f\) 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.

Going Global: Allocation in CFGs (I)

First step is to compute live variables before each statement. In this example, assume that variable \(b\) is live at exit.

\[
\begin{align*}
\{b, c, f\} &\rightarrow a := b + c & \{c, d, e, f\} &\rightarrow d := -a \\
\{a, c, f\} &\rightarrow d := -a & \{c, d, f\} &\rightarrow e := d + f \\
\{c, d, e, f\} &\rightarrow f < 0 \\
\{c, e\} &\rightarrow f := 2 * e & \{c, d, e, f\} &\rightarrow b := d + e \\
\{b, c, f\} &\rightarrow b := f + c & \{c, d, e, f\} &\rightarrow e := e - 1 \\
\{c, f\} &\rightarrow b := f + c & \{b, c, e, f\} &\rightarrow e > 0 \\
\end{align*}
\]

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).

\[
\begin{align*}
\text{The RIG extracts exactly the information needed to characterize legal register assignments} \\
\text{Gives global (over the entire CFG) picture of the register requirements}
\end{align*}
\]
**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, *colors = 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

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>A: a := b + c</td>
<td>A: r2 := r3 + r4</td>
</tr>
<tr>
<td>d := -a</td>
<td>r3 := -r2</td>
</tr>
<tr>
<td>e := d + f</td>
<td>r2 := r3 + r1</td>
</tr>
<tr>
<td>if f &gt;= 0 jump C</td>
<td>if r1 &gt;= 0 jump C</td>
</tr>
<tr>
<td>B: f := 2 * e</td>
<td>B: r1 := 2 * r2</td>
</tr>
<tr>
<td>jump D</td>
<td>jump D</td>
</tr>
<tr>
<td>C: b := d + e</td>
<td>C: r3 := r3 + r2</td>
</tr>
<tr>
<td>e := e - 1</td>
<td>r2 := r2 - 1</td>
</tr>
<tr>
<td>if e &lt;= 0 jump E</td>
<td>if r2 &lt;= 0 jump E</td>
</tr>
<tr>
<td>D: b := f + c</td>
<td>D: r3 := r1 + r4</td>
</tr>
<tr>
<td>if b &lt;= c jump A</td>
<td>if r3 &lt;= r4 jump A</td>
</tr>
<tr>
<td>E:</td>
<td>E:</td>
</tr>
</tbody>
</table>

**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* ≤ *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 \( k = 4 \):

![Graph Coloring Example (I) Diagram](image)

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.

![Graph Coloring Example (II) Diagram](image)

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

![Spilling Diagram](image)

- ... 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)...

... 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:

  A: $a := b + c$
  d := -a
  f := *fa
  if $f >= 0$ jump C
  B: $f := 2 * e$
  if $b <= c$ jump A
  *fa := f
  jump D

Recomputing Liveness Information
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:

```
a
f
b
c
d
```
- 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
  ```
  for(j = 1; j < 10; j += 1)
  for(i = 1; i < 1000000; i += 1)
  a[i] *= b[i]
  ```
- Why does this have terrible cache performance?
- On the other hand,
  ```
  for(i = 1; i < 1000000; i += 1)
  for(j = 1; j < 10; j += 1)
  a[i] *= b[i]
  ```
  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.