9 静态单赋值

徐辉，xuh@fudan.edu.cn

本章学习目标：

- 了解静态单赋值形式
- 掌握 Chaotic Iteration 数据流分析方法
- 掌握静态单赋值形式的构造方法

9.1 静态单赋值

静态单赋值（SSA: Static Single Assignment）的目标是为了显式表示变量的 def-use 关系，便于数据流分析和优化 [1]。SSA 一般有如下要求：

- 命名：每个变量（id）只赋值一次[1]，如存在多次赋值，则需重命名（上节课翻译的 LLVM IR 已经满足）。
- Phi 指令：如由于控制流原因导致变量在某处的 use 对应不同的 def，应使用 phi 指令表示。
- 最简化：使用最少数目的 phi 指令，简化数据流关系。

9.2 IR 优化：删除冗余 Load/Store 指令

AST 翻译 IR 时会引入冗余的 Load 和 Store 指令，我们采用 Chaotic Iteration 的数据流分析方法分别对这两种冗余进行分析和优化。

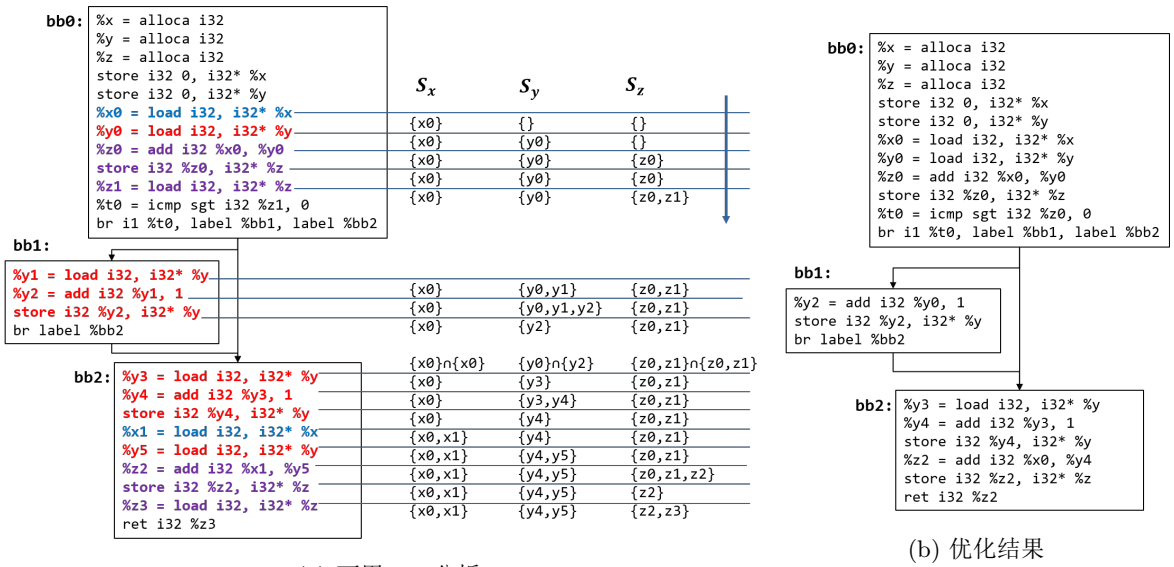
9.2.1 消除冗余 Load

图 9.1a是一段 IR，其中冗余的 load 操作或虚拟寄存器有 x1, y1, y5, z1，z3。以 x1 为例，由于在 def(x0) 和 def(x1) 之间没有 store(x)，def(x0) 和 def(x1) 完全相同，因此 use(x1) 都可以使用 use(x0) 代替。基于该规律，我们总结出表 9.1中三种指令对应的 transfer 函数，按照算法 1将其应用于图 9.1a便可得到每个程序节点每个变量对应的可用虚拟寄存器。以 load(x1) 这一程序节点为例，由于已经存在 x0，便可知其冗余。优化后的结果如图 9.1b所示。

表 9.1: Load 分析 Transfer 函数定义

IR	举例	Transfer 函数
load 指令	`%t = load i32, i32* %x`	$S_x = S_x \cup \{t\}$
store 指令	`store i32 %t, i32* %x`	$S_x = \{t\}$
二元运算	`%t = bop %t1, %t2`	$S_x = S_x \cup \{t\}$, s.t. $t \in *x$

[1]原论文 [1] 只要求每个 use 对应一个 def



(a) 可用 def 分析

(b) 优化结果

图 9.1: Load 指令优化

算法 1 可用 def 分析

Require: IR and variables of a target function

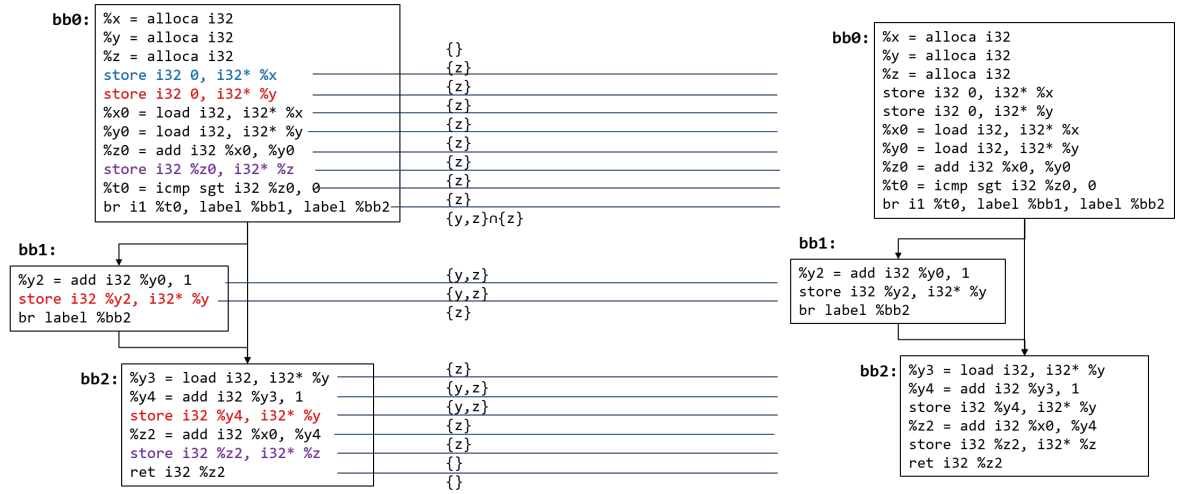
1: **for each** $i \in irs$ **do**
2: $IN[i] \leftarrow \{S_v = \emptyset \mid v \in Var\}$;
3: $OUT[i] \leftarrow \{S_v = \emptyset \mid v \in Var\}$;
4: **end for**
5: **repeat**
6: **for each** $i \in irs$ **do**
7: **for each** $p \in$ Predecessor(n) **do**
8: $IN[i] \leftarrow IN[i] \cup OUT[p]$;
9: **end for**
10: $OUT[i] \leftarrow$ Transfer(i);
11: **end for**
12: **until** $IN[i]$ and $OUT[i]$ stops changing for all i

9.2.2 消除冗余 Store

如果一个变量的两条 store 语句之间没有 load 操作，则前一条 store 是冗余操作，可以直接删除。以图 9.2a为例，由于 bb2 中的 store(z1) 语句和 bb1 中的 store(z0) 之间没有 load(z) 操作，因此可以删除 store(z0)。表 9.2定义了不同指令对应的 transfer 函数，根据算法 2对 IR 控制流图进行逆向遍历变可识别所有符合条件的冗余 store 操作。化简结果如图 9.2b所示。

表 9.2: Store 分析 Transfer 函数定义

IR	举例	Transfer 函数
load 指令	`%t = load i32, i32* %x`	$S = S \setminus \{x\}$
store 指令	`store i32 %t, i32* %x`	$S = S \cup \{x\}$
二元运算	`%t = alloc, i32* %x`	$S = S \setminus \{x\}$



(a) 可用 Store 分析

(b) 优化结果

图 9.2: Store 指令优化

算法 2 可用 store 分析

Require: IR and variables of a target function

1: **for each** $i \in irs$ **do**
2: $IN[i] \leftarrow \{S = \emptyset\}$;
3: $OUT[i] \leftarrow \{S = \emptyset\}$;
4: **end for**
5: **repeat**
6: **for each** $i \in irs$ **do**
7: **for each** $p \in$Successor(n) **do**
8: $OUT[i] \leftarrow OUT[i] \cap IN[p]$;
9: **end for**
10: $IN[i] \leftarrow$ Transfer(i);
11: **end for**
12: **until** $IN[i]$ and $OUT[i]$ stops changing for all i

9.3 IR => 静态单赋值

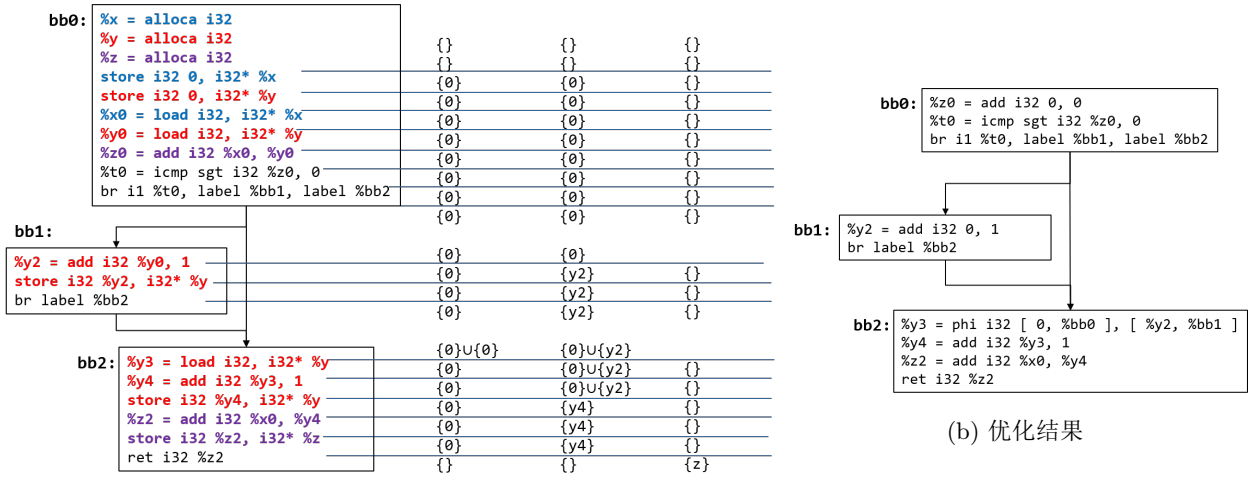
下面介绍 LLVM IR 到 SSA 的翻译方法，共分为两步：1）转化为纯寄存器表示；2）简化数据流关系。

9.3.1 转化为纯寄存器表示

这一步的目的是消除所有针对局部变量的 store 和 load 指令，即不使用栈帧内存。其关键问题是有些 load 可能对应多个 store 或多种 def，需要引入 phi 指令来表示。以图 9.3a为例，bb2 中的 load(y3) 对应 bb0 中的 y0（路径：bb0->bb2）或 bb1 中的 y2（路径：bb0->bb1->bb2）。因此可以采用数据流分析方法分析 store 导致的 def-use 关系，即正序遍历控制流图，遇到 store 节点则应用表 9.3中定义的 transfer 函数，遇到合并节点则取并集。

表 9.3: 数值流分析 Transfer 函数定义

IR	举例	Transfer 函数
store 指令	`store i32 %t, i32* %x`	$S_x = \{t\}$



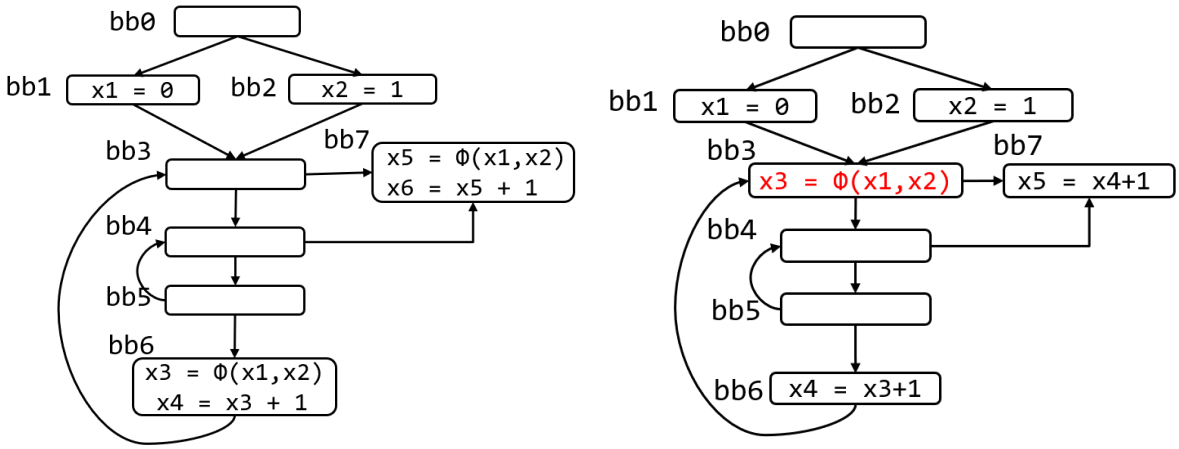
(a) 数值流分析

(b) 优化结果

图 9.3: 使用 Phi 指令替换 Store-Load

9.3.2 Phi 指令优化

纯寄存器表示形式的 IR 通过单赋值和 phi 指令将 def-use 关系显式表示出来，但未能有效优化 def-use 关系的复杂度。以图 9.4a为例，在 bb4 和 bb5 中插入同样的 phi 指令后，其 def-use 关系数量是 2*2，并且随控制流深度增加呈指数增加。如将 phi 指令前移到 bb3（图 9.4b），可将 def-use 关系变为 2+2，避免指数爆炸。该优化亦可以通过数据流分析方法逆向遍历控制流图实现，即识别重复的 phi 指令（参数相同），并将其前移。



(a) 未优化 phi 指令的 IR

(b) 优化结果

图 9.4: Phi 指令优化

对于如何确定 phi 指令的放置位置还有一种标准的基于支配边界的构造方法。亦可以先确定 phi 指令，然后对 IR 进行重新编号和优化。

定义 1 (支配). 给定有向图 $G(V, E)$ 与起点 v_0，如果从 v_0 到某个点 v_j 均需要经过点 v_i ，则称 v_i 支配 v_j 或 $v_i \in Dom(v_j)$ 如果 $v_i \neq v_j$，则称 v_i 严格支配 v_j。

定义 2 (支配边界). v_i 的支配边界是所有满足条件的 v_j 的集合：

- v_i 支配 v_j 的一个前序节点
- v_i 并不严格支配 v_j

以图 9.4a为例，每个节点的支配边界分析结果如下：

$$DF(bb_0) = \emptyset$$

$$DF(bb_1) = \{bb_3\}$$

$$DF(bb_2) = \{bb_3\}$$

$$DF(bb_3) = \{bb_3\}$$

$$DF(bb_4) = \{bb_3, bb_7\}$$

$$DF(bb_5) = \{bb_4\}$$

$$DF(bb_6) = \{bb_3\}$$

$$DF(bb_7) = \emptyset$$

如果在某节点对变量 x 进行了赋值，则应在其支配边界放置 phi(x)。以图 9.4a为例，由于 bb1 的支配边界是 bb3，并且 bb1 对 x 进行了赋值，因此应在 bb3 插入 phi(x)。

Bibliography

[1] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "An efficient method of computing static single assignment form." In Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages, pp. 25-35. 1989.