8 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
- 掌握基于循环迭代的数据流分析方法
- 掌握静态单赋值形式的构造方法

8.1 静态单赋值

静态单赋值（SSA: Static Single Assignment） [1] 是一类特殊的线性 IR，其提出目的是为了简明表示变量的 def-use 关系，便于后续的代码优化。SSA 一般有如下要求：

- 标识符定义：每个标识符只能被定义或赋值（def）一次，如需要修改其值，则只能使用其它的标识符。
- Phi 指令：如果由于控制流原因导致标识符在某处被使用时（use）对应多种不同来源的 def，应使用 phi 指令表示。
- 优化：使用最少数目的 phi 指令，简化数据流关系。

我们上一章使用的 LLVM IR 已经满足标识符只定义一次的要求，但并未使用 phi 指令。当存在不同控制流对应不同的变量值的时候，我们是使用 store-load 解决的，而非 phi 指令。接下来我们讨论如何将上一章使用的 LLVM IR 中的 load-store 替换为 phi，并最终转化为最优的 SSA 形式。

8.2 消除 IR 中冗余的 load/store

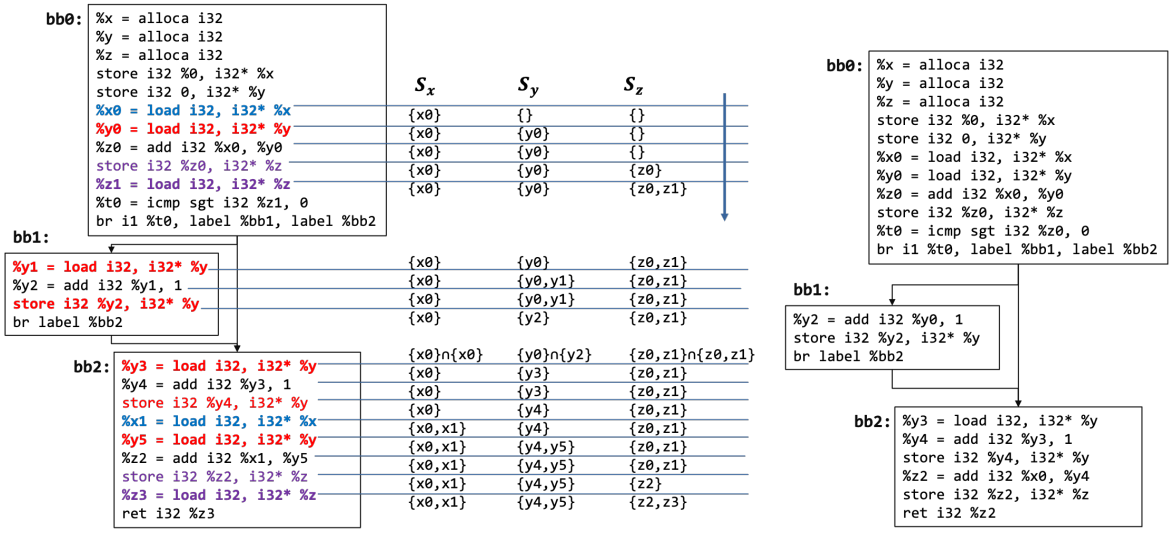
AST 翻译 IR 时为了降低 def-use 的复杂性，我们要求使用变量前必须先 load，更新变量值后必须立即 store，这样会引入大量冗余的 load 和 store 指令。本节我们采用基于循环迭代（Chaotic Iteration）的数据流分析方法消除 IR 中冗余的 load 和 store。

8.2.1 消除冗余 load

图 8.1a展示了一段 IR，其 bb2 代码块中将 x 的值 load 到 x1 的操作是冗余的，可以直接使用 bb0 代码块中定义的 x0。其规律是 def(x0) 和 def(x1) 之间没有 store(x)，则 def(x0) 和 def(x1) 完全相同，因此 use(x1) 都可以使用 use(x0) 代替。这段代码中类似的冗余 load 指令还包括 y1, y5, z1，z3。

为了实现自动化的冗余 load 指令检测，我们可以对每个变量的可用 load 指令或 def 进行记录。如果两次 def 之间没有 store 操作，则说明后一条 def 冗余；反之，则说明之前的 def 失效。基于上述分析，我们总结出与可用寄存器分析相关的指令及其影响，即表 8.1定义的 transfer 函数。我们可以将上述 transfer 函数应用于代码块中的指令序列，但如果涉及到控制流和循环，还需要更多的设计。循环迭代算法是一种应对控制流的常用分析框架，根据具体的分析任务需要设计不同的操作。如算法 1所示，其对每条指令 `i` 进行分析，得到 `OUT[i]`。如果该指令有若干个前驱节点，则取并集处理。如果有循环则迭代该分析过程

直到每个程序节点的分析结果不再变化为止。将该算法应用于图 8.1a便可得到所有变量在每个程序节点对应的可用虚拟寄存器。优化后的结果如图 8.1b所示。



(a) 可用 def 分析　　(b) 优化结果

图 8.1: Load 指令优化

表 8.1: Transfer 函数定义：可用 load 指令分析

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

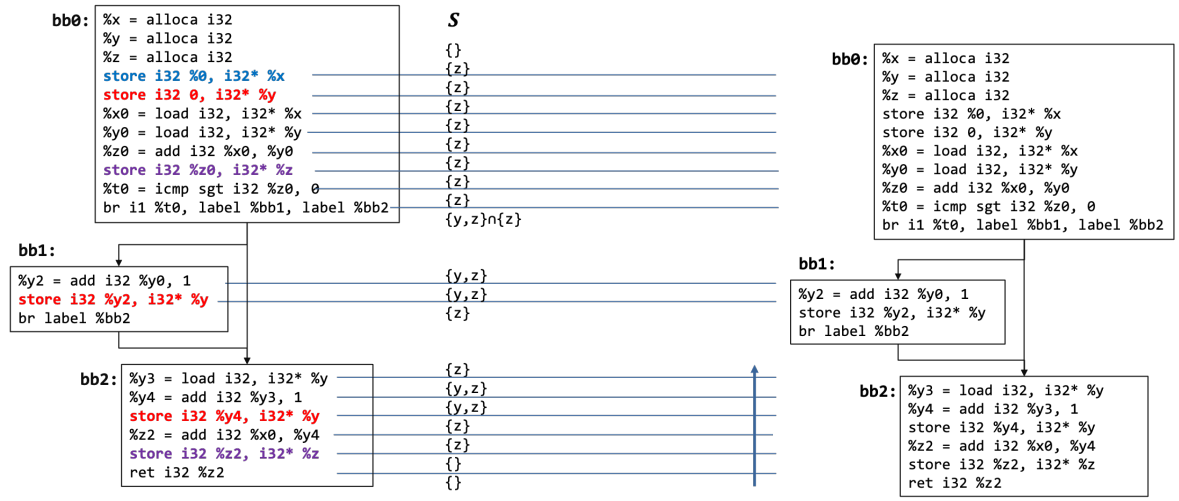
算法 1 循环迭代算法：可用 load 指令分析

Require: IR and variables of a target function

for each $i \in irs$ **do**
  $IN[i] \leftarrow \{S_v = \emptyset \mid v \in Var\}$;
  $OUT[i] \leftarrow \{S_v = \emptyset \mid v \in Var\}$;
end for
repeat
  for each $i \in irs$ **do**
    for each $p \in$ Predecessor(i) **do**
      $IN[i] \leftarrow IN[i] \cap OUT[p]$;
    end for
    $OUT[i] \leftarrow$ Transfer(i);
  end for
until $IN[i]$ and $OUT[i]$ stop changing for all i

8.2.2 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 8.2a为例，由于 bb1 中的 store(z0) 和 bb2 中的 store(z1) 语句之间没有 load(z) 操作，因此可以删除 store(z0)。表 8.2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可识别出所有符合条件的冗余 store 操作。化简结果如图 8.2b所示。



(a) 可用 Store 分析　　(b) 优化结果

图 8.2: Store 指令优化

表 8.2: Transfer 函数定义：可用 store 分析

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

算法 2 循环迭代算法：可用 store 分析

Require: IR and variables of a target function

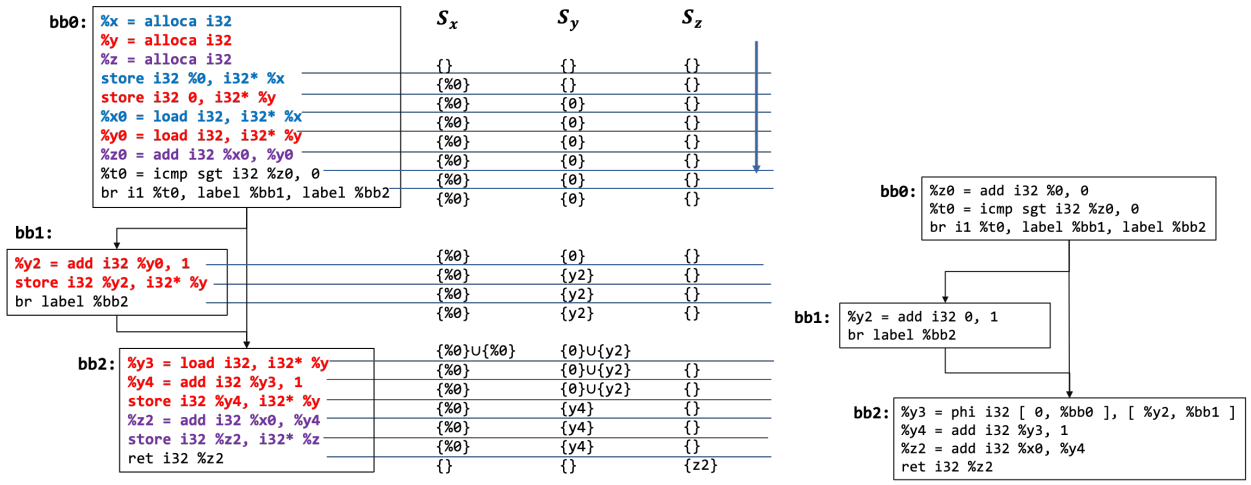
1: **for each** $i \in irs$ **do**
2: $IN[i] \leftarrow \emptyset$;
3: $OUT[i] \leftarrow \emptyset$;
4: **end for**
5: **repeat**
6: **for each** $i \in irs$ **do**
7: **for each** $s \in$Successor(n) **do**
8: $OUT[i] \leftarrow OUT[i] \cap IN[s]$;
9: **end for**
10: $IN[i] \leftarrow$ Transfer(i);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

8.3 转换为静态单赋值形式

这一步的目的是消除 IR 中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 可能对应多个控制流带来的不同定义，需要引入 phi 指令来表示。以图 8.3a为例，bb2 中的 load(y3) 可能对应 bb0 中的 y0（路径：bb0->bb2）或 bb1 中的 y2（路径：bb0->bb1->bb2）。下面介绍 LLVM IR 到 SSA 的翻译方法，分为两步：1）变量数值定义分析；2）使用 phi 指令替换 store-load。

8.3.1 变量数值定义分析

对于变量数值定义分析，我们可以继续采用循环迭代方法分析 store 对 def-use 关系的影响，即正向遍历控制流图，遇到 store 指令则应用表 8.3中定义的 transfer 函数，遇到合并节点则取并集。



(a) 数值流分析　　(b) 优化结果

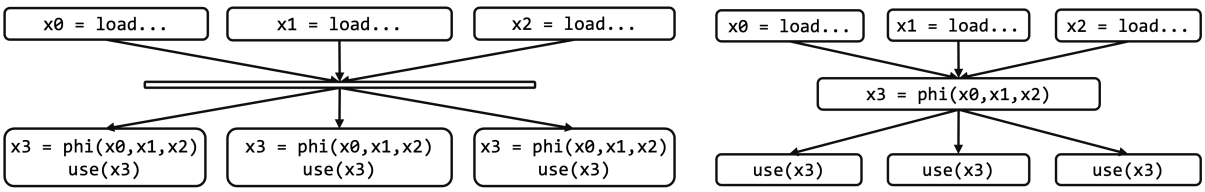
图 8.3: 使用 phi 指令替换 store-load

表 8.3: Transfer 函数定义：def-use 分析

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

8.3.2 使用 phi 指令替换 store-load

确定了每个程序节点可能的变量数值定义，只需在存在多个来源的数值定义处使用 phi 指令即可。值得注意的是，纯寄存器表示形式的 IR 将变量的 def-use 关系显式表示出来，但未必可以有效优化 def-use 关系的复杂度。为了达到最优的 phi 指令使用方法，应当尽量在最靠近起始代码块的地方插入 phi 指令。以图 8.4a为例，使用时寄存器表示后，其 def-use 关系数量是 3*3，并且随控制流深度增加呈指数增加。如果将 phi 指令前移（图 8.4b），def-use 关系变为 3+3，避免了指数爆炸问题。



(a) 非最优 phi 指令放置　　(b) 最优 phi 指令放置

图 8.4: Phi 指令放置位置与 def-use 关系的优化举例

对于如何确定 phi 指令的放置位置，实际中更常用的是基于支配边界的构造方法，即先确定 phi 指令的放置位置，再对 IR 进行重新编号和优化。下面介绍支配和支配边界的概念。

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 或 $v_i \in IDom(v_j)$。

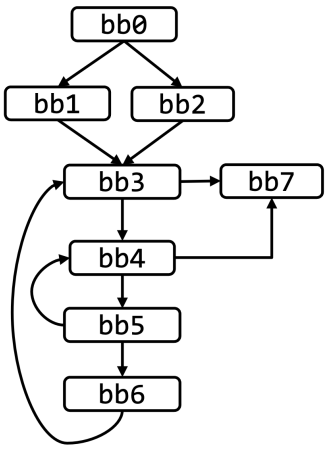


图 8.5: 控制流图举例

支配节点的计算可以采用基于循环迭代的分析方法实现，即正向遍历控制流，并维护从起始节点到当前节点所有经过的代码块；如果遇到分支节点取交集即可。以图 8.5为例，每个节点的支配节点分析结果如下：

$$Dominator(bb_0) = \{bb_0\}$$

$$Dominator(bb_1) = \{bb_0, bb_1\}$$

$$Dominator(bb_2) = \{bb_0, bb_2\}$$

$$Dominator(bb_3) = \{bb_0, bb_3\}$$

$$Dominator(bb_4) = \{bb_0, bb_3, bb_4\}$$

$$Dominator(bb_5) = \{bb_0, bb_3, bb_4, bb_5\}$$

$$Dominator(bb_6) = \{bb_0, bb_3, bb_4, bb_5, bb_6\}$$

$$Dominator(bb_7) = \{bb_0, bb_3, bb_7\}$$

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

有了控制流图每个节点的 $v_j \in V$ 的前驱节点集合 P_j 和支配节点集合 D_j，则节点的支配关系可以直接基于集合分析得到，即 $\forall v_p \in P_j$, $\forall v_i \in D_p \backslash ID_j$, $v_j \in DF(v_i)$。图 8.5中每个节点的支配边界分析结果如下：

$$DF(bb_0) = \emptyset$$

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

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

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

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

$$DF(bb_5) = \{bb_3, bb_4\}$$

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

$$DF(bb_7) = \emptyset$$

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 8.5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)。

练习

1. 代码 8.1是实现阶乘函数的 IR，1）消除其中冗余的 load-store 指令；2）将其转转换纯寄存器表示；3）如果存在 phi 指令冗余，对 phi 指令进行优化。

```
define i32 @fac(i32 %0) {
bb0:
    %n = alloca i32
    %r = alloca i32
    store i32 %0, i32* %n
    store i32 1, i32* %r
    br label %bb1
bb1:
    %t1 = load i32, i32* %n
    %t2 = icmp sgt i32 %t1, 0
    br i1 %t2, label %bb2, label %bb3
bb2:
    %t3 = load i32, i32* %r
    %t4 = load i32, i32* %n
    %t5 = mul i32 %t3, %t4
    store i32 %t5, i32* %r
    %t6 = load i32, i32* %n
    %t7 = sub i32 %t6, 1
    store i32 %t7, i32* %n
    br label %bb1
bb3:
    %t8 = load i32, i32* %r
    ret i32 %t8
}
```

代码 8.1: IR 代码

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.