Register Allocation

Lecture 16

Prof. Aiken  CS 143  Lecture 16

The Memory Hierarchy

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>0.3 ns</td>
<td>1-16 KB</td>
</tr>
<tr>
<td>Cache L1</td>
<td>0.5 ns</td>
<td>8-64 KB</td>
</tr>
<tr>
<td>Cache L2</td>
<td>7 ns</td>
<td>256-512 KB</td>
</tr>
<tr>
<td>Cache L3</td>
<td>12 ns</td>
<td>1-8 MB</td>
</tr>
<tr>
<td>Main memory</td>
<td>100 ns</td>
<td>1-32 GB</td>
</tr>
</tbody>
</table>

HD seek | 10M ns (10ms) | up to 10 TB |
SSD seek | 0.1M ns (0.1ms) | up to 1 TB |
Network | 150M ns (150 ms) |

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 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

The Register Allocation Problem (Cont.)

- The problem:
  \[
  \text{Rewrite the intermediate code to use no more temporaries than there are machine registers}
  \]
- 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
  ```

  - Can allocate `a`, `e`, and `f` all to one register (`r_1`):
    
    ```
    r_1 := r_2 + r_3
    r_1 := r_1 + r_4
    r_1 := r_1 - 1
    ```

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

**The Idea**

*Temporaries `t_1` and `t_2` can share the same register if at any point in the program at most one of `t_1` or `t_2` is live.*

Or

*If `t_1` and `t_2` are live at the same time, they cannot share a register*

**Algorithm: Part I**

- Compute live variables for each point:
  
  ```
  a := b + c
  d := -a
  e := d + f
  f := 2 * e
  b := d + e
  e := e - 1
  ```

  - For our example:
    
    ![Example Diagram](example_diagram)

    - 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

- Under this coloring the code becomes:

Example After Register Allocation

- Under this coloring the code becomes:
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 \( k = 4 \):

  ![Graph Coloring Example (1) Diagram]

  Stack: {}  

  • Remove \( a \)

Graph Coloring Example (2)

• Remove \( d \)

  ![Graph Coloring Example (2) Diagram]

  Stack: \{(a)\}

Graph Coloring Example (3)

• Note: all nodes now have fewer than 4 neighbors

  ![Graph Coloring Example (3) Diagram]

  Stack: \{(d, a)\}

  • Remove \( c \)
Graph Coloring Example (4)

Stack: \{c, d, a\}

• Remove b

Graph Coloring Example (5)

Stack: \{b, c, d, a\}

• Remove e

Graph Coloring Example (6)

Stack: \{e, b, c, d, a\}

• Remove f

Graph Coloring Example (7)

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

Stack: \{f, e, b, c, d, a\}

Graph Coloring Example (8)

Stack: \{e, b, c, d, a\}

Graph Coloring Example (9)

Stack: \{b, c, d, a\}

• e must be in a different register from f
Graph Coloring Example (10)

Stack: \( \{c, d, a\} \)

Prof. Aiken  CS 143  Lecture 16  34

Graph Coloring Example (11)

Stack: \( \{d, a\} \)

Prof. Aiken  CS 143  Lecture 16  35

Graph Coloring Example (12)

Stack: \( \{a\} \)

\( d \) can be in the same register as \( b \)

Prof. Aiken  CS 143  Lecture 16  36

Graph Coloring Example (13)

Prof. Aiken  CS 143  Lecture 16  37

What if the Heuristic Fails?

- What if all nodes have \( k \) or more neighbors?

Example: Try to find a 3-coloring of the RIG:

Prof. Aiken  CS 143  Lecture 16  38

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

Prof. Aiken  CS 143  Lecture 16  39
What if the Heuristic Fails?

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

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 $\text{store } f, fa$

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
Recomputing Liveness Information

- The new liveness information after spilling:
  \[ a := b + c \]
  \[ d := a \]
  \[ f_1 := \text{load } f_a \]
  \[ e := d + f_1 \]
  \[ f_2 := 2 \times e \]
  \[ \text{store } f_2, f_a \]
  \[ b := d + e \]
  \[ e := e - 1 \]
  \[ f_3 := \text{load } f_a \]
  \[ b := f_3 + c \]

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

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
  ```
  \text{for}(j := 1; j < 10; j++)
  \text{for}(i=1;i<1000000; i++)
  a[i] *= b[i]
  ```
- This program has terrible cache performance
  - Why?
Cache Optimization (Cont.)

- Consider the program:
  
  ```c
  for(i=1; i<1000000; i++)
  for(j := 1; j < 10; j++)
  a[i] *= b[j]
  ```
  
  - 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