arXiv:2602.07627v1 [cs.PL] 7 Feb 2026

控制流图的串并循环分解

Xuran Cai[a]、Amir Kafshdar Goharshady[a]、S Hitarth[b]、Chun Kit Lam[b]

[a]牛津大学计算机科学系，沃尔夫森楼，帕克斯路，牛津，英国[b]香港科技大学计算机科学与工程系，清水湾，新界，香港

摘要

众所周知，结构化程序的控制流图（控制流图）具有显著的稀疏性。传统上，人们使用树宽和路径宽度等图参数对这种稀疏性进行建模，从而为编译器优化、模型检测和程序分析中的任务开发出更快速的参数化算法。然而，这些参数只能近似刻画控制流图的结构约束：尽管每个结构化控制流图的树宽至多为 7，但许多树宽至多为 7 的图并不能作为控制流图出现。因此，现有的参数化技术针对的图类远比实践中遇到的图类宽泛。

在这项工作中，我们提出了一种新的基于文法的分解框架，能够精确地刻画由结构化程序生成的控制流图类。我们的分解直观明了，与程序的语法结构相呼应，并且与基于树宽的方法所采用的动态规划范式完全兼容。借助这一框架，我们为两个经典的编译器优化问题设计了改进的算法：寄存器分配和生命周期最优推测式部分冗余消除（*LOSPRE*）。大量实验评估表明，与以往最先进的方法相比，我们的方法显著提升了性能，凸显了采用专为控制流图设计的分解方法的优势。

关键词：静态分析，图分解，编译器优化，寄存器分配，冗余消除

1. 引言与相关工作

程序分析、编译器优化和形式化验证中的许多基础任务，传统上都通过将其归约为图论问题来处理。经典且广为研究的例子包括寄存器分配 [1, 2], LOSPRE [3], 缓存优化技术，例如数据打包 [4] 和缓存感知数据放置 [5, 6], 过程间数据流分析 [7], 代数程序分析 [8], 等式饱和与提取 [9, 10], 以及 μ-演算模型检测 [11], ，还有许多其他例子。

由这些任务引发的图问题通常是 NP难 [1,4, 12], ，在某些情况下，除非 P =NP [13, 14], 甚至难以近似。这自然引出了一个问题：求解这些图问题最一般形式的做法——即将所有 图都视为可能的输入实例——是否适合作为原始程序分析或编译器优化任务的模型？在实践中，这些任务所依赖的图几乎总是良结构程序的控制流图（CFG），或由其直接派生的图（例如通过复制顶点得到的图）。此类控制流图十分稀疏，而且许多图不可能成为任何真实程序的控制流图。因此，针对不受限制的图类设计的算法，可能解决的是一个严格来说并无必要的更难问题。

因此，大量研究致力于形式化控制流图及相关图表示的稀疏性和结构属性。例如，程序依赖图（PDG） [15]同时捕获数据依赖和控制依赖，从而支持更精确的优化分析。近年来，研究者提出了用于指针分析的广义指向图（GPG） [16] ，仅保留与别名相关的控制流图部分。关于控制流图稀疏性的一项颇具影响力的成果来自 Thorup [17],：他证明，在许多广泛使用的语言中，结构化（即不使用 goto 语句）程序的控制流图树宽至多为 6。树宽 [18]是衡量图具有多强树状结构的经典指标：树宽为 k 的图可以分解为一系列彼此重叠的小型顶点集合（“袋”），每个集合的大小至多为 $k+1$，并按照树状结构排列。图1展示了这样一种分解，更多背景介绍见 [19, 20]。这种结构支持高效的自底向上和自顶向下动态规划技术，以及分治策略 [21, 22, 23, 24, 25, 26, 27, 28]。

[17] 的研究结果表明，在 C语言 和 Pascal 等语言中，控制流图的树宽至多为 6；此后，许多类似研究将这一结果扩展到了其他语言，例如 Java、Ada [30], 以及 [31] 。

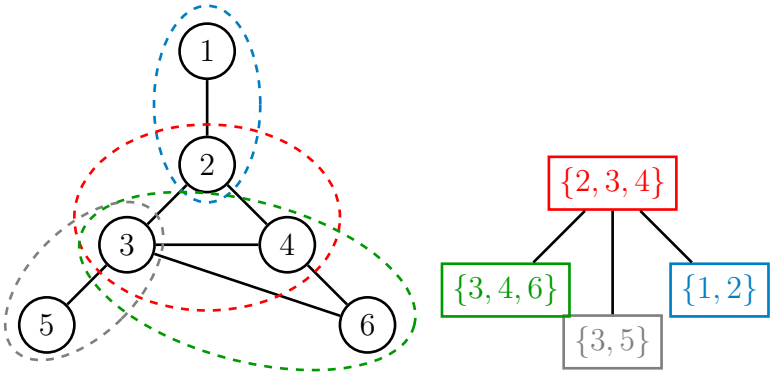


图1：图 G （左）以及 G 的树分解（右）。此示例取自 [29]。

Solidity[32], ，无论从理论上还是实验上都是如此。 [33] 这项研究表明，在某些 C语言变体中，树宽可以达到 7。

这些有界树宽结果实际上开辟了一个全新的研究方向。当与程序相关的任务（例如编译器优化或验证中的任务）被归约为图问题时，我们通常不必在一般图上求解，而只需在有界树宽的图上求解，因为它们已经覆盖了所有可能的控制流图。因此，我们不再寻找运行时间为 $O(n^c)$的多项式时间算法（其中 n 是输入规模），而是旨在寻找所谓的分段多项式（*XP*）或固定参数可解（*FPT*） 算法 [34] ，其运行时间形式分别为 $O(n^{f(k)})$ 和 $O(n^c \cdot f(k))$，其中参数 k 是树宽， f 是任意可计算函数，可能不是多项式。直观地说，由于我们知道在所有关心的实例（即控制流图）中，树宽很小且至多为7，因此运行时间为 $O(n^c \cdot f(k))$的算法对于所有实际目的而言实际上都是多项式时间的。例如，一个耗时 $O(n \cdot 2^k)$的算法在控制流图上的表现将如同线性时间算法，因为 $k \leq 7$ ，从而 2^k 是一个常数。

这种方法在为程序相关任务中出现的 NP困难图问题寻找多项式时间（FPT）算法方面极为成功，使这些问题的实际实例变得可处理。以下是一些著名示例：

- 在证明了树宽界之后， [17] 指出，Chaitin式寄存器分配，即将程序变量分配到数量有限的寄存器中，使得任何两个生存期有交集的变量（不允许拆分生存期时

）都不会被分配到同一个寄存器，在有界树宽的图上可以在线性时间内实现近似，且与最优解的差距不超过 k 倍$\lfloor k/2+1 \rfloor$ 。这使得控制流图可以获得 4-近似解。相比之下，众所周知，该问题等价于图着色 [1] ，因此，除非 P=NP，否则在一般图上无法在任何常数因子范围内有效近似。

- 后来，这一结果被改进为一个针对寄存器分配判定版本的最优线性时间FPT算法，即判断在固定数量的 r 寄存器 [35]下，是否可以不发生溢出。

- 后来有研究证明，当树宽有界时，最优成本寄存器分配问题（即在寄存器数量固定的情况下最小化溢出的总成本）也可以在多项式时间（XP）内求解 [36]。

- 在 [37, 38], 中，研究者证明，图的单子二阶逻辑中的任意公式都可以在有界树宽图上于线性时间内进行模型检验。这一结果在 [39] 中得到推广，适用于一类更一般的、可嵌入且树宽有界的图文法。

- [40, 41] 研究者指出，数据打包和缓存感知型数据布局这两个编译器优化中的经典缓存管理问题，在树宽有界的实例上可以在多项式时间内求解或近似求解，尽管它们在一般实例上是困难的 [13, 12]。

- 文献 [11] 证明，对于底层图具有有界树宽的奇偶博弈，可以在多项式时间内判定胜者。奇偶博弈与 μ-演算/线性时序逻辑模型检测之间众所周知的对应关系，使这一结果适用于从控制流图中获得的实际实例。

即使原图问题已能在所有图上通过多项式时间算法求解，有界树宽的结果仍然很有用。在许多问题中，利用树宽可以得到次数更低的多项式算法。例如：

- 该研究 [42, 43, 44, 45, 43] 考虑过程间数据流分析，即空指针识别和可用表达式等问题，

并为其按需变体提供线性时间的 FPT算法。这与非参数化方法的二次运行时间 [7, 46]形成对比。在 [47]中，代数程序分析也取得了类似结果。

- 线性代数中的各种问题 [48], ，包括高斯消元，以及马尔可夫链和马尔可夫决策过程中的许多经典定性与定量任务 [49, 50] ，如果底层图具有有界树宽，都可以在线性时间内通过 FPT 算法求解。这与一般图上已知的最佳算法形成对比，后者需要 $O(n^\omega)$，其中ω 是矩阵乘法常数。近期，线性规划也取得了类似的改进 [51]。

上述程序相关图问题的所有进展都基于同一个基本直觉：控制流图是稀疏的，在控制流图上求解问题，与在一般图上求解相同问题截然不同，而且通常容易得多。因此，顺着这一直觉自然推演，我们不禁要问：有界树宽是否足以刻画控制流图的稀疏性?

一方面，很容易构造出树宽有界、却无法表示为控制流图的图。例如，考虑由 n 个连通分量组成的图，其中每个分量都是一个含 6 个顶点的完全图，即 $K_{6.}$。这样的图树宽为 5，但显然不可能是任何结构化程序的控制流图，因为控制流图中的任何节点的出度最多为 2。另一方面，近期的研究 [29] 表明，控制流图不仅具有有界树宽，而且通常还具有较小的路径宽度；也就是说，它们可以分解为较小的袋，这些袋彼此之间的连接方式不仅类似于树，也类似于路径。该研究进一步表明，这使得动态规划算法的效率大幅提升，并显著缩短了寄存器分配等问题的运行时间。直观而言，这是因为在路径上进行动态规划通常比在树上更简单。然而，根本问题依然存在：有界路径宽度能否恰好刻画控制流图的稀疏性? 答案是否定的，反例与上文相同。因此，我们考虑这样一个问题：能否找到一种标准的图分解方法，恰好刻画控制流图的集合? 是否存在一种分解方法，既能（i）为编译器优化和形式化验证中的问题实现快速动态规划，又（ii）恰好适用于与控制流图相同的图集合?

在这项工作中，我们提出了一种基于图文法的分解方法，该图文法与用于定义结构化编程语言语法的文法十分相似。我们的方法既与串并联图的标准定义 [52] 相似，也与此前基于图文法的程序分析方法 [53, 54]相似。不过，我们的图更为一般，能够恰好覆盖所有控制流图，包括含有break语句 和 continue语句 的结构化程序的控制流图。我们基于图文法定义了控制流图的一种自然分解，并证明该分解可用于在控制流图上高效地进行动态规划。作为两个具体应用案例，我们考虑 Chaitin式寄存器分配问题和生命周期最优推测式部分冗余消除（LOSPRE）。这两个问题在文献中都属于经典且研究充分的问题，并且已有参数化和非参数化的解。我们基于所提出的分解方法设计了算法，其速度不仅快于以往的非参数化解，甚至也快于利用有界树宽的最先进方法。

最后，我们在第6节展示了全面的实验结果，表明我们的方法在实践中显著缩短了运行时间。值得注意的是，对于无溢出寄存器分配，我们的方法是首个能够处理最多使用20个寄存器的实际实例的精确算法，相较于此前仅能处理8个寄存器的方法取得了重大进展。这一点尤为重要，因为 x86 系列等标准架构通常配备16个寄存器。因此，我们的算法是首个适用于这些架构的精确解，无须再为无溢出寄存器分配采用近似或启发式方法。对于 LOSPRE，与此前基于树宽的最先进方法相比，我们的方法平均提升约五倍。鉴于此前的解已经过高度优化，这一提升意义重大。

尽管 Chaitin式寄存器分配和 LOSPRE 的运行时间改进本身已十分显著，但我们相信，若采用我们的串并联环分解概念，而非树宽／路径宽度，同样的改进也能惠及范围更广的程序相关分析。

总而言之，这项工作将利用控制流图的稀疏性来加速图算法这一思路发挥到了极致，并定义了一种分解概念。与树宽等参数不同，它精确地 刻画了可作为结构化程序控制流图的图集。它

随后，针对编译器优化中的两个经典问题——最小代价寄存器分配问题和LOSPRE，利用这一新的分解概念提出了更快的算法。我们预计，这种分解也有助于解决许多过去采用树宽／路径宽度参数化的问题。

2. 背景

本节回顾了支撑本文后续所提出算法框架的基础概念。首先介绍参数化算法理论，它为利用困难问题的结构性质提供了工具；随后讨论树分解和树宽，这两种参数是用于刻画图稀疏性、影响最深远的参数。这些概念在许多编译器优化任务中发挥着关键作用，尤其是在涉及控制流图（CFG）的任务中。

2.1. 参数化算法

参数化算法 [20] 通过分离出一个或多个能够刻画输入结构特征的参数，为分析计算问题提供了一个更精细的框架。参数化复杂性不再仅以输入规模 n来衡量复杂度，而是引入辅助参数 k，从而能够开发出这样的算法：即使对于 NP 困难问题，只要 k 相对较小，算法仍能保持高效。这一范式推动了理论计算机科学多个领域的突破，影响了图算法、形式化验证和编译器优化等领域 [55, 56]。

该领域的一个核心概念是固定参数可解性 （FPT） [57]。如果一个问题能在以下时间内求解，则称其为固定参数可解问题

$$O(f(k) \cdot n^c),$$

其中， f 是仅取决于参数 k的可计算函数，而 c是与 n 和 k均无关的常数。尽管 $f(k)$通常呈指数级增长，但其关键优势在于组合爆炸仅限于输入中的一小部分，因此当 k较小时，该算法仍然高效。

一个典型示例是顶点覆盖 问题 [58]。给定一个图$G = (V, E)$，目标是找出一个大小最小的顶点子集，使其包含每条边的至少一个端点。尽管一般情况下该问题

是NP难问题，但参数化 问题是，

是否存在大小至多为 k的顶点覆盖?

具有一种优雅的 FPT算法。下方展示的分支策略体现了这一思路：每一步选择一条未覆盖边（u, v），并分支考虑将 u 或 v 纳入解中。

1 2 定义 顶点_覆盖(图, k): 3 如果 k < 0: 4 返回 None 5 如果 图.为空(): 6 返回 集合() 7 8 u, v =图.边[0]9 覆盖_用_u = 顶点_覆盖(图.移除_顶点(u), k - 1) 10 覆盖_用_v = 顶点_覆盖(图.移除_顶点(v), k - 1) 11 12 如果 覆盖_用_u 不为 None: 13 返回 覆盖_用_u.并集({u}) 14 如果 覆盖_用_v 不为 None: 15 返回 覆盖_用_v.并集({v}) 16 返回 None

这一递归过程最多产生 2^k 个分支，且每个分支都执行多项式时间工作，因此运行时间为 $O(2^k \cdot n)$。因此，$f(k) = 2^k$，该问题属于 FPT。

除了 FPT，参数化复杂性还研究更广泛的 XP 类 [57]。如果一个问题存在运行时间为

$$O(n^{f(k)}),$$

的算法，则该问题属于 XP。这意味着，对于每个固定的参数值 k，该问题都可以在多项式时间内求解——尽管多项式的次数可能取决于 k。XP算法特别适用于参数值适中的情况，并在经典多项式时间算法与严格意义上的固定参数可处理算法之间起到桥梁作用。

例如，考虑求解大小为k的

顶点覆盖问题的朴素枚举算法。可以简单地枚举所有 $\binom{n}{k}$ 子集，并在线性时间内检查每个候选集。这种方法的运行时间为

$$O(n^{k+1}),$$

这符合带有 $f(k) = k + 1$ 的 XP 定义。尽管该策略远不如 FPT 分支算法高效，但它提供了一个正确且概念上简单的 XP 解。

参数化推理也已成为编译器优化和模型检测等实际领域中的重要工具。针对控制流图的许多分析都受益于较小的结构参数（例如树宽或路径宽度），从而在这些参数有界时使算法能够在线性时间内运行。这一优势推动了越来越多将参数化技术应用于优化现实世界编程语言和验证流程的研究。

2.2. 树分解与树宽

树分解由 Robertson 和 Seymour [59], 提出，为理解一个图在多大程度上“像树”提供了有力途径。图 $G = (V, E)$ 的树分解由一棵树 T 以及一族顶点子集（袋） $\{B_t\}_{t \in T}$组成。图1中给出了一个示例。该示例最初出现在 [29] 中。该分解必须满足以下条件：

- **覆盖条件：** 每个顶点 $v \in V$ 至少出现在一个袋中B_t。
- **边条件：** 对于每条边 $(u, v) \in E$，都存在一个同时包含 u 和 v的袋。
- **连通性条件：** 对于每个顶点 $v \in V$，所有满足 $v \in B_t$ 的节点 t 都必须构成 T 的一棵连通子树。

分解的宽度是其最大袋的大小减一，而 G 的树宽 是所有可能分解中的最小宽度。树宽较小的图在很多方面与树相似，因此可以使用多种在任意图上无法实现的动态规划技术。

树分解在解决 NP 困难问题的特殊情形时已证明特别有效。将图分解成较小且相互重叠的部分后，可以在树结构上进行动态规划。由于每个袋的大小都有上限，每个节点所需的计算量都可控，因此算法的整体运行时间通常与 $|V|$呈线性关系，而指数级增长仅取决于树宽。

编译器分析中有一个著名的应用。近期的理论结果表明，每个无 goto 的结构化控制流图（控制流图）的树宽至多为 7 [17]。这使得在许多程序分析中，可以将树宽视为一个固定常数。因此，利用有界树宽结构，控制流图上的许多优化问题——例如寄存器分配、数据流分析和代码移动——都能高效求解 [60, 61]。

树分解上的动态规划过程如下：设P是定义在图 G上的一个问题，且 $(T, \{B_t\}_{t \in T})$ 是一个树分解。对于每个节点 $t \in T$，令 G_t 表示袋 B_t上的诱导子图。假设我们在每个节点处构造一个表 S_t ，并满足：

1. S_t 包含足够的信息，以便在 G_t上求解问题 P 。
2. 对于叶节点， S_t 仅使用 G_t计算。
3. 对于内部节点， S_t 通过结合其子节点的表以及 G_t的结构来计算。

当这些条件成立时，可以沿树自底向上进行动态规划。由于每个袋的大小至多为 $w+1$，其中 w 是树宽，因此每个袋的计算量仅由 w 决定。由此，总运行时间为 $O(f(w) \cdot n)$；当 w 是一个较小的常数时，该时间与程序的大小呈线性关系。这一原理是许多用于编译器优化和静态分析的现代参数化算法的基础。

3. SPL 分解

为了定义结构化程序，我们采用类似于 [17] 的语法。程序由以下文法生成：

$$\begin{aligned} P := \;& \epsilon \mid \texttt{break} \mid \texttt{continue} \mid P; P \\ & \mid \texttt{if } \varphi \texttt{ then } P \texttt{ else } P \texttt{ fi} \mid \texttt{while } \varphi \texttt{ do } P \texttt{ od}. \end{aligned} \tag{1}$$

这里， ϵ 是不影响控制流的中性语句，例如变量赋值； φ 是布尔表达式。如果每个 break 和 continue 语句都出现在 while 循环的主体内，我们就称程序 P 为闭合的 。程序 P 的语义将按通常方式定义。本节只关注程序 $P.$ 的控制流图。我们还注意到，for 循环和 switch 语句等其他常见结构都可以定义为语法糖 [17]。具体而言，带有 k 个跳转的 switch 语句可以建模为 k 个 if 语句。

受串并联图 [62] 以及 [54], 的方法启发，我们定义了一种图文法，通过组合由同一文法生成的较小图来构建新图。为涵盖所有控制流图，我们考虑三种组合操作：串联、并联和循环。因此，我们将由该文法生成的图称为 SPL 图。此外，与具有两个特殊终端（即起点和终点）的串并联图不同，我们的图具有四个特殊顶点，分别是起点（S）、终止点（T）、break语句（B）和 continue语句（C）。作为基本情形，图 2 中的三个图都是 SPL 图，称为 原子的 图 A_ϵ， $A_{\texttt{break}}$ 和 $A_{\texttt{continue}}$。

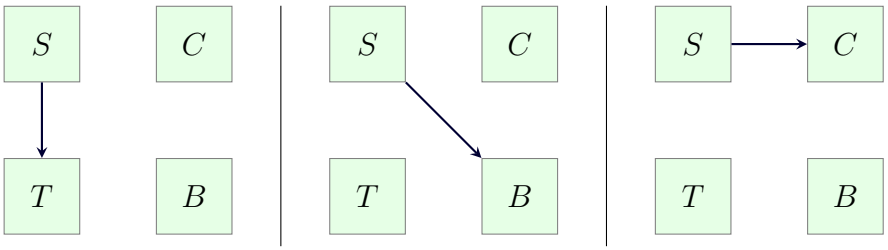


图 2：原子 SPL 图： A_ϵ （左）、 $A_{\texttt{break}}$ （中）和 $A_{\texttt{continue}}$ （右）。

SPL 图通过反复应用串联、并联和循环操作生成，分别记作 $\otimes$、 $\oplus$、和 $\circledast$ 。形式上，SPL 图的集合由以下上下文无关文法定义：

$$G = A_\epsilon \mid A_{\texttt{break}} \mid A_{\texttt{continue}} \mid G \otimes G \mid G \oplus G \mid G^{\circledast}. \tag{2}$$

我们用元组 $G = (V, E, S, T, B, C)$, 表示一个 SPL 图，其中 V是该图顶点的有限集合， E 是其边的有限集合，而$S, T, B, C \in V$是指定的起始、终止、break语句和 continue语句顶点。当上下文已明确时，我们有时会省略 V 和 E 。

我们现在定义三种运算。设 $G_1 = (V_1, E_1, S_1, T_1, B_1, C_1)$ 与$G_2 = (V_2, E_2, S_2, T_2, B_2, C_2)$ 为两个不相交的 SPL 图。

1. 串联操作。图 $G_1 \otimes G_2$ 的构造方式是：先对 G_1 和 G_2 取不交并，然后合并特定的顶点对。具体而言，我们合并并认定

$$T_1 \equiv S_2, \qquad B_1 \equiv B_2, \qquad C_1 \equiv C_2,$$

从而得到三个合并后的顶点

$$M = (T_1, S_2), \quad B = (B_1, B_2), \quad C = (C_1, C_2).$$

请注意，这不会引入新边，也不会移除现有边。相反，G_1 或 G_2 中与某个被识别顶点相连的每条边，现在都与对应的合并顶点相连。所得图的特殊顶点为 $(S_1,\ T_2,\ B,\ C)$。串联操作满足结合律，图 3 展示了两个示例。

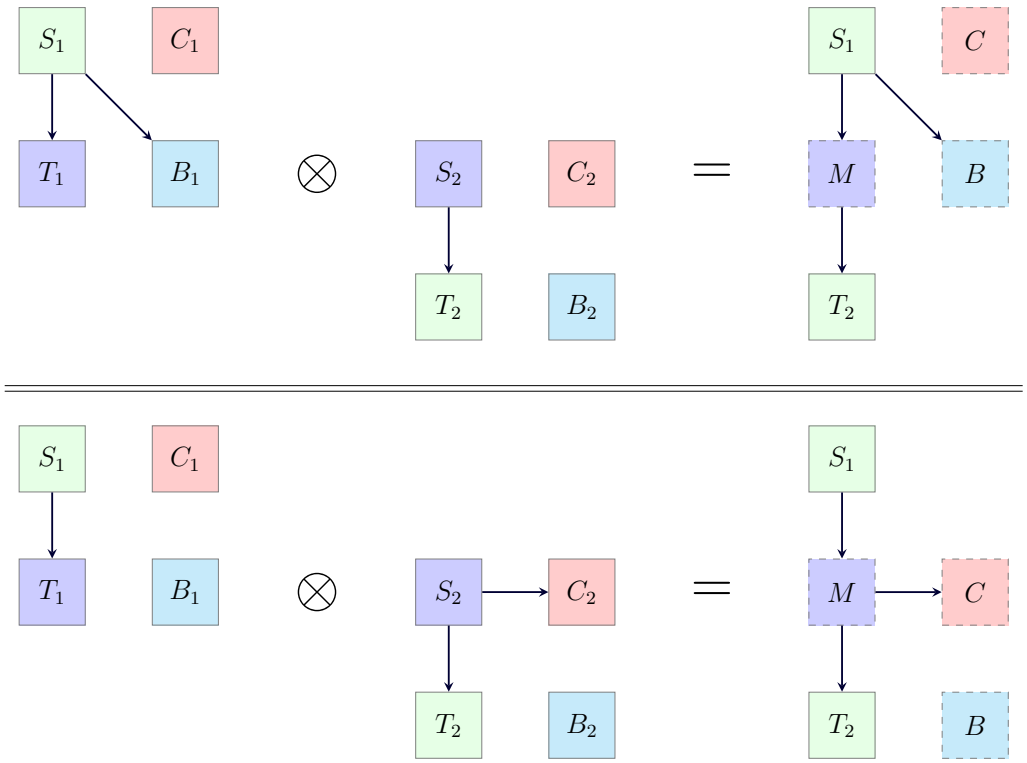


图 3：串联操作的两个示例 $\otimes$。在每种情况下，虚线顶点都是通过合并原始图中颜色相同的顶点而得到的新顶点。例如， T_1 和 S_2 合并形成 M.

2. 并联操作。 图 $G_1 \oplus G_2$ 是通过对 G_1 和 G_2 取不交并，然后合并全部四对指定顶点而得到的：

$$S_1 \equiv S_2, \qquad T_1 \equiv T_2, \qquad B_1 \equiv B_2, \qquad C_1 \equiv C_2.$$

这些识别操作产生了新的顶点

$$S = (S_1, S_2), \quad T = (T_1, T_2), \quad B = (B_1, B_2), \quad C = (C_1, C_2).$$

与串联情况一样，不会添加或删除任何边。所有与任一合并顶点相连的边都会重定向到新的合并顶点，而来自 G_1 和 G_2 且因此重复的边会自然合并

到并集中。 $G_1 \oplus G_2$ 的特殊顶点元组为（S， T， B， C）。图4中给出了一个示例。

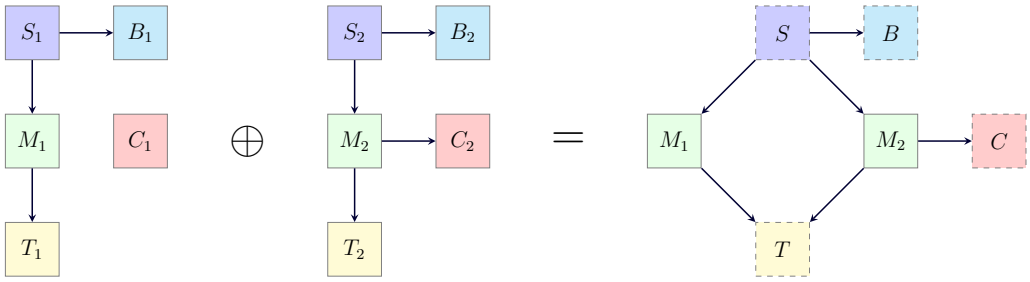


图4：并联操作的一个示例 $\oplus$。虚线顶点是将原始图中颜色相同的顶点合并后得到的新顶点。例如， S_1和 S_2 合并形成 S. 。请注意，尽管两个原始图彼此不相交，但结果图中的边(S, B)是原始图中两条边(S_1, B_1)和(S_2, B_2)合并而成的。

3. 循环操作。图 $G_1^{\circledast}$ 通过添加四个新顶点构成

$$S,\ T,\ B,\ C,$$

它们分别成为新的指定起始顶点、终止顶点、break语句顶点和继续顶点。然后，我们引入五条新边，将这些顶点连接到 G_1:

$$(S, S_1), \qquad (S, T), \qquad (T_1, S), \qquad (C_1, S), \qquad (B_1, T).$$

不会修改 G_1 的其他边。因此，循环操作会在 G_1 外创建一个新的“循环框架”，同时保留 G_1 的所有内部结构，从而扩展 G_1。 $G_1^{\circledast}$ 的特殊顶点元组为 $(S,\ T,\ B,\ C)$，图5给出了一个示例。

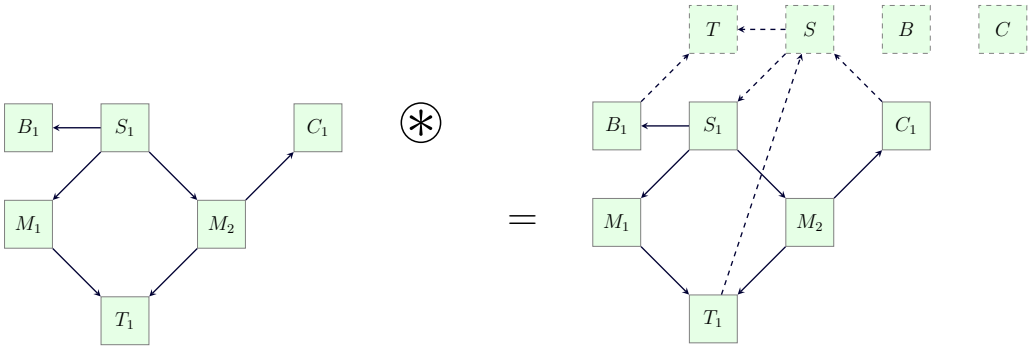


图5：循环操作示例。新引入的顶点和边以虚线表示。

如果 SPL 图 $G = (V,\ E,\ S,\ T,\ B,\ C)$ 的 B 和 C. 均没有入边，我们就称其为闭合的

给定一个 SPL 图 $G = (V,\ E,\ S,\ T,\ B,\ C)$，其文法分解是根据文法 (2) 得到的 G 的解析树。换句话说，文法分解是一棵树，其叶节点是原子 SPL 图，而内部节点对应于三种 SPL 操作$c_1 \otimes c_2$、 $c_1 \oplus c_2$和 $c_1^{\circledast}$在其子节点 c_1 和 c_2上的应用。

直观地说，分解树中的每个节点都表示程序控制流图中一个定义明确的子结构。叶节点编码最小的不可分割片段，通常是单条边或基本块边界；内部节点则表示通过按顺序（串联）、并行（分支/合并）组合其子区域，或将其作为循环体而得到的更大区域。因此，每棵子树都对应于原程序的一个连续部分，而沿树向上移动则对应于逐步组装更大的程序片段。当我们到达文法分解的根节点时，整个控制流图 G 已被重建，这意味着根节点恰好对应于整个程序。

例如，图6（左下）展示了图6（右下）中 SPL 图的文法分解。该分解的每棵子树都表示控制流图中一个有意义的片段，而根则表示完整的程序结构。

程序与.....之间存在一个自然的满同态 cfg(·),

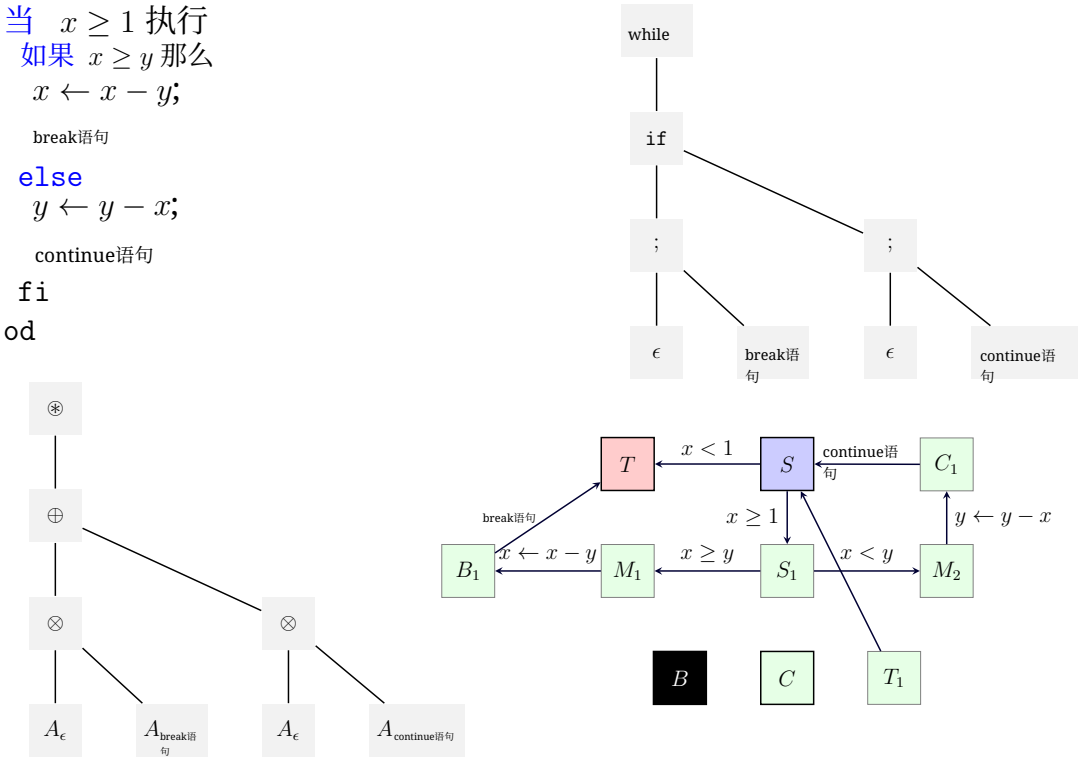


图6：程序 P （左上）、其解析树（右上）、 G 的对应解析树（左下），即其文法分解，以及图 $G = \mathsf{cfg}(P)$（右下）。图中的边根据程序的指令/条件标记。

SPL 图，其定义如下：

$$\mathsf{cfg}(\epsilon) = A_\epsilon \quad \mathsf{cfg}(\texttt{break}) = A_{\texttt{break}} \quad \mathsf{cfg}(\texttt{continue}) = A_{\texttt{continue}}$$

$$\mathsf{cfg}(P_1; P_2) = \mathsf{cfg}(P_1) \otimes \mathsf{cfg}(P_2)$$

$$\mathsf{cfg}(\texttt{if } \varphi \texttt{ then } P_1 \texttt{ else } P_2 \texttt{ fi}) = \mathsf{cfg}(P_1) \oplus \mathsf{cfg}(P_2)$$

$$\mathsf{cfg}(\texttt{while } \varphi \texttt{ do } P_1 \texttt{ od}) = \mathsf{cfg}(P_1)^{\circledast}$$

很容易验证，这个同态符合程序控制流图的通常定义，并且 P 是闭合的，当且仅当 $\mathsf{cfg}(P)$ 是闭合的。因此，SPL 图的集合与结构化程序的控制流图集合完全相同。

此外，给定一个程序 P，我们只需按照

文法 (1) 对其进行语法分析并得到一棵解析树，然后将我们的同态应用于这棵解析树，从而根据文法 (2) 找到其控制流图的一棵等价解析树。这正是控制流图的文法分解，它展示了当前程序的控制流图如何通过使用我们的三种操作之一合并较小部分的控制流图而得到。由于对上下文无关文法进行语法分析所需的时间与程序大小呈线性关系，即 $O(n)$，因此我们也能在 $O(n)$ 时间内得到控制流图的文法分解。图6展示了一个示例。请注意，在此示例中，我们还使用程序中的指令/条件为控制流图的边标注。标签并非图的一部分，只是为了提高可读性而添加。

在第6节的实验中，我们通过对程序进行语法分析来找出文法分解。不过，也可以直接基于控制流图来找出这些分解。形式上，判断图 G 是否为 SPL 图并找出其文法分解的问题，等价于对上下文无关文法进行语法分析。因此，CYK算法也适用 [63]。

我们注意到，从概念上说，我们的文法分解与串并联图的相似度，高于与树分解及基于树宽的算法的相似度。具体而言，我们没有树宽或路径宽度这类单独的参数。相反，我们只关注能够通过上述三种操作得到的图集合。下一节中，我们正是借此避开寄存器分配的 NP 困难性。我们不在所有图上求解，而只在 SPL 图上求解；SPL 图与控制流图相同，因此涵盖了该问题的所有实际实例。直观地说，树宽／路径宽度算法之所以高效，主要是因为它们能够反复在控制流图中找到较小的割，并利用这些割递归地将图切分为较小的部分。这也是我们方法背后的思路。区别在于，基于树宽的方法找到的割的大小至多为 8，而基于路径宽度的方法找到的割的大小为 $2 \cdot d + 1$，其中d 是嵌套深度；相比之下，我们的割恰好是特殊顶点，因此大小为 4。我们很快就会看到，较小的割能带来更高效的算法。

4. 寄存器分配

作为文法分解这一概念的一项应用，我们考虑 [36] 中形式化定义的经典问题——最小成本寄存器分配。为变量分配寄存器时，会为每种分配方式指定一个成本，用于模拟溢出或重物化所浪费的时间。这

个成本旨在模拟溢出或重物化所浪费的时间。我们注意到，相较于 [17, 29] 中仅关注能否完全避免溢出、从而实现零成本的表述，这种表述对问题作了更一般化的定义。参见第 4.3 节。 [36]，提供了一种基于树宽的算法，可在 XP 时间内获得最优寄存器分配 $O(|G| \cdot |\mathbb{V}|^{2\cdot(t+1)\cdot r})$，其中 G 是控制流图，$\mathbb{V}$ 是程序变量集合，t 是 G 的树宽，r 是可用寄存器的数量。如果 r 和 t 都是常数，那么这就是一个多项式时间算法。然而，由于 C语言等语言中的程序树宽最高可达 7 [33],，该算法在控制流图上的最坏情况运行时间为 $O(|G| \cdot |\mathbb{V}|^{16\cdot r})$。本节将介绍一种采用文法分解的替代算法，它显著缩短了运行时间，运行时间为 $O(|G| \cdot |\mathbb{V}|^{5\cdot r})$。与 [33] 的算法相比，其渐近性能有了巨大提升。

4.1. 问题定义

假设给定一个程序 P，其控制流图为 $G = \mathsf{cfg}(P) = (V, E, S, T, B, C)$。令 $[r] = \{0, 1, \ldots, r-1\}$ 为可用寄存器集合，$\mathbb{V}$ 为程序变量集合。每个变量 $v \in \mathbb{V}$ 都有一个生存期 $\mathsf{lt}(v)$，它是 $G.$ 的一个连通子图。有关生存期的详细说明，请参见 [64]。由于生存期可以通过简单的数据流分析计算得出，因此不失一般性，我们假设它们是作为输入提供给我们的算法的。对于 G 中的顶点 或边，我们用 $L(v)$ 或 $L(e)$ 表示在该顶点/边处存活的变量集合。 赋值是一个函数$f\colon \mathbb{V} \to [r] \cup \{\perp\}$，它将每个变量映射到一个寄存器或 $\perp$.。后者表示该变量被溢出到内存。若一个赋值不会将两个生存期相交的变量映射到同一个寄存器，则称其为有效赋值。我们用 $F.$ 表示所有有效赋值的集合

我们程序的干扰图 P 是一个图 $\mathbb{I} = (\mathbb{V}, E_{\mathbb{I}})$，其中每个程序变量对应一个顶点；如果变量 u 和 v 可以同时处于活跃状态，即 $\mathsf{lt}(u) \cap \mathsf{lt}(v) \neq \emptyset.$，则它们之间存在一条边 $\{u, v\}$。任何有效赋值 f 都是 $\mathbb{I}$ 中一个顶点子集的有效着色，颜色取自 $[r].$。寄存器分配与图着色之间的这种对应关系众所周知，源自 Chaitin [1]。我们注意到，对于每个顶点 v，在 v 处活跃的变量集合$L(v)$构成 $\mathbb{I}.$ 中的一个团。

我们提供一个取自 [65] 的示例。图 7 展示了一个程序 P 及其控制流图 $G = \mathsf{cfg}(P)$，其中包括每个顶点处的活跃变量，以及干扰图 $\mathbb{I}.$。我们的目标是使用 r 种颜色为 $\mathbb{I}$ 的一部分顶点着色，其中 r 是可用寄存器的数量。一个完全着色

使用 4 种颜色的结果如图所示。这避免了任何溢出。我们还展示了使用 3 种颜色并产生一些溢出的部分着色。

一个代价函数 [36] 是一个函数 $c\colon E \times F \to [0, \infty)$。对于控制流图中一条对应程序某条指令的边 $e \in E$，$c(e, f)$ 表示按照 f. 进行寄存器分配时执行该指令的成本。我们假设 $c(e, f)$ 仅取决于在 e. 处存活的变量的分配决策。按照 [36], ，我们进一步假设可以通过常数时间预言机访问 c. 的求值结果。实践中， c 通常通过性能分析获得。总运行时间或代码大小等不同优化目标，可以通过选择合适的函数 c.来建模。

最优寄存器分配问题以 P、G、r、$\mathbb{V}$、c 以及变量的生命周期为输入，并要求求出赋值 $f \in F$，使总成本最小化；也就是说，我们的目标是

$$\text{minimize} \sum_{e \in E} c(e, f)$$

通过选择最优的 f.

4.2. 我们的算法

现在，我们将说明如何在控制流图 G 的文法分解上执行动态规划，以求得最优寄存器分配。我们的算法简单而优雅。我们自底向上处理文法分解，并针对文法分解中出现的每个子图$H =(V_H$、E_H、S_H、T_H、B_H、$C_H)$，定义以下动态规划变量：

$$\begin{aligned}\textsc{Opt}[H, f'] = \;& \text{The minimum total cost } \textstyle\sum_{e \in E_H} c(e, f) \\ & \text{of an assignment } f \text{ over } H \text{ such that} \\ & f_{|L(S_H) \cup L(T_H) \cup L(B_H) \cup L(C_H)} = f'.\end{aligned}$$

直观地说，对于特殊顶点中任意一个顶点处仍然存活的变量的每一种可能赋值 f' （S_H，T_H，B_H，C_H），我们都要在所有与其一致且包含该赋值的变量赋值 f 中，求最小总成本。计算出 $\textsf{Opt}[\cdot, \cdot]$ 值后，算法的最终答案，也就是寄存器分配的最小成本，只需取 $\min_f \textsf{Opt}[G, f]$.

现在，我们将说明如何组合计算任意 SPL 图 H 的 $\textsf{Opt}[H, f']$，假设我们已经计算出了以下对象的 $\textsf{Opt}[\cdot, \cdot]$ 值

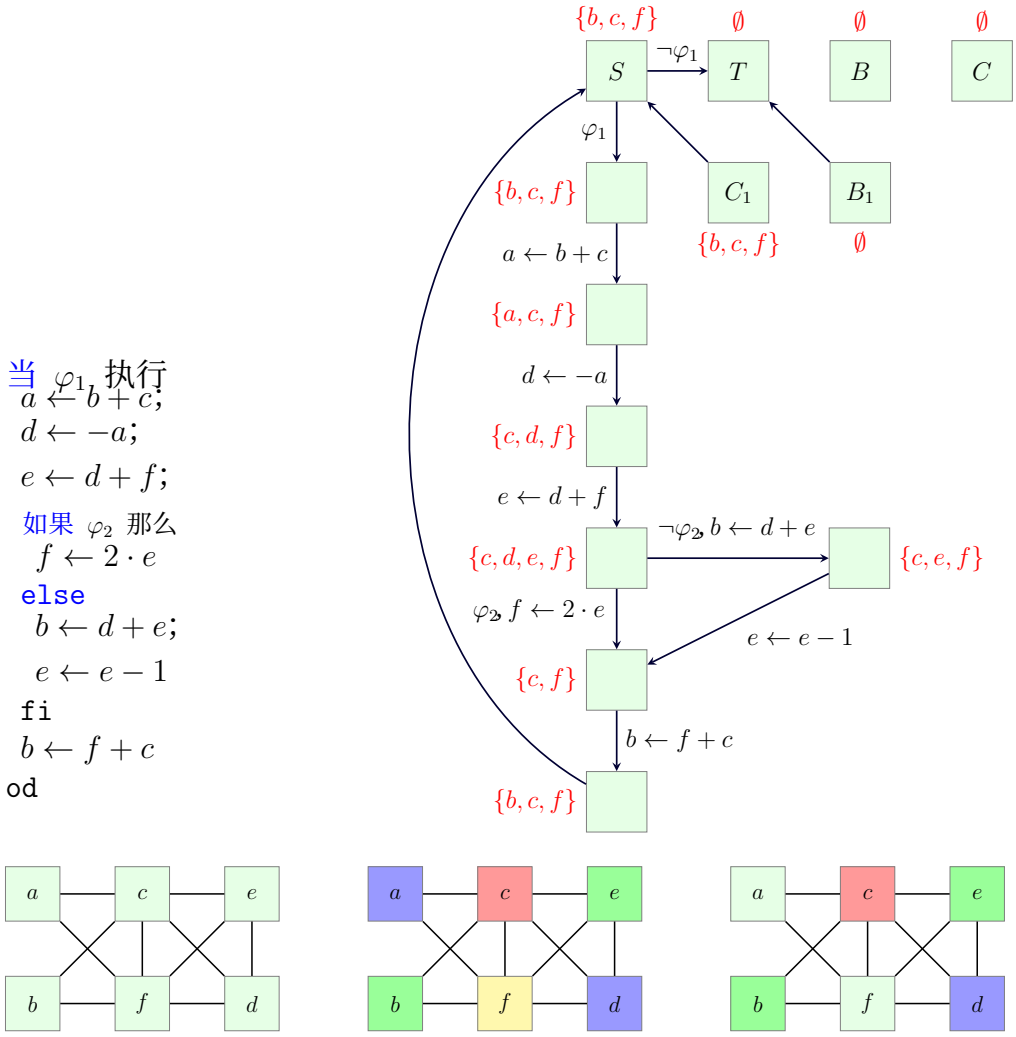


图 7：一个示例程序 P（左上），其控制流图 $G = \mathsf{cfg}(P)$（右上），其中每个顶点都用红色标注了其活跃变量集合；干扰图 $\mathbb{I}$（左下）；用 4 种颜色为 $\mathbb{I}$ 的所有顶点着色，表示将所有变量分配到 4 个寄存器（中下）；以及用 3 种颜色为 $\mathbb{I}$ 的部分顶点着色，表示将变量 a 和 f 溢出（右下）。

SPL 子图 H. 这通过分类讨论来完成:

- 原子图: 如果 $H \in \{A_\epsilon, A_{\text{break语句}}\ A_{\text{continue语句}}\}$，那么 H 除了特殊顶点（S_H， T_H， B_H， C_H）之外不含任何顶点。因此，在 H 的任意位置处存活的所有变量，也会在某个特殊顶点处存活；对于每个部分分配 $f'.$ ，我们只需设置 $\mathsf{Opt}[H, f'] = c(e,\ f')$。这里， e 是 H.中的唯一一条边

相容赋值。我们称两个部分赋值 $f_1 : \mathbb{V}_1 \to [r] \cup \{\perp\}$ 和 $f_2 : \mathbb{V}_2 \to [r] \cup \{\perp\}$ 相容，并记作 $f_1 \leftrightarrows f_2$ ，如果$\forall v \in \mathbb{V}_1 \cap \mathbb{V}_2 \quad f_1(v) = f_2(v)$。通俗地说， f_1 和 f_2 不会对任何变量 v 作出相互冲突的决定，但对于仅由 f_1 或仅由 f_2.决定的变量，我们不作任何限制。换言之， f_1 和 f_2 可以合并到同一个完整赋值中。

- *Series Operation:* If $H = H_1 \otimes H_2$, then we have

$$\textsc{Opt}[H_1 \otimes H_2, f'] =$$

$$\min_{\substack{f' \leftrightarrows f_1' \\ f' \leftrightarrows f_2' \\ f_1' \leftrightarrows f_2'}} \left(\textsc{Opt}[H_1, f_1'] + \textsc{Opt}[H_2, f_2'] - \sum_{e \in E_{H_1} \cap E_{H_2}} c(e, f_1') \right).$$

  这一计算的正确性直接由串联操作的定义得出。根据构造，我们有 $L(B_{H_1}) = L(B_{H_2}) = L(B_H)$ 和 $L(C_{H_1}) = L(C_{H_2}) = L(C_H)$，此外还有 $L(T_{H_1}) = L(S_{H_2})$。此外， H_1 和 H_2 中的每条边都保留在 $H_1 \otimes H_2$.中。因此，总成本就是两个分量中成本之和。我们还应注意，不能重复计算同时出现在 H_1 和 H_2中的边的成本。因此，我们要减去这些成本。当然，部分赋值 f'， f_1'和 f_2'应当两两兼容。

- 并联操作: 此情况的处理方式与串联情况完全相同:

$$\textsc{Opt}[H_1 \oplus H_2, f'] =$$

$$\min_{\substack{f' \leftrightarrows f'_1 \\ f' \leftrightarrows f'_2 \\ f'_1 \leftrightarrows f'_2}} \left(\textsc{Opt}[H_1, f'_1] + \textsc{Opt}[H_2, f'_2] - \sum_{e \in E_{H_1} \cap E_{H_2}} c(e, f'_1) \right).$$

这是因为我们的并联操作也会保留 H_1 和 H_2. 中的所有边。请注意，可能存在同时出现在 H_1 和 H_2 中的边，例如，(S_{H_1}， B_{H_1}) 和 (S_{H_2}， B_{H_2}) 可能都与边 (S_H， B_H) 相同。因此，总成本是分量 H_1 和 H_2 中成本之和，减去它们的公共边的成本。与之前一样，我们还应确保所有部分赋值彼此兼容。

- 循环操作：假设 $H = H_1^{\circledast}$. 在这种情况下，根据我们的构造，H 与 H_1 具有相同的顶点和边，除了引入了四个新的特殊顶点（S_H、T_H、B_H、C_H）和五条新边 e_1 =(S_H、S_{H_1}) e_2 =(S_H、T_H) e_3 =(T_{H_1}、S_H) e_4 =(C_{H_1}、S_H) 和 e_5 =(B_{H_1}、T_H)。因此，我们的总成本就是 H_1中的总成本加上这些新边产生的成本。因此，我们有：

$$\textsc{Opt}[H_1^{\circledast}, f'] = \min_{f'_1 \leftrightarrows f'} \left(\textsc{Opt}[H_1, f'_1] + \sum_{i=1}^{5} c(e_i, f' \cup f'_1) \right).$$

至此，我们的算法便完成了，它计算最优赋值 f. 的成本。按照动态规划方法中的惯例，可以通过回溯算法的步骤，并记住每一步使取值达到最小的选择，来获得 f 本身。

定理 1。 给定程序 P，其变量为 $\mathbb{V}$，控制流图为 $G = \mathit{cfg}(P)$，可用寄存器的数量为 r，代价函数为 $c(\cdot, \cdot)$，作为输入，上述算法可在 $O(|G| \cdot |\mathbb{V}|^{5 \cdot r})$ 时间内找到最优寄存器分配，即最优赋值函数 f。

证明。 上文已论证其正确性。我们对运行时间分析进行分类讨论：

- 对于原子图，我们考虑对在四个特殊顶点中任一顶点处均活跃的变量进行部分赋值。令 a 为这些特殊顶点之一。在 a处活跃的变量集合 L (a) 在干扰图 $\mathbb{I}$. 中构成一个团。因此，任何有效的 f' 都可以赋值

$f'(v) \neq \perp$ 至多为 r 个变量 v 分配寄存器，这些变量属于 $L(a)$。此外，任意两个变量都不能分配到同一个寄存器。鉴于 $|L(a)| \leq |\mathbb{V}|$，为 $L(a)$ 中的变量进行赋值的可能方式总数至多为

$$\binom{|\mathbb{V}|}{r} \cdot r! + \binom{|\mathbb{V}|}{r-1} \cdot (r-1)! + \cdots + \binom{|\mathbb{V}|}{0} \cdot 0! \in O(r \cdot |\mathbb{V}|^r).$$

因此，鉴于我们有四个特殊顶点，f' 函数的总数至多为 $O(r^4 \cdot |\mathbb{V}|^{4 \cdot r})$。我们的算法对每个 f'，都只需查询一条边的代价，因此耗费常数时间。

- 当 $H = H_1 \otimes H_2$，我们注意到有 $B_H = B_{H_1} = B_{H_2}$ 和 $C_H = C_{H_1} = C_{H_2}$.。类似地，有 $T_{H_1} = S_{H_2}$.。因此，f'，f'_1 和 f'_2需要共同为五个顶点之一上存活的变量选择寄存器赋值：S_{H_1}，T_{H_1}，T_{H_2}，B 和 C.。与前一种情况类似的论证表明，这样的赋值有 $O(r^5 \cdot |\mathbb{V}|^{5 \cdot r})$ 个。我们还注意到，$E_{H_1} \cap E_{H_2}$ 有 $O(1)$ 条边，因为任意这样的边都必须连接两个特殊顶点，而我们只有四个这样的顶点。因此，这里的总运行时间也是 $O(r^5 \cdot |\mathbb{V}|^{5 \cdot r})$。

- 当 $H = H_1 \oplus H_2$，时，类似的论证同样适用。在这种情况下，我们有 $S_H = S_{H_1} = S_{H_2}$，$T_H = T_{H_1} = T_{H_2}$，$B_H = B_{H_1} = B_{H_2}$ 和 $C_H = C_{H_1} = C_{H_2}$. 因此，我们只需检查四个不同顶点处活跃变量的赋值，运行时间为 $O(r^4 \cdot |\mathbb{V}|^{4 \cdot r})$。

- 最后，当 $H = H_1^{\circledast}$，时，我们引入了四个新的特殊顶点。因此，我们似乎必须考虑总共八个顶点处的活跃变量，也就是 H 和 H_1.的特殊顶点。不过请注意，在我们的控制流图中，B_{H_1} 只有一条出边 G，该边指向 T_H.。因此，我们有 $L(B_{H_1}) \subseteq L(T_H)$。出于类似的原因，$L(T_{H_1}) \subseteq L(S_H)$ 和 $L(C_{H_1}) \subseteq L(S_H)$。因此，我们只需考虑在五个顶点 S_H，T_H，B_H，C_H和 S_{H_1}.之一处存活的程序变量。与前面几种情况类似的论证表明，运行时间为 $O(r^5 \cdot |\mathbb{V}|^{5 \cdot r})$。

最后，我们的算法必须以自底向上方式处理文法分解，并计算每个节点处的 $\mathsf{Opt}[\cdot, \cdot]$ 值。共有 $O(|G|)$ 个节点。因此，最坏情况运行时间总计为 $O(|G| \cdot r^5 \cdot |\mathbb{V}|^{5 \cdot r})$。遵循 [36] 等关于最小成本寄存器分配的研究，我们假设 r 是一个常数。因此，我们的运行时间为 $O(|G| \cdot |\mathbb{V}|^{5 \cdot r})$。 □

并行化。我们注意到，我们的算法完全可以并行化，因为对于每个 SPL 子图 H，都可以并行计算不同 f 函数的 Opt$[H, f]$ 值。因此，如果我们有 k 个线程，且 k 小于可能的部分函数数量 f，那么我们的运行时间将减少到 $O\left(\frac{|G|\cdot|\mathbb{V}|^{5}\cdot r^{5\cdot r+5}}{k}\right)$。

4.3. 无溢出寄存器分配

值得一提的是，[35]（SODA 1998）和 [29]（OOPSLA 2023）的研究针对是否存在无溢出寄存器分配这一判定问题，提供了相对于控制流图大小的线性时间算法。具体来说，设置 $c(e, f) = 0$，当且仅当 f 有效且能将所有变量分配到寄存器（即不会将任何内容映射到 $\perp$）时成立；否则设置 $c(e, f) = +\infty$，然后只需判断能否实现总成本为零的赋值。这是我们上文所讨论问题的一个特例。与一般情况不同，针对这一特例的研究通常不将 r 视为常数，而是根据 $|G|$ 和 r. 分析运行时间。前一项研究提出的算法运行时间为 $O(|G| \cdot r^{2\cdot t\cdot r+2\cdot r})$，其中 t 是控制流图的树宽。后一项研究采用了不同的参数，即路径宽度，并得到运行时间 $O($ $|G| \cdot p \cdot r^{p\cdot r+r+1})$。鉴于 C语言等语言中的结构化程序，其控制流图的树宽最高可达 7 [33]，且在 [29], 中观察到路径宽度在实证中也不超过 17，因此这些方法的运行时间分别为 $O(|G| \cdot r^{16\cdot r})$ 和 $O(|G| \cdot r^{18\cdot r+1})$。此外，[29]（图 9）观察到，绝大多数真实世界程序的路径宽度为 6 或更低。对于这些实例，其运行时间为 $O(|G| \cdot r^{7\cdot r+1})$。虽然这适用于大多数真实世界控制流图，但并不涵盖所有情况。一般而言，控制流图的路径宽度不存在已知的常数上界，因此 [29] 的算法在理论最坏情况下的运行时间要高得多。

我们现在证明，上述算法也显著降低了无溢出寄存器分配问题的时间复杂度，即对 r, 的依赖。

定理 2. 第 *4.2* 节的算法可直接应用于无溢出寄存器分配问题，即当 f 不映射任何内容到 $\perp$ 时，代价 $c(e, f)$ 为零，否则代价为 $+\infty$，并且该算法能在时间 $O(|G| \cdot r^{5\cdot r+5})$内解决该问题。

证明。 证明与定理 1 完全相同，只需额外注意以下一点：如果对于控制流图中的某个顶点 a，我们有 $|L(a)| > r$

，即如果同时存活的变量多于 r 个，那么无溢出寄存器分配问题的答案就是“否”。这是因为 $L(a)$ 中的顶点在 $\mathbb{I}$. 中构成一个团。因此，我们只需考虑对每个 a. 都有 $|L(a)| \leq r$ 的情况。所以，在定理 1 的证明分析中，当我们考虑在 $k \leq 5$ 个顶点处存活的变量时，可以确定这样的变量至多有 $k \cdot r$ 个。因此，总运行时间为 $O(|G| \cdot r^5 \cdot r^{5 \cdot r}) = O(|G| \cdot r^{5 \cdot r+5})$。

□

进一步优化。 此外，还有一种简单的优化，在实践中可以提升我们算法的性能。对于无溢出寄存器分配问题，我们没有任何寄存器偏好。因此，重命名和置换寄存器不会使有效赋值失效。同样，如果赋值无效，重命名寄存器也无法解决冲突。因此，对我们而言，除寄存器重命名外相同的寄存器分配都视为相同。这意味着，我们只需从每个等价类中存储一个代表作为规范表示。这样可以显著减少我们的算法需要计算的动态规划值数量。

与其他编译器优化结合。 我们的寄存器分配方法能够与其他编译器优化稳健地协同工作，前提是这些优化不会引入或移除程序变量，从而保持变量之间的生命周期交集不变。这是因为我们可以将此类优化执行前的控制流图视为寄存器分配器的输入。还有许多常见的优化，例如生命周期最优推测式部分冗余消除（LOSPRE） [3]，它们会改变变量，但保持程序结构化。在这种情况下，我们的方法可以在应用这些优化后得到的控制流图上运行。不过，如果编译器应用的优化既引入新变量，又将控制流图变为非结构化图，那么我们的方法就无法与之结合。

5. LOSPRE

冗余消除（RE），即避免对同一表达式进行重复且不必要的计算，自编译器诞生之初便一直是优化编译器的目标。简单来说，如果同一表达式 e 在程序中的多个不同位置被使用，那么可以考虑只计算 e一次，将其存储在临时变量中，随后每当程序执行到需要 e. 的位置时使用该变量。这一思想最早的形式化定义之一

于1970年以全局公共子表达式消除（GCSE） [66]的形式提出。后来的方法考虑消除仅出现在控制流图部分路径中的冗余，从而发展出部分冗余消除（PRE） [67]。惰性代码移动（LCM） [68], 对PRE进行了改进，其重点是通过缩短所引入临时变量的生存期来实现生命周期最优性。这也有助于降低寄存器压力。另一项经典改进是推测式部分冗余消除（SPRE） [69, 70], ，它根据性能分析信息选择添加计算的路径，以最大化PRE带来的收益。将LCM和SPRE的思想结合起来，便得到生命周期最优推测式部分冗余消除（LOSPRE）。目前，LOSPRE是冗余消除中表现力最强的方法，并且涵盖了上述所有其他方法。

现在，给定控制流图 $G = \{V, E\}$，我们正式将问题定义如下: :

•使用集。 考虑表达式 $e.$ 。我们将 e 的使用集 U 定义为控制流图中所有计算表达式 e 的节点所构成的集合。

• 活跃集。 我们的目标是在少数几个位置预先计算表达式 e ，将结果保存到临时变量 temp 中，然后在 $U.$ 的每个节点中用 temp 代替 e 。我们用 L表示变量 temp 的生存期，并称之为活跃集。

•失效集。 如果节点 v 处的语句改变了 $e.$ 的值，我们就称控制流图中的节点 v 使 e 失效。例如，如果 e =a+b，那么语句 a = 0 会使 $e.$ 失效。我们用 $I.$ 表示所有失效节点构成的集合。这些节点在 LOSPRE 中起着关键作用，因为它们会迫使我们通过重新计算 $e.$ 来更新保存在 temp 中的值。我们假设入口节点和出口节点都会使其失效，因为 LOSPRE 是一种过程内分析，无法获取当前函数执行之前或之后程序运行情况的信息。

• 计算集。 根据上述集合 U, L 和 I ，我们必须确保临时变量 temp 的值在$U \cup L.$ 中的每个节点处都正确。因此，对于控制流图中满足 $x \notin L$ 和 $y \in U \cup L,$ 条件的每条边 $(x, y) \in E$ ，我们都必须在 x 和 $y.$之间插入计算 temp $= e$ 。类似地，如果 $x \in I,$ ，那么执行 $x,$ 后，存储在 temp 中的值就会失效，因此我们需要在两者之间插入相同的计算

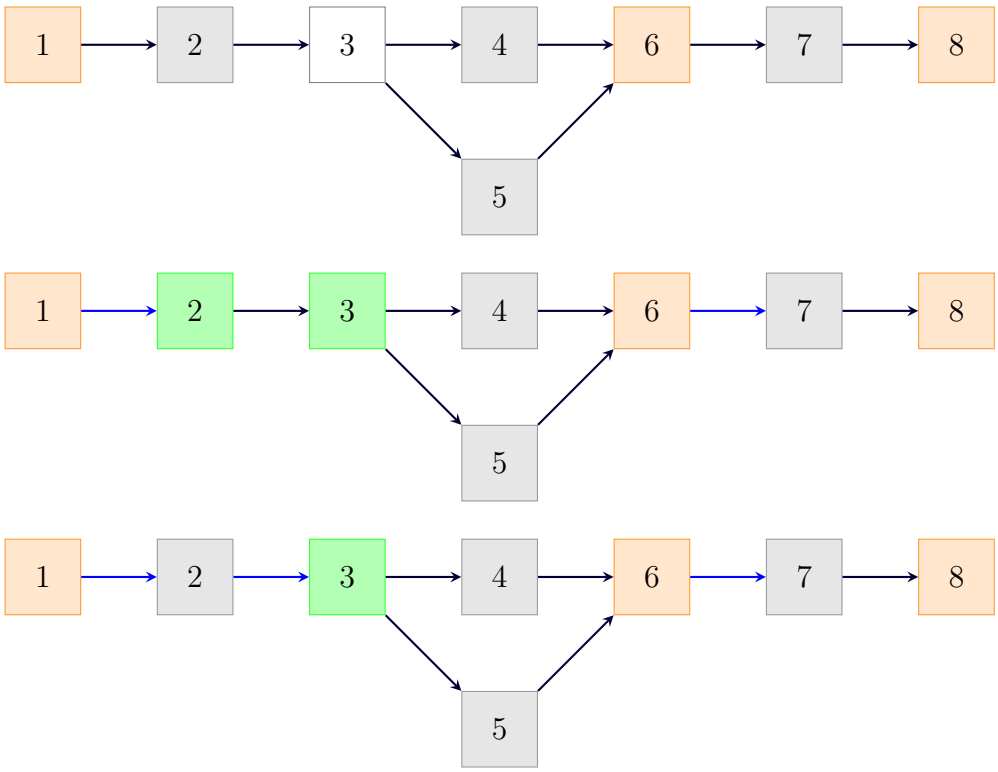


图 8：LOSPRE 的一个示例。使用集以灰色显示，失效集以橙色显示，生命周期集以绿色显示。计算集的边以蓝色显示。

x 和 y. 形式化地说，必须将计算 temp $= e$ 插入控制流图的以下边集：

$$C(U, L, I) = \{(x, y) \in E \mid x \notin L \setminus I \ \wedge \ y \in U \cup L\}.$$

图 8 展示了 LOSPRE 的一个示例。图的上半部分是一个控制流图，其中表达式 e 的使用集以灰色显示，也就是说，我们需要在顶点 $U = \{2, 4, 5, 7\}$. 处取得 e 的值。失效集以橙色显示，也就是说， $I = \{1, 6, 8\}$ 中的顶点会使 e. 失效。中间部分和底部部分分别展示了一种可能的优化。我们以绿色显示临时变量的生存期。

在中间部分，临时变量在 $\{2, 3\}$. 处处于活跃状态。因此，必须将计算 temp $= e$ 注入边 (1，2)。 接下来我们可以

在位置 2，4 和 5处使用 *temp* 替代 。 不过，我们需要在边 (6， 7)处重新计算。在这种情况下，我们的计算集是 {(1， 2)，(6，7)}. 计算集中的边以蓝色显示。

在底部部分，临时变量仅在位置 3 处存活。因此，我们首先在经过 (1，2) 时计算 e ，以便在位置 2 处获得它的值。然后，我们在经过 (2，3) 时重新计算 e ，并将其保存到临时变量 temp。接着，在位置 4 和 5 处使用这个临时变量代替 e 。这个示例展示了一种权衡：减少重复计算会延长临时变量的生存期，从而增加寄存器压力，不利于寄存器分配。

这一次，上述过程涉及两类成本：(i) 将计算注入 C 中的边（U， L， I），以及 (ii) 在 $L.$ 中的每个节点都保留一个额外的变量 temp。这些成本取决于编译器所追求的目标。例如，旨在减小代码大小的编译器会着重考虑 (i)。另一方面，如果我们的目标是缓解寄存器压力，就应尽量减小 (ii)。LOSPRE 是一个表达能力强的框架，它通过两个函数对这些成本进行建模

$$c : E \to K$$

and

$$l : V \to K.$$

其中， K 是一个带有加法运算符的全序集， c 是一个函数，用于将每条边映射到在该边添加 e 计算的成本； l 同样是一个函数，用于将控制流图的每个顶点映射到使临时变量 temp 在该顶点保持存活的成本。

形式化定义。 给定一个控制流图 $G = (V, E)$、一个使用集 U、一个失效集 I 以及两个代价函数 $c : E \to K$ 和 $l : V \to K$，LOSPRE 问题是求一个生命周期集 L，使总成本最小

$$\textsc{Cost}(G, U, I, L, c, l) = \sum_{e \in C(U,L,I)} c(e) + \sum_{v \in L} l(v).$$

我们的算法。 我们提出一种基于 SPL 分解、用于 LOSPRE 的线性时间算法。算法的输入包括一个闭合程序P、其控制流图 $G = (V, E)$、一个使用集 $U \subseteq V$、一个失效集$I \subseteq V$以及两个代价函数 $c : E \to K$ 和 $l : V \to K.$ 。我们的目标是找到

一个生命周期集 $L \subseteq V$，使其最小化

$$\textsc{Cost}(G, U, I, L, c, l) = \sum_{e \in C(U,L,I)} c(e) + \sum_{v \in L} l(v).$$

步骤 1（初始化）。我们的算法首先对 P 进行语法分析，然后应用上一节中的同态，从而计算 $G = \textsf{cfg}(P)$ 的 SPL 分解。

步骤 2（动态规划）。我们的算法在 SPL 分解上进行自底向上的动态规划。请注意，SPL 分解中的每个节点 u 都对应于 G 的一个 SPL 子图 G_u；该子图要么是原子 SPL 图（当 u 是叶节点时），要么是通过对与 $u.$ 的子节点相对应的图应用某个 SPL 操作而得到的。参见图6。令 $\Gamma_u = \{S_u, T_u, B_u, C_u\}$ 为 $G_u.$ 的特殊顶点集。对于每个 $X \subseteq \Gamma_u$，我们定义一个动态规划变量 $\textsf{dp}[u, X].$。我们的目标是计算这个动态规划值，使得

$$\texttt{dp}[u, X] = \min_{L \subseteq V_u \ \wedge \ L \cap \Gamma_u = X} \textsc{Cost}(G_u, U, I, L, c, l).$$

直观来说，我们考虑的是原始 LOSPRE 的一个子问题，其中图仅限于 $G_u.$。此外，我们只考虑满足 $L \cap \Gamma_u = X.$ 的解（生命周期集）L。那么，$\textsf{dp}[u, X]$ 中的值应给出所有此类解中的最小成本。下面，我们将介绍我们的算法如何根据各子节点处的 $\textsf{dp}[\cdot, \cdot]$ 值，计算分解中每个顶点 u 对应的 $\textsf{dp}[u, X]$：

- 原子节点：如果 G_u 是一个原子 SPL 图，那么 G_u 中仅有四个特殊顶点。因此，我们必须有 $L = X.$。我们的算法将每个 $\textsf{dp}[u, X]$ 计算为：

$$\texttt{dp}[u, X] = \textsc{Cost}(G_u, U, I, X, c, l) = \sum_{e \in C(U,X,I) \cap G_u} c(e) + \sum_{v \in X} l(v).$$

- 串联节点：设 $G_u = G_v \otimes G_w$，其中 v 和 w 是 SPL 分解中 u 的子节点。设 $X \subseteq \Gamma_u$ 和 $X_v \subseteq \Gamma_v$ 分别是 G_u 和 G_v、的特殊顶点子集。我们称 X 和 X_v 是兼容的，并记作 $X \leftrightharpoons X_v$，如果满足以下条件：

  - $S_v \in X_v \Leftrightarrow S_u \in X$;

- $B_v \in X_v \Leftrightarrow B_u \in X$;
- $C_v \in X_v \Leftrightarrow C_u \in X$.

直观地说，兼容性意味着子集 X 和 X_v 在是否将顶点纳入生命周期集 L. 这一点上做出相同的决定。由于 $S_u = S_v$，它们应该同时将其纳入，或同时将其排除。类似地， B_u 是通过合并 B_v 和 B_w得到的。因此，对 B_u和 B_v 做出的决定必须一致。对于 C_u 也是如此，它是由 C_v和 C_w.合并而成的。

现在考虑 $X_w \subseteq \Gamma_w$. 我们称 X_w 和 X 兼容，并记作 $X \leftrightharpoons X_w$ ，当且仅当满足以下条件：

- $T_w \in X_w \Leftrightarrow T_u \in X$;
- $B_w \in X_w \Leftrightarrow B_u \in X$;
- $C_w \in X_w \Leftrightarrow C_u \in X$.

直观上与前一种情况相同，只不过现在我们有$T_u = T_w$. 最后，我们称 X_v 和 X_w 兼容，并记作$X_v \leftrightharpoons X_w$ ，如果

- $T_v \in X_v \Leftrightarrow S_w \in X_w$.

这是因为 T_v 和 S_w 是控制流图中的同一个顶点。

在这一步中，我们的算法设置

$$\mathtt{dp}[u, X] = \min_{\substack{X \leftrightharpoons X_v \\ X \leftrightharpoons X_w \\ X_v \leftrightharpoons X_w}} \mathtt{dp}[v, X_v] + \mathtt{dp}[w, X_w] - [T_v \in X_v] \cdot l(T_v) - [B_v \in X_v] \cdot l(B_v) - [C_v \in X_v] \cdot l(C_v).$$

这是因为 G_u 中的每条边都出现在 G_v 或 G_w 中，但不会同时出现在两者中。因此，边的代价就是它们在两个子图中的代价之和。然而，对于顶点， T_v 和S_w 会合并， B_v 和 B_w， 以及 C_v 和 C_w. 也会合并。因此，我们必须确保不会重复计算这些顶点的活跃性代价。由于子节点处的两个动态规划值都已计入这项代价，我们应将其减去。

- 并联节点：我们可以用与处理串联节点相同的方式处理并联节点，即在两个子节点处找到兼容掩码，并确保

不会重复计算顶点成本。更准确地说，令 $G_u = G_v \oplus G_w$. 我们需要检查的兼容条件如下：

$$\begin{aligned} X \leftrightharpoons X_v \Leftrightarrow & (S_u \in X \Leftrightarrow S_v \in X_v \ \wedge \ T_u \in X \Leftrightarrow T_v \in X_v \ \wedge \ B_u \in X \Leftrightarrow B_v \in X_v \ \wedge \ C_u \in X \Leftrightarrow C_v \in X_v); \\ X \leftrightharpoons X_w \Leftrightarrow & (S_u \in X \Leftrightarrow S_w \in X_w \ \wedge \ T_u \in X \Leftrightarrow T_w \in X_w \ \wedge \ B_u \in X \Leftrightarrow B_w \in X_w \ \wedge \ C_u \in X \Leftrightarrow C_w \in X_w); \\ X_v \leftrightharpoons X_w \Leftrightarrow & (S_v \in X_v \Leftrightarrow S_w \in X_w \ \wedge \ T_v \in X_v \Leftrightarrow T_w \in X_w \ \wedge \ B_v \in X_v \Leftrightarrow B_w \in X_w \ \wedge \ C_v \in X_v \Leftrightarrow C_w \in X_w). \end{aligned}$$

基于与前一种情况相同的论证，我们的算法设定

$$\mathtt{dp}[u, X] = \min_{\substack{X \leftrightharpoons X_v \\ X \leftrightharpoons X_w \\ X_v \leftrightharpoons X_w}} \mathtt{dp}[v, X_v] + \mathtt{dp}[w, X_w] - [S_v \in X_v] \cdot l(S_v) - [T_v \in X_v] \cdot l(T_v) - [B_v \in X_v] \cdot l(B_v) - [C_v \in X_v] \cdot l(C_v).$$

• 循环节点：最后，我们应当处理 $G_u = G_v^{\circledast}$. 这种情况。这种情况相当简单。根据构造，与 G_v，相比，图G_u 有四个新顶点

$$V_{\text{new}} = \{S_u, T_u, B_u, C_u\}$$

以及五条新边

$$E_{\text{new}} = \{(S_u, S_v), (S_u, T_u), (T_v, S_u), (C_v, S_u), (B_v, T_u)\}.$$

这两个图 G_u 和 G_v 不共享任何特殊顶点，即 $\Gamma_u \cap \Gamma_v = \emptyset$. 此外，对于每条边 $(x, y) \in E_{\text{new}}$，我们都可以仅根据 X 和 X_v. 判断 (x, y) 是否属于计算集。这是因为$x, y \in X \cup X_v$. 更具体地说，(x, y) 属于计算集，当且仅当

$$\varphi(X, X_v, x, y) := [x \notin X \cup X_v \setminus I \ \wedge \ y \in U \cup X \cup X_v]$$

因此，我们的算法设置：

$$\mathtt{dp}[u, X] = \sum_{x \in V_{\text{new}} \cap X} l(x) + \min_{X_v \subseteq \Gamma_v} \mathtt{dp}[v, X_v] + \sum_{(x,y) \in E_{\text{new}}} \varphi(X, X_v, x, y) \cdot c(x, y).$$

第 3 步（计算最终答案）。令 r 为 SPL 的根

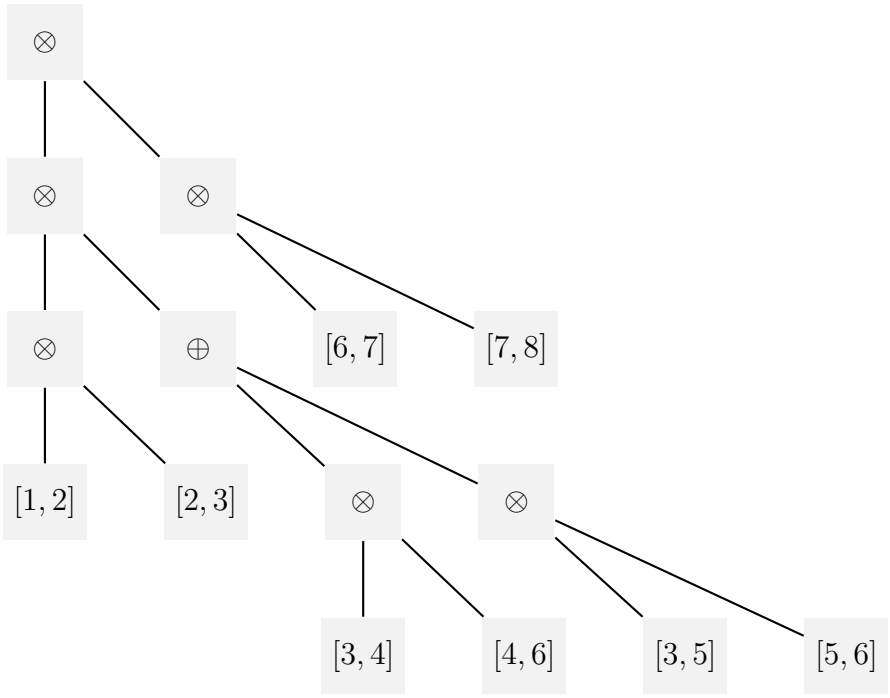


图 9：图 8 中示例的 SPL 分解

分解。根据定义，我们有 $G_r = G.$ 。算法输出 $\min^{X \subseteq \Gamma_r} \mathrm{dp}[r, X]$ ，作为给定 LOSPRE 输入的最小可能成本。这是因为 G_r 是整个控制流图 G ，并且任意解 L 都会恰好符合 $r.$ 处 X 的不同可能取值之一。与动态规划方法中的惯例一样，可以通过回溯算法的步骤，并记住每一步中导致最优值的选择，重构出达到该最小成本的最优活跃集 L。

示例。 现在，我们详细说明算法如何处理图 8 中的控制流图。为简单起见，我们再次假设所有边的边插入代价均相同$c(e) = 1$ ，且所有顶点的活跃性代价均相同 $l(v) = 0.1$ 。这些统一的代价使我们能够突出动态规划的结构性行为，而不必让常数因子模糊示例。回顾该示例中的相关集合：

$$U = \{2, 4, 5, 7\}, \qquad I = \{1, 6, 8\}.$$

图 9 展示了控制流图的一种 SPL 分解。由于该示例不包含循环，整个分解仅由原子的、串联、

以及并联 节点构成。对于这棵树中的每个节点 u，我们都会为其特殊顶点的每个子集 $X \subseteq \Gamma_u$ 计算动态规划表 $\mathsf{dp}[u, X]$。

叶节点。 分解的每个叶节点对应于一个简单双顶点片段 $[i, j]$，它编码一条控制流图边。由于该示例没有循环块，每个原子子图恰好有两个特殊顶点 $S_u = i$ 和$T_u = j$；在这种非循环设定中，break/continue 顶点无关紧要，可以忽略。因为 $\Gamma_u = \{i, j\}$，我们评估全部四个子集

$$X \in \big\{\emptyset,\, \{i\},\, \{j\},\, \{i, j\}\big\}.$$

对于每个子集，我们评估

$$\mathsf{dp}[u, X] \;=\; \sum_{e \in C(U,X,I) \cap G_u} c(e) \;+\; \sum_{v \in X} l(v).$$

在叶节点 $[i, j]$处，唯一的边是(i, j)，因此唯一的问题是这条边是否属于计算集。这取决于 $i \notin X \setminus I$和 $j \in U \cup X$是否成立。如果必须在(i, j)处注入更新，则我们增加代价1；如果顶点 i 或 j 包含在 X中，则我们加上相应的活跃性代价。

作为一个具体示例，考虑片段 $[3, 4]$。由于 $4 \in U$，该表达式的任何计算在节点4处都必须是正确的。因此，除非 $3 \in X$，边(3,4)必须出现在计算集中，并贡献代价1。我们得到：

$$\mathsf{dp}([3, 4], \emptyset) = 1, \qquad \mathsf{dp}([3, 4], \{3\}) = 0.1,$$
$$\mathsf{dp}([3, 4], \{4\}) = 1.1, \qquad \mathsf{dp}([3, 4], \{3, 4\}) = 0.2.$$

对所有叶节点执行这一操作，就能得到一张完整的表，每个叶节点对应 4 个条目。尽管每个叶节点都是一个很小的图，其接口（特殊顶点）上的掩码仍至关重要，因为它们使动态规划能够区分 e 的值是否必须已经在 S_u 或 T_u 处保持有效，而这又会影响是否需要重新计算。

串联节点。串联节点表示将两个 SPL 子图拼接起来，使左侧子图的终止顶点与右侧子图的起始顶点重合。我们来看看连接 $[1, 2]$ and $[2, 3]$ 的串联节点。这里，合并顶点是

$$T_{[1,2]} = S_{[2,3]} = 2,$$

因此，父节点的接口是

$$\Gamma_u = \{1, 3, B_u, C_u\}.$$

同样， B_u 和 C_u 不起作用，但它们是统一接口结构的一部分。

对于每个可能的 $X \subseteq \Gamma_u$，动态规划都会考察来自两个子节点的所有掩码 $X_v \subseteq \Gamma_v$ 和 $X_w \subseteq \Gamma_w$ ，并确保它们满足兼容性约束。兼容性要求合并顶点 2 在这三个掩码中始终同时被包含或排除。因此，即使父节点掩码 X 没有明确提及顶点 2，每个兼容对 (X_v, X_w)也会隐式地考虑合并顶点在子节点接口处是否处于活动状态。

让我们通过详细的求值来说明这一点。假设我们要计算 $\mathtt{dp}([1,3], X)$，其中 $X = \{3\}$表示临时变量必须在顶点 3 处处于活动状态，但在顶点 1 处不处于活动状态。为满足这一要求，动态规划会枚举子节点 $[1,2]$ 和 $[2,3]$的兼容对 (X_v, X_w)。

具体而言，会出现两个这样的兼容对：

- $X_v = \{2\}$ 和 $X_w = \{2,3\}$。这一选择确保合并顶点 2 在两个子节点中都处于活跃状态。组合成本必须减去顶点 2 重复计数的存活状态。

- $X_v = \emptyset$ 和 $X_w = \{3\}$。这对应于不要求顶点 2 处于活跃状态的情况。

评估这两对组合可得：

$$\begin{aligned}\mathtt{dp}([1,3],\{3\}) &= \min\Big\{\mathtt{dp}([1,2],\{2\}) + \mathtt{dp}([2,3],\{2,3\}) - l(2),\ \mathtt{dp}([1,2],\emptyset) + \mathtt{dp}([2,3],\{3\})\Big\} \\ &= \min\{1.1 + 0.2 - 0.1,\ \ 1 + 1.1\} = 1.2.\end{aligned}$$

在完整实现中，动态规划会在此节点评估所有 $2^{|\Gamma_u|} = 4$ 掩码，但上述示例说明了兼容性如何传递有关中间存活状态的信息，以及如何消除顶点存活成本中的重复计数。

并联节点。并联节点用于表示控制流图中的分支。在示例中，两个子图 $[3,4,6]$ 和 $[3,5,6]$ 构成并行分支，并在顶点 6 处重新汇合。因此，组合节点的特殊顶点为 $\{3,6\}$，任何掩码都必须对这些顶点中哪些在两个子节点中均保持活跃达成一致。

例如，考虑 $X = \{3\}$。动态规划只需将两个子节点的动态规划值相加，但如果顶点 3 被重复计数，则要减去其活跃性代价：

$$\texttt{dp}([3,4,5,6],\{3\}) = \texttt{dp}([3,4,6],\{3\}) + \texttt{dp}([3,5,6],\{3\}) - l(3).$$

类似的推理也适用于其他掩码：

$$X \in \{\emptyset, \{3\}, \{6\}, \{3,6\}\}.$$

这些情况都要求在两个子节点上采用一致的容斥处理，并对重复计数的活跃性使用一个修正项。

直观来说，串联节点沿着控制流传播信息，而并联节点跨越分支传播信息，从而确保正确处理对分支敏感的代价。

向根传播。处理完所有内部节点后，动态规划到达代表整个控制流图的根。根包含特殊顶点$\Gamma_r = \{1, 8\}$，用于标记函数的入口和出口。我们计算全部四种掩码：

$$X \in \big\{\emptyset, \{1\}, \{8\}, \{1,8\}\big\}.$$

最小值在 $X = \emptyset$ 处取得，其值为

$$\texttt{dp}(r, \emptyset) = 2.2.$$

这表示临时值在入口顶点和出口顶点处都不需要；在我们的代价模型下，满足所有使用所需插入的计算次数最少为 2。2（加上活跃性代价）。

回溯每个分解步骤中兼容掩码的选择，即可得出相应的最优活跃集合：

$$L = \{2, 3\}.$$

因此，只有让临时变量在顶点 2 和 3 处活跃，才能达到最小成本。其他每个节点要么重新计算该表达式，要么不需要其值，这完全由动态规划确定。

这个扩展的演示说明了 SPL 分解如何通过揭示小型、结构化的接口来指导动态规划，并确保通过局部兼容性约束实现全局最优。

定理 3。 给定一个由闭合结构化程序 P、其控制流图 G （含 n 个顶点）、一个使用集 U、一个失效集 I 和两个代价函数 c： $E \to K$ 及 l： $V \to K$ 构成的 ***LOSPRE*** 实例，上述算法可在 O（n） 内求解 ***LOSPRE*** 问题，并输出

$$\min_{L} \textsc{Cost}(G, U, I, L, c, l) \qquad and \qquad \arg\min_{L} \textsc{Cost}(G, U, I, L, c, l).$$

证明。 正确性已在上文论证。因此，我们将重点放在运行时间分析上。SPL 分解有 $O(n)$ 个顶点，可在 $O(n)$ 时间内计算。对于分解中的每个顶点 u，$2^4 = 16 = O$ 有 (1) 个不同的可能取值。原子节点中的计算针对仅含四个顶点的图进行，因此耗时 $O(1)$。在串联节点和并联节点中，每个 $X.$ 至多有两个兼容的 X_v。这是因为，顶点 S_v、 B_v 和 C_v 是否包含在 X_v 中，由 X 唯一确定，只剩下 T_v 需要选择。类似地，对于每个固定的 X、X_v，都存在唯一的 $X_w.$ 。因此，在这一步中，计算每个 $\mathsf{dp}[u, X]$ 需要 $O(1)$ 时间。在循环节点中，每个 X 都会导出唯一的 X_v 和唯一的 $X_w.$ 。因此，这一步计算每个 $\mathsf{dp}[u, x]$ 值需要 $O(1)$ 时间。在第 3 步中，对于每个 $X.$，我们尝试 $2^4 = O(1)$ 个不同的 X_v。因此，第 2 步的总运行时间为 $O(n)$。最后，第 3 步取 $2^4 = O(1)$ 个值中的最大值。 □

6. 实验结果

本节将提供详尽的实验结果，比较我们用于无溢出寄存器分配和 LOSPRE 的算法与先前基于树宽的方法 [36, 3]。

实现。 我们使用 C语言++ 实现了我们的方法，并将其集成到 Small Device C Compiler (SDCC) [71]中。SDCC 已经包含了对 [17, 33, 36] 中用于寻找树分解的算法以及用于解决这两项编译器优化任务的基于树宽的算法进行高度优化的变体。尽管我们的方法完全可以并联执行，但为了与现有的先前方法实现进行公平比较（这些方法并不支持并联），我们在实验中没有采用并联化。

机器。 实验结果是在一台运行 Oracle Linux（ARM64 位）的机器上获得的，该机器配备 Apple M2 的 1 核 CPU 和 4GB 内存。
基准测试。我们采用了 [29] 中所述的设置。我们使用 HC08 的 SDCC 回归测试套件作为基准测试集。这些基准测试由以下内容组成

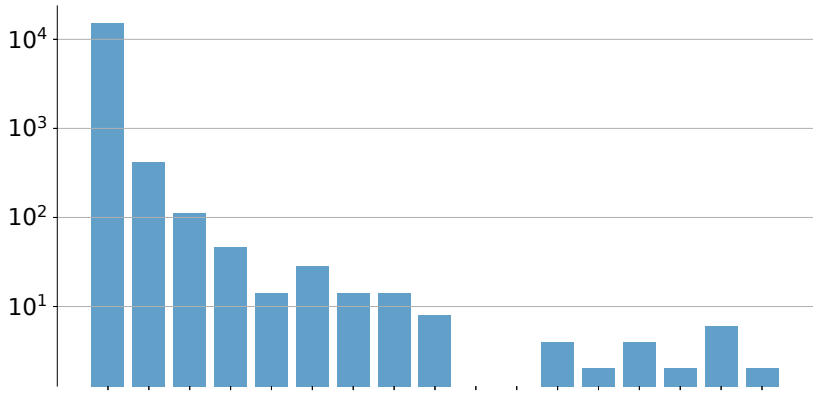


图10：各基准测试中控制流图顶点数（代码行数）的直方图。 x 轴表示控制流图规模，y 轴表示实例数。 y 轴采用对数刻度。

嵌入式程序，旨在有限资源的系统上运行，因此编译器优化是影响其性能的关键瓶颈。这些基准测试中的函数，其控制流图（CFG）规模从 1 到 800 个顶点（代码行）不等，平均规模为 15.7 个顶点。图10展示了函数规模的直方图。我们对每个测试用例运行一次寄存器分配，并对每个表达式运行一次 LOSPRE，因此 LOSPRE 会有更多实例。

计算文法分解的运行时间。 这三项编译器优化任务都依赖于将控制流图（CFG）分解为 SPL 图的文法分解。如上所述，这种分解可以在线性时间内从原始控制流图计算得出。图11展示了为各项基准测试计算文法分解所需的运行时间。平均运行时间为 13.8 微秒，最长为 570 微秒。因此，计算文法分解的效率极高，获取分解所花费的时间对总编译时间的影响并不显著。

寄存器分配。 给定一个输入程序，我们的目标是确定无溢出分配所需的最少寄存器数。如前所述，无溢出寄存器分配是寄存器分配的一种特殊情况，其特点是成本最小；当不存在

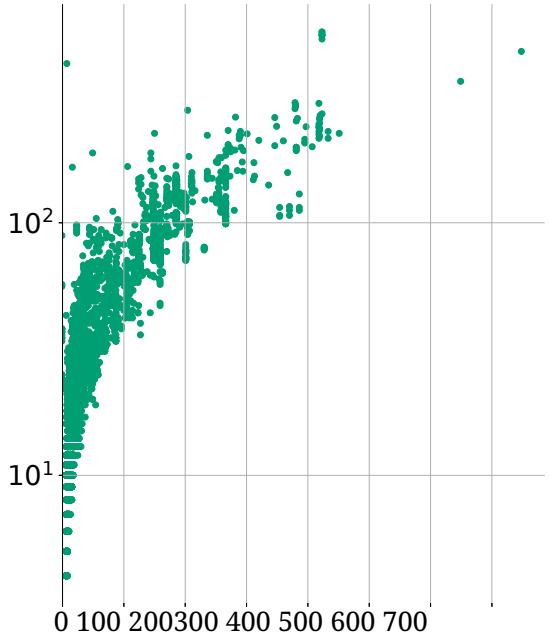


图11：通过对程序进行语法分析来计算控制流图的文法分解所需的运行时间。每个点对应一个实例。 x 轴表示控制流图的大小， y 轴表示运行时间（微秒）。 y 轴采用对数刻度。

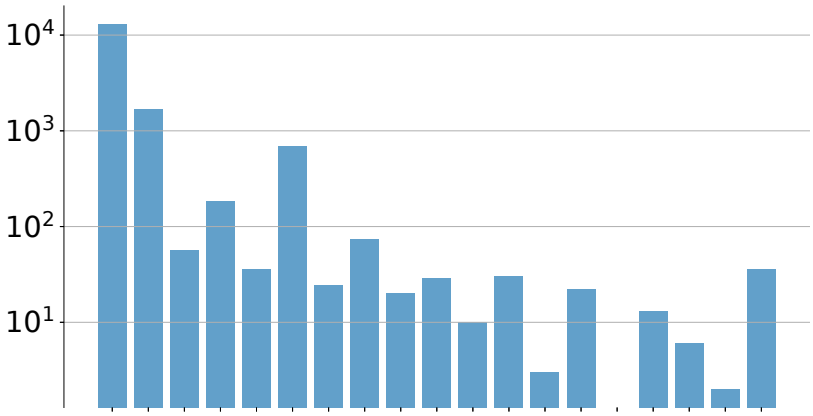
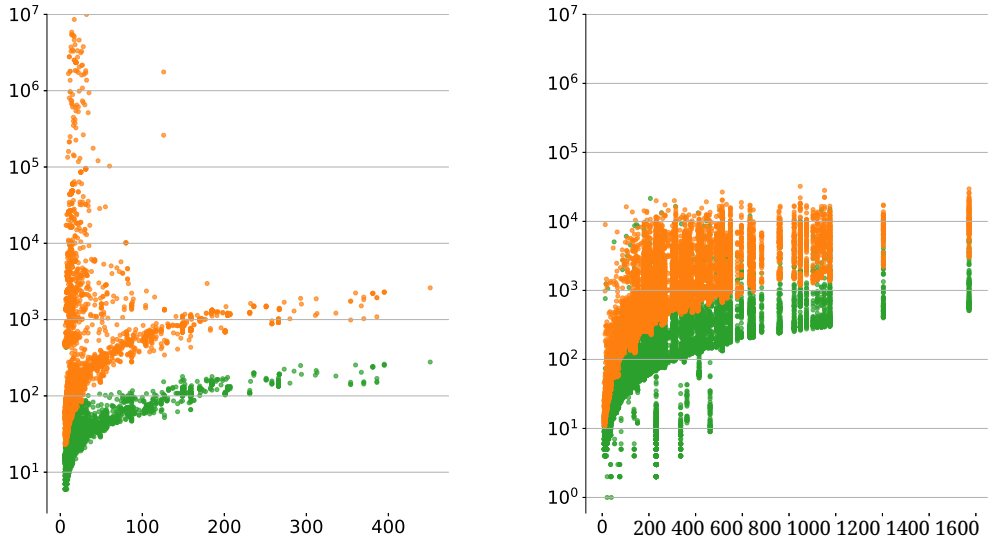


图12：无溢出分配所需最少寄存器数量的直方图。 x 轴表示寄存器数量， y 轴表示需要相应数量寄存器的实例数。 y 轴采用对数刻度。



（a）寄存器分配

（b）LOSPRE

图13：基于树宽的算法（橙色）与我们的方法（绿色）在（a）寄存器分配和（b）LOSPRE中的运行时间比较。 x 轴表示CFG中的顶点数量， y 轴表示（a、b）中的时间，单位为微秒。

溢出，否则为无穷大。我们选择这个问题进行实验评估，原因有二：（i）文献中大多数以往研究都关注这一变体；（ii）通常没有标准方法来选择代价函数 c；每个编译器都会根据自身的上下文和用例以不同方式定义该函数，且通常依赖动态分析和性能分析。相比之下，无溢出分配定义明确，并且在所有编译器中保持一致。

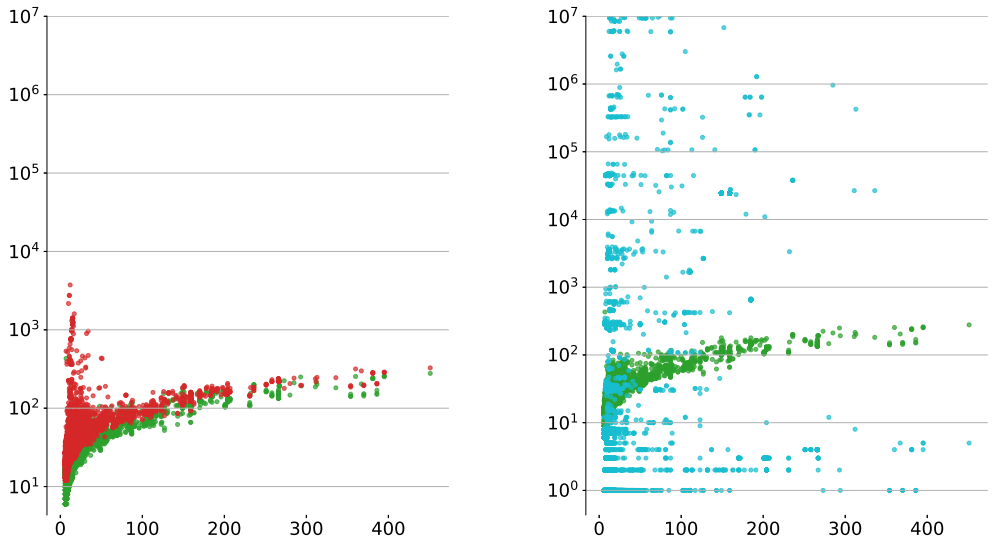
我们的方法成功地在规定的时间和内存限制内处理了所有输入实例，要么找出无溢出分配所需的最优寄存器数量，要么报告所需寄存器数量超过 20 个。图12展示了所需寄存器数量的直方图。相比之下， [36] 的基于树宽的方法在 554 个实例上失败，其中包括所有需要超过 8 个寄存器的实例。

图13（a）比较了在设置 $r \leq 20.$ 时，我们的算法与 [36], 的基于树宽的方法的运行时间。我们的方法平均运行时间为 3.87 微秒，而 [36] 为 1,191,284 微秒。这些平均值未计入以往方法失败的实例。基于树宽的方法的运行时间主要受 704 个实例影响，这可能是由于树宽较大。排除这些离群实例后， [36] 的平均运行时间为 372.34 微秒。

图14展示了我们的方法与 [29] 的路径宽度方法以及 [1]的经典图着色方法在运行时间上的比较。当我们设置 $r \leq 20.$ 时， [29] 的平均运行时间为21,544微秒，而我们的方法的平均运行时间为3.87微秒。这些平均值不包括先前方法无法处理的实例。

我们观察到，Chaitin图着色方法是唯一使用最优数量寄存器的经典非参数化方法，但其可扩展性很差。在1分钟的时限内，它只能处理测试集中的6,042个基准测试，平均运行时间为153,602微秒。值得注意的是，其中不包括任何需要8个以上寄存器的基准测试。图着色方法在所有这类基准测试上都超时了。

与图着色方法不同，两种启发式方法，即线性扫描 [64]和 [72], 的组合方法，在实践中都相当高效。对于前者，所有基准测试的平均运行时间不到1微秒，最长为9微秒。对于后者，平均运行时间为15微秒，最长运行时间为1,363微秒。因此，它们的运行时间与我们的方法相当，线性扫描甚至更快。然而，它们无法保证使用所需的最少数量的



(a) 路径宽度

(b) 图着色

图14：我们的方法与其他寄存器分配算法的运行时间比较。(a) 基于 [29], ，(b) 基于 [1]。横轴表示控制流图中的顶点数，纵轴表示耗时（微秒）。纵轴采用对数刻度。

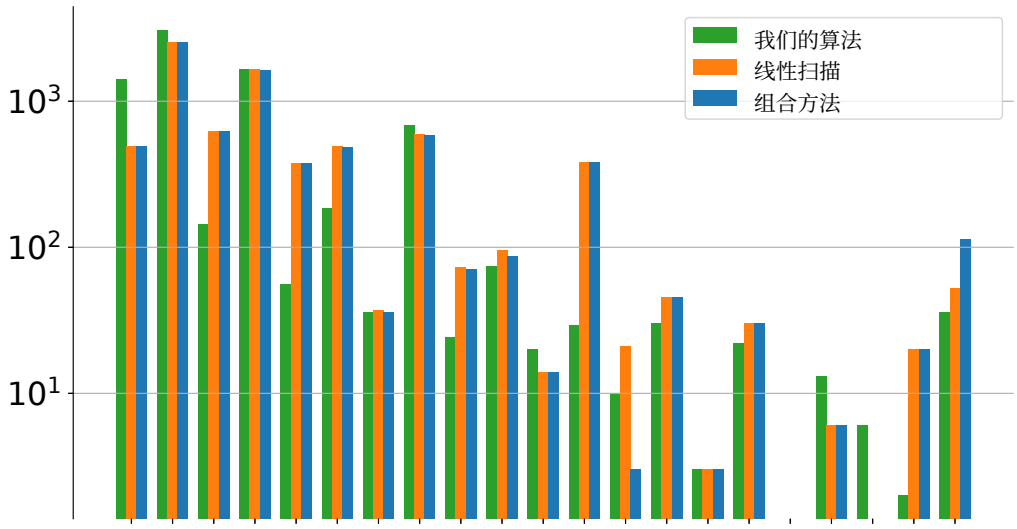


图15：我们的算法、线性扫描 [64]和 [72]的组合方法所使用的寄存器数量直方图。 x 轴表示使用的寄存器数量， y 轴表示实例数量，采用对数刻度。我们的算法始终使用可能的最少寄存器数量。此图不包括无需使用任何寄存器的简单实例。

寄存器，因为它们只是启发式方法，并不附带最优性证明。由于线性扫描和组合方法并不精确，我们根据所使用的寄存器数量，将它们的输出与我们的方法进行比较。图15展示了各方法所使用寄存器数量的直方图。我们观察到，这两种启发式方法通常比我们的最优解使用更多寄存器。事实上，线性扫描有46%的概率找到最优寄存器数量，而组合方法在45%的实例中生成最优结果。

LOSPRE。 在我们的 LOSPRE 实例中，目标是最小化所得三地址码中的计算总数。因此，我们使用 $K = \mathbb{Z}^2$ 并采用字典序。为每条边（$x,\ y$）分配的成本为 c（$x,\ y$）＝（1，0）。我们还通过为每个顶点 $x.$ 分配成本 l（x）＝（0，1）来保证生命周期最优性。由于两种方法都能找到最小成本，我们只比较所用的时间。

图 13 (b) 展示了 [3] 与我们的方法之间的运行时间比较。平均而言，我们的算法耗时 222.38 微秒，而 [3] 基于树宽的方法平均运行时间为 1349.14 微秒。我们的算法最长运行时间为 21,524 微秒，相比之下， [3] 为 32,284 微秒。在绝大多数基准测试中，我们的算法显著优于 [3] 。我们仅发现 19 个运行时间超过 10,000 微秒的实例，而 [3] 在 277 个实例中的运行时间超过 10,000 微秒。

讨论。 在上述所有情况下，我们的算法都优于此前最先进的方法。对于寄存器分配而言，我们的方法是首个能够扩展到最多拥有 20 个寄存器的现实架构（例如 x86 系列架构）的无溢出寄存器分配精确算法。鉴于我们的方法效率很高，每个实例的平均运行时间仅为 4 微秒，我们认为，在无溢出寄存器分配中继续使用近似算法或启发式方法已无必要。尽管该问题是NP难的，且在理论上具有近似困难性，我们的方法仍能高效求解所有实际实例。对于 LOSPRE，我们的方法比基于树宽的算法快约 6 倍。

7. 更广泛的影响与未来方向

在本节中，我们将进一步阐述本工作的广泛概念意义，并探讨 SPL 框架如何揭示编译器优化任务之间的新联系。我们还将 SPL 分解与

经典的基于树宽的技术进行比较，并概述若干具体的未来研究方向。我们的目标是将 SPL 分解置于更广泛的程序分析方法体系中，并阐明这一视角为何有望帮助优化流水线扩展至日益复杂的规模。

7.1. 寄存器分配与 *LOSPRE* 之间的联系

尽管寄存器分配和 LOSPRE 追求不同的目标，但这两个问题都具有一种结构性属性，使其特别适合采用基于分解的推理：依赖局部性。在这两种应用中，程序中某个给定位置的决策都取决于一组有限的变量或状态，而这些变量或状态只通过一个很小的边界相互作用。这种现象也体现在它们经典的图论归约中：寄存器分配归约为对干扰图部分区域进行着色，而 LOSPRE 则归约为在局部数据依赖约束下计算表达式求值的可行放置方案。

一个核心观察是，这两个问题都不需要一次性掌握整个控制流图的全局信息。相反，只需一个简洁的摘要——即边界区域中活跃、活动或以其他方式相关的变量集合——便可在分解过程中传播信息。对于寄存器分配，该边界由活跃变量构成，这些变量的寄存器分配必须在相邻区域之间保持一致。对于 LOSPRE，该边界则反映了表示该表达式的临时值是否必须在某个区域之前或之后保持活跃，以及该区域是否会使此值失效。

这种局部性特征与经典的基于宽度的图问题类似。例如，使用 SPL 风格分解在控制流图上求解 k 着色问题时，只需存储特殊边界节点处的颜色赋值。只要边界规模较小，动态规划就仍然易于处理，从而得到一个关于颜色数量的 FPT算法。同一原理也适用于许多数据流分析，例如可用表达式、到达定义和活跃性传播。这些分析通常会在每个边界处维护一个有限的抽象状态，因此非常适合采用 SPL 风格的推理。

因此，寄存器分配与 LOSPRE 之间的联系表明，SPL 分解为广泛的优化问题提供了统一的视角。具有相似局部性模式的任务，或许可以采用高效算法，在该分解上而非整个控制流图上运行。识别此类任务并将其边界

摘要形式化，是拓展基于分解的优化应用范围的一个令人振奋的方向。

7.2. SPL 分解与树分解的比较

树宽长期以来一直是分析控制流图的重要参数，尤其是因为研究已表明，结构化程序的树宽至多为七。然而，树分解针对无向图进行操作，并将所有边一视同仁，因此无法体现控制流的方向性。相比之下，SPL 分解本质上具有方向性，并且基于生成结构化程序的同一套语法结构。这种与程序语法的契合带来了若干重要优势。

首先，对于结构化程序，SPL 分解始终能得到较小的割集大小——特殊顶点从不超过四个。相比之下，即使对于结构良好的输入，树分解也可能需要包含多达八个顶点的袋。由于动态规划的复杂度通常随割集大小呈指数增长，这种缩减对性能有显著影响。我们的实验印证了这一判断：边界越小，动态规划表就越小，兼容性检查越少，逐节点处理也越快。

其次，SPL 构造忠实地保留了控制流语义。串联节点体现顺序组合；并联节点表示分支与合并；循环节点则准确地刻画迭代结构的行为。这种语义上的忠实性使问题特定优化成为可能，例如沿控制流方向传播信息、利用循环结构，或根据支配关系简化局部状态。由于无法获取这种结构，基于树宽的方法无法如此直接地利用这些特性。

尽管如此，树分解仍然是一种稳健且通用的工具。引入 goto 语句或其他非结构化控制流结构后，控制流图可能无法再用 SPL 文法进行分解。此时，基于树宽的方法可作为后备方案：尽管树宽可能会增加，但实证研究表明，许多真实世界程序在实践中仍表现出相对较小的树宽。理解为何非结构化结构很少导致树宽大幅增长，以及这种现象究竟是根本性的还是偶然的，仍是一个引人入胜的开放问题。

7.3. 未来研究方向

我们的研究成果为未来探索开辟了若干前景广阔的方向。

将 *SPL* 扩展至更广泛的优化任务。许多经典分析——例如可用表达式、到达定义、常量传播和部分死代码消除——都具有与寄存器分配和 LOSPRE 类似的边界摘要。这些摘要通常取决于少量变量或表达式状态，这表明 SPL 分解有望支持高效的动态规划解法。将这些摘要形式化，并开发基于 SPL 的数据流分析通用框架，可以显著扩大这一方法的影响。

处理非结构化控制流。自然的下一步是扩展 SPL 分解，以适应 goto 语句、异常处理以及其他形式的非结构化控制流。一种方法是在 SPL 文法中加入用于描述不可约控制流的特殊结构，或者将此类边视为注释，由动态规划单独处理。另一种可能是结合 SPL 和树分解：找出适用 SPL 的结构化区域，并在不规则区域改用基于树宽的方法。这样的混合分解或许能兼具两者的优势。

图着色之外的应用。一些与寄存器分配或 LOSPRE 并不相似的问题，仍可能受益于较小的分解边界。例如，可以在有界树宽图上高效地进行 μ-演算模型检测及相关模态逻辑的求解。SPL 分解能否带来类似优势——或许通过利用方向性或控制流结构来改进符号传播——是一个值得研究的方向。研究这类问题在 SPL 分解下的固定参数分类和 XP 分类，或许能带来新的复杂性理论见解。

集成到编译器流水线中。最后，在现有编译器基础设施（例如 LLVM、GCC 或领域专用编译器）中将 SPL 分解用作通用优化框架，可能会提升一系列分析的性能。了解 SPL 与静态单赋值形式、寄存器合并、指令调度及其他优化遍的交互方式，是一个重要的实践研究方向。

总体而言，我们的研究指向了在程序分析中推进基于分解的技术的广阔机遇图景。SPL

框架通过将程序结构与图论分解紧密耦合，为构建可扩展且有原则的编译器优化提供了颇具前景的基础。

8. 结论

本文提出了一种基于文法的结构化程序控制流图分解方法，可增强自底向上的动态规划，并为传统树宽方法提供一种计算效率更高的替代方案，从而显著缩短编译器优化任务的运行时间。我们应用这一分解方法，改进了无溢出寄存器分配和最小成本寄存器分配的渐近运行时间，实现了与变量和寄存器数量相关的指数级改进。此外，我们为无 goto 程序开发了生命周期最优推测式部分冗余消除（LOSPRE）算法。该算法利用串联-并联-循环（SPL）分解来发挥图稀疏性的优势，使 Small Device C Compiler 基准测试的运行时间提升了 6 倍。我们的方法有望显著缩短各种编译器优化任务、程序验证和模型检测的运行时间，但目前仅适用于无 goto 程序。未来的工作将致力于扩展 SPL 分解，使其涵盖 goto 语句，并处理带有异常处理或带标签循环的程序。

致谢与注释

本工作是对会议论文 [73, 74]的扩展。研究部分由 ERC Starting Grant 101222524 (SPES) 和以太坊基金会研究资助 FY24-1793 资助。Chun Kit Lam 获得了香港博士奖学金资助。作者按字母顺序排列。

参考文献

[1] G. J. Chaitin，通过图着色进行寄存器分配与溢出（附回顾），载于：Best of PLDI，1982，第 66–74 页。

[2] F. Bouchez、A. Darte、C. Guillon、F. Rastello，寄存器分配：Chaitin等人的NP 完全性证明究竟证明了什么？或重访寄存器分配：原因与方法，载于：LCPC，2006年，第283–298页。

[3] P. K. Krause，线性时间内的LOSPRE，载于：SCOPES，2021，第35–41页。[4] K. O. Thabit，编译器的缓存管理，莱斯大学，1982。[5] B. Calder、C. Krintz、S. John、T. M. Austin，缓存感知数据放置，载于：ASPLOS，ACM，1998，第139–149页。[6] M. O. Beg、P. van Beek，面向直接映射缓存的缓存感知数据放置的图论方法，载于：ISMM，ACM，2010，第113–120页。[7] T. W. Reps、S. Horwitz、S. Sagiv，通过图可达性进行精确的过程间数据流分析，载于：POPL，ACM，1995，第49–61页。[8] Z. Kincaid、T. W. Reps、J. Cyphert，代数程序分析，载于：CAV，第12759卷，Springer，2021，第46–83页。[9] M. Willsey、C. Nandi、Y. R. Wang等，egg：快速且可扩展的等式饱和，ACM编程语言汇刊（2021）。[10] A. K. Goharshady、C. K. Lam、L. Parreaux，稀疏等式图的快速最优提取，ACM编程语言汇刊（2024）2551–2577。[11] J. Obdrzálek，树宽有界时的快速μ演算模型检查，载于：CAV，第2725卷，Springer，2003，第80–92页。[12] R. Lavaee，数据打包的困难性，载于：POPL，ACM，2016，第232–242页。[13] E. Petrank、D. Rawitz，缓存感知数据放置的困难性，载于：POPL，ACM，2002，第101–112页。[14] E. Petrank、D. Rawitz，缓存感知数据放置的困难性，《北欧计算机杂志》12（2005）275–307。[15] J. Ferrante、K. J. Ottenstein、J. D. Warren，程序依赖图及其在优化中的应用，《ACM程序设计语言与系统汇刊》9（1987）319–349。[16] P. M. Gharat、U. P. Khedker、A. Mycroft，广义指向图：面向指向分析的精确且可扩展抽象，《ACM程序设计语言与系统汇刊》42（2020）8:1–8:78。

[17] M. Thorup, 所有结构化程序都具有较小的树宽和良好的寄存器分配, Information and Computation 142 (1998) 159–181. doi:https://doi.org/10.1006/inco.1997.2697.[18] N. Robertson, P. D. Seymour, 图小面 III：平面树宽, J. Comb. Theory, Ser. B 36 (1984) 49–64.[19] H. L. Bodlaender, 发现树宽, 见：SOFSEM, 第3381卷, Springer, 2005, 第1–16页.[20]M. Cygan, F. V. Fomin, Ł. Kowalik, D. Loks htanov, D. Marx, M. Pilipczuk, M. Pilipczuk, S. Saurabh, 参数化算法, 第5卷, Springer, 2015.[21] N. Robertson, P. D. Seymour, 图小面 II：树宽的算法方面, J. Algorithms 7 (1986) 309–322.[22] H. L. Bodlaender, 在有界树宽的图上进行动态规划, 见：ICALP, 第317卷, Springer, 1988, 第105–118页.[23] A. K. Goharshady, M. R. Hooshmandasl, M. A. Meybodi, 树中的 $[1, 2]$集与$[1, 2]$总集及其算法, Discret. Appl. Math. 198 (2016) 136–146.[24] A. K. Goharshady, F. Mohammadi, 一种高效计算小树宽网络可靠性的算法, Reliab. Eng. Syst. Saf. 193 (2020) 106665.[25] M. A. Meybodi, A. K. Goharshady, M. R. Hooshmandasl, A. Shakiba, 最优挖矿：最大化比特币矿工的交易手续费收入, 见：区块链, 2022, 第266–273页.[26] A. Ahmadi, K. Chatterjee, A. K. Goharshady, T. Meggendorfer, R. Safavi, Đ. Zikelic, 计算马尔可夫链核的算法与难度结果, 见：FSTTCS, 2022, 第29:1–29:20页.[27]T. Barakbayeva, S. Farokhnia, A. K. Goharshady, M. Gufler, S. Novozhilov, Pixiu：卡尔达诺上的最优区块生产收益, 见：区块链, 2024, 第491–496页。

[28] M. A. Meybodi, A. K. Goharshady, M. R. Hooshmandasl, A. Shakiba, 实现比特币矿工交易费收入最大化的最优区块, 组合优化期刊 49 (2025) 15.[29] G. K. Conrado, A. K. Goharshady, C. K. Lam, 控制流图的有界路径宽, ACM编程语言汇刊 7 (2023) 292–317.[30] J. Gustedt, O. A. Mæhle, J. A. Telle, Java程序的树宽, 见：ALENEX, 2002, 第86–97页.[31] B. Burgstaller, J. Blieberger, B. Scholz, Ada程序的树宽, 见：Ada-Europe, 第3063卷, Springer, 2004, 第78–90页.[32] K. Chatterjee, A. K. Goharshady, E. K. Goharshady, 智能合约的树宽, 见：SAC, ACM, 2019, 第400–408页.[33] P. K. Krause, L. Larisch, F. Salfelder, C语言的树宽, Discret. Appl. Math. 278 (2020) 136–152.[34] R. G. Downey, M. R. Fellows, 参数化复杂性, Springer, 2012.[35] H. L. Bodlaender, J. Gustedt, J. A. Telle, 针对固定数量寄存器的线性时间寄存器分配, 见：SODA, ACM/SIAM, 1998, 第574–583页.[36] P. K. Krause, 多项式时间内的最优寄存器分配, 见：CC, 第7791卷, 2013, 第1–20页.[37] B. Courcelle, 图的一元二阶逻辑 I：有限图的可识别集, 信息与计算 85 (1990) 12–75.[38] R. B. Borie, R. G. Parker, C. A. Tovey, 根据谓词演算描述自动生成适用于递归构造图族问题的线性时间算法, Algorithmica 7 (1992) 555–581.[39] M. Chimes, R. Iosif, F. Zuleger, 树可验证图文法, 见：LPAR, 第100卷, 2024, 第165–180页.[40] K. Chatterjee, A. K. Goharshady, N. Okati, A. Pavlogiannis, 用于数据打包的高效参数化算法, ACM编程语言汇刊 3 (2019) 53:1–53:28。

[41] A. Ahmadi, M. Daliri, A. K. Goharshady, A. Pavlogiannis，《缓存感知数据放置的高效近似算法》，载于：PLDI，2022年，第857–871页。[42] K. Chatterjee, A. K. Goharshady, R. Ibsen-Jensen, A. Pavlogiannis，《常数树宽分量并发系统中代数路径性质的算法》，载于：POPL，ACM，2016年，第733–747页。[43] K. Chatterjee, A. K. Goharshady, R. Ibsen-Jensen, A. Pavlogiannis，《用于按需数据流分析的最优且完全并联的算法》，载于：欧洲编程研讨会，第12075卷，Springer，2020年，第112–140页。[44] K. Chatterjee, A. K. Goharshady, A. Pavlogiannis，《Jtdec：一种用于 Soot 中树分解的工具》，载于：自动化技术：理论与应用，第10482卷，2017年，第59–66页。[45] K. Chatterjee, A. K. Goharshady, P. Goyal, R. Ibsen-Jensen, A. Pavlogiannis，《常数树宽基本递归状态机中动态代数查询的更快算法》，《ACM程序设计语言与系统汇刊》41（2019）23:1–23:46。[46] S. Horwitz, T. W. Reps, S. Sagiv，《按需过程间数据流分析》，载于：软件工程基础研讨会，ACM，1995年，第104–115页。[47] G. K. Conrado, A. K. Goharshady, K. Kochekov, Y. C. Tsai, A. K. Zaher，《利用控制流图和调用图的稀疏性进行高效按需代数程序分析》，ACM编程语言汇刊 7（2023）1993–2022。[48] F. V. Fomin, D. Lokshtanov, S. Saurabh, M. Pilipczuk, M. Wrochna，《针对低树宽图和矩阵的完全多项式时间参数化计算》，ACM《算法汇刊》（TALG）14（2018）1–45。[49] K. Chatterjee, J. Lacki，《低树宽马尔可夫决策过程的更快算法》，载于：CAV，2013年，第543–558页。[50] A. Asadi, K. Chatterjee, A. K. Goharshady, K. Mohammadi, A. Pavlogiannis，《小树宽马尔可夫链和马尔可夫决策过程定量分析的更快算法》，载于：自动化技术：理论与应用，第12302卷，2020年，第253–270页。

[51] S. Dong, Y. T. Lee, G. Ye, 《一种用于小树宽线性规划的近线性时间算法：鲁棒中心路径的多尺度表示》，载于：STOC，ACM，2021年，第1784–1797页。[52] K. Takamizawa, T. Nishizeki, N. Saito, 《串并联图上组合问题的线性时间可计算性》，ACM 会刊（JACM）29（1982）623–641。[53] N. Wirth, 《论良结构程序的组合》，ACM 计算调查（CSUR）6（1974）247–259。[54] K. Kennedy, L. Zucconi, 《图文法在程序控制流分析中的应用》，载于：第四届 ACM SIGACT-SIGPLAN 编程语言原理研讨会论文集，1977年，第72–85页。[55] A. K. Goharshady, S. Hitarth, F. Mohammadi, H. J. Motwani, 《用于多项式程序基于模板综合的代数几何算法》，ACM编程语言汇刊（2023）。网址：https://doi.org/10.1145/3586052。[56] A. K. Goharshady, C. K. Lam, L. Parreaux, 《稀疏等式图的快速最优提取》，ACM编程语言汇刊（2024）。doi:10.1145/3689801。[57] R. G. Downey, M. R. Fellows 等, 《参数化复杂性基础》，Springer，2013年。[58] S. Khuller, 《算法专栏：顶点覆盖问题》，ACM SIGACT 新闻 33（2002）31–33。[59] N. Robertson, P. D. Seymour, 《图小面 II：树宽的算法方面》，算法杂志 7（1986）309–322。[60] A. K. Goharshady, A. K. Zaher, 《利用树深度和树宽进行高效过程间数据流分析》，载于：验证、模型检验与抽象解释，Springer，2023年，第177–202页。[61] G. K. Conrado, A. K. Goharshady, P. Hudec, P. Li, H. J. Motwani, 《维纳指数的更快速树宽近似算法》，载于：SEA’24，第301卷，2024年。doi:10.4230/LIPIcs.SEA.2024.6。

[62] J. Valdes、R. E. Tarjan、E. L. Lawler，串并联有向图的识别，载于：STOC，ACM，1979，第 1–12 页。[63] I. Sakai，通用翻译中的句法，载于：《国际机器翻译与应用语言分析会议论文集》，1961。[64] M. Poletto、V. Sarkar，线性扫描寄存器分配，TOPLAS 21（1999）895–913。[65]H. C. Grossman，寄存器分配示例，2013。URL：https://www.cs.clemson.edu/course/cpsc827/material/Optimization/Register%20Allocation%20Example.pdf。[66]J. Cocke，全局公共子表达式消除（1970）20–24。doi：https://doi.org/10.1145/390013.808480。[67] E. Morel、C. Renvoise，通过消除部分冗余实现全局优化，Commun. ACM 22（1979）96–103。[68] J. Knoop、O. Rüthing、B. Steffen，惰性代码移动，载于：PLDI，1992，第 224–234 页。[69] Q. Cai、J. Xue，基于推测的最优高效部分冗余消除，CGO，2003，第 91–102 页。[70] R. Gupta、D. Berson、J. Fang，使用推测的路径剖面引导部分冗余消除，载于：1998 年国际计算机语言会议论文集（目录号 98CB36225），1998，第 230–239 页。[71] S. Dutta、D. Drotos、K. Vigor 等，小型设备 C 编译器，2003。URL：http://sdcc.sourceforge.net/。[72] R. C. Lozano、M. Carlsson、G. H. Blindell、C. Schulte，组合寄存器分配与指令调度，TOPLAS 41（2019）1–53。[73] X. Cai、A. K. Goharshady、S. Hitarth、C. K. Lam，通过控制流图的文法分解实现更快的 Chaitin 式寄存器分配，载于：ASPLOS，2025，第 463–477 页。

[74] X. Cai, A. K. Goharshady, 《无 goto 程序的更快速生命周期最优推测式部分冗余消除》，载于：SETTA，第 15469 卷，2024 年，第 382–398 页。