Register Allocation

Lecture 16
Register Allocation

• This is one of the most sophisticated things that compiler do to optimize performance
• Also illustrates many of the concepts we’ve been discussing in global flow analysis
Lecture Outline

• Register Allocation
  - Register interference graph
  - Graph coloring heuristics
  - Spilling

• Memory Hierarchy Management

• Cache Management
The Register Allocation Problem

• Intermediate code uses unlimited temporaries
  - Simplifies code generation and optimization
    • E.g., we don’t have to worry about preserving the right number of registers in the code
  - Complicates final translation to assembly code
    • Because we might be using too many temporaries

• Typical intermediate code uses too many temporaries
  - Fairly common problem in practice
  - Not uncommon for intermediate code to use more registers than there are registers on the target machine
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
    • So need a many-to-one mapping of temporaries to registers
    • Clearly if we are using too many temporaries we will not be able to fit them into a single register (so we need some kind of a trick, which will almost, but not always work)
    • And we’ll need a backup plan for when trick doesn’t work
  - But without changing the program behavior
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
    
    • Given all that, we still want to be able to fit as many temporaries into a single register as possible

  
  - But without changing the program behavior
An Example

- Consider the program
  
  \[
  \begin{align*}
  a &:= c + d \\
  e &:= a + b \\
  f &:= e - 1
  \end{align*}
  \]

- Assume \(a\) and \(e\) dead after use
  
  - Temporary \(a\) can be “reused” after read of \(a\) in \(e := a + b\)
  - Temporary \(e\) can be reused after read of \(e\) in \(f := e - 1\)

- Can allocate \(a\), \(e\), and \(f\) all to one register \((r_1)\):
  
  (here \(c, d,\) and \(b\) assigned to \(r_2, r_3,\) and \(r_4\))

  \[
  \begin{align*}
  r_1 &:= r_2 + r_3 \\
  r_1 &:= r_1 + r_4 \\
  r_1 &:= r_1 - 1
  \end{align*}
  \]

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

7
An Example

• Consider the program
  
  \[
  \begin{align*}
  a &:= c + d \\
  e &:= a + b \\
  f &:= e - 1
  \end{align*}
  \]

• Assume a and e dead after use
  
  - Temporary a can be “reused” after read of a in \( e := a + b \)
  
  - Temporary e can be reused after read of e in \( f := e - 1 \)

• Can allocate a, e, and f all to one register (\( r_1 \)):
  
  (here \( c, d, \) and \( b \) assigned to \( r_2, r_3, \) and \( r_4 \))

  \[
  \begin{align*}
  r_1 &:= r_2 + r_3 \\
  r_1 &:= r_1 + r_4 \\
  r_1 &:= r_1 - 1
  \end{align*}
  \]

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

• Register allocation is as old as compilers
  - Register allocation was used in the original FORTRAN compiler in the ‘50s
  - Very crude algorithms
    • It was quickly noticed that this was a bottleneck in the quality of code that a compiler could produce
    • I.e., Limitations on the ability to perform register allocation had a significant effect on the overall quality of the code that was generated

• A breakthrough came in 1980
  - Researchers at IBM: Register allocation scheme based on graph coloring
  - Relatively simple, global and works well in practice
History

• A breakthrough came in 1980
  - Researchers at IBM: Register allocation scheme based on graph coloring
  - Relatively simple, global and works well in practice
    • Simple: fairly easy to explain
    • Global: takes advantage of information from entire control flow graph at the same time
The Basic Principle

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.

Equivalently

If $t_1$ and $t_2$ are live at the same time at any point in the program, they cannot share a register.
Algorithm: Part I

- Compute live variables for each point:

```
{a,c,f}  a := b + c  {b,c,f}
{c,d,f}  d := -a
         e := d + f
{c,e}   f := 2 * e
{c,f}   b := f + c
{c,f}   e := e - 1
{c,f}   b := d + e
{b}     b := f + c
```

Assume that at output of program, only b is live
Algorithm: Part I

- Compute live variables for each point:

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

Work backwards (recall liveness is a backward analysis)
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 in the register interference graph
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
  – Note we haven’t yet discussed how to get the register assignment from the register interference graph

• Gives a global (i.e., over the entire flow graph) picture of the register requirements
  – Helps us make good global decisions about what values are important to live in registers

• After RIG construction the register allocation algorithm is architecture independent
  – We’ll see only concern is number of registers
So, How to Use Register Interference Graphs

• We want to come up with register assignments
• One popular method for doing this involves using graph colorings
  - So it behooves us to take a look at these.
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

- For this graph there is no coloring with less than 4 colors

- There are 4-colorings of this graph
  - Obviously, since it is illustrated here
Graph Coloring Example

- Consider the example RIG

- This is not the only 4-coloring of this graph
- And how to assign registers is obvious in this case (since we assign them)
Example Review

\[
a := b + c \\
d := -a \\
e := d + f \\
f := 2 \times e \\
b := d + e \\
e := e - 1 \\
b := f + c
\]
Example After Register Allocation

• Under this coloring the code becomes:

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

\[ r_1 := 2 \times r_2 \]

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

\[ r_3 := r_1 + r_4 \]
Computing Graph Colorings

• Knowing that graph colorings can help us assign temporaries to registers, the question becomes how exactly to color graphs.
Computing Graph Colorings

• How do we compute graph colorings?

• It isn’t easy:
  1. This problem is very hard (NP-hard in fact). 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
    - If none, then graph not $k$-colorable
  - 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 Heuristic

• The following works well in practice:
  - Pick a node $\uparrow$ with fewer than $k$ neighbors
  - Put $\uparrow$ on a stack and remove it from the RIG
  - Repeat until the graph has one node

• 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 $k = 4$:

- Remove $a$

```
Stack: {}
```
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)

• Remove $f$

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

\[ r_1 \}

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

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

- \textit{e} must be in a different register from \textit{f}
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
    • Because frankly that’s the only other kind of storage that we have!
    • The term comes from the analogy of a bucket (the registers). When the bucket is full, any other stuff added will spill
When does the Heuristic Fail?

- When we reach a point in our algorithm where all remaining nodes have $k$ or more neighbors?

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

```
  a
 /|
/  |
 f - e - d - c - b
```
When does the Heuristic Fail?

- Remove $a$ and get stuck (as shown below)
So what now?

- Pick a node as a candidate for spilling
  - Assume, for now, that \( f \) is picked as a candidate
  - Later: how we choose candidate to spill
What if the Heuristic Fails?

• Remove $f$ and continue the simplification
  - Simplification now succeeds: use heuristic algorithm removing nodes in order $b, d, e, c$
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
  - Basically, we could get lucky
What if the Heuristic Fails?

- **Optimistic coloring**: we pick a candidate to spill, color the resulting subgraph, and hope that we get lucky and can fit candidate into a register after all.
  - In which case we can just continue coloring the rest of the graph as if nothing had happened.
What if the Heuristic Fails?

• So, what happens in our example?
  - optimistic coloring fails, since $f$ has neighbors that are assigned to three different registers
Spilling

• If optimistic coloring fails, we spill \( f \)
  - Allocate a memory location for \( f \)
    • Typically in the current stack frame
    • Call this address \( fa \)

• Then modify the control flow graph:

• Before each operation that reads \( f \), insert
  \[
  f := \text{load} \ fa
  \]

• After each operation that writes \( f \), insert
  \[
  \text{store} \ f, \ fa
  \]
Spilling Example: The original CFG

reads and writes off

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

- This is the new code after spilling f

\[
\begin{align*}
a & := b + c \\
d & := -a \\
f & := \text{load } fa \\
e & := d + f \\
f & := 2 \times e \\
\text{store } f, fa \\
b & := d + e \\
e & := e - 1 \\
f & := \text{load } fa \\
b & := f + c
\end{align*}
\]
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 \( f \)

\[
\begin{align*}
a &:= b + c \\
d &:= -a \\
f1 &:= \text{load } \text{fa} \\
e &:= d + f1 \\
f2 &:= 2 \times e \\
\text{store } f2, \text{fa} \\
b &:= d + e \\
e &:= e - 1 \\
f3 &:= \text{load } \text{fa} \\
b &:= f3 + c
\end{align*}
\]
Recomputing Liveness Information

• The **original** liveness information before spilling:

```
\begin{align*}
  a & := b + c \\
  d & := -a \\
  f1 & := \text{load } fa \\
  e & := d + f1 \\
  f2 & := 2 \times e \\
  \text{store } f2, fa \\
  b & := d + e \\
  e & := e - 1 \\
  f3 & := \text{load } fa \\
  b & := f3 + c
\end{align*}
```
Recomputing Liveness Information

- The **new** liveness information after spilling:

\[
\begin{align*}
a &:= b + c \\
d &:= -a \\
f1 &:= \text{load } fa \\
e &:= d + f1 \\
f2 &:= 2 \times e \\
\text{store } f2, fa \\
b &:= d + e \\
e &:= e - 1 \\
f3 &:= \text{load } fa \\
b &:= f3 + c \\
\end{align*}
\]
Recomputing Liveness Information

- The new liveness information after spilling:

\[
\begin{align*}
    a &:= b + c \\
    d &:= -a \\
    f1 &:= \text{load } fa \\
    e &:= d + f1 \\
    f2 &:= 2 \times e \\
    b &:= d + e \\
    e &:= e - 1 \\
    f3 &:= \text{load } fa \\
    b &:= f3 + c
\end{align*}
\]
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

Note that f used to be live in many places, but it’s not now
Recomputing Liveness Information

- The **new** liveness information after spilling:

```
<table>
<thead>
<tr>
<th></th>
<th>a := b + c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>d := -a</td>
</tr>
<tr>
<td></td>
<td>f1 := load fa</td>
</tr>
<tr>
<td></td>
<td>e := d + f1</td>
</tr>
</tbody>
</table>

f2 := 2 * e
store f2, fa
b := d + e
e := e - 1

{b, c, e, f}

Also, we’ve distinguished the different uses of f
Recomputing Liveness Information

• New liveness information is *almost* as before
  - Note *f* has been split into three temporaries
  - And we’ve changed the program (added *loads* and *stores*)
    • So now we have a different program
  - And that changes our register allocation problem
    • We have to recompute the liveness information
    • We have to rebuild the register interference graph
    • We have to try again to color the RIG
    • But we can still use much of the information from before
      - E.g., for all of the non-spilled temporaries, they are still live wherever they were live before
      - Though info for *f* has changed fairly dramatically
Recomputing Liveness Information

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

- Each $f_i$ is live only in a very small area
  - Between a $f_i := \text{load } f_a$ and the next instruction
  - Between a $\text{store } f_i, f_a$ 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

- Note that 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

![Graph Diagram]

- $a$
- $b$
- $c$
- $d$
- $e$
- $f1$
- $f2$
- $f3$
Spilling Notes

• Additional spills might be required before a coloring is found

• The tricky part is deciding what to spill
  – Any choice is correct (will lead to a correct program)
    • But some choices lead to better code than others

• Possible heuristics:
  – Spill temporaries with most conflicts
  – Spill temporaries with few definitions and uses
  – Avoid spilling in inner loops
Spilling Notes

• Additional spills might be required before a coloring is found

• Possible heuristics:
  - Spill temporaries with most conflicts
    • Reason: This is the one thing you can move into memory that will most effect the number of interferences in the RIG
    • So, possibly, by spilling just this one variable, we’ll remove enough edges from the graph that it becomes colorable with the number of registers we have
Spilling Notes

• Additional spills might be required before a coloring is found

• Possible heuristics:
  – Spill temporaries with few definitions and use
    • Reason: By spilling these, since they are not used very much, the number of loads and stores that will need to be added is small, so the additional cost in terms of extra instructions is small
Spilling Notes

• Additional spills might be required before a coloring is found

• Possible heuristics:
  - Avoid spilling in inner loops
    • Pretty much all compilers implement this
    • Reason: Similar to previous. Presumably the inner loop is iterated quite a bit, so want to avoid adding additional loads and stores to that loop
We’ve talked about managing registers. Let’s talk now about managing caches.
# The Memory Hierarchy

<table>
<thead>
<tr>
<th></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-1M</td>
</tr>
<tr>
<td>Main memory</td>
<td>20-100 cycles</td>
<td>512M-64G</td>
</tr>
<tr>
<td>Disk</td>
<td>0.5-5M cycles</td>
<td>80G-2T</td>
</tr>
</tbody>
</table>

Recall from CS 301: Fast vs slow, expensive vs cheap, big vs small
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 well with large main memory

• If you want your program to perform well, it is very important to:
  - Manage registers properly
  - Manage caches properly
Caches

• **Compilers are very good at managing registers**
  - These days, most would agree that compilers do a much better job than a programmer could do
    • So this job is typically left to compiler

• **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

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

• This program has terrible cache performance
  • Why? Well, consider the order in which the values are accessed (and thus placed in cache)
    - a[1], b[1], a[2], b[2], a[3], b[3],....
Cache Optimization

• This program has terrible cache performance
  • Why? Well, consider the order in which the values are accessed (and thus placed in cache)
    - a[1], b[1], a[2], b[2], a[3], b[3],....
  • Each of these are cache misses because each iteration of the loop refers to “new” elements
    - True, some might be in the same cache line, but even then, since $i$ is much larger than the cache block size, you’ll be constantly swapping lines in/out of the cache
  • Which means program runs at memory, rather than cache, speed
Cache Optimization (Cont.)

- Consider the program:
  
  ```c
  for(i=1; i<1000000; i++)
    for(j := 1; j < 10; j++)
      a[i] *= b[i]
  ```
  - Computes the same thing
  - But with much better cache behavior
    - Miss on first reference, but hit on the next nine!
    - Might actually be more than 10x faster

- A compiler can perform this optimization
  - called *loop interchange*
  - Though not many compilers do, because in general it is not easy to decide whether you can reverse the order of the loops
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
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