GCC Compiler Overview

C  Front End  Middle End  Back End
C++  GENERIC  GIMPLE  RTL  Assembly
Java
Fortran

SSA Optimizers
- vectorization
- loop optimizations
- scalar optimizations: CCP, DCE, DSE, FRE, PRE, VRP, SRA
- field-sensitive, points-to alias analysis

RTL Optimizers
- RTL has infinite registers
- register allocation
- scheduling, SW pipelining, CSE, ...

**LLVM Compiler Infrastructure**

**Goals**
- lifelong analysis and optimization
- modular compiler components

**IR is low level, control-flow graph and SSA**
- language-independent types: any-bit ints, float, double, pointers, arrays, structures, functions
- interprocedural analysis and transformation

**Status**
- Apple is paying main developer (Chris Lattner) to continue development
- built OpenGL JIT compiler with it in two weeks

**Static Single Assignment Form**

**Last Time**
- Induction variable detection and elimination and strength reduction

**Today**
- Program representations
- Static single assignment (SSA) form
  - Program representation for sparse data-flow
- Conversion to and from SSA

**Next Time**
- Applications of SSA
Data Dependence

Definition
– Data dependences are constraints on the order in which statements may be executed
We say statement \( s_2 \) depends on \( s_1 \)
– Flow (true) dependence: \( s_1 \) writes memory that \( s_2 \) later reads (RAW)
– Anti-dependence: \( s_1 \) reads memory that \( s_2 \) later writes (WAR)
– Output dependences: \( s_1 \) writes memory that \( s_2 \) later writes (WAW)
– Input dependences: \( s_1 \) reads memory that \( s_2 \) later reads (RAR)

True dependences
– Flow dependences represent actual flow of data

False dependences
– Anti- and output dependences reflect reuse of memory, not actual data flow; can often be eliminated

Example

\[
\begin{align*}
\text{s}_1 & : a = b; \\
\text{s}_2 & : b = c + d; \\
\text{s}_3 & : e = a + d; \\
\text{s}_4 & : b = 3; \\
\text{s}_5 & : f = b \times 2;
\end{align*}
\]
Representing Data Dependences

Implicitly
- Using variable defs and uses
- Pros: simple
- Cons: hides data dependence (analyses must find this info)

Def-use chains (du chains)
- Link each def to its uses
- Pros: explicit; therefore fast
- Cons: must be computed and updated, space consuming

Alternate representations
- e.g., Static single assignment form (SSA), Program Dependence Graph (PDG), dependence flow graphs (DFG), value dependence graphs (VDG),

DU Chains

Definition
- du chains link each def to its uses

Example

\[
\begin{align*}
  s_1 & \quad a = b; \\
  s_2 & \quad b = c + d; \\
  s_3 & \quad e = a + d; \\
  s_4 & \quad b = 3; \\
  s_5 & \quad f = b * 2;
\end{align*}
\]
**UD Chains**

**Definition**
- ud chains link each use to its defs

**Example**

\[
\begin{align*}
    s_1 & : a = b; \\
    s_2 & : b = c + d; \\
    s_3 & : e = a + d; \\
    s_4 & : b = 3; \\
    s_5 & : f = b * 2;
\end{align*}
\]

- ud chain

**Role of Alternate Program Representations**

**Advantage**
- Allow analyses and transformations to be simpler & more efficient/effective

**Disadvantage**
- May not be “executable” (requires extra translations to and from)
- May be expensive (in terms of time or space)

**Process**

Original Code (RTL) \[\rightarrow\] SSA Code1 \[\xrightarrow{T1}\] SSA Code2 \[\xrightarrow{T2}\] SSA Code3 \[\rightarrow\] Optimized Code (RTL)
Static Single Assignment (SSA) Form

Idea

– Each variable has only one static definition
– Makes it easier to reason about values instead of variables
– Similar to the notion of functional programming

Transformation to SSA

– Rename each definition
– Rename all uses reached by that assignment

Example

\[
\begin{align*}
  v &:= \ldots \\
  \ldots &:= \ldots v \ldots \\
  v &:= \ldots \\
  \ldots &:= \ldots v \ldots \\
\end{align*}
\]

\[
\begin{align*}
  v_0 &:= \ldots \\
  \ldots &:= \ldots v_0 \ldots \\
  v_1 &:= \ldots \\
  \ldots &:= \ldots v_1 \ldots \\
\end{align*}
\]

What do we do when there’s control flow?

SSA and Control Flow

Problem

– A use may be reached by several definitions
SSA and Control Flow (cont)

Merging Definitions
– $\phi$-functions merge multiple reaching definitions

Example

Another Example
SSA vs. ud/du Chains

SSA form is more constrained

Advantages of SSA
- More compact
- Some analyses become simpler when each use has only one def
- Value merging is explicit
- Easier to update and manipulate?

Furthermore
- Eliminates false dependences (simplifying context)

```cpp
for (i=0; i<n; i++)
    A[i] = i;
for (i=0; i<n; i++)
    print(foo(i));
```

SSA vs. ud/du Chains (cont)

Worst case du-chains?

```cpp
switch (c1) {
    case 1:  x = 1; break;
    case 2:  x = 2; break;
    case 3:  x = 3; break;
}
x_4 = \phi(x_1, x_2, x_3)
switch (c2) {
    case 1:  y1 = x; break;
    case 2:  y2 = x; break;
    case 3:  y3 = x; break;
    case 4:  y4 = x; break;
}
m \text{ defs and } n \text{ uses leads to } m \times n \text{ du chains}
```
Transformation to SSA Form

Two steps
- Insert $\phi$-functions
- Rename variables

Where Do We Place $\phi$-Functions?

Basic Rule
- If two distinct (non-null) paths $x \rightarrow z$ and $y \rightarrow z$ converge at node $z$, and nodes $x$ and $y$ contain definitions of variable $v$, then a $\phi$-function for $v$ is inserted at $z$

```
x  v_1 := ...  y  v_2 := ...
      \downarrow       \downarrow
        z  v_3 := \phi(v_1, v_2)  \ldots v_3 \ldots
```
Approaches to Placing $\phi$-Functions

Minimal
- As few as possible subject to the basic rule

Briggs-Minimal
- Same as minimal, except $v$ must be live across some edge of the CFG

Pruned
- Same as minimal, except dead $\phi$-functions are not inserted

What’s the difference between Briggs Minimal and Pruned SSA?

Briggs Minimal vs. Pruned

Briggs Minimal will add a $\phi$ function because $v$ is live across the blue edge, but Pruned SSA will not because the $\phi$ function is dead.

Neither Briggs Minimal nor Pruned SSA will place a $\phi$ function in this case because $v$ is not live across any CFG edge.

Why would we ever use Briggs Minimal instead of Pruned SSA?
Machinery for Placing $\phi$-Functions

Recall Dominators
- $d \text{ dom } i$ if all paths from entry to node $i$ include $d$
- $d \text{ sdom } i$ if $d \text{ dom } i$ and $d \not\text{ i}$

Dominance Frontiers
- The dominance frontier of a node $d$ is the set of nodes that are “just barely” not dominated by $d$; i.e., the set of nodes $n$, such that
  - $d$ dominates a predecessor $p$ of $n$, and
  - $d$ does not strictly dominate $n$
- $DF(d) = \{n \mid \exists p \in \text{pred}(n), d \text{ dom } p \text{ and } d \not\text{ sdom } n\}$

Notational Convenience
- $DF(S) = \bigcup_{n \in S} DF(n)$

Dominance Frontier Example

$DF(d) = \{n \mid \exists p \in \text{pred}(n), d \text{ dom } p \text{ and } d \not\text{ sdom } n\}$

$Dom(5) = \{5, 6, 7, 8\}$

$DF(5) = \{4, 5, 12, 13\}$

What’s significant about the Dominance Frontier?
In SSA form, definitions must dominate uses
Dominance Frontier Example II

\[ DF(d) = \{ n \mid \exists p \in \text{pred}(n), d \text{ dom } p \text{ and } d \not\text{sdom } n \} \]

\[ \text{Dom}(5) = \{ 5, 6, 7, 8 \} \]

\[ DF(5) = \{ 4, 5, 13 \} \]

In this graph, node 4 is the first point of convergence between the entry and node 5, so do we need a \( \phi \) function at node 13?


SSA Exercise

\[ DF(8) = \{ 10 \} \]

\[ DF(9) = \{ 10 \} \]

\[ DF(2) = \{ 6 \} \]

\[ DF(\{ 8, 9 \}) = \{ 10 \} \]

\[ DF(10) = \{ 6 \} \]

\[ DF(\{ 2, 8, 9, 6, 10 \}) = \{ 6, 10 \} \]
**Dominance Frontiers Revisited**

Suppose that node 3 defines variable x

\[ \text{DF}(3) = \{5\} \]

Do we need to insert a \( \phi \)-function for x anywhere else?

Yes. At node 6. Why?

---

**Dominance Frontiers and SSA**

Let

- \( \text{DF}_1(S) = \text{DF}(S) \)
- \( \text{DF}_{i+1}(S) = \text{DF}(S \cup \text{DF}_i(S)) \)

**Iterated Dominance Frontier**

- \( \text{DF}_\infty(S) \)

**Theorem**

- If \( S \) is the set of CFG nodes that define variable v, then \( \text{DF}_\infty(S) \) is the set of nodes that require \( \phi \)-functions for v
The dominance tree shows the dominance relation

### Dominance Tree Example

![Dominance Tree Diagram]

### Inserting Phi Nodes

**Calculate the dominator tree**
- A lot of research has gone into calculating this quickly

**Computing dominance frontier from dominator tree**
- \( DF_{local}[n] \) = successors of \( n \) (in CFG) that are not strictly dominated by \( n \)
- \( DF_{up}[n] \) = nodes in the dominance frontier of \( n \) that are not strictly dominated by \( n \)’s immediate dominator

\[
DF[n] = DF_{local}[n] \cup \bigcup_{c \in \text{children}[n]} DF_{up}[c]
\]
**Algorithm for Inserting \( \phi \)-Functions**

\[
\text{for each variable } v \\
\text{WorkList } \leftarrow \emptyset \\
\text{EverOnWorkList } \leftarrow \emptyset \\
\text{AlreadyHasPhiFunc } \leftarrow \emptyset \\
\text{for each node } n \text{ containing an assignment to } v \\
\text{WorkList } \leftarrow \text{WorkList } \cup \{n\} \\
\text{EverOnWorkList } \leftarrow \text{WorkList} \\
\text{while } \text{WorkList } \neq \emptyset \\
\text{Remove some node } n \text{ for WorkList} \\
\text{for each } d \in \text{DF}(n) \\
\text{if } d \notin \text{AlreadyHasPhiFunc} \\
\text{Insert a } \phi \text{-function for } v \text{ at } d \\
\text{AlreadyHasPhiFunc } \leftarrow \text{AlreadyHasPhiFunc } \cup \{d\} \\
\text{if } d \notin \text{EverOnWorkList} \\
\text{WorkList } \leftarrow \text{WorkList } \cup \{d\} \\
\text{EverOnWorkList } \leftarrow \text{EverOnWorkList } \cup \{d\}
\]

**Transformation to SSA Form**

**Two steps**

- Insert \( \phi \)-functions
- Rename variables
Variable Renaming

Basic idea
- When we see a variable on the LHS, create a new name for it
- When we see a variable on the RHS, use appropriate subscript

Easy for straightline code

\[
\begin{align*}
x &= x \\
x &= x \\
\end{align*}
\]

\[
\begin{align*}
x_0 &= x_0 \\
x_1 &= x_1 \\
\end{align*}
\]

Use a stack when there’s control flow
- For each use of \(x\), find the definition of \(x\) that dominates it

Variable Renaming (cont)

Data Structures
- \(\text{Stacks}[v] \forall v\)
  Holds the subscript of most recent definition of variable \(v\), initially empty
- \(\text{Counters}[v] \forall v\)
  Holds the current number of assignments to variable \(v\); initially 0

Auxiliary Routine

\[
\text{procedure GenName}(\text{variable } v) \\
i := \text{Counters}[v] \\
push i \text{ onto Stacks}[v] \\
\text{Counters}[v] := i + 1
\]

Use the Dominance Tree to remember the most recent definition of each variable
Variable Renaming Algorithm

```
procedure Rename(block b)
  if b previously visited return  

  for each statement s in b (in order)
    for each variable v ∈ RHS(s) (except for φ-functions)
      replace v by v_i, where i = Top(Stacks[v])
    for each variable v ∈ LHS(s)
      GenName(v) and replace v with v_i, where i=Top(Stack[ɔ])
  for each s ∈ succ(b) (in CFG)
    j ← position in s’s φ-function corresponding to block b
    for each φ-function p in s
      replace the jth operand of RHS(p) by v_i, where i = Top(Stack[v])

  for each s ∈ child(b) (in DT)
    Rename(s)
  for each φ-function or statement t in b
    for each v_i ∈ LHS(t)
      Pop(Stack[v])
```

Transformation from SSA Form

Proposal
- Restore original variable names (*i.e.*, drop subscripts)
- Delete all φ-functions

Complications (the proposal doesn’t work!)
- What if versions get out of order? (simultaneously live ranges)

Alternative
- Perform dead code elimination (to prune φ-functions)
- Replace φ-functions with copies in predecessors
- Rely on register allocation coalescing to remove unnecessary copies
**Concepts**

**Data dependences**
- Three kinds of data dependences
  - du-chains

**Alternate representations**

**SSA form**

**Conversion to SSA form**
- $\phi$-function placement
- Variable renaming

---

**Next Time**

**Assignments**
- Read Alpern and Zadeck paper on value numbering

**Lecture**
- Using SSA to apply optimizations