第 3 章

基于数据流分析的优化

上一章讨论的数据流分析的基本原理和方法，为基于数据流分析的编译器优化奠定了基础。本章将讨论几种重要的基于数据流分析的优化算法，包括无用代码删除、常量与拷贝传播、公共子表达式删除和部分冗余删除等。这些优化算法不但本身非常重要且常用，更作为具体实例，展示了基于数据流分析，进行程序进行优化的一般性方法和过程。

3.1 引言

基于数据流分析的编译器优化都具有类似的概念架构，即编译器首先执行数据流分析算法，得到待优化程序的数据流信息；然后，编译器利用这些数据流信息，对程序进行改写以优化代码。图3.1 给出了基于数据流分析的编译优化执行架构图。待优化的输入程序被一系列的程序优化算法处理，直到得到最终的输出。

在设计和实现基于数据流分析的编译优化流程时，还需要考虑几个关键点。首先，从编译优化的视角，程序分析（也包括这里讨论的数据流分析）的主要作用是提供程序分析的

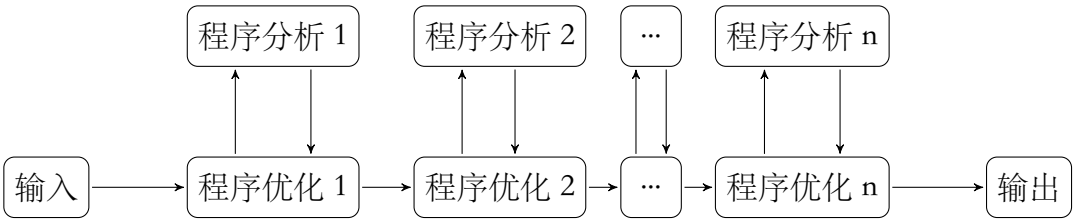


图 3.1: 基于数据流分析的程序优化架构

结果，来帮助编译器判定编译优化的安全性和收益。安全性指的是编译器必须保证待执行的优化不会改变程序的语义，这是对优化的基线要求。而收益指的是编译器必须能够判定待执行的优化，能够到达预期的优化目标。并且，主要的程序分析往往仅从程序中分析和抽取必要的程序信息，而不会对程序进行修改。按这种函数式风格进行的程序分析，大大简化了基于数据流分析的优化器架构。

其次，编译器设计者必须决策要执行哪些编译优化，以及这些编译优化需要哪些程序分析的支持。编译优化选择的决策，既取决于编译的源语言和目标机器的具体特点，也取决于具体优化目标。而一般的，数据流分析对支持全局或过程间优化非常有效，这些优化包括但不限于删去无用代码、代码移动、或冗余计算的移除等。还需要特别注意的是，

其次，编译器设计者必须决策和布置优化的顺序和排列。【优化可能进行多遍】并且，还需要注意到不同的优化往往可能共享同一个数据分析

最后，【程序优化和 IR 的关系】。

在本章中，我们将深入探讨数据流分析的关键内容。首先，介绍支配节点的概念及其在控制流图中的应用，帮助理解控制流的支配关系。接着，讨论活跃分析，通过确定程序中活跃的变量来优化代码执行。然后，研究可用表达式分析，

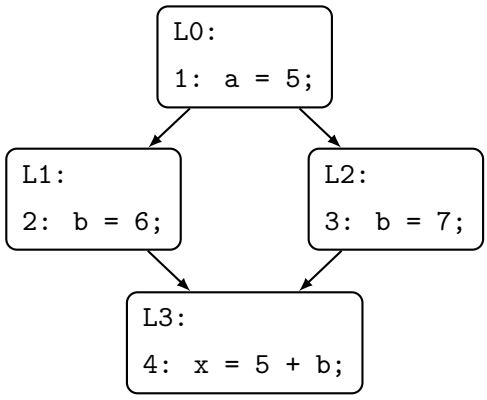


图 3.2: 死代码删除示例

识别程序中可以重用的表达式，减少冗余计算。随后，探讨忙碌表达式分析，确定即将使用的表达式，以优化计算资源的分配。接着，介绍到达定值分析，帮助确定变量的值是否可以到达特定位置，以提高执行效率。此外，进一步介绍了控制流分析算法的改进与优化，以提高其效率。最后，对不同数据流分析技术进行比较，分析其优缺点及实际应用效果。通过这些内容，读者将全面理解数据流分析在编译器优化中的作用，并掌握其实际应用方法。

3.2 无用代码删除

程序中经常包含一些对外部结果没有任何影响的操作，称为*无用代码*（Useless code）。如果编译器能够确定某个操作不会影响程序的最终结果，那么它就可以安全地移除该操作。这种优化技术称为*无用代码删除*（Useless code elimination, UCE），是程序优化中常见且关键的步骤。例如，图3.2中，语句 1 中的定义的变量 a 在语句 1 之后未被使用，因此这条语句是死代码，可以删除。

删除这些无用的代码可以减小程序的体积，进而降低内存和计算资源的消耗，最终提升程序的执行效率。

算法 3.1 无用代码删除

输入： 控制流图 G

输出： 经过无用代码删除后的控制流图

```
1: function uselessCodeRemove(G)
2:     ⟨_, Out⟩ =livenessAnalysis(G)
3:     for each statement x = e ∈ G do
4:         if x ∉ Out[n] then
5:             remove the statement n : x = e from G
```

实现死代码删除的一种有效方法是利用活跃分析的结果。通过活跃分析，编译器能够确定哪些变量在特定程序点之后仍然会被使用，而哪些变量已经不再被使用。从而，编译器可将对不会被使用的变量赋值语句删除。

算法3.1给出了一个基于活跃分析的死代码删除算法。算法接受一个控制流图 G 作为输入，并通过活跃分析 livenessAnalysis 算法计算每个语句的 Out 集合，这一步旨在确定在每个程序点之后哪些变量是活跃的。接着，算法遍历控制流图中的每个赋值语句 $n : x = e$，检查变量 x 是否出现在该语句的 Out 集合中。如果变量 x 不在 $Out[n]$ 中，则说明该赋值语句的结果在后续程序执行中不会被使用，因此可以安全地删除这条语句。最后，算法返回经过死代码删除优化后的控制流图。

在应用死代码删除算法时，必须确保删除的代码 $n : x = e$ 没有副作用。副作用包括修改全局变量、输入输出操作、或调用可能改变程序状态的函数。如果语句 $n : x = e$ 存在副作用，则即使变量 $x \notin out[n]$，该语句也不能被删除。例如，语句 $a = print(42)$ 执行时会输出值 42，因此即使 a 不在该语句的 $Out[]$ 集合中，这条语句也不能删除，否则会改变程序的可观察行为。

3.3 常量与拷贝传播

在??节，我们介绍了到达定值分析并简要介绍了到达定值信息的用途。为了详细说明到达定值分析的用途，本节我们介绍两种借助到达定值信息的程序优化：常量传播优化和拷贝传播优化。

3.3.1 常量传播

如果程序中有一个定值 $d: t = c$，其中 c 为一个常量。该定值可以到达一条使用变量 t 的语句 $n: x = t \oplus y$，并且 d 为唯一一个到达该语句 n 的定值。那么作为一种优化，可以将语句 n 中变量 t 的替换为常量 c，即 $x = c \oplus y$。这种优化称为**常量传播**（Constant propagation）。

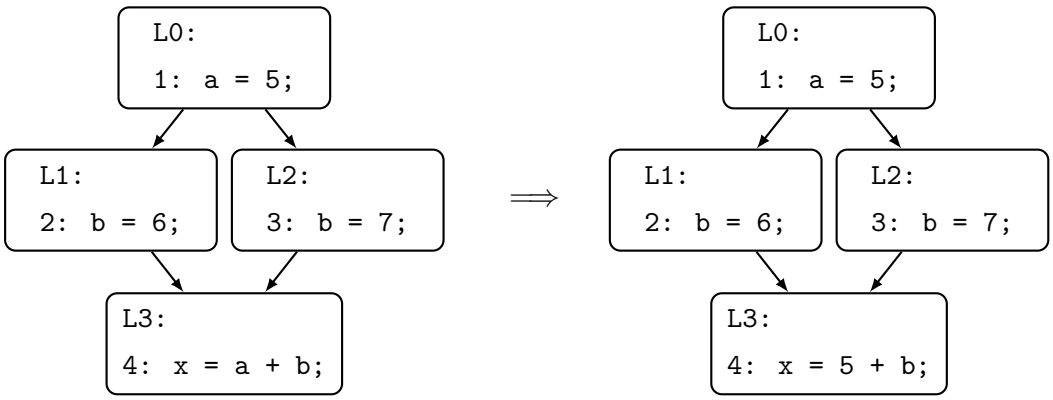


图 3.3: 常量传播示例

图3.3给出了常量传播优化的一个例子。在左侧的流图中，在语句 4 的入口处共有三个定值，分别为 1、2 和 3。但是只有变量 a 的定值是唯一确定的，只包含唯一定值 $a = 5$，所以可以将语句 4 中变量 a 替换为 5，结果如右侧流图所示。

通过常量传播，可以缩短程序中某些变量的活跃区间，并且可以发现其他优化的机会。例如在图3.3中，变量 a 由常量

传播优化前的活跃到语句 4 缩短到只在语句 1 处活跃。这种变量活跃区间的缩短非常有利于后续程序的寄存器分配，因为我们通常认为具有较短活跃区间的变量会更容易分配到一个寄存器中。此外，如果仔细观察右侧流图的话，变量 a 在被定值后并没有被使用，语句 1 成为了死代码，这就为死代码优化引入了新的优化机会。

3.3.2 拷贝传播

拷贝传播（Copy propagation）类似于常量传播优化，只不过传播的是变量而不是常量。

如果程序中有一个定值 $d : t = z$，其中 z 是一个变量。该定值可以到达一条使用变量 t 的语句 $n : x = t \oplus y$。如果除了定值 d 之外，没有其他对 t 的定值到达 n，并且在定值 d 到达 n 的所有路径中，没有再对变量 z 的定值，那么我们就可以使用变量 z 来替换 t，将语句 n 改写为 $x = z \oplus y$。这种优化称为拷贝传播。

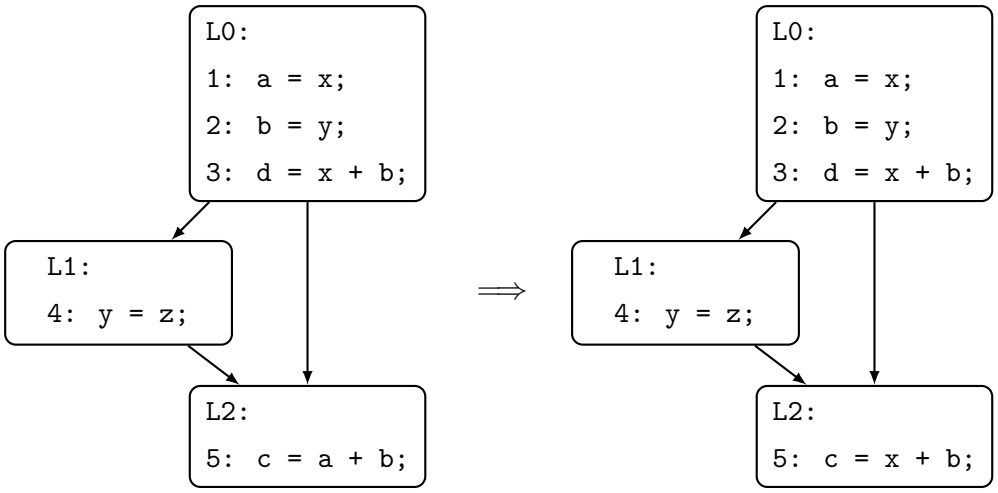


图 3.4: 拷贝传播示例

图3.4给出了拷贝传播的一个例子。在左侧控制流图中，定

值 1 和 2 都可以到达语句 5，分别是对变量 a 和 b 的定值。由于定值 1 到达语句 5 的路径上都没有对变量 x 的重新定值，所以可以用变量 x 替换语句 5 中的变量 a。但是却不能用变量 y 替换语句 5 中的变量 b，因为在路径 $B_0B_1B_2$ 上有对变量 y 的重新定值。拷贝传播优化后的结果如右侧流图所示。

拷贝传播并不能提升程序的性能，但可以发现其他的优化机会。例如在图 3.4中，优化后的语句 3 和 5 拥有公共子表达式 $x+b$，这就为公共子表达式删除引入了优化机会，此外语句 1 也成为了死代码，可以应用死代码删除优化。然而，拷贝传播可能会延长变量的活跃区间。在优化后，变量 x 的活跃区间将由语句 1 延长到了语句 5。变量活跃区间的延长可能会给后续的寄存器分配带来压力，导致过多的变量溢出，所以在编译器设计时需要根据实际情况考虑拷贝传播的应用。

拷贝传播需要考虑定值 d 到语句 n 的路径上没有对拷贝变量的重新定值，然而在实际的实现中，仅通过到达定值信息无法判断是否存在重新定值的情况。一种朴素的想法是对 d 到达 n 的所有路径进行搜索，判断是否存在重新定值的情况，但这所带来的时间开销往往是难以接受的。一种解决方法是引入一个新的临时变量 v，并将定值 $d : t = z$ 改写成

$$d : v = z$$
$$d' : t = v,$$

然后使用临时变量 v 替换语句 n 中的变量 t。由于 v 是新引入的变量，定值 d 到达 n 的路径上不会存在对它的重新定值，这样就避免了复杂的路径搜索。这种方式能够产生死代码消除的优化机会，但很难引入公共子表达式。

另一种解决方法是使用一个到达定值分析变种：我们只

考虑形如 $d : t = z$ 的赋值语句，让

- $Gen[d]$ 依旧与原始的到达定值相同，
- 而 $Kill[d]$ 包含除 d 外，所有对变量 t 和 z 的定值。

并且使用交作为交汇运算而不是使用并。由于将对变量 z 的定值包含进了杀死集中，若定值 d 能够到达 n，那么中间路径上一定不存在对变量 z 的定值。并且使用交作为交汇运算也避免了在 n 的入口对定值是否唯一的判断。

算法3.2给出了拷贝传播的一般框架。算法接收一个控制流图 G，首先使用到达定值分析算法 $ReachDefAnalyze$ 计算图 G 中每个语句的到达定值信息（方便起见，这里我们使用到达定值分析的变种），拷贝传播只用到了 In 集合。之后遍历流图中的每一个语句 n，判断 $In[n]$ 中是否存在对 t 的唯一定值 $t = z$（对于到达定值分析的变种，这里一定是唯一的），若存在则对语句 n 进行重写，使用变量 z 替换语句 n 中的 t。该算法框架同样适用于常量传播，作为拷贝传播的一个特例，只需将 z 看做常量即可。

算法 3.2 拷贝传播（常量传播）算法框架

输入： 一个控制流图 G

输出： 经过拷贝传播（常量传播）后的控制流图

```
1: function Propagate(G)
2:     ⟨In, _ ⟩ = ReachDefAnalyze(G)
3:     for each statement n : x = t ⊕ y ∈ G do
4:         if t (or y) only has an unique defination t = z in In[n] then
5:             rewrite x = t ⊕ y to x = z ⊕ y
6:     return G
```

3.4 公共子表达式删除

对于一个语句 $n : t = x \oplus y$，如果表达式 $x \oplus y$ 在语句 n 的入口处为一个可用表达式，说明它已经被计算过，我们称该表达式为公共子表达式。那么一种自然的想法是将语句 n 中的对 $x \oplus y$ 的冗余计算消除掉，这种优化称为**公共子表达式删除**（Common-subexperssion elimination）优化。在??节中，我们已经探讨过了可用表达式分析，本节我们将利用可用表达式信息，来对程序进行公共子表达式删除优化。

假设可用表达式 $x \oplus y$ 由语句 $m : u = x \oplus y$ 生成，那么我们可以引入一个临时变量 v，对于所有生成可用表达式 $x \oplus y$ 的语句，将它们改写为

$$m : v = x \oplus y$$
$$m' : u = v$$

并且将语句 n 改写为 $n : t = v$。

在公共子表达删除的过程中，并没有使用语句 m 中定值的变量 u 直接替换语句 n 中的公共子表达式。这是因为我们无法确定在语句 m 到达 n 的路径上，是否存在对变量 u 的重新定值。当存在对变量 u 的重新定值时，u 很可能已经不是 $x \oplus y$ 的计算结果，直接使用变量 u 去替换公共子表达式会导致程序语义错误。尽管这种做法会使得优化后的程序变得复杂，但是可以通过后续的拷贝传播和死代码删除等优化进行简化。

图3.5给出了公共子表达式删除的一个例子。在左侧控制流图中，表达式 $a + x$ 和 $c + y$ 为两个公共子表达式，我们分别为它们引入临时变量 p 和 q 来对程序进行改写，优化后的

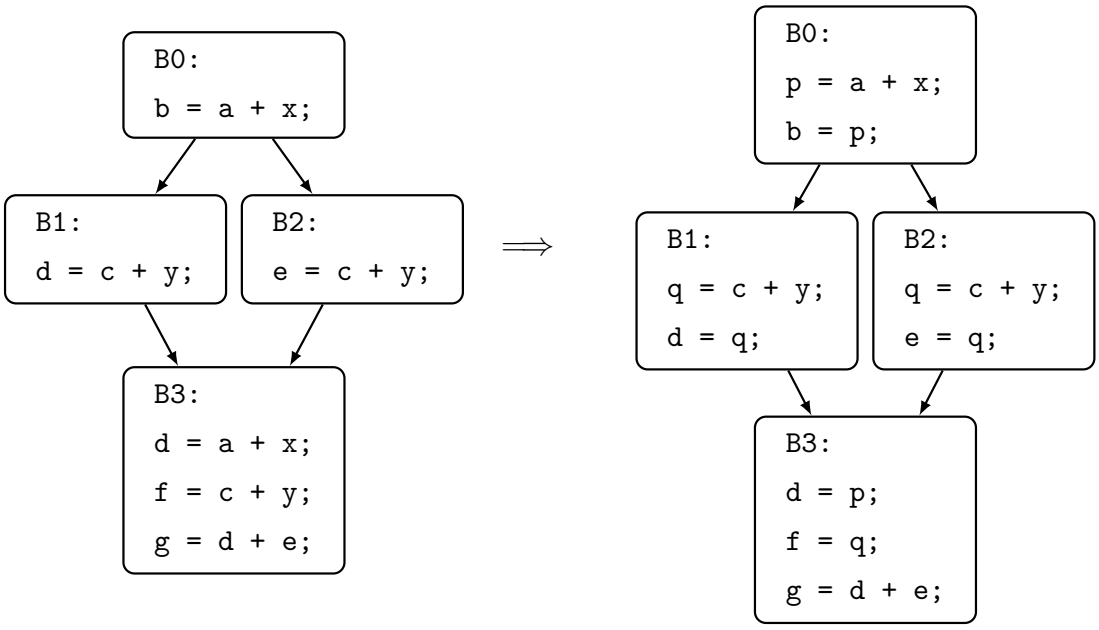


图 3.5: 公共子表达式删除示例

结果如右侧控制流图所示。

最后，算法3.3给出了公共子表达式删除的算法框架。该算法接受一个控制流图 G 作为输入，并返回公共子表达式删除后的控制流图。算法首先使用函数 `AvailableExpression`对图 G 进行可用表达式分析，之后对图中的每一条语句 $n : t = x \oplus y$，如果 $x \oplus y$ 在 n 的入口处为一个可用表达式，即属于集合 $In[n]$，则按照前文的重写规则进行重写：算法首先创建一个临时变量 u，然后对生成了 $In[n]$ 集合中可用表达式 $x \oplus y$ 的所有的语句，以及语句 n 本身进行修改。

3.5 部分冗余删除

尽可能减少程序中表达式求值次数是程序优化的一个重要思路。通过移动各个对表达式 $x \oplus y$ 求值的位置，并在必要的时候把求值结果保存在临时寄存器中，我们能够在很多执行路径上较少对该表达式的求值次数，且保证不增加任何路

算法 3.3 公共子表达式删除算法框架

输入： 一个控制流图 G

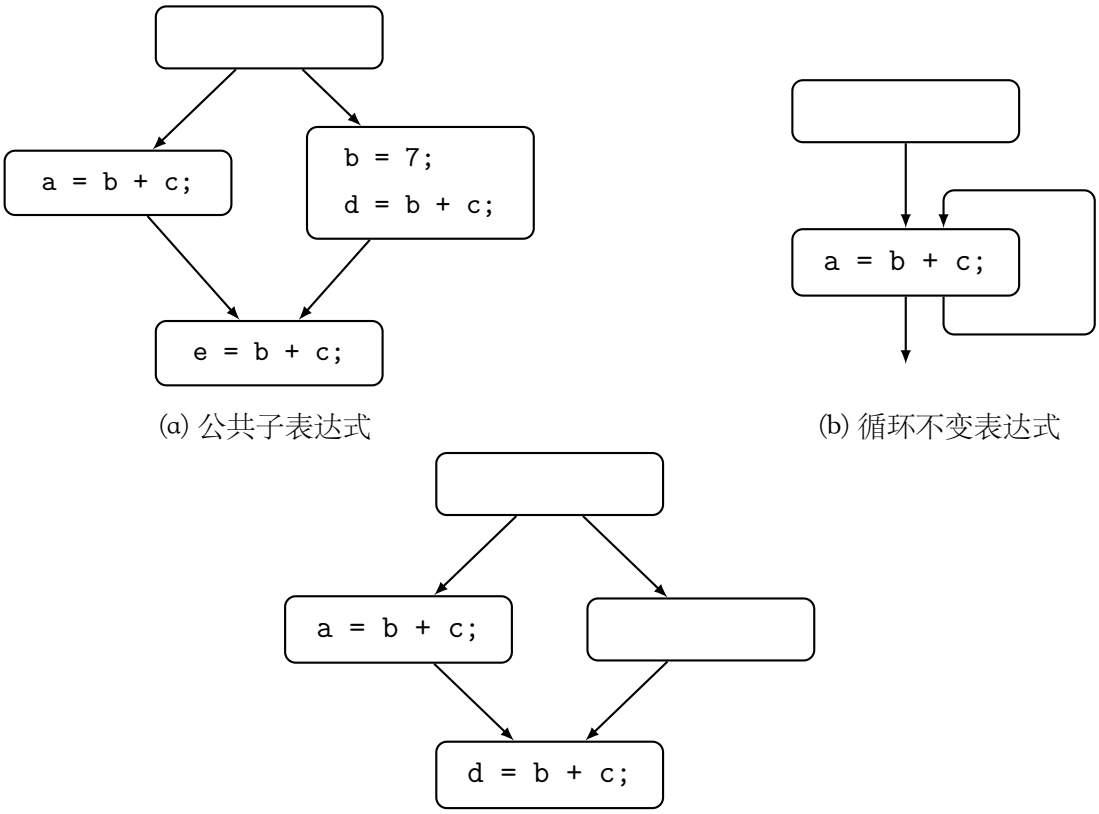
输出： 经过公共子表达式删除后的控制流图

```
1: function CSE(G)
2:     ⟨In, _ ⟩ = AvailableExpression(G)
3:     for each n : t = x ⊕ y ∈ G do
4:         if x ⊕ y ∈ In[n] then
5:             u = CreateTempVariable()
6:             let m_0, m_1, ...m_k be the statements that generate x ⊕ y in In[n]
7:             for i = 0 to k do
8:                 rewrite m_i : v_i = x ⊕ y to m_i : u = x ⊕ y; m' : v_i = u
9:             rewrite t = x ⊕ y to t = u
10:    return G
```

径中的求值次数。

如图3.6所示，程序中的冗余可能以多种形式存在：（a）公共子表达式，即表达式 $x \oplus y$ 在某次出现之前已经被计算过，且变量 x 或者 y 在那次计算之后一直未被改变；（b）循环不变表达式，即循环体中计算表达式 $x \oplus y$，其包含的变量 x 以及 y 在循环中没有被重新定值；（c）部分冗余表达式，即表达式 $x \oplus y$ 的计算在某条路径上是冗余的，但在另一条路径上不冗余。

减少程序中部分冗余计算的优化被称为**部分冗余删除**（Partial Redundancy Elimination，PRE）。在之前的章节中，我们介绍了利用可用表达式进行公共子表达式的删除。受限于可用表达式的判定条件，以及公共子表达式消除对控制流图有限的操作，该优化对循环不变表达式以及部分冗余表达式无能为力。为了实现部分冗余表达式的删除，我们需要设计一个部分冗余删除的算法。事实上，由于公共子表达式以及循环不变表达式可以视为部分冗余的特例，该算法同样可以



(a) 公共子表达式

(b) 循环不变表达式

(c) 部分冗余表达式

图 3.6: 各种冗余表达式的例子

消除这两种冗余。

部分冗余删除算法是程序优化中最复杂的算法之一。为了方便读者理解算法中每一步的行为动机，此处先给出该算法的概述。

消除部分冗余的第一步，便是设法将程序中的部分冗余转变为完全冗余。程序点 p 处对表达式 $x \oplus y$ 的计算是部分冗余的，是因为存在某些到达 p 的路径 S 上有过对该表达式的计算而其他路径 R 上则没有。如果我们将程序点 p 处对表达式 $x \oplus y$ 的计算移动到各条路径上，那么路径 S 上对表达式的计算便是完全冗余的，并且路径 R 上对表达式的计算次数也不会增加。图3.7展示了将部分冗余转变为完全冗余的过程。对于左侧的原始控制流图，在 $L4$ 块中对表达式 $b+c$ 的计算是

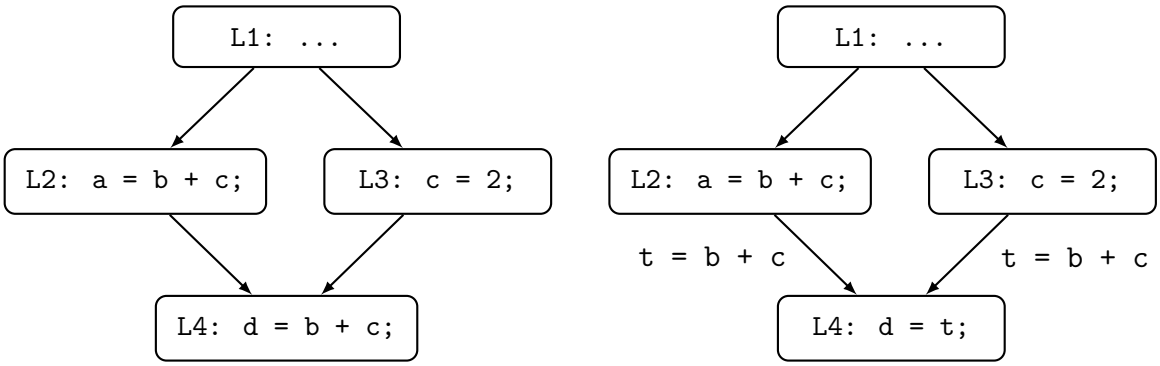


图 3.7: 将部分冗余转变为完全冗余

部分冗余的。如果我们将表达式 $b+c$ 的计算放置到 $L2 \to L4$ 以及 $L3 \to L4$ 的边上，那么路径 $L1 \to L2 \to L4$ 上对 $b+c$ 的计算便是完全冗余的。注意，我们此处使用了边放置的概念，在实际中，放置在边上的计算将会成为一个新的块并插入到原始的边上。

在第一步中，我们将表达式的计算移动到了程序点的入边上，实现了部分冗余到完全冗余的转换。在此基础上，如果我们能在不改变程序语义的前提下，将表达式的计算进一步向上转移，便有可能消除更多的冗余表达式。如图3.8所示，在将表达式 $b+c$ 的计算从边 $L2 \to L4$ 向前传递到边 $L1 \to L2$ 上后，$L2$ 块中对 $b+c$ 的运算也变成了冗余的计算，并且这样的移动是合法的，因为 $L1 \to L2$ 边上 $b+c$ 的结果一定等于 $L2 \to L4$ 边上的计算结果。然而，$L3 \to L4$ 上 $b+c$ 的运算并不能提前到 $L1 \to L3$ 上，因为 $L3$ 块中出现了对变量 c 的重新定值，这会改变 $b+c$ 的计算结果。$L1 \to L2$ 上的运算也不能提前到 $L1$ 块之前，因为这会增加 $L1 \to L3 \to L4$ 路径上 $b+c$ 的计算次数。更进一步，如果我们将程序中包含的所有表达式的计算都尽可能放置到最早可提前位置，便可能消除所有的冗余表达式计算。这种采取最早放置策略的算法被称为 MR 算法。

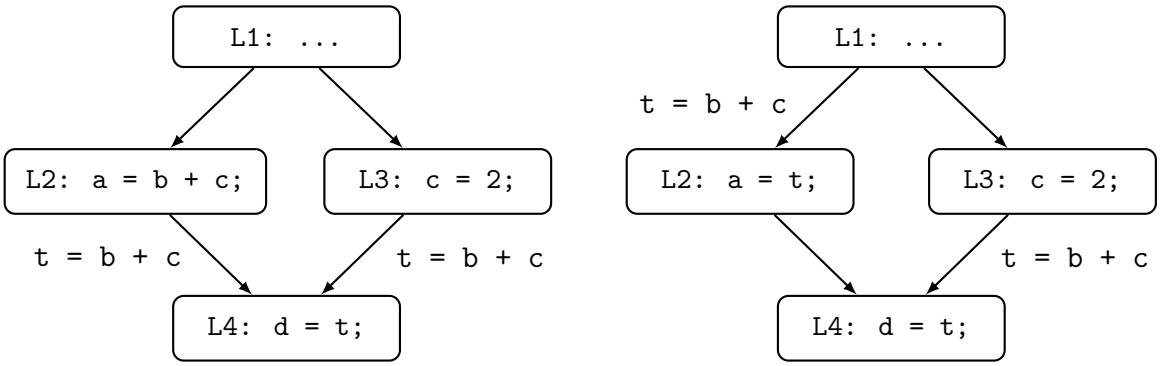


图 3.8: 表达式向前放置

在 MR 算法提出后，人们发现该算法采用的最早放置策略在实际实现中存在两个问题：（1）提前计算某些表达式是没有必要的；（2）最前放置导致表达式生命周期过长。如图3.9左图所示，该程序中 $L5$ 块中表达式 $b+c$ 的计算并不是冗余的，因此理论上不需要将该表达式的计算提前。但如果按照最前放置策略，那么算法将会将表达式的计算提前放置到 $L1 \rightarrow L2$ 以及 $L1 \rightarrow L3$ 的边上。这样的修改并不会改变运行时 $L1 \rightarrow L5$ 路径上对表达式 $b+c$ 的计算次数，但由于原本只在 $L5$ 块中出现一次的 $b+c$ 被分别提前到了 $L1 \rightarrow L2$ 和 $L1 \rightarrow L3$ 两处，反而造成了代码体积的增大。又如图3.9右图所示，如果使用 MR 算法对该程序进行优化，那么表达式 $b+c$ 的计算应该提前到①位置。原始程序表达式 $b+c$ 仅在 $L3$ 以及 $L5$ 块中活跃，将其提前后，该表达式将会从 $L2$ 的入口处一直活跃到 $L5$ 块中。而过长的生命周期会为之后进行的寄存器分配带来压力。

为了解决 MR 算法最前放置策略造成的问题，一个显而易见的想法是将提前放置的表达式在不影响优化效果的前提下尽可能延后，尽量缩短表达式的活跃范围。对于图3.9左侧程序，MR 算法提前放置的 $b+c$ 的计算可以延后到 $L5$ 块中。这能够去除 MR 算法引入的不必要的表达式提前。对于图3.9

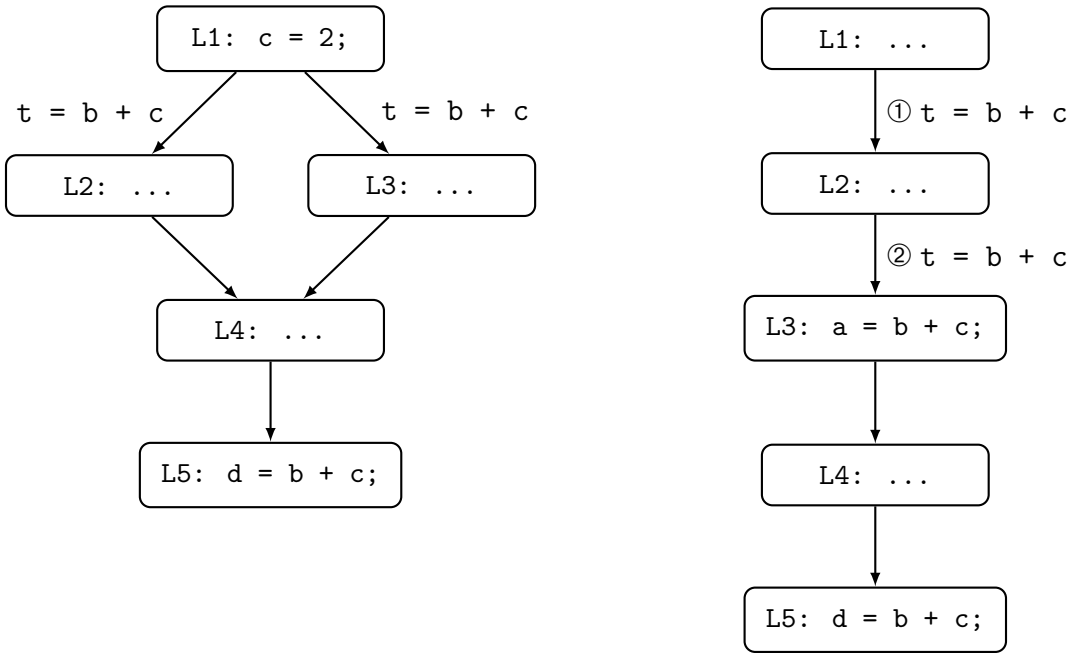


图 3.9: 表达式延后放置

右侧程序，可以将①位置的 $b+c$ 计算推迟到②的位置，使得 $L3$ 和 $L5$ 块中表达式 $b+c$ 冗余删除的同时，尽可能缩短该表达式的活跃范围。使用表达式后延改进的 MR 算法被称为*懒惰代码移动*（Lazy Code Motion，LCM）。

依据*懒惰代码移动*算法概述的指导，我们给出该算法的具体步骤。为了简化算法，我们假设每个节点中只包含一条语句。对于每个节点 i，使用符号 $Gen[i]$ 表示 i 中生成的表达式，符号 $Kill[i]$ 表示 i 中杀死的表达式。以图3.9左侧程序为例。对于 $L1$ 块，其对变量 c 进行了重新定值，因此会杀死所有包含变量 c 的表达式（$b+c$）。对于 $L5$ 块，其计算了表达式 $b+c$，即生成了表达式 $b+c$。

第一步，计算可用表达式。可用表达式分析内容请参考前文??节。此处我们使用 AvailIn 以及 AvailOut 分别表示程序中每个节点的入口可用表达式和出口可用表达式。

第二步，计算忙碌（预期执行）表达式。忙碌表达式分析

内容请参考前文??节。此处我们使用 AntIn 以及 AntOut 分别表示程序中每个节点的入口忙碌表达式和出口忙碌表达式。

第三步，计算表达式提前放置位置。表达式最早放置的边 $E\langle i,j\rangle$ 应该满足三个条件：（1）表达式可以安全地移动到边 $E\langle i,j\rangle$ 上；（2）在边 $E\langle i,j\rangle$ 上插入表达式不是冗余的；（3）在边 $E\langle i,j\rangle$ 上的表达式计算不能继续提前放置。第一个条件可以使用忙碌表达式分析结果确定能够安全提前的位置。如果表达式可以安全放置在边 $E\langle i,j\rangle$ 上，那么该表达式一定会在节点 j 的入口处忙碌。第二个条件可以使用可用表达式判定。如果某条边 $E\langle i,j\rangle$ 是表达式 $b+c$ 可以安全提前的位置，但这条边上该表达式是可用的，即节点 i 的出口可用表达式包含 $b+c$，这意味着在边 e 上插入表达式 $b+c$ 是冗余的。第三个条件确保插入位置是最前的。一个放置在边 $E\langle i,j\rangle$ 上的表达式 $b+c$ 不能继续提前存在两种可能，一种可能是节点 i 上存在对变量 b 或 c 的重新定值，可以使用 $Kill[i]$ 判断；另一种可能是节点 i 存在某个后继节点 k，边 $E\langle i,k\rangle$ 上表达式 $b+c$ 不是忙碌的。此时对于节点 i, 表达式 $b+c$ 在该节点的出口处不是忙碌的。综上所述，我们可以使用方程

$$Earliest[i,j] = AntIn[j] \cap \overline{AvailOut[i]} \cap \left(Kill[i] \cup \overline{AntOut[i]}\right)$$

确定表达式能够最前放置的位置。

图3.10展示了表达式最早放置分析结果。图中，我们使用灰色方块标记表达式 $b+c$ 在节点入口忙碌，黑色方块标记表达式 $b+c$ 在节点出口可用，粗体的边标记表达式 $b+c$ 的最前放置位置。通过忙碌表达式分析可知，表达式 $b+c$ 的计算不能提前到边 $\langle L1,L2\rangle$ 上，因为该表达式在节点 $L2$ 入口不是忙碌的。边 $\langle L2,L4\rangle$ 是表达式 $b+c$ 的一个最早放置点，因为在这条边上放置该表达式安全且不冗余，且由于节点 $L2$ 对变量

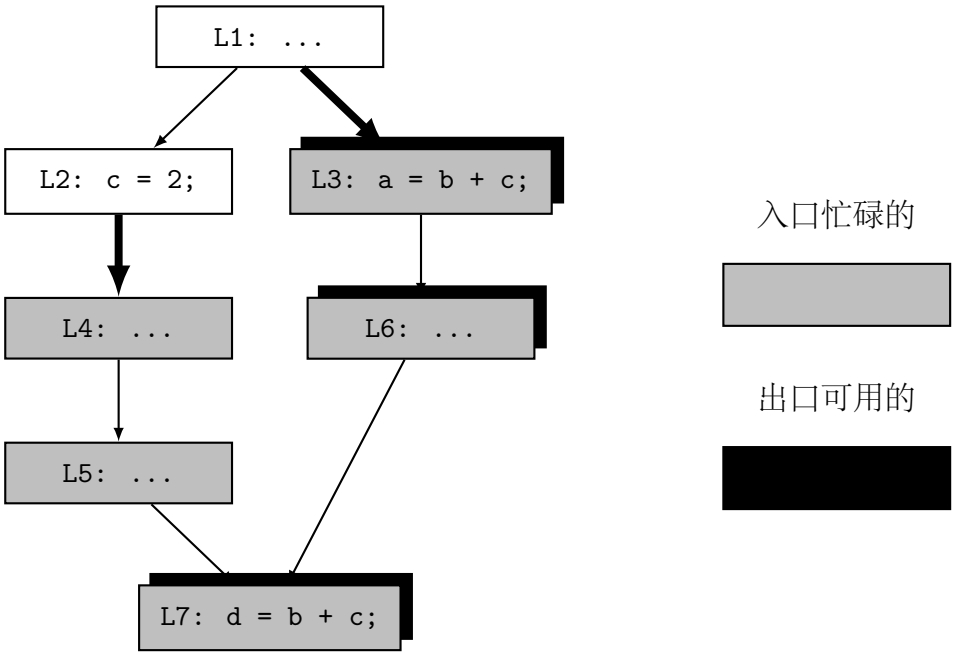


图 3.10: 最早放置表达式示例

c 进行了重新定值，使得 $b+c$ 的计算无法继续向前移动。类似的，边 $\langle L1, L3\rangle$ 也是表达式 $b+c$ 的一个最早放置点，因为该表达式在节点 $L1$ 的出口处并不是忙碌的，阻止了表达式继续向前移动。

第四步，确定表达式延后放置位置。表达式的最晚放置位置可以通过前一步得到的最前放置位置不断延后至某个不可延后的程序点为止确定，因此我们需要确定表达式在哪些位置是可延后的。对于某个节点 j，如果表达式 $x \oplus y$ 属于 j 入口可延后的表达式，意味着这条表达式可以放置在节点 j 的入口处计算而不会影响程序语义。这要求 j 的所有前驱节点 i，表达式 $x \oplus y$ 在边 $\langle i, j\rangle$ 上都是可延后的，否则将该表达式延后到节点 j 的入口时，某些路径上会出现对 $x \oplus y$ 的冗余甚至错误计算。对于某条边 $\langle i, j\rangle$，表达式 $x \oplus y$ 在这条边上是可延后的，意味着将该表达式放置在边 $\langle i, j\rangle$ 上计算不会影响程序的语义。边 $\langle i, j\rangle$ 上的可延后表达式包含两个来源：（1）这条边上所确定的所有最前放置的表达式；（2）来自于节点

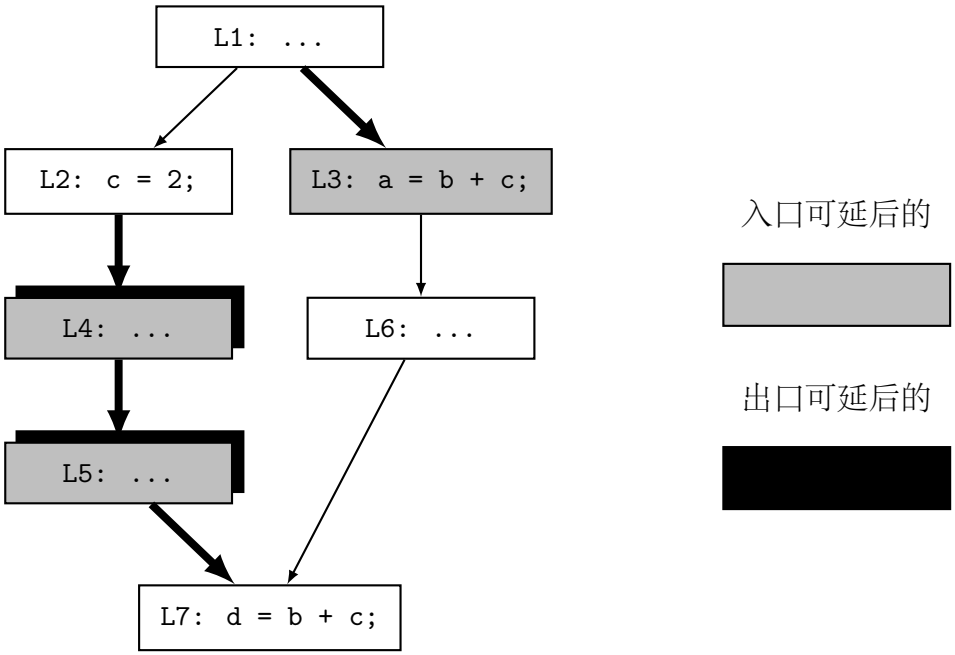


图 3.11: 可延后表达式示例

i 出口处的可延后表达式。前一个来源在第三步中已经确定。而节点 i 出口处的可延后表达式则是其入口可延后表达式减去该节点所生成（使用）的表达式。去除本节点生成的表达式的理由是显而易见的，如果我们将节点生成的表达式延后，那么当程序运行到这个节点时，此处的表达式的值将不可用。我们使用符号 $LaterIn[j]$ 表示节点 j 的入口可用表达式，使用符号 $LaterOut[j]$ 表示节点 j 的出口可用表达式，使用符号 $Later[i,j]$ 表示边 $\langle i,j\rangle$ 上的可延后表达式，那么上述的程序可延后位置信息可以使用方程

$$LaterIn[j] = \bigcap_{i \in pred[j]} Later[i,j]$$
$$LaterOut[j] = LaterIn[j] - Gen[j]$$
$$Later[i,j] = Earliest[i,j] \cup LaterOut[i]$$

进行描述。该数据流方程可以使用迭代算法3.4进行计算。

图3.11展示了可后延表达式分析结果。图中，我们使用灰

算法 3.4 可延后表达式数据流方程迭代算法

输入： 一个控制流图 G 及表达式最早放置信息 $Earliest$

输出： 表达式可延后放置信息

```
1:  function LaterExpression(G, Earliest)
2:      for each i ∈ G do
3:          LaterIn[i] = LaterOut[i] = U
4:      for each e⟨i, j⟩ ∈ G do
5:          Later[i, j] = Earliest[i, j]
6:      LaterIn[ENTRY] = ∅
7:      while any LaterIn[i] or LaterOut[i] changed do
8:          for each i ∈ G do
9:              LaterIn[i] = ⋂_{p∈pred[i]} Later[p, i]
10:             LaterOut[i] = LaterIn[i] − Gen[i]
11:             for each s ∈ succ[i] do
12:                 Later[i, s] = Earliest[i, s] ∪ LaterOut[i]
13:     return Later, LaterIn
```

色方块标记表达式 $b+c$ 在节点入口可延后，黑色方块标记表达式 $b+c$ 在节点出口可延后，粗体的边标记表达式 $b+c$ 可延后放置的边。我们可以发现，由于 $L3$ 块中计算了 $b+c$，边 $\langle L1, L3\rangle$ 上对该表达式的计算不能延后。而边 $\langle L2, L4\rangle$ 上对表达式 $b+c$ 的计算则可以延后到边 $\langle L5, L7\rangle$ 上，并且由于该表达式在 $L7$ 块的入口处不是可延后的，使得其不能继续延后。

第五步，改写程序。为了实现冗余表达式的删除，我们需要在最晚放置点插入表达式以及删除冗余计算的表达式。首先是确定表达式的最晚放置点。如果某条边 $\langle i, j\rangle$ 是表达式 $b+c$ 的最晚放置点，其应该满足条件：（1）表达式 $b+c$ 在边 $\langle i, j\rangle$ 上是可延后的，否则将表达式放置在该处会改变程序语义；（2）表达式 $b+c$ 在节点 j 的入口处不可延后，否则表达式可以放置在节点 j 的开头，使得这条边不是表达式的最晚放置点。随后需要删除的冗余表达式的位置。如果节点 i 处的

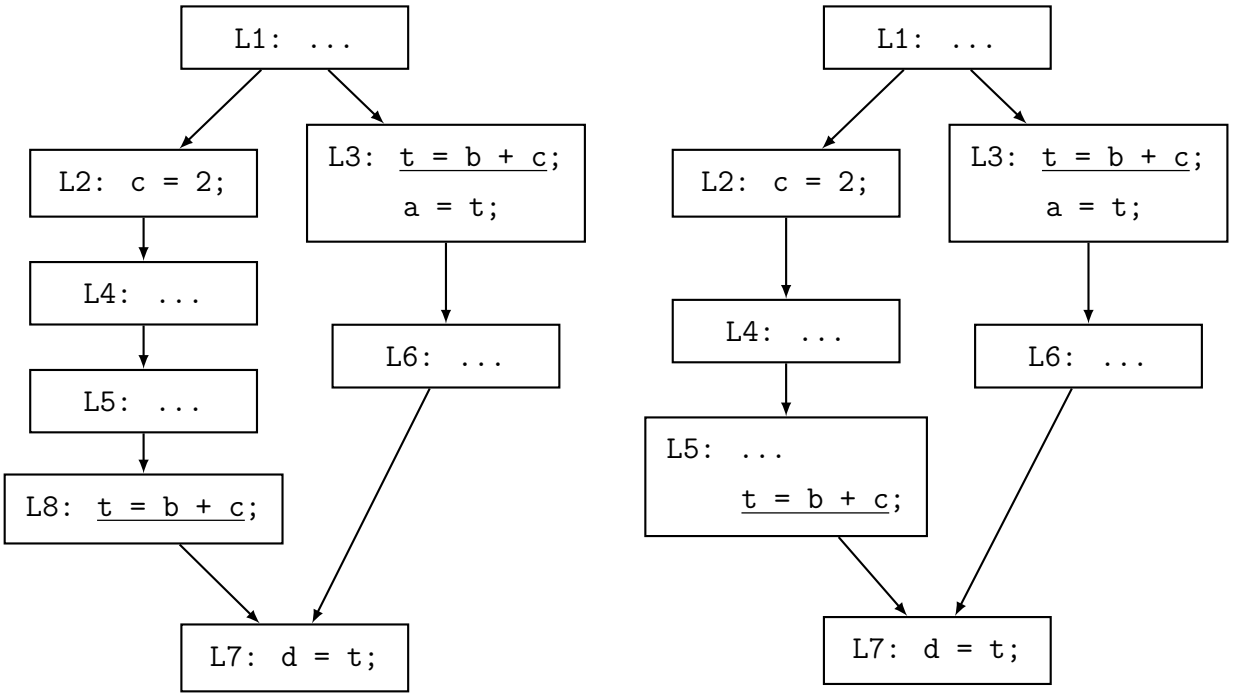


图 3.12: 程序改写示例

表达式 $b+c$ 是冗余的，其应该满足条件：（1）节点 i 存在对表达式 $b+c$ 的计算；（2）表达式 $b+c$ 在节点 i 的入口处是不可延后的。如果一个表达式可以延后到节点 i 入口处计算，说明在节点 i 前并没有一个可用的 $b+c$ 表达式，此时节点 i 处的表达式 $b+c$ 并不是冗余的，因此不能删除。我们使用符号 $Insert[i,j]$ 表示表达式的最晚放置边 $\langle i,j\rangle$，$Delete[i]$ 表示节点 i 处可以删除的冗余表达式，以上条件可以使用方程

$$Insert[i,j] = Later[i,j] - LaterIn[j]$$
$$Delete[i] = Gen[i] - LaterIn[i]$$

形式化表述。

以图3.11为例。由于表达式 $b+c$ 在边 $\langle L2, L4\rangle$ 上可延后，且在节点 $L4$ 的入口处也可延后，因此边 $\langle L2, L4\rangle$ 不是表达式 $b+c$ 的最晚放置点。边 $\langle L4, L5\rangle$ 与边 $\langle L1, L3\rangle$ 同理。表达式 $b+c$ 在边 $\langle L5, L7\rangle$ 上可延后，但在节点 $L7$ 入口不可延后，

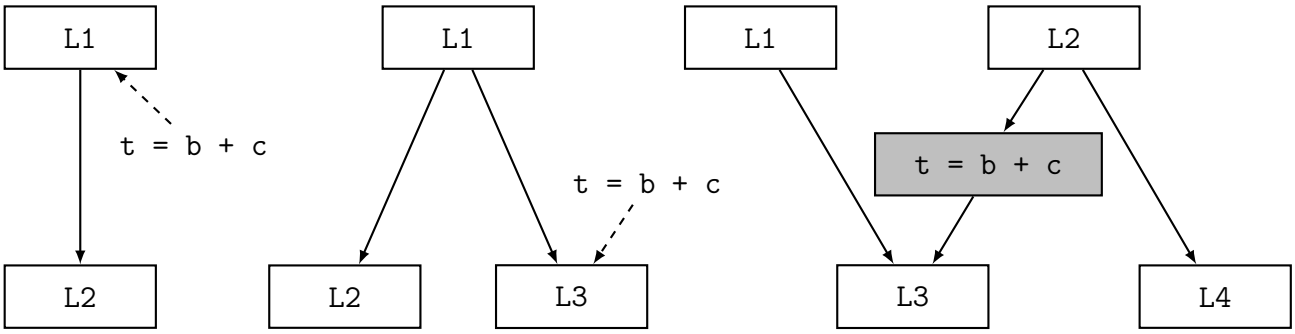


图 3.13: 指令插入位置示例

因此边 $\langle L5, L7\rangle$ 是表达式 $b+c$ 的一个最晚放置点。对于节点 $L3$，表达式 $b+c$ 在其出口处可延后，说明该处的 $b+c$ 运算首次出现不可删除。对于节点 $L7$，表达式 $b+c$ 则在其出口处不可延后，说明在该程序点前存在可用的 $b+c$ 计算，因此此处的 $b+c$ 是冗余的。

在得到 $Insert[i,j]$ 和 $Delete[i]$ 信息后，我们可以对程序进行改写以实现冗余的删除。一种改写规则如下：（1）如果边 $\langle i,j\rangle$ 的 $Insert[i,j]$ 集合包含表达式 $b+c$，则在该边上插入新的块并放置表达式 $t=b+c$；（2）如果节点 i 的 $Delete[i]$ 集合包含表达式 $b+c$，则删除该节点中的第一个 $b+c$ 计算，并将结果替换为 t；（3）如果节点 i 计算了 $b+c$ 但 $Delete[i]$ 不包含表达式 $b+c$，则在节点的开头插入 $t=b+c$，并将节点中的第一个 $b+c$ 计算替换为 t。按照以上规则改写的程序如图3.12左侧所示。当然，这种方式在所有 $Insert[i,j]$ 集合包含表达式 $b+c$ 的边上插入新的块，这会增加跳转指令的数量，影响程序的性能。而在某些情况下，我们可以在现有的块中插入指令实现同样的效果。如图3.12左侧程序，我们可以将节点 $L8$ 中插入的 $t=b+c$ 直接放置在 $L5$ 节点的末尾，且不会改变程序的语义。为此，我们需要修改第一条规则：某条边 $\langle i,j\rangle$ 的 $Insert[i,j]$ 集合包含表达式 $b+c$，如果节点 i 只有一个后继节点 j，则在节点 i 结尾插入 $t=b+c$（图3.13左）；或

算法 3.5 懒惰代码删除算法框架

输入： 一个控制流图 G，其中每条语句的 Gen 集和 $Kill$ 集都已完成计算

输出： 经过冗余表达式删除后的控制流图

```
1:  function LCM(G)
2:      ⟨AvailIn, AvailOut⟩ = AvailableExpression(G)
3:      ⟨AntIn, AntOut⟩ = BusyExpression(G)
4:      for each e⟨i, j⟩ ∈ G do
5:          Earliest[i, j] = AntIn[j] ∩ ¬AvailOut[i] ∩ (Kill[i] ∪ ¬AntOut[i])
6:      ⟨Later, LaterIn⟩ = LaterExpression(G, Earliest)
7:      for each e⟨i, j⟩ ∈ G do
8:          Insert[i, j] = Later[i, j] − LaterIn[j]
9:      for each i ∈ G do
10:         Delete[i] = Gen[i] − LaterIn[i]
11:     for each e⟨i, j⟩ ∈ Insert do
12:         toInsert = ∅
13:         for each x ⊕ y ∈ Later[i, j] do
14:             // if x ⊕ y has no temporary variable, we will create one
15:             u = GetTempVariable()
16:             toInsert ∪= {u = x ⊕ y}
17:         if | succ[i] | = 1 then
18:             insert all statements in toInsert into the tail of i
19:         else if | pred[j] | = 1 then
20:             insert all statements in toInsert into the head of j
21:         else
22:             split e⟨i, j⟩ with a new block n
23:             insert all statements in toInsert into n
24:     for each n : t = x ⊕ y ∈ G and x ⊕ y has a temporary variable u do
25:         if n ∉ Delete then
26:             insert u = x ⊕ y into the head of n
27:         rewrite n : t = x ⊕ y to n : t = u
28:     return G
```

者节点 j 只有一个前驱节点 i，则在节点 j 开头插入 $t = b + c$（图3.13中）；否则该边是一条关键边（??节），需要在该边上插入新块并放置表达式 $t = b + c$（图3.13右）。使用新规则改写

后的程序如图3.12右侧所示。

最后，根据上述的步骤，算法3.5给出了懒惰代码移动进行部分冗余消除的框架。

3.6 本章小结

本章探讨了几种基于数据流分析的关键优化技术。首先讨论了死代码删除，通过移除冗余代码简化了程序结构；接着，分析了常量与拷贝传播，以优化变量的使用；随后，介绍了公共子表达式删除，以减少重复计算；最后，探讨了部分冗余删除，以消除部分冗余。这些优化技术利用数据流分析的结果，使编译器能够生成更加高效和精简的代码。

3.7 深入阅读

死代码删除是一种常见的基于数据流分析的优化技术，在许多编译器和数据流分析相关的书籍 [1, 2, 5, 9] 中均有详细的介绍。

常量传播通常与常量折叠交错使用，可以尽可能地发现对方的优化的机会。稀疏有条件的常量传播比本章介绍的常量传播算法更加强大，除可以进行传统的常量传播，还能够结合分支条件执行死代码的消除，更多可参阅文献 [13, 3]。许多文献 [1, 2] 都有对拷贝传播的介绍，但并未涉及拷贝变量在中间路径上的定值问题，文献 [9] 给出了一个到达定值分析的变种来解决这个问题。

Cocke[4] 和 Ullman[12] 介绍了可用表达式分析以及它在公共子表达式消除中的使用。

部分冗余删除算法最早由 Morel 与 Renvoise[11] 于 1979 年提出。该算法提出后，衍生出了多种改进版本 [6, 7]。而为了解决 MR 算法引入的寄存器分配压力，Knoop[10] 等人提出了懒惰代码移动算法。本书介绍的部分冗余删除算法则是 Drechsler 与 Stadel[8] 提出的懒惰代码移动算法的一种变种。

3.8 思考题

1. 思考对图??中的控制流图进行活跃分析，进而实现死代码删除的具体过程。

2. 在编译器优化过程中，死代码删除通常与其他优化技术结合使用。讨论死代码删除与其他优化技术之间的相互作用。分析如果调整这些优化技术的顺序，可能会对最终的优化结果产生什么影响。

3. 在3.3.2节，为了高效地确定定值 d 到语句 n 的路径上没有对拷贝变量的重新定值，我们介绍了一种到达定值分析的变种。写出该变种的数据流方程，并像??节一样给出它的各个属性。

4. 算法3.3中，我们需要知道 $In[n]$ 中的可用表达式 $x \oplus y$ 是由哪些语句生成的，这些语句可以在可用表达式分析的过程中记录下来。尝试对可用表达式分析进行扩展，设计一种你认为比较合适的数据结构，使得在分析的过程中记录下这些语句，并能够在使用的时候高效地查找。

5. 正如3.5节所述，公共子表达式是部分冗余的一种特例。尝试使用部分冗余消除算法，重新对图3.5进行冗余消除。

6. 如图3.14所示。该图左侧为优化前的控制流图，右侧为使

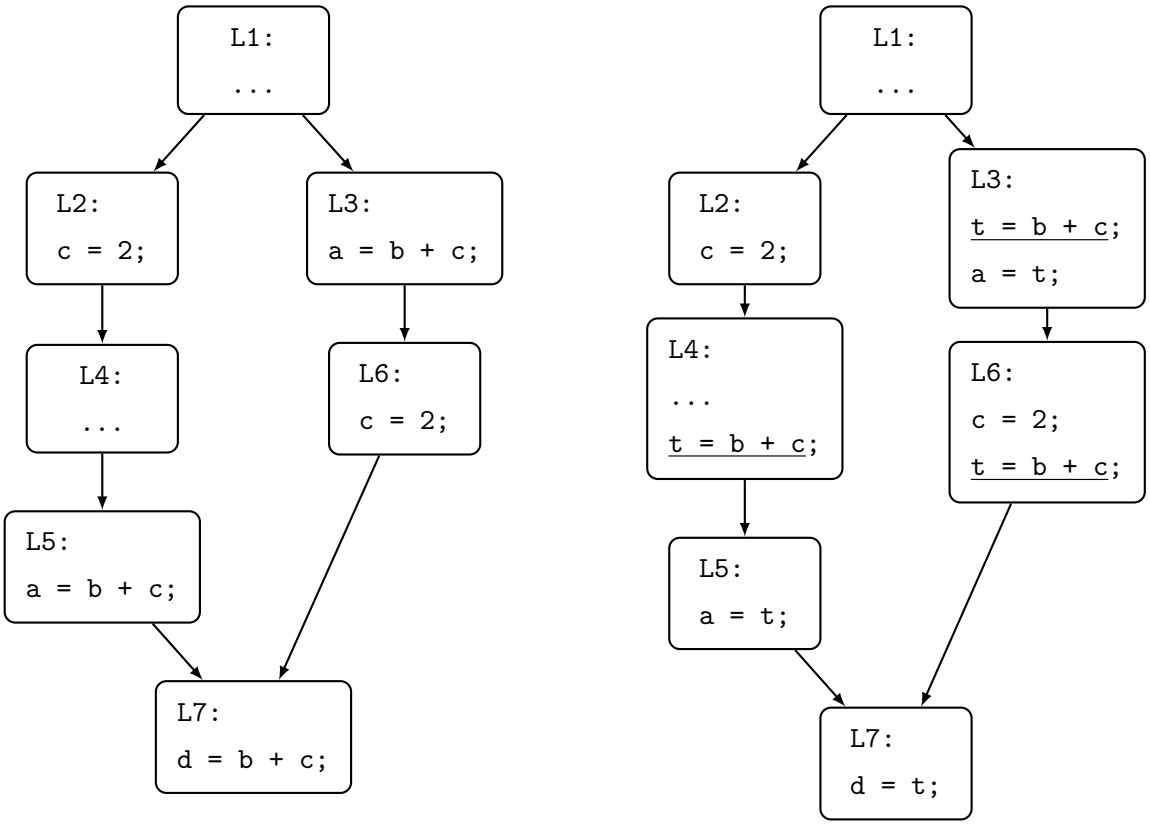


图 3.14: 部分冗余删除示例

用了 3.5节介绍懒惰代码移动算法进行部分冗余消除后的控制流图。在块 L_3 中，$b+c$ 的计算被提前到该块的开头处，但提前计算的值仅在 L_3 块内被使用。显然，这样的提前是没有意义的，并且还会延长表达式 $b+c$ 的活跃范围。为了解决这个问题，可以进行被使用表达式分析，通过判断节点内的表达式是否之后会被使用，确定是否提前计算表达式。根据以上的解决方法，尝试修改现有的懒惰代码移动算法。

参考文献

[1] Alfred V. Aho and Alfred V. Aho, editors. Compilers: Principles, Techniques, & Tools. Pearson/Addison Wesley, 2nd ed edition.

[2] Andrew W Appel. Modern Compiler Implementation in ML. Cambridge university press.

[3] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. 17(2):181 – 196.

[4] John Cocke. Global common subexpression elimination. 5(7):20 – 24.

[5] Keith D. Cooper and Linda Torczon. Engineering a Compiler. Elsevier/Morgan Kaufmann, 2nd ed edition.

[6] D. M. Dhamdhere. A fast algorithm for code movement optimisation. 23(10):172 – 180.

[7] D. M. Dhamdhere. Practical adaption of the global optimization algorithm of morel and renvoise. 13(2):291 – 294.

[8] Karl-Heinz Drechsler and Manfred P. Stadel. A variation of knoop, rüthing, and steffen's lazy code motion. 28(5):29 – 38.

[9] Uday Khedker, Amitabha Sanyal, and Bageshri Sathe. Data Flow Analysis: Theory and Practice. CRC Press.

[10] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. In Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation - PLDI '92, pages 224 – 234. ACM Press.

[11] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. 22(2):96 – 103.

[12] J. D. Ullman. Fast algorithms for the elimination of common subexpressions. 2(3):191 – 213.

[13] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. 13(2):181 – 210.