ECE 5775 (Fall’17)
High-Level Digital Design Automation

Static Single Assignment
Announcements

- HW 1 released (due Friday)

- Student-led discussions on Tuesday 9/26
  - Sign up on Piazza: 3 students / group
  - Meet with the instructor 3 days before the talk
Agenda

- More on dominance relation
- Dataflow analysis – static single assignment (SSA)
  - Definition
  - PHI node (Φ-node) placement
  - Code optimizations with SSA
- LLVM overview
Revisiting CFG and Dominators

- What becomes a leader statement of a basic block?

- Does a node strictly (properly) dominate itself?

- Does a predecessor of a node B always dominate B?

- Suppose A that dominates all of B’s predecessors. Does A always dominate B?
Dominator Tree

- A node (basic block) \( N \) in CFG may have multiple dominators, but only one of them will be closest to \( N \) and be dominated by all other dominators of \( N \).

- A dominator tree is a useful way to represent the dominance relation:
  - The entry node \( s \) is the root.
  - Each node in the tree is the immediate dominator of its children:
    - Each node \( d \) dominates only its descendants in the tree.
Example: Dominator Tree

 CFG

 entry

 Dominator Tree

 1

 2

 3

 4

 5

 6

 7

 8

 9

 10
Static Single Assignment

- Static single assignment (SSA) form is a restricted IR where
  - Each variable definition has a unique name
  - Each variable use refers to a single definition

- SSA simplifies data flow analysis & many compiler optimizations
  - Eliminates artificial dependences (on scalars)
    - Write-after-write
    - Write-after-read
SSA within a Basic Block

- Assign each variable definition a unique name
- Update the uses accordingly

Original code

\[
\begin{align*}
x &= \text{read()} \\
x &= x \times 5 \\
x &= x + 1 \\
y &= x \times 9
\end{align*}
\]

SSA form

\[
\begin{align*}
x_0 &= \text{read()} \\
x_1 &= x_0 \times 5 \\
x_2 &= x_1 + 1 \\
y &= x_2 \times 9
\end{align*}
\]

Corresponding data flow graph
Consider a situation where two control-flow paths merge – e.g., due to an if-then-else statement or a loop

\[
\begin{align*}
x &= \text{read()} \\
\text{if } (x > 0) &\quad y = 5 \\
\text{else} &\quad y = 10 \\
x &= y
\end{align*}
\]

\[
\begin{align*}
x_0 &= \text{read()} \\
\text{if } (x_0 > 0) &\quad y_0 = 5 \\
\quad &\quad y_1 = 10 \\
x_1 &= y
\end{align*}
\]

Should this be \(y_0\) or \(y_1\)?
Introducing $\phi$-Node

- Inserts special join functions (called $\phi$-nodes or PHI nodes) at points where different control flow paths converge.

$y_0 = 5$

$y_1 = 10$

$y_2 = \phi(y_0, y_1)$

$x_1 = y_2$

**Note:** $\phi$ is not an executable function!

To generate executable code from this form, appropriate copy statements need to be generated in the predecessors (in other words, reversing the SSA process for code generation).
SSA in a Loop

- Insert $\phi$-nodes in the loop header block

```
x = 0
i = 1
while (i<10) {
    x = x+i
    i = i+1
}
```
ϕ-Node Placement

- When and where to add φ-nodes?
  - If two control paths $A \to C$ and $B \to C$ converge at a node $C$, and both $A$ and $B$ contain assignments to variable $X$, then φ-node for $X$ must be placed at $C$
    - We often call $C$ a join node or convergence point
    - Can be generalized to more than two converging control paths

- Objective: Minimize the number of φ-nodes
  - Need to compute dominance frontier sets
Dominance Frontier

- A basic block $F$ is in the dominance frontier set of basic block $B$ iff
  - $B$ does NOT strictly dominate $F$
  - $B$ dominates some predecessor(s) of $F$

- Intuitively, the nodes in the dominance frontier set of $B$ are *almost* strictly dominated by $B$

- Useful for efficiently computing the SSA form
Dominance Frontier and SSA

- B3 does not dominate B7, but dominates one of its predecessors (i.e., B6)
- Hence B7 is in the dominance frontier set of B3
- B7 is the destination of some edge(s) leaving an area dominated by B3
- For each variable definition in B3, a $\phi$ node is needed in B7

Area dominated by B3
Example: Dominance Frontiers

 CFG

 Dominance frontiers (DF)

<table>
<thead>
<tr>
<th>B</th>
<th>DF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>–</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>
**Dominance Frontiers and Dominator Tree**

**Algorithm to compute DF set**

```plaintext
foreach convergence point\(^1\) X in CFG
foreach predecessor, P, of X in CFG
Run up to Q=IDOM(X) in the dominator tree,
adding X to DF(Y) for each Y between [P, Q)
```

\(^1\) convergence point is a node (basic block) with more than one predecessors
Iterative $\phi$-Node Insertion

- $a$ is defined in 0, 3 need $\phi$ in 7, then $a$ defined in 7 need $\phi$ in 1
- $b$ is defined in 0, 2, 6 need $\phi$ in 7 then $b$ defined in 7 need $\phi$ in 1
- $c$ is defined in 0, 2, 5 need $\phi$ in 6, 7 then $c$ defined in 7 need $\phi$ in 1
- $d$ is defined in 2, 3, 4 need $\phi$ in 6, 7 then $d$ defined in 7 need $\phi$ in 1
- $i$ is defined in 7 need $\phi$ in 1

<table>
<thead>
<tr>
<th>B</th>
<th>DF</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SSA Applications

- SSA form simplifies data flow analysis and many code transformations
  - Primarily due to explicit & simplified (sparse) def-use chains

- Here we show two simple examples
  - Dead code elimination
  - Loop induction variable detection
Dead Code in CDFG

- Dead code is either
  - Unreachable code
  - Definitions never used

- Dead statements?

```plaintext
x = a + b
y = c + d
x = a - b
z = z + 1
y = c - d
z = x + y
f(x, y)
```
Dead Code Elimination with SSA

Iteratively remove unused definitions:
1. Remove \( y_1, z_2 \) (and B4) \( \rightarrow \) 2. Remove \( z_1 \)
An induction variable is a variable that
- Gets increased or decreased by a fixed amount (loop invariant) on every iteration of a loop (basic induction variable)
  - $i = i + c$
- or is an affine function of another induction variable (mutual induction variable)
  - $j = a \times i + b$
Find basic loop induction variable(s)

1. Inspect back edges in the loop
2. Each back edge points to a $\phi$ node in the loop header, which may indicate a basic induction variable
3. $\phi$ is a function of an initialized variable and a definition in the form of “$i + c$” (i.e., increment operation)
“The LLVM Project is a collection of modular and reusable compiler and toolchain technologies...”
What is LLVM?

- Formerly Low Level Virtual Machine
  - Brainchild of Chris Lattner and Vikram Adve back in 2000
  - 2012 ACM Software System Award

- The core of LLVM is the SSA-base IR
  - Language independent, target independent, easy to use
  - RISC-like virtual instructions, unlimited registers, exception handling, etc.

- Many high-quality libraries (components) with clean interfaces
  - Optimizations, analyses, modular code generator, profiling, link time optimization, ARM/X86/PPC/SPARC code generator …
  - Tools built from the libraries
    - C/C++/ObjC compiler, modular optimizer, linker, debugger, LLVM JIT …
What is a Compiler

- Compiler is a tool that inspects and manipulates a representation of programs
  - Intentionally a very board definition
  - Examples:
    - Traditional C compiler (cc), Java JIT compiler (hotspot), system assembler (as), system linker (ld), IDEs, refactoring tools, ...

GCC is a compiler

LLVM is not a compiler
What is a Compiler Infrastructure

- Provides modular & reusable components for building compilers
  - Components are ideally language/target independent
  - Allows choice of the right component for the job

- Allows components to be shared across different compilers
  - Improvements made to one compiler benefits the others

**LLVM is a compiler infrastructure**

source: http://llvm.org
The LLVM C/C++ Compiler

- Clang is a C, C++, and Objective-C front end (sub-project of LLVM)
  - Converts source to LLVM IR
  - GCC compatibility

- Distinguishing features:
  - `.bc` files contain LLVM IR/bitcode, not machine code
  - Executable can be bytecode (JIT’d) or machine code
Events at Compile Time

C/C++ file → Clang LLVM → .o file

C to LLVM Frontend → Compile-time Optimizer

LLVM IR Parser → LLVM Verifier → LLVM Analysis & Optimization Passes → LLVM .bc File Writer

Dead Global Elimination, IP Constant Propagation, Dead Argument Elimination, Inlining, Reassociation, LICM, Loop Opts, Memory Promotion, Dead Store Elimination, ADCE, …
LLVM Program Structure

- Module contains Functions/GlobalVariables
  - Module is unit of compilation/analysis/optimization

- Function contains BasicBlocks/Arguments
  - Functions roughly correspond to functions in C

- BasicBlock contains list of instructions
  - Each block ends in a control flow instruction

- Instruction is opcode + vector of operands
  - All operands have types
  - Instruction result is typed
LLVM Flow Analysis

- LLVM IR is in SSA form
  - use-def and def-use chains are always available
  - All objects have user/use info, even functions

- Control flow graph (CFG) is always available
  - Exposed as BasicBlock predecessor/successor lists
  - Many generic graph algorithms usable with the CFG

- Higher-level info implemented as passes
  - CallGraph, Dominators, LoopInfo, …

source: http://llvm.org
High-level information exposed in the code
- Explicit dataflow through SSA form
- Explicit control-flow graph
- Explicit language-independent type-information
- Explicit typed pointer arithmetic
  - Preserve array subscript and structure indexing

```
for (i=0; i<N; ++i)
  foo(A[i], &P);
```

```
loop:
  %i.1 = phi i5 [ 0, %bb0 ], [ %i.2, %loop ]
  %AiAddr = getelementptr float* %A, i32 %i.1
call void %foo(float %AiAddr, %pair* %P)
  %i.2 = add i5 %i.1, 1
  %tmp = icmp eq i5 %i.1, 16
  br i1 %tmp, label %loop, label %outloop
```
Arbitrary Precision Integers

- LLVM allows arbitrary fixed width integers since version 2.0
  - i2, i13, i128, i1024, etc.

- Essential for hardware synthesis
  - An 11b multiplier is significantly cheaper/faster than a 16b implementation
  - Can leverage other LLVM analyses/optimizations to perform bitwidth minimization
Summary

- Importance of compilers
  - Essential component of SoC software development flow
  - Essential component of high-level synthesis

- A good intermediate representation (IR) enables efficient and effective analysis and optimization
  - Dominance relation helps effective CFG analysis
  - SSA form facilitates efficient IR-level optimization

- LLVM: a modular and powerful open-source compiler infrastructure with a wide range of applications
Before Next Class

- Sign up for paper discussions on Piazza
- Next lecture: Scheduling algorithms
Acknowledgements

★ These slides contain/adapt materials developed by
  – Prof. Scott Mahlke (UMich)
  – Dr. Chris Lattner (Apple Inc.)