Register Allocation

Lecture 13
Back-End (Revisited)

Back-End:

• Translate IR into machine code

• Choose instructions for each IR operation

• Decide what to keep in registers at each point
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:
  
  *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
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 \((r_1)\):

  \[
  r_1 := r_2 + r_3 \\
  r_1 := r_1 + r_4 \\
  r_1 := r_1 - 1
  \]

- A dead temporary is not needed
  - A dead temporary can be reused

Prof. Aiken
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
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:

\[
\begin{align*}
    a &:= b + c \\
    d &:= -a \\
    e &:= d + f \\
    f &:= 2 \times e \\
    b &:= f + c \\
    e &:= e - 1
\end{align*}
\]
The Register Interference Graph

• Construct an undirected graph
  - A node for each temporary
  - An edge between $t_1$ and $t_2$ 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
  - It does not depend on any property of the machine except for the number of registers
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

\[
\begin{align*}
a &:= b + c \\
d &:= -a \\
e &:= d + f \\
f &:= 2 \times e \\
b &:= d + e \\
e &:= e - 1
\end{align*}
\]
Example After Register Allocation

- Under this coloring the code becomes:

```
r_2 := r_3 + r_4
r_3 := -r_2
r_2 := r_3 + r_1

r_1 := 2 * r_2
r_3 := r_3 + r_2
r_2 := r_2 - 1

r_3 := r_1 + r_4
```
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 $\dagger$ with fewer than $k$ neighbors in RIG
  - Eliminate $\dagger$ 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 $\dagger$ in the reduced graph
  - Since $n < k$ we can pick some color for $\dagger$ that is different from those of its neighbors
Graph Coloring Heuristic

1. The following works well in practice:
   - Pick a node \( t \) with fewer than \( k \) neighbors
   - Put \( t \) on a stack and remove it from the RIG
   - Repeat until the graph has one node

2. Assign colors to nodes on the stack
   - Start with the last node added
   - At each step pick a color different from those assigned to already colored neighbors
Graph Coloring Example (1)

- Start with the RIG and with $k = 4$:
  
  Stack: {}

- Remove a
Graph Coloring Example (2)

- Remove \( d \)

Stack: \( \{a\} \)
Graph Coloring Example (3)

• Note: all nodes now have fewer than 4 neighbors

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)

- Empty graph - done with the first part!

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

- Now start assigning colors to nodes, starting with the top of the stack
Graph Coloring Example (8)

Stack: \{e, b, c, d, a\}
Graph Coloring Example (9)

- e must be in a different register from f

Stack: \{b, c, d, a\}
Graph Coloring Example (10)

Stack: \{c, d, a\}
Graph Coloring Example (11)

Stack: \{d, a\}
Graph Coloring Example (12)

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

Stack: \( \{a\} \)
Graph Coloring Example (13)
What if the Heuristic Fails?

• What happens if the graph coloring heuristic fails to find a coloring?

• In this case, we can’t hold all values in registers.
  - Some values are spilled to memory
What if the Heuristic Fails?

• What if all nodes have $k$ or more neighbors?

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

![Graph Diagram]

Prof. Aiken
What if the Heuristic Fails?

- Remove a and get stuck (as shown below)
  - There is no node with fewer than 3 neighbors
- Pick a node as a candidate for spilling
  - A spilled temporary “lives” in memory
  - Assume that f is picked as a candidate
What if the Heuristic Fails?

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

In this ex., it doesn’t work
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

- Original code

\[
\begin{align*}
a &:= b + c \\
d &:= -a \\
e &:= d + f \\
f &:= 2 \times e \\
b &:= f + c \\
b &:= d + e \\
e &:= e - 1
\end{align*}
\]
Spilling Example

- This is the new code after spilling $f$

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

  f := 2 * e  
  store f, fa

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

  f := load fa
  b := f + c
```
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 $f$

\[
\begin{align*}
a &:= b + c \\
d &:= -a \\
f1 &:= \text{load } fa \\
e &:= d + f1 \\
f2 &:= 2 \times e \\
\text{store } f2, fa \\
f3 &:= \text{load } fa \\
b &:= f3 + c \\
b &:= d + e \\
e &:= e - 1
\end{align*}
\]
The new liveness information after spilling:

- \( a := b + c \)
- \( d := -a \)
- \( f1 := \text{load} f_a \)
- \( e := d + f1 \)
- \( f2 := 2 \times e \) (store \( f2, f_a \))
- \( b := d + e \) (store \( f2, f_a \))
- \( e := e - 1 \)
- \( f3 := \text{load} f_a \)
- \( b := f3 + c \)
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
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
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
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
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
  
  - Because each iteration of the inner loop refers to a new element of arrays (i.e., fresh data) = [a cache miss]

Prof. Aiken
Cache Optimization (Cont.)

- Consider the program:

```java
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
Question?

Which of the following pairs of temporaries interfere in the code fragment given at right?

- [ ] A and B
- [ ] A and C
- [ ] B and C
- [ ] C and D
Question?

Which of the following colorings is a valid minimal coloring of the given RIG?
Question?

For the given RIG and $k = 3$, which of the following are valid deletion orders for the nodes of the RIG?

- $\{d, e, c, b, a, f\}$
- $\{e, f, a, b, c, d\}$
- $\{d, c, b, a, f, e\}$
- $\{d, e, b, c, a, f\}$
For the given code fragment and RIG, find the minimum cost spill. In this example, the cost of spilling a node is given by:

# of occurrences (use or definition) - # of conflicts + 5 if the node corresponds to a variable used in a loop