CSE 401 - Compilers
Section 8

3/6/2013
12:30 - MEB 238
1:30 - EE 037

Project

Code Due: March 15th
Report Due: March 17th

Workload closer to part 3 than part 1 or 2
Start soon

How did testing go?

Questions?

Debugging

Generate commented assembly

Draw expected memory layouts

```
$ gcc -g boot.c minijava_out.s
```

```
$ ./a.out
$ gdb a.out
```

The most important gdb command:

```
help [command]
```

Debugging

View source:

```
list [<linenum> | <function>]
```

Make it go:

```
run
step
continue
```

Examine state:

```
backtrace
info registers
x <address>
```

Make it stop:

```
break [<linenum>]
info breakpoints
delete [<breakpoint number>]
```

Debugging

Retyping the same commands?

```
gdb -x commands.txt
```

More References (via 351 website):

Quick summary:

http://csapp.cs.cmu.edu/public/docs/gdbnotes-x86-64.pdf

Extensive tutorial:

http://heather.cs.ucdavis.edu/~matloff/UnixAndC/CLanguage/Debug.html

Object Representation, Method Table, or Other Code Gen Questions?

Optimizations Review

Overview

Optimization are program transformations that "usually make something better" while preserving semantics

Kinds:

- Peephole: Adjacent Instructions
- Local: Basic blocks
- Intraprocedural: Entire procedure
- Interprocedural: Entire program

Larger scope = more useful and more expensive

Peephole Optimizations

Generate "good" code to start with

Jump chaining
Strength reduction (multiplication => shift)
Constant folding
Other mathematical simplifications

What does "good" mean?

Local Optimizations

Handwritten transformations on basic blocks

Constant propagation
Dead store elimination
Common subexpression elimination
Copy propagation (skip through pointer chains)

Could we discover these automatically?

Intra/Interprocedural Optimizations

All of the easier optimizations

Code motion:

- Lift loop-invariant computations out of loops
- Remove redundant checks in libraries

```
increment_count(key, value) {
    check_valid(key)
    x = get_count(key)
    ...
}
```

```
get_count(key) {
    check_valid(key)
    ...
}
```

Inlining (interprocedural)

Useful Graphs

Control flow graphs:
- Nodes are basic blocks
- Edges represent control flow

Data flow graphs:
- Nodes are declarations and references
- Edges show data dependencies

Analysis: propagate info through the graph

An Example

Constant propagation/folding with the CFG

Walk the CFG:

- What info do we need to keep track of?

An Example

Constant propagation/folding with the CFG

Walk the CFG:

- What info do we need to keep track of?
  - Variable -> Constant(c), NonConstant, or Undefined

An Example

Constant propagation/folding with the CFG

Walk the CFG:

- What info do we need to keep track of?
  - Variable -> Constant(c), NonConstant, or Undefined
- What transformations can we perform with this info?

An Example

Constant propagation/folding with the CFG

Walk the CFG:

- What info do we need to keep track of?
  - Variable -> Constant(c), NonConstant, or Undefined
- What transformations can we perform with this info?
  - If Var(x) -> Constant(c), replace x with c
  - If RHS contains only constants, perform folding

Dataflow Analysis

General framework:

IN(b) = facts true when entering b

OUT(b) = facts true when exiting b

GEN(b) = facts created and not killed in b

KILL(b) = facts killed in b

$$OUT(b) = GEN(b) \cup (IN(b) - KILL(b))$$

Solve forward or backward, iteratively

Dataflow Analysis Example

Common Subexpression Elimination

Define:

AVAIL(b) = Expressions available at block b

NKILL(b) = Expressions not killed in b

DEF(b) = Expressions defined & not killed in b

$$AVAIL(b) = \bigcap_{x \text{ in preds}(b)} (DEF(x) \cup (AVAIL(x) \cap NKILL(x)))$$

Solving the Dataflow Problem

Compute DEF and KILLED

Compute NKILL

Compute AVAIL via fixed-point algorithm

```
while (nodes_to_compute_for) {...}
```

Defining the right things is the hard part

Single Static Assignment Form

Limit each variable to one static definition:

- Gives a unique name to everything
- Disambiguates def-use chains
- Static definition may be executed multiple times at runtime

Merge points:

```
if (...)
  a = x;
else
  a = y;
b = a;
```



```
if (...)
  a1 = x;
else
  a2 = y;
a3 = Φ(a1,a2);
b1 = a3;
```

Loop Example

Loop back edges are also merge points:

```
a = 0;
do {
  b = a + 1;
  c = c + b;
  a = b * 2;
} while (a < N);
return c;
```



```
a1 = 0;
do {
  a3 = Φ(a1,a2);
  b1 = Φ(b0,b2);
  c1 = Φ(c0,c2);
  b2 = a3 + 1;
  c2 = c1 + b2;
  a2 = b2 * 2;
} while (a2 < N);
return c2;
```

Add Φ's, rename vars

Translating Back

For each selector $x = \Phi(x_1, x_2)$

Insert $x = x_i$ at the end of the proceeding block

```
if (...)
   a1 = x;
else
   a2 = y;
a3 = Φ(a1,a2);
b1 = a3;
```



```
if (...) {
   a1 = x;
   a3 = a1;
} else {
   a2 = y;
   a3 = a2;
}
b1 = a3;
```

Dominance Frontiers

x dominates y:
  Every path through the CFG to y includes x
  So x dominates x
x strictly dominates y:
  x dominates y and x != y

The dominance frontier of x:
  {w | x dominates a predecessor of w and
       x does not strictly dominate w}

Dominance Frontiers

The dominance frontier of x:

{w | x dominates a predecessor of w and
x does not strictly dominate w}

{w | x dominates a predecessor of w and
(x does not dominate w or x = w)}

{w | (x = w and x is its own predecessor) or
(x dominates a predecessor of w but not w)}

Placing Φ Functions

For nodes x containing definitions of a:

- Place Φ functions for a in each node y in x's dominance frontier
- A different definition of a will reach each y (say that the initial node defines everything)
- Adding Φ functions adds definitions
  - Use a fixed-point algorithm

Aside: Concurrency Breaks Things

If other threads can write to memory at any time,
which optimizations change program behavior?

But we still want to do these optimizations...
  Require the programmer to "be safe"

Questions?