Lecture 21

COMPILER DESIGN
Announcements

- **HW5**: OAT v. 2.0
  - Records, function pointers, type checking, array-bounds checks, etc.

- **HW6**: Analysis & Optimizations *(the final homework)*
  - Alias analysis, constant propagation, dead code elimination, register allocation
Plan

• Next: Register allocation

• Upcoming
  – Dataflow analysis (part 2)
  – Control-flow analysis & SSA
  – Garbage collection (GC)
  – Compiler testing & validation
    • How to find thousands of bugs in GCC & LLVM?
  – Compiler verification
    • How to build a fully verified realistic compiler?
  – MLIR
  – Guest lecture on GraalVM, PGQL & Green-Marl
  – Summary
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 AST level or at a lower level?
  – Loop optimizations benefit from other IR-level optimizations & vice-versa
  – Thus, it’s 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 CFG
  - One distinguished entry: the **header**

- Each node reachable from header
- Header reachable from each node

- No edges enter a loop except to header
- **Exit nodes**: nodes with outgoing edges

Loop is a strongly connected component (SCC)
Nested Loops

- A CFG may contain many loops
- Loops may contain other loops

Control Tree

Control tree depicts nesting structure
Control-flow Analysis

- **Goal**: Identify loops & nesting structure in a CFG

- Control flow analysis is based on the idea of *dominators*
  
  **A dominates B**: if the only way to reach B from start node is via A

- An edge in the CFG is a *back edge* if its target dominates the source

- A loop contains >= 1 back edge
Dominator Trees

- Domination is transitive: \( A \text{ dom } B, B \text{ dom } C \implies A \text{ dom } C \)
- Domination is anti-symmetric: \( A \text{ dom } B, B \text{ dom } A \implies A = B \)

- Every flow graph has a dominator tree
  - The Hasse diagram of the dominates relation

![Dominator Tree Diagram](image-url)
Dominator Dataflow Analysis

• We can define \textbf{Dom}[n] as a forward dataflow analysis
  – Using the framework we saw earlier: \textbf{Dom}[n] = \text{out}[n] where
  – B is dominated by A if A dominates all B’s predecessors

\[
in[n] := \bigcap_{n' \in \text{pred}[n]} \text{out}[n']
\]
  – Every node dominates itself

\[
\text{out}[n] := \text{in}[n] \cup \{n\}
\]

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

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

• **Dom[b]:** just those nodes along the path in dominator tree from 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 to represent Dom sets by storing dominator tree
  – **doms[b]** = immediate dominator of b

• To compute Dom[b] walk through doms[b]

• Need to efficiently compute **Dom[a] ∩ Dom[b]**
  – Traverse up tree & look for least common ancestor

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

- Dominator analysis identifies back edges
  - Edge $n \rightarrow h$ 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 $n \rightarrow h$, find the natural loop
  - $\{n' \mid n \text{ is reachable from } n' \text{ in } G \setminus \{h\}\} \cup \{h\}$
- 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

Control tree depicts nesting structure
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
- It makes order of evaluation explicit
- Each %uid is assigned only once
  - Contrast with the mutable quadruple form
  - Dataflow analyses had kill[n] sets due to variable updates

- Naïve backup implementation: map %uids to stack slots
- Better: map %uids to registers (as much as possible)

- Question: How to convert a source to maximally use %uids (vs. alloca’s)?
  - Two problems: control flow & location in memory

- Then, how to convert SSA code to x86, mapping %uids to registers?
  - Register allocation
• Current compilation strategy

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

%\texttt{x} = \texttt{alloca i64}
%\texttt{y} = \texttt{alloca i64}
\texttt{store i64* } %\texttt{x}, 3
\texttt{store i64* } %\texttt{y}, 0
%\texttt{x1} = \texttt{load i64* } %\texttt{x}
%\texttt{tmp1} = \texttt{add i64 } %\texttt{x1}, 1
\texttt{store i64* } %\texttt{x}, %\texttt{tmp1}
%\texttt{x2} = \texttt{load i64* } %\texttt{x}
%\texttt{tmp2} = \texttt{add i64 } %\texttt{x2}, 2
\texttt{store i64* } %\texttt{y}, %\texttt{tmp2}

• Directly map source variables into %\texttt{uids}?

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

int x1 = 3;
int y1 = 0;
x2 = x1 + 1;
y2 = x2 + 2;

%\texttt{x1} = \texttt{add i64 3, 0}
%\texttt{y1} = \texttt{add i64 0, 0}
%\texttt{x2} = \texttt{add i64 } %\texttt{x1}, 1
%\texttt{y2} = \texttt{add i64 } %\texttt{x2}, 2

• Q: Does this always work?
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;
```

```c
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
```

- What do we put for ???
Phi Functions

- **Solution:** \( \phi \) functions
  - Fictitious operator, used only for analysis
    - Implemented by Mov at x86 level
  - Chooses versions of a variable by the path how control enters phi node

\[
\text{%uid} = \phi <\text{ty}> v_1, <\text{label}_1>, \ldots, v_n, <\text{label}_n>
\]

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

```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:
  %x4 = phi i64 %x2, %then, %x3, %else
  %z2 = %add i64 %x4, 3
```
Phi Nodes and Loops

- Importantly, `%uids` on RHS of phi nodes can be defined “later” in CFG
  - Meaning 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:
  ...
```
Alloca Promotion

- Not all source variables can be allocated to registers
  - If the address of the variable is taken (as permitted in C, for example)

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

  - If the address of the variable “escapes” (by being passed to a function)

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

- An alloca inst. is **promotable** if neither of the above conditions holds

- Happily, most local variables declared in source programs are promotable
  - That means they can be register allocated
Converting to SSA: Overview

• Start with the ordinary CFG 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 $B$ if $A \text{ dom } B \land A \neq B$
  – 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
- \( \text{DF}[1] = \{1\}, \quad \text{DF}[2] = \{1,2\}, \quad \text{DF}[3] = \{2\}, \quad \text{DF}[4] = \{1\}, \quad \text{DF}[5] = \{8,0\}, \quad \text{DF}[6] = \{8\}, \quad \text{DF}[7] = \{7,0\}, \quad \text{DF}[8] = \{0\}, \quad \text{DF}[9] = \{7,0\}, \quad \text{DF}[0] = \{\} \)

Control-flow Graph

Dominator Tree
Dominance Frontiers

- Example of a dominance frontier calculation results
- $DF[1] = \{1\}$ because
  - $1 \text{ dom } 4$ (4 is a predecessor of 1)
  - $1$ doesn’t strictly dominate 1

Control-flow Graph

Dominator Tree
Example of a dominance frontier calculation results

- $DF[2] = \{1,2\}$ because
  - $2 \text{ dom } 4$ (a predecessor of 1), 2 doesn’t strictly dominate 1
  - $2 \text{ dom } 3$ (a predecessor of 2), 2 doesn’t strictly dominate 2
Dominance Frontiers

- Example of a dominance frontier calculation results
- $DF[5] = \{8,0\}$ because
  - For 0, 5 dom 9 (a predecessor of 0), but 5 doesn’t dominate 0
Algorithm For Computing DF[n]

• Assume doms[n] stores the dominator tree
  – 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]) ≥ 2   // (just an optimization)
    for each p ∈ pred[B] {
      runner := p   // start at the predecessor of B
      while (runner ≠ doms[B])   // walk up the tree adding B
        DF[runner] := DF[runner] U \{B\}
        runner := doms[runner]
    }

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] = \{

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 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 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?
Example Join-point Calculation

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

- $DF_0[\{3,6\}] = DF[\{3,6\}] = DF[3] \cup DF[6] = \{2,8\}$
- $DF_1[\{3,6\}]$
  - $= DF[DF_0[\{3,6\}] \cup \{3,6\}]$
  - $= DF[\{2,3,6,8\}]$
  - $= \{1,2\} \cup \{2\} \cup \{8\} \cup \{0\} = \{1,2,8,0\}$
- $DF_2[\{3,6\}]$
  - $= \ldots$
  - $= \{1,2,8,0\}$

- So $J[\{3,6\}] = \{1,2,8,0\}$, and we need to add phi nodes at those 4 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
  \[
  \%x = \text{phi} \ t \ \%y, \ %\text{pred1} \ t \ \%y \ %\text{pred2} \ldots \ t \ %y \ %\text{predK}
  \]

  // code that uses \%x

  ⇒

  // 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
Example SSA Optimizations

- How to place phi nodes without breaking SSA?

  - Insert
    - Loads at the end of each block

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

\text{l_2}: \text{store 1, %p}
\text{\%x_2 = load \%p}
\text{br \%l_3}

\text{l_3}: \text{\%x = load \%p}
\text{ret \%x}
```
Example SSA Optimizations

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

- How to place phi nodes without breaking SSA?

- Insert
  - Loads at the end of each block
  - Insert φ-nodes at each block
  - Insert stores after φ-nodes

```
l₁:  %p = alloca i64
    store 0, %p
    %b = %y > 0
    %x₁ = load %p
    br %b, %l₂, %l₃

l₂:  %x₃ = φ[%x₁, %l₁]
    store %x₃, %p
    store 1, %p
    %x₂ = load %p
    br %l₃

l₃:  %x₄ = φ[%x₁, %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

\[ l_1: \%p = \text{alloca i64} \]
\[ \text{store 0, } \%p \]
\[ \%b = \%y > 0 \]
\[ \%x_1 = \text{load } \%p \]
\[ \text{br } \%b, \%l_2, \%l_3 \]

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

\[ l_3: \%x_4 = \phi[\%x_1, \%l_1, \%x_2, \%l_2] \]
\[ \text{store } \%x_4, \%p \]
\[ \%x = \text{load } \%p \]
\[ \text{ret } \%x \]
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 = allocas i64
    store 0, %p
    %b = %p > 0
    %x_1 = load %p
    br %b, %l_2, %l_3

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

l_3: %x_4 = phi[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

```
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, %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

\[ l_1: \%p = \text{alloca i64} \]
\[ \text{store 0, } \%p \]
\[ \%b = \%y > 0 \]
\[ \text{br } \%b, \%l_2, \%l_3 \]

\[ l_2: \%x_3 = \phi[0, \%l_1] \]
\[ \text{store } \%x_3, \%p \]
\[ \text{store 1, } \%p \]
\[ \%x_2 = \text{load } \%p \]
\[ \text{br } \%l_3 \]

\[ l_3: \%x_4 = \phi[0; \%l_1, 1; \%l_2] \]
\[ \text{store } \%x_4, \%p \]
\[ \%x = \text{load } \%p \]
\[ \text{ret } \%x \]
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 = \phi[0;%l₁]
    store %x₃, %p
    store 1, %p
    br %l₃

l₃: %x₄ = \phi[0;%l₁, 1;%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

```
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_4
```
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.

```
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
    br %l_3

l_3: %x_4 = φ[0;%l_1, 1;%l_2]
    store %x_4, %p
    ret %x_4
```
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

\[ \text{l}_1: \]
\[ \%b = \%y > 0 \]
\[ \text{br } \%b, \%l_2, \%l_3 \]

\[ \text{l}_2: \%x_3 = \phi[0,\%l_1] \]
\[ \text{br } \%l_3 \]

\[ \text{l}_3: \%x_4 = \phi[0;\%l_1, 1:\%l_2] \]
\[ \text{ret } \%x_4 \]

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

Find alloca
max \( \phi \)s
LAS/LAA
DSE
DAE
elim \( \phi \)s
Example SSA Optimizations

1:
\%
\texttt{b} = \%y > 0
\texttt{br} \%\texttt{b}, \%\texttt{l}_2, \%\texttt{l}_3

2:
\%\texttt{x}_3 = \varphi[0,\%\texttt{l}_4]
\texttt{br} \%\texttt{l}_3

3:
\%\texttt{x}_4 = \varphi[0;\%\texttt{l}_1, 1:\%\texttt{l}_2]
\texttt{ret} \%\texttt{x}_4

- Eliminate \( \varphi \) 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 it “mem2reg” memory to register promotion

- LLVM combines it with \textit{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