Register Allocation

Lecture 16

Lecture Outline

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

The Register Allocation Problem

• Intermediate code uses unlimited temporaries
  - Simplifies code generation and optimization
  - Complicates final translation to assembly
• Typical intermediate code uses too many temporaries

Managing the Memory Hierarchy

• Most programs are written as if there are only two kinds of memory: main memory and disk
  - Programmer is responsible for moving data from disk to memory (e.g., file I/O)
  - Hardware is responsible for moving data between memory and caches
  - Compiler is responsible for moving data between memory and registers

Current Trends

• Power usage limits
  - Size and speed of registers/caches
  - Speed of processors
• But
  - The cost of a cache miss is very high
  - Typically requires 2-3 caches to bridge fast processor with large main memory
• It is very important to:
  - Manage registers properly
  - Manage caches properly
• Compilers are good at managing registers

The Memory Hierarchy

<table>
<thead>
<tr>
<th>Memory Level</th>
<th>Access Time</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>1 cycle</td>
<td>256-8000 bytes</td>
</tr>
<tr>
<td>Cache</td>
<td>3 cycles</td>
<td>256k-40MB</td>
</tr>
<tr>
<td>Main memory</td>
<td>20-100 cycles</td>
<td>4GB-32+G</td>
</tr>
<tr>
<td>Disk</td>
<td>0.5-5M cycles</td>
<td>1-10TB’s</td>
</tr>
</tbody>
</table>
The Register Allocation Problem (Cont.)

- The problem: 
  Rewrite the intermediate code to use no more temporaries than there are machine registers.

- Method:
  - Assign multiple temporaries to each register
  - But without changing the program behavior.

History

- Register allocation is as old as compilers
  - Register allocation was used in the original FORTRAN compiler in the ’50s
  - Very crude algorithms

- A breakthrough came in 1980
  - Register allocation scheme based on graph coloring
  - Relatively simple, global and works well in practice

An Example

- Consider the program:
  a := c + d
  e := a + b
  f := e - 1

- Assume a and e dead after use
  - Temporary a can be "reused" after e := a + b
  - So can temporary e

- Can allocate a, e, and f all to one register (r₁):
  r₁ := r₂ + r₃
  r₁ := r₁ + r₄
  r₁ := r₁ - 1

The Idea

- Temporaries t₁ and t₂ can share the same register if at any point in the program at most one of t₁ or t₂ is live.

  Or

- If t₁ and t₂ are live at the same time, they cannot share a register.

Algorithm: Part I

- Compute live variables for each point:

  (a,c,f) ——— (b,c,f)
  (c,d,f) ——— (c,d,e,f)
  (b) ——— (b,c,e,f)
  (c,e) ——— (c,f)
  (f := e * e) ——— (b,c,e,f)
  (b := f + c) ——— (b)

The Register Interference Graph

- Construct an undirected graph
  - A node for each temporary
  - An edge between t₁ and t₂ if they are live simultaneously at some point in the program

- This is the register interference graph (RIG)
  - Two temporaries can be allocated to the same register if there is no edge connecting them.
Example

- For our example:
  - E.g., b and c cannot be in the same register
  - E.g., b and d could be in the same register

Notes on Register Interference Graphs

- Extracts exactly the information needed to characterize legal register assignments
- Gives a global (i.e., over the entire flow graph) picture of the register requirements
- After RIG construction the register allocation algorithm is architecture independent

Definitions

- 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

Register Allocation Through Graph Coloring

- In our problem, colors = registers
  - We need to assign colors (registers) to graph nodes (temporaries)
- Let k = number of machine registers
- If the RIG is k-colorable then there is a register assignment that uses no more than k registers

Graph Coloring Example

- Consider the example RIG
  - There is no coloring with less than 4 colors
  - There are 4-colorings of this graph

Example Review

- a := b + c
d := -a
e := d + f
f := 2 * e
b := d + e
e := e - 1
b := f + e
Example After Register Allocation

- Under this coloring the code becomes:

\[
\begin{align*}
  r_2 &:= r_3 + r_4 \\
  r_3 &:= r_2 \\
  r_3 &:= r_0 + r_1 \\
  r_2 &:= 2 \times r_3 \\
  r_3 &:= r_1 + r_4
\end{align*}
\]

Computing Graph Colorings

- How do we compute graph colorings?

  - It isn’t easy:
    1. This problem is very hard (NP-hard). No efficient algorithms are known.
       - Solution: use heuristics
    2. A coloring might not exist for a given number of registers
       - Solution: later

Graph Coloring Heuristic

- Observation:
  - Pick a node \( t \) with fewer than \( k \) neighbors in RIG
  - Eliminate \( t \) and its edges from RIG
  - If resulting graph is \( k \)-colorable, then so is the original graph

- Why?
  - Let \( c_1, \ldots, c_n \) be the colors assigned to the neighbors of \( t \) in the reduced graph
  - Since \( n < k \) we can pick some color for \( t \) that is different from those of its neighbors

Graph Coloring Example (1)

- Start with the RIG and with \( k = 4 \):

  Stack: {}

- Remove \( a \)

Graph Coloring Example (2)

- Remove \( d \)
Graph Coloring Example (3)

- Note: all nodes now have fewer than 4 neighbors

- Remove c

Graph Coloring Example (4)

- Remove b

Graph Coloring Example (5)

- Remove e

Graph Coloring Example (6)

- Remove f

Graph Coloring Example (7)

- Now start assigning colors to nodes, starting with the top of the stack

Graph Coloring Example (8)
**Graph Coloring Example (9)**

- e must be in a different register from f

**Graph Coloring Example (10)**

**Graph Coloring Example (11)**

- d can be in the same register as b

**Graph Coloring Example (12)**

**What if the Heuristic Fails?**

- What if all nodes have k or more neighbors?
- Example: Try to find a 3-coloring of the RIG:
What if the Heuristic Fails?

- Remove \( a \) and get stuck (as shown below)

- Pick a node as a candidate for spilling
  - A spilled temporary “lives” in memory
  - Assume that \( f \) is picked as a candidate

```
\[
\begin{align*}
  &f & b \\
  &e & d & c \\
\end{align*}
\]
```

What if the Heuristic Fails?

- Remove \( f \) and continue the simplification
  - Simplification now succeeds: \( b, d, e, c \)

```
\[
\begin{align*}
  &b & e \\
  &d & c \\
\end{align*}
\]
```

What if the Heuristic Fails?

- Eventually we must assign a color to \( f \)

- We hope that among the 4 neighbors of \( f \) we use less than 3 colors \( \Rightarrow \) optimistic coloring

```
\[
\begin{align*}
  &f & b & r_3 \\
  &e & d & r_1 \end{align*}
\]
```

Spilling

- If optimistic coloring fails, we spill \( f \)
  - Allocate a memory location for \( f \)
    - Typically in the current stack frame
    - Call this address \( fa \)
  - Before each operation that reads \( f \), insert
    \[
    f := \text{load } fa
    \]
  - After each operation that writes \( f \), insert
    \[
    \text{store } f, fa
    \]

Spilling Example

- This is the new code after spilling \( f \)

```
a := b + c \\
d := -a \\
f := \text{load } fa \\
e := d + f \\
f := 2 * e \\
\text{store } f, fa
```

A Problem

- This code reuses the register name \( f \)

- Correct, but suboptimal
  - Should use distinct register names whenever possible
  - Allows different uses to have different colors
Spilling Example

- This is the new code after spilling f

```
fa := b + c
d := -a
f1 := load fa
e := d + f1

f2 := 2 * e
store f2, fa

f3 := load fa
b := f3 + c
```

Recomputing Liveness Information

- The new liveness information after spilling:

```
a := b + c
d := -a
f1 := load fa
e := d + f1

f2 := 2 * e
store f2, fa

b := d + e
e := e - 1

f3 := load fa
b := f3 + c
```

Recompute RIG After Spilling

- Some edges of the spilled node are removed
- In our case f still interferes only with c and d
- And the resulting RIG is 3-colorable

Recomputing Liveness Information

- New liveness information is almost as before
  - Note f has been split into three temporaries

- fi is live only
  - Between a fi := load fa and the next instruction
  - Between a store fi, fa and the preceding instr.

- Spilling reduces the live range of f
  - And thus reduces its interferences
  - Which results in fewer RIG neighbors

Spilling Notes

- Additional spills might be required before a coloring is found

- The tricky part is deciding what to spill
  - But any choice is correct

- 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 a programmer could be

- Compilers are not good at managing caches
  - This problem is still left to programmers
  - It is still an open question how much a compiler can do to improve cache performance

- Compilers can, and a few do, perform some cache optimizations
Cache Optimization

- Consider the loop
  for(j := 1; j < 10; j++)
  for(i=1; i<1000; i++)
  a[i] *= b[i]

- This program has terrible cache performance
  - Why?

Cache Optimization (Cont.)

- Consider the program:
  for(i=1; i<1000; i++)
  for(j := 1; j < 10; j++)
  a[i] *= b[i]

  - Computes the same thing
  - But with much better cache behavior
  - Might actually be more than 10x faster

- A compiler can perform this optimization
  - called loop interchange

Conclusions

- Register allocation is a “must have” in compilers:
  - Because intermediate code uses too many temporaries
  - Because it makes a big difference in performance

- Register allocation is more complicated for CISC machines