Announcements

• HW6: Analysis & Optimizations
  – Alias analysis, constant propagation, dead code elimination, register allocation
  – Due: Wednesday, April 25th
LOOPS AND DOMINATORS
Loops in Control-flow Graphs

• Taking into account loops is important for optimizations.
  – The 90/10 rule applies, so optimizing loop bodies is important

• Should we apply loop optimizations at the AST level or at a lower representation?
  – Loop optimizations benefit from other IR-level optimizations and vice-versa, so it is good to interleave them.

• Loops may be hard to recognize at the quadruple / LLVM IR level.
  – Many kinds of loops: while, do/while, for, continue, goto…

• Problem: How do we identify loops in the control-flow graph?
Definition of a Loop

• A *loop* is a set of nodes in the control flow graph.
  – One distinguished entry point called the *header*

• Every node is reachable from the header & the header is reachable from every node.
  – A loop is a *strongly connected component*

• No edges enter the loop except to the header
• Nodes with outgoing edges are called loop exit nodes
Nested Loops

- Control-flow graphs may contain many loops
- Loops may contain other loops:

Control Tree:

The control tree depicts the nesting structure of the program.
Control-flow Analysis

• Goal: Identify the loops and nesting structure of the CFG.

• Control flow analysis is based on the idea of dominators:
  • Node A dominates node B if the only way to reach B from the start node is through node A.

• An edge in the graph is a back edge if the target node dominates the source node.

• A loop contains at least one back edge.
Dominator Trees

- **Domination is transitive:**
  - if A dominates B and B dominates C then A dominates C
- **Domination is anti-symmetric:**
  - if A dominates B and B dominates A then A = B
- **Every flow graph has a dominator tree**
  - The Hasse diagram of the dominates relation
Dominator Dataflow Analysis

• We can define $\text{Dom}[n]$ as a forward dataflow analysis.
  – Using the framework we saw earlier: $\text{Dom}[n] = \text{out}[n]$ where:
• “A node B is dominated by another node A if A dominates all of the predecessors of B.”
  – $\text{in}[n] := \bigcap_{n' \in \text{pred}[n]} \text{out}[n']$
• “Every node dominates itself.”
  – $\text{out}[n] := \text{in}[n] \cup \{n\}$

• Formally: $\mathcal{L} = \text{set of nodes ordered by } \subseteq$
  – $T = \{\text{all nodes}\}$
  – $F_n(x) = x \cup \{n\}$
  – $\sqcap$ is $\cap$

• Easy to show monotonicity and that $F_n$ distributes over meet.
  – So algorithm terminates and is MOP
Improving the Algorithm

- Dom[b] contains just those nodes along the path in the dominator tree from the root to b:
  - e.g. Dom[8] = \{1,2,4,8\}, Dom[7] = \{1,2,4,5,7\}
  - There is a lot of sharing among the nodes

- More efficient way to represent Dom sets is to store the dominator tree.
  - doms[b] = immediate dominator of b

- To compute Dom[b] walk through doms[b]
- Need to efficiently compute intersections of Dom[a] and Dom[b]
  - Traverse up tree, looking for least common ancestor:

- See: “A Simple, Fast Dominance Algorithm” Cooper, Harvey, and Kennedy
Completing Control-flow Analysis

• Dominator analysis identifies *back edges*:
  – Edge $n \to h$ where $h$ dominates $n$

• Each back edge has a *natural loop*:
  – $h$ is the header
  – All nodes reachable from $h$ that also reach $n$ without going through $h$

• For each back edge $n \to h$, find the natural loop:
  – $\{n' \mid n$ is reachable from $n'$ in $G - \{h\}\} \cup \{h\}$

• Two loops may share the same header: merge them

• Nesting structure of loops is determined by set inclusion
  – Can be used to build the control tree
Example Natural Loops

Natural Loops

Control Tree:

The control tree depicts the nesting structure of the program.
Uses of Control-flow Information

• Loop nesting depth plays an important role in optimization heuristics.
  – Deeply nested loops pay off the most for optimization.

• Need to know loop headers / back edges for doing
  – loop invariant code motion
  – loop unrolling

• Dominance information also plays a role in converting to SSA form
  – Used internally by LLVM to do register allocation.
Phi nodes
Alloc “promotion”
Register allocation

REVISITING SSA
Single Static Assignment (SSA)

• LLVM IR names (via %uids) all intermediate values computed by the program.
• It makes the order of evaluation explicit.
• Each %uid is assigned to only once
  – Contrast with the mutable quadruple form
  – Note that dataflow analyses had these kill[n] sets because of updates to variables...
• Naïve implementation of backend: map %uids to stack slots
• Better implementation: map %uids to registers (as much as possible)

• Question: How do we convert a source program to make maximal use of %uids, rather than alloca-created storage?
  – two problems: control flow & location in memory

• Then: How do we convert SSA code to x86, mapping %uids to registers?
  – Register allocation.
Alloc vs. %UID

- Current compilation strategy:

```c
int x = 3;
int y = 0;
x = x + 1;
y = x + 2;
```

```assembly
%x = alloca i64
%y = alloca i64
store i64* %x, 3
store i64* %y, 0
%x1 = load %i64* %x
%tmp1 = add i64 %x1, 1
store i64* %x, %tmp1
%x2 = load %i64* %x
%tmp2 = add i64 %x2, 2
store i64* %y, %tmp2
```

- Directly map source variables into %uid?

```c
int x = 3;
int y = 0;
x = x + 1;
y = x + 2;
```

```assembly
%x1 = add i64 3, 0
%y1 = add i64 0, 0
%x2 = add i64 %x1, 1
%y2 = add i64 %x2, 2
```

- Does this always work?

Zdancewic  CIS 341: Compilers
What about If-then-else?

- How do we translate this into SSA?

```c
int y = ...
int x = ...
int z = ...
if (p) {
    x = y + 1;
} else {
    x = y * 2;
}
z = x + 3;
```

- What do we put for ???

```plaintext
entry:
    %y1 = ...
    %x1 = ...
    %z1 = ...
    %p = icmp ...
    br i1 %p, label %then, label %else
then:
    %x2 = add i64 %y1, 1
    br label %merge
else:
    %x3 = mult i64 %y1, 2
merge:
    %z2 = %add i64 ???, 3
```
Phi Functions

• Solution: φ functions
  – Fictitious operator, used only for analysis
    • implemented by Mov at x86 level
  – Chooses among different versions of a variable based on the path by which control enters the phi node.
    \[%uid = phi <ty> v_1, <label_1>, ..., v_n, <label_n>\]

```c
int y = ...
int x = ...
int z = ...
if (p) {
    x = y + 1;
} else {
    x = y * 2;
}
z = x + 3;
```

```c
entry:
    %y1 = ...
    %x1 = ...
    %z1 = ...
    %p = icmp ...
    br il %p, label %then, label %else
then:
    %x2 = add i64 %y1, 1
    br label %merge
else:
    %x3 = mult i64 %y1, 2
merge:
    %x4 = phi i64 %x2, %then, %x3, %else
    %z2 = %add i64 %x4, 3
```
Phi Nodes and Loops

- Importantly, the %uids on the right-hand side of a phi node can be defined “later” in the control-flow graph.
  - Means that %uids can hold values “around a loop”
  - Scope of %uids is defined by dominance

```c
entry:
   %y1 = ...
   %x1 = ...
   br label %body

body:
   %x2 = phi i64 %x1, %entry, %x3, %body
   %x3 = add i64 %x2, %y1
   %p = icmp slt i64, %x3, 10
   br i1 %p, label %body, label %after

after:
   ...
```
Not all source variables can be allocated to registers

- If the address of the variable is taken (as permitted in C, for example)
- If the address of the variable “escapes” (by being passed to a function)

An alloca instruction is called promotable if neither of the two conditions above holds

- Happily, most local variables declared in source programs are promotable
  - That means they can be register allocated

```assembly
entry:
  %x = alloca i64  // %x cannot be promoted
  %y = call malloc(i64 8)
  %ptr = bitcast i8* %y to i64**
  store i65** %ptr, %x  // store the pointer into the heap

entry:
  %x = alloca i64  // %x cannot be promoted
  %y = call foo(i64* %x)  // foo may store the pointer into the heap
```
Converting to SSA: Overview

• Start with the ordinary control flow graph that uses allocas
  – Identify “promotable” allocas
• Compute dominator tree information
• Calculate def/use information for each such allocated variable
• Insert \( \phi \) functions for each variable at necessary “join points”

• Replace loads/stores to alloc’ed variables with freshly-generated %uids

• Eliminate the now unneeded load/store/alloca instructions.
Where to Place $\phi$ functions?

- Need to calculate the “Dominance Frontier”

- Node A **strictly dominates** node B if A dominates B and $A \neq B$.
  - Note: A does not strictly dominate B if A does not dominate B or $A = B$.

- The **dominance frontier** of a node B is the set of all CFG nodes $y$ such that B dominates a predecessor of $y$ but does not strictly dominate $y$
  - Intuitively: starting at B, there is a path to $y$, but there is another route to $y$ that does not go through B

- Write $DF[n]$ for the dominance frontier of node n.
Dominance Frontiers

- Example of a dominance frontier calculation results
- \( \text{DF}[1] = \{1\}, \ \text{DF}[2] = \{1,2\}, \ \text{DF}[3] = \{2\}, \ \text{DF}[4] = \{1\}, \ \text{DF}[5] = \{8,0\}, \ \text{DF}[6] = \{8\}, \ \text{DF}[7] = \{7,0\}, \ \text{DF}[8] = \{0\}, \ \text{DF}[9] = \{7,0\}, \ \text{DF}[0] = \{\}\)
Algorithm For Computing DF[n]

- Assume that doms[n] stores the dominator tree (so that doms[n] is the immediate dominator of n in the tree)

- Adds each B to the DF sets to which it belongs

for all nodes B
  if #(pred[B]) \geq 2  // (just an optimization)
    for each p \in pred[B] {
      runner := p  // start at the predecessor of B
      while (runner \neq doms[B])  // walk up the tree adding B
        DF[runner] := DF[runner] \cup \{B\}
        runner := doms[runner]
    }
Insert $\phi$ at Join Points

- Lift the $DF[n]$ to a set of nodes $N$ in the obvious way:
  \[ DF[N] = \bigcup_{n \in N} DF[n] \]
- Suppose that at variable $x$ is defined at a set of nodes $N$.

  \[
  DF_0[N] = DF[N] \\
  DF_{i+1}[N] = DF[DF_i[N] \cup N]
  \]

  Let $J[N]$ be the least fixed point of the sequence:
  \[ DF_0[N] \subseteq DF_1[N] \subseteq DF_2[N] \subseteq DF_3[N] \subseteq \ldots \]
  That is, $J[N] = DF_k[N]$ for some $k$ such that $DF_k[N] = DF_{k+1}[N]$
  \[ J[N] \] is called the “join points” for the set $N$

- We insert $\phi$ functions for the variable $x$ at each node in $J[N]$.
  - $x = \phi(x, x, \ldots, x)$; (one “$x$” argument for each predecessor of the node)
  - In practice, $J[N]$ is never directly computed, instead you use a worklist algorithm that keeps adding nodes for $DF_k[N]$ until there are no changes, just as in the dataflow solver.

- Intuition:
  - If $N$ is the set of places where $x$ is modified, then $DF[N]$ is the places where phi nodes need to be added, but those also “count” as modifications of $x$, so we need to insert the phi nodes to capture those modifications too…
Example Join-point Calculation

- Suppose the variable x is modified at nodes 3 and 6
  - Where would we need to add phi nodes?

- \( DF_0[\{3,6\}] = DF[\{3,6\}] = DF[3] \cup DF[6] = \{2,8\} \)
- \( DF_1[\{3,6\}] 
  = DF[DF_0[\{3,6\}] \cup \{3,6\}] 
  = DF[\{2,3,6,8\}] 
  = \{1,2\} \cup \{2\} \cup \{8\} \cup \{0\} = \{1,2,8,0\} \)
- \( DF_2[\{3,6\}] 
  = ... 
  = \{1,2,8,0\} \)

- So \( J[\{3,6\}] = \{1,2,8,0\} \) and we need to add phi nodes at those four spots.
Phi Placement Alternative

• Less efficient, but easier to understand:

• Place phi nodes "maximally" (i.e. at every node with > 2 predecessors)

• If all values flowing into phi node are the same, then eliminate it:
  \%
  \text{x} = \text{phi} \ t \ \%
  \text{y}, \ %\text{pred1} \ t \ %\text{y} \ %\text{pred2} \ ... \ t \ %\text{y} \ %\text{predK}
  \text{// code that uses \%x}
  \Rightarrow
  \text{// code with \%x replaced by \%y}

• Interleave with other optimizations
  – copy propagation
  – constant propagation
  – etc.
Example SSA Optimizations

- How to place phi nodes without breaking SSA?
  - Note: the “real” implementation combines many of these steps into one pass.
    - Places phis directly at the dominance frontier
- This example also illustrates other common optimizations:
  - Load after store/alloca
  - Dead store/alloca elimination

```c
l_1: %p = alloca i64
     store 0, %p
     %b = %y > 0
     br %b, %l_2, %l_3

l_2:
     store 1, %p
     br %l_3

l_3:
     %x = load %p
     ret %x
```
Example SSA Optimizations

- How to place phi nodes without breaking SSA?
- Insert
  - Loads at the end of each block

```
l_1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    %x_1 = load %p
    br %b, %l_2, %l_3

l_2:
    store 1, %p
    %x_2 = load %p
    br %l_3

l_3:
    %x = load %p
    ret %x
```
Example SSA Optimizations

• How to place phi nodes without breaking SSA?

• Insert
  – Loads at the end of each block
  – Insert $\phi$-nodes at each block

```assembly
l_1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    %x_1 = load %p
    br %b, %l_2, %l_3

l_2: %x_3 = $\phi$[%x_1, %l_1]
    store 1, %p
    %x_2 = load %p
    br %l_3

l_3: %x_4 = $\phi$[%x_1, %l_1, %x_2:%l_2]
    %x = load %p
    ret %x
```
Example SSA Optimizations

• How to place phi nodes without breaking SSA?

• Insert
  – Loads at the end of each block
  – Insert \( \phi \)-nodes at each block
  – Insert stores after \( \phi \)-nodes

\[
\begin{align*}
l_1: & \quad \%p = \text{alloca} \ i64 \\
& \quad \text{store} \ 0, \ %p \\
& \quad \%b = \%y > 0 \\
& \quad \%x_1 = \text{load} \ %p \\
& \quad \text{br} \ %b, \ %l_2, \ %l_3 \\
l_2: & \quad \%x_3 = \phi[\%x_1, %l_1] \\
& \quad \text{store} \ %x_3, \ %p \\
& \quad \text{store} \ 1, \ %p \\
& \quad \%x_2 = \text{load} \ %p \\
& \quad \text{br} \ %l_3 \\
l_3: & \quad \%x_4 = \phi[\%x_1; %l_1, \%x_2; %l_2] \\
& \quad \text{store} \ %x_4, \ %p \\
& \quad \%x = \text{load} \ %p \\
& \quad \text{ret} \ %x
\end{align*}
\]
Example SSA Optimizations

- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

```asm
l_1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    %x_1 = load %p
    br %b, %l_2, %l_3

l_2: %x_3 = phi[%(x_1), %l_1]
    store %x_3, %p
    store 1, %p
    %x_2 = load %p
    br %l_3

l_3: %x_4 = phi[%(x_1), %l_1, %x_2, %l_2]
    store %x_4, %p
    %x = load %p
    ret %x
```
Example SSA Optimizations

- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

```c
l_1: %p = alloca i64
     store 0, %p
     %b = %y > 0
     %x_1 = load %p
     br %b, %l_2, %l_3

l_2: %x_3 = phi [%x_1, %l_1]
     store %x_3, %p
     store 1, %p
     %x_2 = load %p
     br %l_3

l_3: %x_4 = phi [%x_1, %l_1, %x_2:%l_2]
     store %x_4, %p
     %x = load %p
     ret %x
```
Example SSA Optimizations

- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

```c
l_1: %p = alloca i64
    store 0, %p
    %b = %b > 0
    %x_1 = load %p
    br %b, %l_2, %l_3

l_2: %x_3 = φ[0, %l_1]
    store %x_3, %p
    store 1, %p
    %x_2 = load %p
    br %l_3

l_3: %x_4 = φ[0; %l_1, %x_2; %l_2]
    store %x_4, %p
    %x = load %p
    ret %x
```
Example SSA Optimizations

- For loads after stores (LAS):
  - Substitute all uses of the load by the value being stored
  - Remove the load

```
// SSA code after optimizations
l1: %p = alloc i64
    store 0, %p
    %b = %y > 0
    br %b, %l2, %l3

l2: %x3 = φ[0, %l1]
    store %x3, %p
    store 1, %p
    %x2 = load %p
    br %l3

l3: %x4 = φ[0; %l1, %x2, %l2]
    store %x4, %p
    %x = load %p
    ret %x
```
Example SSA Optimizations

For loads after stores (LAS):
- Substitute all uses of the load by the value being stored
- Remove the load

```c
l_1: %p = alloca i64
  store 0, %p
  %b = %y > 0
  br %b, %l_2, %l_3

l_2: %x_3 = φ[0,%l_1]
  store %x_3, %p
  store 1, %p
  %x_2 = load %p
  br %l_3

l_3: %x_4 = φ[0;%l_1, 1;%l_2]
  store %x_4, %p
  %x = load %p
  ret %x
```
Example SSA Optimizations

For loads after stores (LAS):
- Substitute all uses of the load by the value being stored
- Remove the load

```
l_1: %p = alloca i64
    store 0, %p
    %b = %y > 0
    br %b, %l_2, %l_3

l_2: %x_3 = phi[0, %l_1]
    store %x_3, %p
    store 1, %p
    br %l_3

l_3: %x_4 = phi[0; %l_1, 1:%l_2]
    store %x_4, %p
    %x = load %p
    ret %x
```
Example SSA Optimizations

For loads after stores (LAS):
- Substitute all uses of the load by the value being stored
- Remove the load

l₁: %p = alloca i64
  store 0, %p
  %b = %y > 0
  br %b, %l₂, %l₃

l₂: %x₃ = φ[0, %l₁]
  store %x₃, %p
  store 1, %p
  br %l₃

l₃: %x₄ = φ[0: %l₁, 1: %l₂]
  store %x₄, %p
  %x = load %p
  ret %x₄

Find alloca
max φs
LAS/LAA
DSE
DAE
elim φs
Example SSA Optimizations

- Dead Store Elimination (DSE)
  - Eliminate all stores with no subsequent loads.

- Dead Alloca Elimination (DAE)
  - Eliminate all allocas with no subsequent loads/stores.
Example SSA Optimizations

- **Dead Store Elimination (DSE)**
  - Eliminate all stores with no subsequent loads.

- **Dead Alloca Elimination (DAE)**
  - Eliminate all allocas with no subsequent loads/stores.

```c
l₁:  %p = alloca i64
   store 0, %p
   %b = %y > 0
   br %b, %l₂, %l₃

l₂:  %x₃ = φ[0,%l₁]
   store %x₃, %p
   store 1, %p
   br %l₃

l₃:  %x₄ = φ[0;%l₁, 1;%l₂]
   store %x₄, %p
   ret %x₄
```

Find alloca

Find alloca

max φs

max φs

LAS/LAA

LAS/LAA

DSE

DSE

DAE

DAE

elim φs

elim φs
Example SSA Optimizations

\[ l_1: \]
\[ \%b = \%y > 0 \]
\[ \text{br } \%b, \%l_2, \%l_3 \]

\[ l_2: \%x_3 = \phi[0,\%l_1] \]
\[ \text{br } \%l_3 \]

\[ l_3: \%x_4 = \phi[0;\%l_1, 1:\%l_2] \]
\[ \text{ret } \%x_4 \]

- Eliminate \( \phi \) nodes:
  - Singletons
  - With identical values from each predecessor
  - See Aycock & Horspool, 2002
Example SSA Optimizations

$l_1$: 
\[
%b = %y > 0 \\
br %b, %l_2, %l_3
\]

$l_2$: 
\[
%x_3 = \phi[0, %l_1]
\]


br %l_3

$l_3$: 
\[
%x_4 = \phi[0; %l_1, 1:%l_2]
\]

ret %x_4

- Eliminate $\phi$ nodes:
  - Singletons
  - With identical values from each predecessor
Example SSA Optimizations

\[ l_1: \]
\[ %b = %y > 0 \]
\[ \text{br } %b, %l_2, %l_3 \]

\[ l_2: \]
\[ \text{br } %l_3 \]

\[ l_3: %x_4 = \phi[0; %l_1, 1:%l_2] \]
\[ \text{ret } %x_4 \]

- Done!

- Find alloca
- max \( \phi \)s
- LAS/LAA
- DSE
- DAE
- elim \( \phi \)
LLVM Phi Placement

- This transformation is also sometimes called register promotion
  - older versions of LLVM called this “mem2reg” memory to register promotion

- In practice, LLVM combines this transformation with *scalar replacement of aggregates* (SROA)
  - i.e. transforming loads/stores of structured data into loads/stores on register-sized data

- These algorithms are (one reason) why LLVM IR allows annotation of predecessor information in the .ll files
  - Simplifies computing the DF