arXiv:2603.09726v1 [cs.PL] 10 Mar 2026

幂等切片及其在代码尺寸缩减中的应用

Rafael Alvarenga de Azevedo 米纳斯吉拉斯联邦大学，贝洛奥里藏特，巴西rafael.alvarenga@dcc.ufmg.br

Daniel Augusto Costa deSa 米纳斯吉拉斯联邦大学，贝洛奥里藏特，巴西danielaugusto191@mat-comp.grad.ufmg.br

RodrigoCaetano Rocha 华为，爱丁堡，英国rodrigo.rocha@huawei.com

Fernando Magno Quint˜ao Pereira 米纳斯吉拉斯联邦大学，贝洛奥里藏特，巴西fernando@dcc.ufmg.br

2026年3月11日

摘要

给定程序中计算出的一个值，相对于该值的幂等后向切片是计算该值的最大子程序。Guimar˜aes 等人此前曾使用幂等切片的非正式概念，将 LLVM中间表示中的急切求值转换为严格求值。然而，该算法不足以正确应用于一般控制流图。本文通过形式化定义幂等后向切片，并提出一种从门控静态单赋值（GSA）形式的程序中提取此类切片的可靠且高效的算法，弥补了上述不足。作为其实际用途的示例，本文说明了如何通过识别和提取幂等后向切片来实现稀疏代码尺寸缩减优化；也就是说，这种优化能够合并单个函数内或跨函数的控制流图中不连续的指令序列。对 LLVM 测试套件的实验表明，对于经 clang 17 的 -Os优化遍次序列高度优化的程序，在特定基准测试中，这种新算法可将代码尺寸最多缩减 -7.24%。

关键词： 幂等切片、控制流图、代码尺寸缩减

1 引言

最近，Guimar˜aes 和 Pereira [10] 提出了一种将按值调用求值转换为按需调用的技术。他们以反向方式计算函数参数 v的依赖关系，将相应的子程序提取为一个独立函数，并在被调用函数中每当使用 v 时调用这个新函数。在本文中，我们将这样的子程序称为以 v 为准则的幂等后向切片。幂等后向切片的应用不止于“惰性化”。例如，提取此类切片可以实现冷热代码拆分、揭示任务级并行、作为代码混淆的机制，并支持代码尺寸缩减，因为可将反复出现的切片合并到一个函数中。

我们发现，Guimar˜aes 和 Pereira 的算法至少在两种情况下无法识别切片。首先，正如作者自己已经指出的那样，他们的方法可能无法从不满足“传统”静态单赋值（CSSA）性质的程序中提取切片 [20]，，例如含有由 ϕ-函数关联且活跃区间重叠的变量的程序。其次，如第 2.4 节所示，对于某些不具有“吊床”结构的控制流图，该算法可能无法提取切片 [9]；也就是说，这些图无法分解为单入口单出口区域构成的树。本文提出了一种算法，能够在这两种情况下识别并提取幂等后向切片。

可靠的幂等后向切片。 在第 3 节中，我们形式化幂等后向切片的概念，并引入一种可靠算法，将它们提取为独立函数。可靠性的关键在于如下观察：可以从处于门控静态单赋值 （GSA）形式的程序中提取正确的幂等后向切片，前提是使用由

Tu 和 Padua [22] 提出的路径编号算法来构建 GSA。单个切片的识别相对于程序控制流图中的边数以线性时间运行，并且既不需要吊床结构，也不需要 CSSA 属性。

作为幂等后向切片的一种应用，第 4 节展示了如何利用它们实现代码尺寸缩减。该优化的思路是尽可能多地识别切片，提取其中同构的切片，并将它们合并为一个函数，从而减少代码重复。与 Rocha 等人 [15] 基于序列比对的函数合并或 LLVM IROutliner pass [14], 等先前工作不同，所提优化能够合并不连续或顺序不一致的指令，也能合并同一函数内的指令组。

贡献总结。 我们在 LLVM 编译器 17.0.6 版中实现了第 3 节所述的切片算法。该实现足够健壮，能够处理整个 LLVM 测试套件，其中包含 2007个程序。第 5 节中的实验表明，基于该实现的代码尺寸缩减算法可使 29 个程序的 .text 段大小缩减，几何平均值为 -7.24%。这些改进并未被任何公开可用的代码尺寸缩减技术所涵盖，例如 Rocha 等人 [15]的函数合并或 LLVM IROutliner。反过来，我们的实现也未涵盖这些技术；它们彼此互补，各自实现了独特的代码尺寸缩减。例如，我们的实现在 AMGmk 基准测试上取得了最大收益，在 clang -Os 的基础上将 .text 段大小缩减了 -12.49%。相比之下，Rocha 等人的方法实现了 -5.59% 的代码大小差异，而 LLVM 提取器仅实现了 -0.94%。

2 稀疏幂等切片计算面临的挑战

本节有两个目标：首先，介绍本文中使用的核心定义；其次，探讨以往工作为何在某些情况下无法正确识别幂等后向切片。

2.1 幂等执行

本文关注实现“幂等执行”的指令序列。定义 1 对这一概念进行了形式化，示例 1 则结合 LLVM 编译基础设施进行了说明：

定义 1（幂等执行）。设 P 是一个采用静态单赋值（SSA）形式的程序，S 是 P 的一组指令。设 I 是在 S 中使用但在 S 外部定义的变量集合（即 S 的输入）。如果对于任意赋值给 I 中变量的值，重复执行 S 中的指令都会使 S 中定义的所有变量得到相同的值，且不会改变程序的可观察状态，则 S 的执行是幂等的。

示例 1. 图 1 展示了几组 LLVM 指令序列，它们或符合幂等执行的概念，或违反该概念。图 1(a) 中的序列是幂等的，因为它只包含纯算术运算：在自由变量绑定相同的情况下，多次执行这些指令总会产生相同的值，且不会修改程序状态。图 1(b) 中的序列不是幂等的，因为它包含一条存储指令，会修改内存。因此，重新执行该序列可能会改变程序的可观察状态。图 1(c) 中的片段也不是幂等的，因为除法指令可能引发异常（例如，%c 为零时）。可能引发异常的指令无法安全地重新执行，否则可能改变程序的控制流。最后，图 1(d) 中的序列包含一条加载指令，但仍可视为幂等，因为加载的值来自一个常量全局变量。由于内存位置 @K 不可变，重复执行该序列总会读取相同的值，且不会改变程序状态。

第 5 节评估了一种编译器编译遍的实现，该编译遍用于识别构成程序切片的幂等指令序列；我们在第 2.2 节对程序切片这一概念进行了形式化定义。为识别此类序列，我们的实现采用尽力而为方法。具体而言，我们会将可能引发异常、调用函数或写入内存的指令排除在幂等集合之外。我们还会排除可能从可变内存位置读取数据的加载指令。不过，我们允许从不可变内存（例如常量全局变量）读取数据的加载指令，如图 1(d) 所示。

```
01 %x1 = 加法 i32 %a, %b
02 %x2 = 乘法 i32 %x1, 4
03 %x3 = 加法 i32 %x2, %c
```
(a)

```
01 %x1 = 加法 i32 %a, %b
02 存储 i32 %x1, ptr %p
03 %x2 = 加载 i32, ptr %p
```
(b)

```
01 %x1 = 加法 i32 %a, %b
02 %x2 = 有符号除法 i32 %x1, %c
03 %x3 = 加法 i32 %x2, 1
```
(c)

```
01 %x1 = 加载 i32, ptr @K
02 %x2 = 加法 i32 %x1, %a
03 %x3 = 乘法 i32 %x2, 2
```
(d)

图 1：LLVM 中间表示中幂等与非幂等指令序列的示例。

2.2 幂等后向切片

程序切片是程序语句的一个子集，这些语句可能会影响程序中的某个特定位置，该位置称为切片准则。Weiser 提出的早期程序切片应用 [24], 旨在帮助开发者在调试和测试等任务中专注于相关代码。后来，切片的概念被证明有助于编译器优化，例如自动并行化 [12], 、冗余消除 [21], ，以及从关键路径中移除冷代码 [23]。

Blazy 等人 [3], 提出的程序切片标准形式化所指的是稠密切片。给定程序 P，稠密切片是一个子程序 $S \subseteq P$，用于确定指令$\iota \in P$的计算。相比之下，Rodrigues 等人 [17], 提出的稀疏切片捕获以静态单赋值（*SSA*） 形式 [6]表示的程序中某个值的计算。本文关注稀疏切片的一种变体，我们称之为幂等切片，定义如下：

定义 2（幂等后向切片）。设 v 是 SSA 形式程序 P中定义的一个变量。关于切片准则 v的幂等后向切片 $S \subseteq P$ 是 P 中满足以下性质的指令子集：

单入口： 包含 S 中指令的基本块集合 $B_S \subseteq P$ 在P中构成一个单入口区域；也就是说，存在唯一的基本块 $b \in B_S$，使得从 P 的入口到 B_S 中任意基本块的每条路径都必须首先经过 b。

幂等性： I 表示 S （其输入）中的自由变量集合。对于 I 中的变量，只要其值绑定相同，多次执行 S 总会得到相同的结果（见定义 1）。

示例 2 展示了定义 2 中引入的一些概念。该示例提到了门控静态单赋值形式，第 2.3 节将对此作更详细的回顾。

示例 2。我们使用图 2(a) 中的程序来说明幂等切片的概念。该程序的语义与下文的讨论无关。图 2(b) 改编自 Blazy 等人工作的图 2(e)，展示了针对第 10 行中指令 $s = s + 1$ 计算得到的稠密切片。图 2(c) 展示了同一程序转换为 SSA 形式后的控制流图。图 2(d) 展示了对应的门控静态单赋值表示，其中循环头处的 ϕ函数被替换为 μ函数。此外，一个 η函数会在变量 s_1 退出循环时将其重命名为 s_4 。请注意， η函数受谓词 p_0 守护，该谓词控制着循环。图 2(e) 展示了相对于变量 s_3的后向幂等切片，图 2(f) 则展示了相对于准则 x_2的切片。图 2(e) 中 s_3的值完全取决于输入集合 $\{N, x_0, s_0\}$。请注意， s_3 在循环外定义；因此，无论该循环的迭代次数如何， s_3 都只会被赋值一次。类似地，图 2(f) 中x_2 的值仅取决于输入 x_0 和 x_2的初始绑定。请注意，在这种情况下，后向切片不会越出包含该切片准则的循环，否则 x_2 会受到多次影响。

示例 2 突显了经典（即稠密）后向切片概念与本文提出的幂等后向切片之间一个重要的区别。Weiser [24] 在原始研究中定义、并由 Blazy 等人 [3], 形式化的稠密后向切片，是程序中标签的一个子集，它对切片准则的影响与原始程序相同。相比之下，幂等切片是一个引用透明函数，用于定义单个值。因此，最大后向切片必须局限于定义其切片准则的循环中；否则，它会为该准则计算出多个值：每次循环迭代对应一个值。基于这个原因，图 2(f) 中 x_2 的切片仅包含两个基本块，且不会超出定义 x_2 的循环。

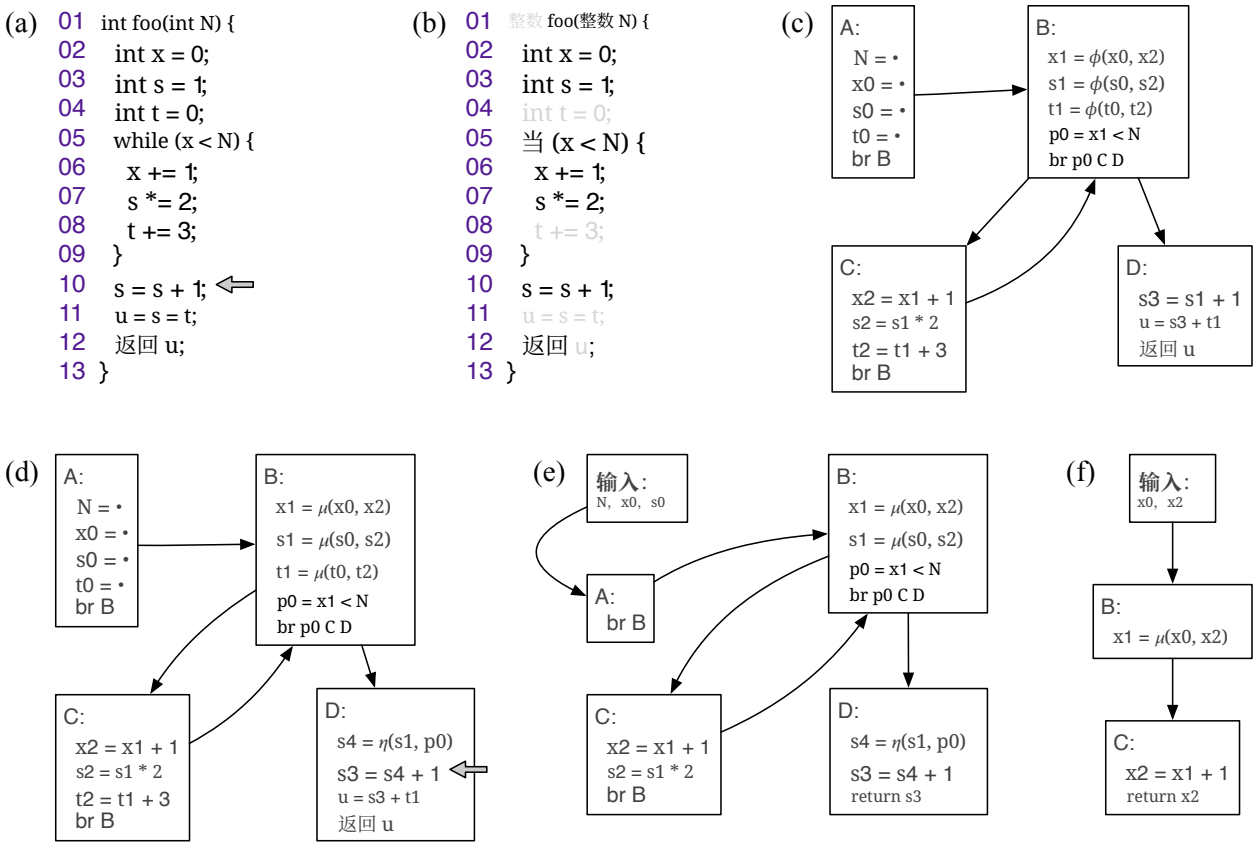


图2: （a）带有切片准则的程序：指令 $s = s + 1$。（b）稠密切片，改编自 Blazy 等人 [3, 图 1]。（c）静态单赋值形式的程序。（d）门控静态单赋值形式的程序。（e）相对于 SSA变量 s_3 的幂等后向切片。（f）相对于 SSA变量 x_2 的幂等后向切片。

2.3 门控静态单赋值形式

示例 2 中提到的门控静态单赋值形式 [22, 13]，通过显式表示控制程序中值流动的控制条件，扩展了传统的静态单赋值表示 [6]。在 SSA 中，ϕ 函数会合并来自多个前驱的值，但不会包含用于确定运行时选择哪个值的控制谓词。GSA 用三类门指令——γ、μ 和 η——替换这些 ϕ 函数，从而共同编码数据依赖和控制依赖：

- **μ-指令：** 以 GSA形式表示循环头。当汇合点对应于循环入口时，它们会用 ϕ-函数替代。A μ-指令通过将每个输入值与启用相应循环回边的控制谓词关联起来，表示跨迭代的值递归。因此，μ-指令明确表示控制迭代结构中值流的控制依赖。

- **γ-指令：** 表示简单连接点。它们通过将每个输入值与一个门配对，推广了 ϕ-函数的概念。该门是一个谓词，用于刻画从到达定义到合并点的控制流路径。在运行时，会选择谓词求值为真的参数。因此，γ-指令比 ϕ-函数编码了更多信息，因为它们明确表示了控制依赖和数据依赖。

- **η-指令：** 表示值门控。它们仅在特定控制条件下传播值，将值的活跃性与控制谓词关联起来。在图2 (d) 中，η-指令确保这些值

在循环或条件区域内产生后，在这些区域之外使用或返回之前都受到适当的守卫。

GSA 表示支持需要联合推理数据流和控制流的分析与变换。正如第 3 节将要讨论的，其中一项应用是计算幂等后向切片。

2.4 计算幂等切片的先前工作的局限性

2023 年，Guimar˜aes 和 Pereira [10]提出了一种用于识别后向切片的快速算法。他们的目标是将函数实参转换为惰性求值对象；也就是说，将 λ 表达式延迟求值，仅在被调用函数需要时才求值。为了生成一个编码实参 p 计算过程的惰性求值对象，Guimar˜aes 和 Pereira 概述了计算 p 值的后向切片。为高效识别这些切片，他们遍历生成值 p 的稀疏 依赖图。为完整起见，定义 3 对稀疏后向依赖图的概念进行了形式化。

定义 3（稀疏后向依赖图）。 相对于 SSA变量 v 、程序 P，依赖图 $G = (V, E)$ 定义如下：

基准： 变量 $v \in V$；

数据： 若 B： $v_0 = v_1 \oplus v_2$ 是基本块 $B \in P$ 中的一条指令，且 $v_0 \in V$，则 $\{v_1, v_2\} \subset V$，并且 $\{v_1 \to v_0, v_2 \to v_0\} \subseteq E$；

控制： 若 B ： 分支 p B_0 B_1 是终止基本块 $B \in P$ 的分支，则对于基本块 $B' \in P$ 中的每条指令B'： $v_0 = \ldots$，满足：(i) B 支配 B'；(ii) B' 不支配 B；且 (iii) $v_0 \in V$，我们有 $p \in V$ 和 $p \to v_0 \in E$。

Guimar˜aes 和 Pereira 使用 Rodrigues 等人提出的稀疏方法 [17] 来计算定义 3 的反向依赖图。该算法又基于 Ferrante 等人提出的控制依赖定义 [9]: 若基本块 B_s 控制 基本块 B_u ，则 B_s 支配 B_u，但 B_u 不后支配 B_s。基于这一观察，Rodrigues 等人使用图 3 (a) 中的四条规则来推断数据依赖和控制依赖。Rodrigues 等人的算法遍历程序的支配树，并将控制分支的谓词压入栈中。访问定义新值的指令时，它会在栈顶谓词与这些值之间建立控制依赖。然而，当这一最初针对程序标签定义的概念被转换应用于 SSA 形式变量时，Rodrigues 等人的算法会遗漏某些控制依赖边。在我们重新实现 Guimar˜aes 和 Pereira 的方法时，这些缺失的边对于正确捕获提取的切片中的所有分支至关重要，如示例 3 所示。

示例 3. 图 3 (b) 展示了一个简化的控制流模式，该模式源自 LLVM 测试套件中 minisat 基准测试的函数 reduceDB 在使用 -Os 标志中的 simplifycfg 优化遍优化后[1] 的编译结果。图 3 (c) 展示了 Rodrigues 等人提出的算法针对准则 x3 构建的反向依赖图 [17] 。在生成 x3 的切片时，Guimar˜ães 和 Pereira 的算法会漏掉基本块 B2。这是因为，根据 Ferrante 等人 [9]对控制依赖的定义，B2 不控制 B3 或 B4，也不控制 B5，因为它不支配这些节点中的任何一个。因此，遵循这种控制依赖定义时，Rodrigues 等人的算法会漏掉谓词 p2 也控制对 x0 和 x1 的赋值这一事实。

图 3 (b) 所展示的模式非常特殊：在 LLVM 测试套件的 2007个程序中，我们仅识别到一次。尽管如此，它的出现会导致按照 Rodrigues 等人的算法错误地计算后向依赖。第 3 节将提出一种新算法，用于计算在此情形下仍然正确的稀疏反向切片。

[1]可在 https://github.com/llvm/llvm-test-suite/blob/main/MultiSource/Applications/minisat/Solver.cpp获取

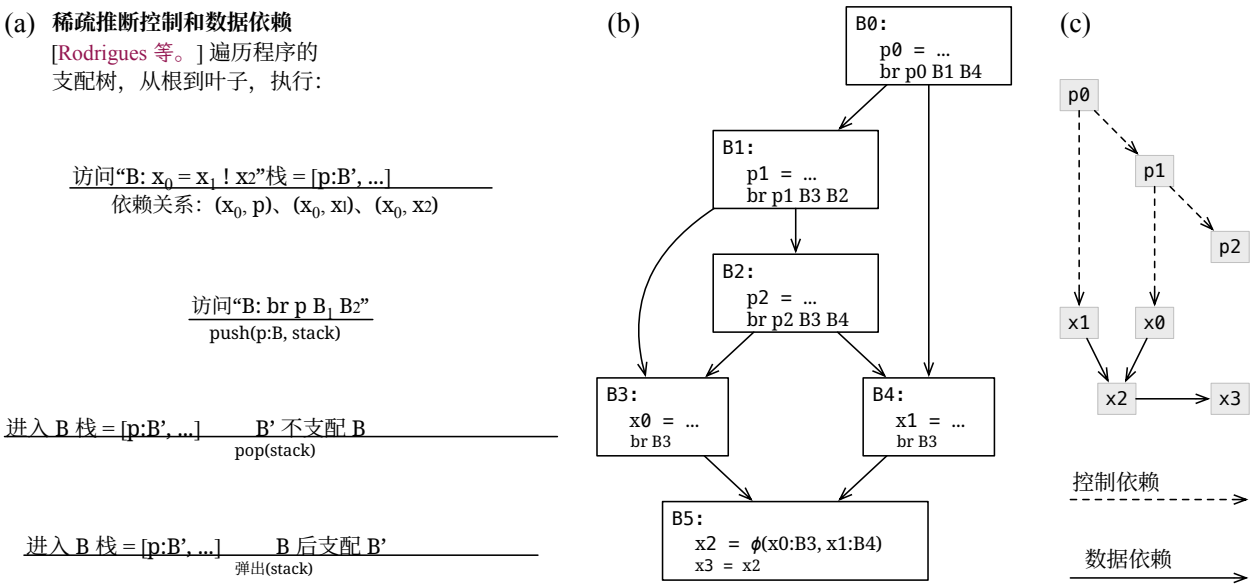


图 3：（a）Rodrigues 等人用于推断数据依赖和控制依赖的四条规则。[17]（b）一个控制流图，其中的子结构会导致 Guimar˜aes 和 Pereira [10] 的算法在针对准则 x_3 的后向切片中漏掉基本块 B_2。（c）为切片准则 x_3 构建的反向依赖图。

3 幂等后向切片

本节提出一种算法，用于识别满足定义 2 所列性质、并克服第 2.4 节所讨论局限性的幂等后向切片。我们首先在第 3.2 节说明如何正确识别此类切片。随后，在第 4.1 节中，我们展示如何将该切片外提为一个独立函数。

3.1 从 SSA 到 GSA

为了构建 GSA 表示，我们采用 Tu 和 Padua [22]，的算法，将门（谓词）添加到 SSA 形式程序中。该转换基于门控路径这一概念，将 ϕ-函数的放置与保护其输入值的控制谓词的推导统一为一个几乎线性时间的过程。给定一个位于合并块 B 中的 ϕ-函数，相对于该 ϕ-函数的门控路径满足以下非正式性质：

1. 它从直接支配点 idom(B) 开始。[2] 的直接支配点是控制流在分向 B之前最后汇合的唯一位置。2. 它对应于一个唯一的分支谓词序列，这些谓词决定哪个到达定义流入 B。3. 它终止于 B，即包含 ϕ-函数的基本块。

因此，在合并点处，ϕ 函数会被 γ-函数取代，后者会显式编码用于刻画其门控路径的布尔谓词。示例 4 说明了这一概念。

示例 4. 请看图4(a)。合并块 B5 包含一个针对 x2 的 ϕ-函数。它的直接支配点是 B0，因此每条门控路径都必须从 B0 出发并终止于 B5。根据控制流结构，我们枚举出四条不同的门控路径：

[2]如果从程序入口到 n 的每条路径都经过 d，那么基本块 d *dominates* 就支配 n。反过来，d *post-dominates* n 如果从 n 到程序出口的每条路径都经过 d，那么 n 就后支配 d。支配关系和后支配关系构成树：基本块的 *immediate dominator*（分别为 *immediate post-dominator*）是控制流图中距离该基本块最近的唯一严格支配者（分别为严格后支配者）。

- 路径 1 (x0)：B0 → B1 → B3 → B5 条件：p0 ∧p1
- 路径 2 (x0)：B0 → B1 → B2 → B3 → B5 条件：p0 ∧ ¬p1 ∧p2
- 路径 3 (x1)：B0 → B1 → B2 → B4 → B5 条件：p0 ∧ ¬p1 ∧ ¬p2
- 路径 4 (x1)：B0 → B4 →B5 条件： ¬p0

因此，选择 x0 的门控路径表达式为：

$$C_{\mathrm{x0}} = (\mathrm{p0} \wedge \mathrm{p1}) \vee (\mathrm{p0} \wedge \neg \mathrm{p1} \wedge \mathrm{p2})$$

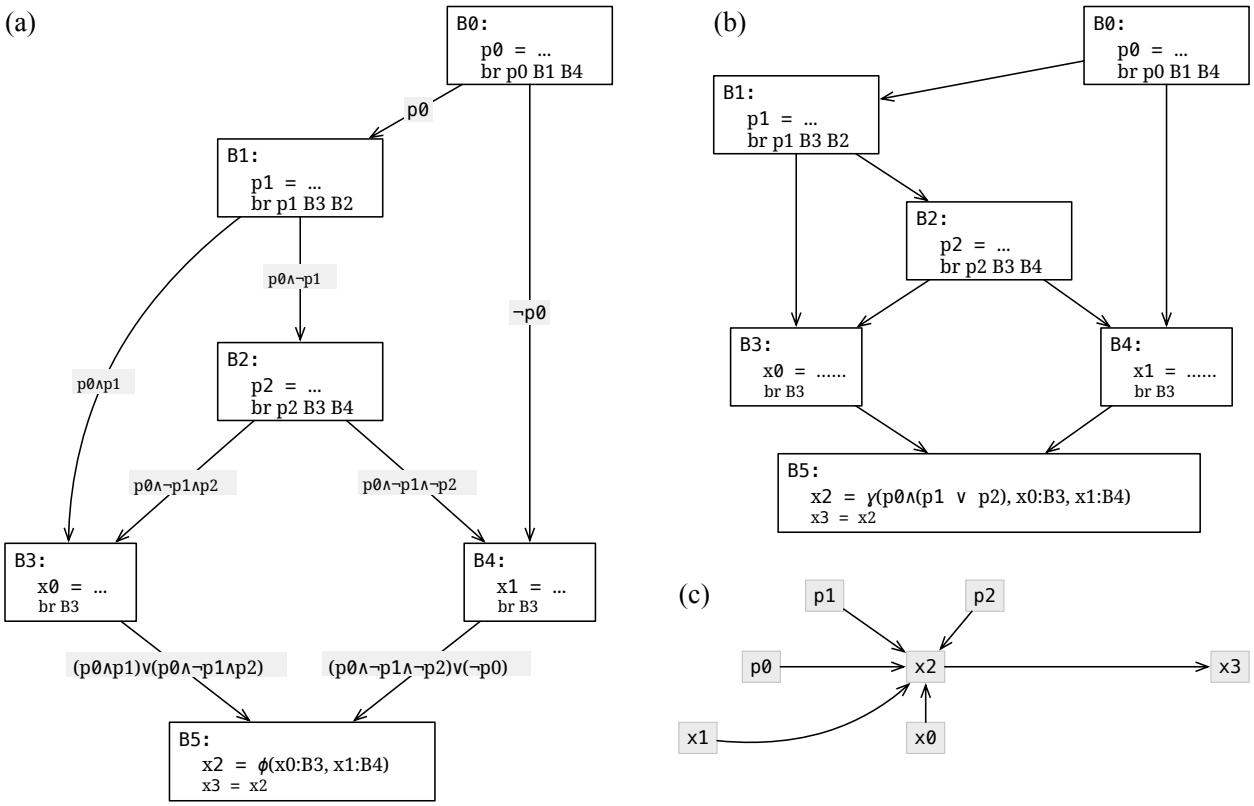


图4：(a) 标注了门控路径谓词的 SSA 控制流图。(b) 消除 ϕ 后的 GSA 形式。(c) 变量 x3 的反向依赖图。

通过分析门控路径，我们为每个 ϕ-函数补充相应的布尔守卫，使所选值显式化。在 GSA 中，这会产生 γ-指令，其形式为 $x = \gamma(P, x_{\mathrm{true}}, x_{\mathrm{false}})$，含义如下：

- 若 P 的计算结果为真，则 x 接收 $x_{\text{真}}$；
- 否则， x 接收$x_{\text{假}}$。

例5. 在图4(b)中， ϕ-函数 $\mathrm{x2} = \phi(\mathrm{x0}:\mathrm{B3},\ \mathrm{x1}:\mathrm{B4})$ 被 γ-指令 $\mathrm{x2} = \gamma\big(\mathrm{p0} \wedge (\mathrm{p1} \vee \mathrm{p2}),\ \mathrm{x0},\ \mathrm{x1}\big)$ 替代。这里，守卫 $\mathrm{p0} \wedge (\mathrm{p1} \vee \mathrm{p2})$ 概括了控制经由 B3 到达 B5 时所满足的所有条件。该条件的否定涵盖了到达 B4 的情况。因此， γ-函数精确地捕获了此前在 SSA 中隐含的值选择。

虽然 Tu 和 Padua 的算法会显式构造 μ- 和 γ-指令 [22]，但 η-指令早在 Ottenstein 等人提出的 GSA 原始定义中就已引入。 [13]

诸如 $v_0 = \eta(v_{exit},\ p)$ 的 η-指令会出现在一个退出循环的基本块中，其中定义了 v_{exit}，且循环迭代由谓词 p 控制。引入这类指令十分简单，因为它们自然源自循环闭合形式的概念。[3]

3.2 切片识别

程序转换为 GSA 形式后，我们通过对程序的依赖图（定义 3）进行反向遍历，识别构成其幂等后向切片（定义 2）的变量集合。

在 GSA 中，与切片相关的所有依赖都显式体现在指令的语法中。因此，与经典切片不同，无需单独计算数据依赖和控制依赖。例如，η-指令 $x = \eta(p,\ x_{exit})$引入一个控制依赖 $p \to x$ 和一个数据依赖 $x_{exit} \to x$。类似地，γ-指令 $x = \gamma(P,\ x_{\text{true}},\ x_{\text{false}})$引入两个数据依赖 $x_{\text{true}} \to x$ 和 $x_{\text{false}} \to x$，此外，对于每个谓词 $p \in P$，还会引入一个控制依赖 $p \to x$ 。

示例 6。 图4(c)展示了变量 x3 的反向依赖图。指向 x2 的三条入边——即 $\text{p}0 \to \text{x}2$、$\text{p}1 \to \text{x}2$ 和 $\text{p}2 \to \text{x}2$——对应控制依赖；其余边均为数据依赖。然而，对于切片提取而言，这一区别无关紧要：反向遍历只需沿着所有边进行。

遍历采用 两种停止准则；也就是说，当以下任一条件成立时，遍历即终止 ：

循环： 针对准则 v 的切片不能离开定义 v 的循环。

函数： 切片仅限于过程内部；它们不能跨越函数边界。

这些条件定义了针对准则 v的切片的停止集：即无需探索其入向边的变量集合。函数参数是自然的停止点。它们没有入向依赖，因此满足**函数**条件。为满足**循环**条件，我们为每个 GSA 变量指定一个循环深度，其值等于该变量定义所在位置的循环嵌套深度。针对准则 v 的反向遍历在遇到以下任一情况时停止：(i) 定义循环深度与 v相同变量的 μ-指令，或 (ii) 任意深度更小的变量。这确保切片始终局限于定义 v的循环内。

示例 7。 图5展示了变量 s3、x2 和 p0 的反向依赖图（此前见图2(d)）。图5(a)重现了 GSA 程序；(b)部分为每个变量标注了循环深度。(c)部分展示了 s3 的依赖图，其深度为 0。遍历只到达函数参数，因此这些参数构成停止集。(d)部分展示了 x2 的依赖图（深度为 1）。遍历在 x1 处停止，因为 x1 由同一深度上的 μ-指令定义。最后，(e)部分展示了 p0 的依赖图（深度同样为 1）。遍历在深度为 1 的 x1 定义处以及深度为 0 的参数 N 处停止。

定义 3 得出构成后向切片的完整且正确的变量集合，前提是每个节点都表示一条程序语句或一个 SSA 变量，且每条边都表示数据依赖或控制依赖。

4 基于切片的代码尺寸缩减

我们将基于切片的代码尺寸缩减（SBCR）称为一种编译器变换，它对程序 P 应用以下一系列步骤：

1. 在 P 中识别一组幂等后向切片（见第 3 节）；2. 将这些切片提取为一组函数（见第 4.1 节）；3. 识别一组计算相同切片的函数子集（见第 4.2 节）；

4. 合并 F' 中的函数，并根据用于估算代码尺寸缩减的成本模型，用一次调用替换它们的切片（见第 4.3 节）。

[3]在 LLVM 中，循环闭合 SSA（LCSSA）形式强制规定，在循环内定义的所有值只能在该循环内使用。如果某个值需要在循环外使用，LCSSA 会在循环出口处插入一个唯一的 ϕ-函数来捕获该值。在我们的场景中，我们会在这个 ϕ-函数中加入控制循环的谓词，如图2(d)所示。

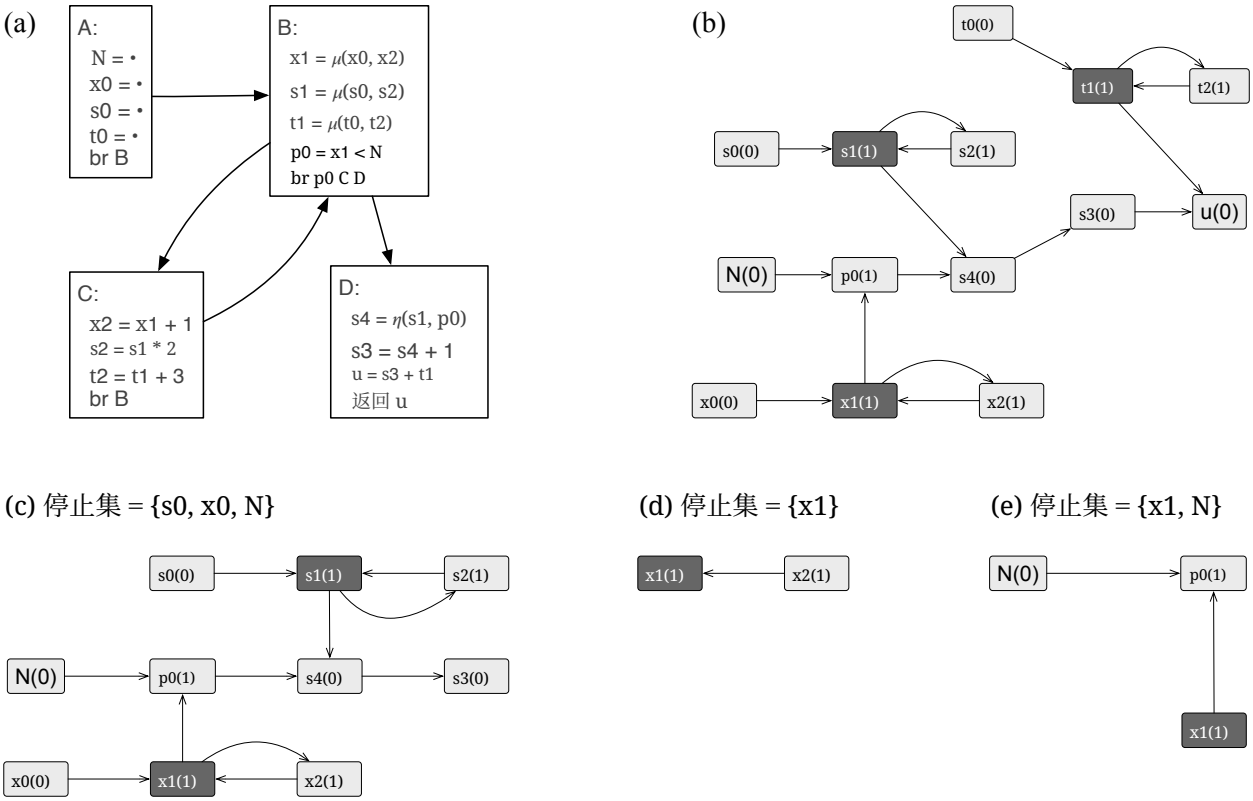


图5：（a）GSA 程序（改编自图2）。（b）根据定义-使用链构建的依赖图，标注了循环深度。由 μ-函数定义的变量以深灰色标出：它们充当停止准则。（c–e）分别为 s3、x2 和 p0 的反向依赖图。

4.1 函数提取

确定切片后，可以通过提取将其提取为一个独立函数。此变换受一组合法性检查的约束，以确保该切片具有幂等性；也就是说，它不会产生副作用，且只读取内存。如果所有检查均通过，提取将分四步进行：

1. **区域识别：** 通过反向遍历识别出的基本块集合（V_s）定义了待提取的切片区域（定义4）。 2. **接口定义：** 创建函数接口。其参数为切片停止集中的元素，返回类型与切片准则 v 的类型相同。 3. **代码克隆：** 将区域中的基本块克隆到新函数中。使用值映射重新映射引用：内部依赖重定向到克隆指令，外部依赖则替换为新函数参数。 4. **指令替换：** 删除父函数中的原始区域，并将其替换为对新提取的切片函数的一次调用。

定义4（切片区域）。设 $G = (V, E)$ 为程序的控制流图， S 为切片准则 v 对应的反向依赖图中的变量集合。切片区域 $V_s \subseteq V$ 是包含至少一条定义 S 中变量的指令的基本块集合。

恢复切片中的控制流重建切片的控制流图需要重写分支指令，因为父控制流图中的控制流路径可能经过 V_s 以外的基本块。重建依赖于 Guimar˜ães 和 Pereira [10] 引入、并在下文重新讨论的首个支配节点概念。

定义 5（首个支配节点）。给定一个控制流图 $G = (V, E)$、一个子集 $V' \subseteq V$ 和一个节点 $b_0 \in V'$，b_0 在 V' 中的首个支配节点是满足以下条件的唯一节点 $b_1 \in V'$：(i) b_1 在 G 中支配 b_0；(ii) 对于任意其他的 $b_2 \in V'$，若 b_2 支配 b_0，则 b_2 也支配 b_1。在实际操作中，从 b_0 出发，沿 G 的支配树向上回溯，直到遇到 V' 中的首个节点，即可找到 b_1。

虽然 Guimar˜ães 和 Pereira 依赖这一概念，但他们并未证明切片区域总是包含一个首个支配节点。在我们的情形中，这一性质可直接由引理 4.1[4] 推得。

引理 4.1（单入口区域）。设从一个准则 v出发，通过反向依赖遍历构造一个幂等后向切片 S，并确定一个切片区域 V_s。那么，存在唯一一个基本块$b_{entry} \in V_s$，它支配 V_s 中的所有其他基本块。

给定一个控制流图 $G = (V, E)$ 和一个切片区域 $V_s \subseteq V$，按照以下两条规则重建切片的控制流图边 E_s：

转置：如果 $b_0, b_1 \in V_s$ 且 $b_0 \to b_1 \in E$，那么 $b_0 \to b_1 \in E_s$。

吸引：如果 $b_1 \in V_s$、$b_0 \to b_1 \in E$ 和 $b_0 \notin V_s$，那么令 b 为 V_s 中 b_0 的首个支配节点；将 $b \to b_1$ 插入 E_s。

示例 8. 图 6 展示了如何通过转置与吸引重建控制流图。切片区域为 $V_s = \{$BB0，BB1、BB2、BB5、BB9$\}$。V_s 内部节点之间的边予以直接保留（例如，BB5 $\to$ BB9）。其他边则必须重定向。例如，由于 BB6 $\notin V_s$，BB6 $\to$ BB8 被移除；取而代之的是 BB2 $\to$BB8，因为 BB2 是 V_s中 BB6 的首个支配节点。

通过反向依赖遍历以及随后的转置与吸引生成的切片会形成单入口函数，并保留原始程序在切片准则上的可观察行为。定理 4.2 对此进行了形式化表述。

定理 4.2 (语义可靠性)。设 P 为一个程序，I_v 为定义变量 v的切片准则，并设 $S = \text{Slice}(P, I_v)$ 为提取的单入口切片，其入口为 L_{entry}。对于任意初始存储 σ_{in}：

$$If\ P, I_v \vdash \langle L_{entry}, \sigma_{in} \rangle \to^* \sigma_P \quad then\ S, I_v \vdash \langle L_{entry}, \sigma_{in} \rangle \to^* \sigma_S \quad and \quad \sigma_S[v] = \sigma_P[v].$$

4.2 公共切片识别与消除

公共切片的识别与消除利用了 LLVM 编译器中已有的逻辑。在当前原型中，我们提取每个准则为二元指令的切片。因此，如果一个程序包含 $O(N)$ 条此类指令，切片提取器最初可能会生成 $O(N)$ 个切片函数，并执行最坏情况下 $O(N^2)$ 次操作，如定理 4.3[5] 所述。

定理 4.3. 设 N 为给定程序中的指令数。如果候选切片准则的数量为 $O(N)$，且对单个切片执行反向依赖遍历和提取的最坏情况时间复杂度为 $O(N)$，那么完整切片提取阶段的最坏情况总时间复杂度为 $O(N^2)$。

一旦所有候选 切片都已提取，我们就使用 LLVM 标准的 mergefunc pass 消除重复项。过程分为两个阶段：

阶段 1：结构哈希（快速筛选）。每个外提函数都会根据其指令、常量和控制流骨架分配一个结构哈希值。哈希值不同的函数会立即被归类为不同函数。此步骤将原本数量呈二次方增长的两两比较，减少为近似线性的分组过程。

[4]引理和定理的证明见第一作者的 4 2 [8] 硕士学位论文。证明见 de Azevedo [5] [8]的硕士学位论文。

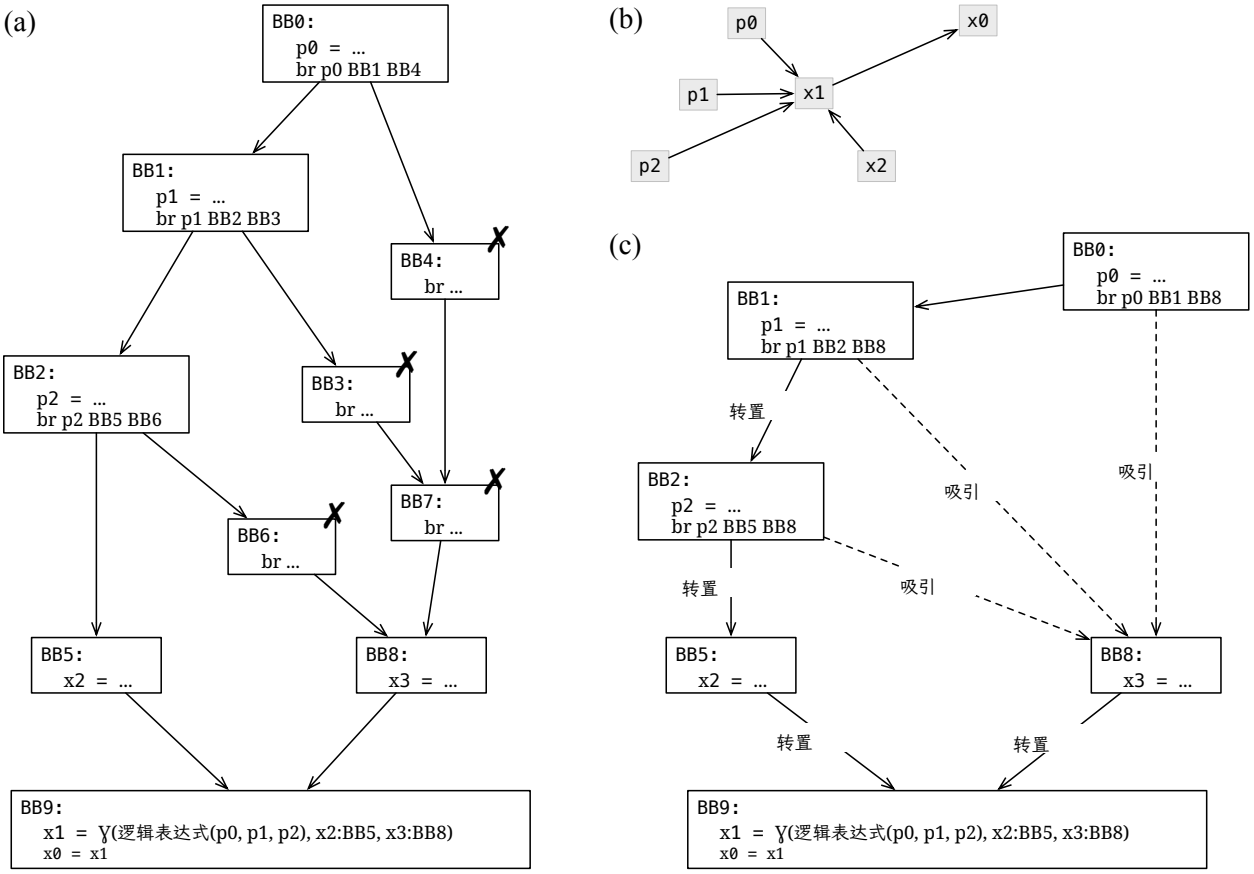


图 6：（a）包含常见“梯形”控制流图模式的程序。（b）针对准则 x1 的反向依赖图。（c）通过转置与吸引重构的控制流图。

阶段2：规范比较（等价性检查）。在每个哈希组内，都会执行递归等价性测试。该过程尝试在参数、指令和基本块之间建立双射，以证明两个函数除重命名外完全相同。确认等价后，会选出一个实现作为规范实现，并将其重复实现替换为轻量级别名或桩调用。

规范化之后，第4.3节中引入的成本模型会确定哪些切片函数具有盈利性：只有预计能够减小代码大小的切片函数才会被保留。请注意，如果认为提取某一组相似切片不值得，那么代表这些切片的函数会直接从程序中移除，不会对目标代码造成任何修改。换言之，我们的实现不会先进行提取再重新内联，而是直接移除被判定为无盈利性的外提函数。否则，切片准则（原始定义指令）会被替换为对合并切片函数的调用，而 LLVM 的 simplifyCFG 优化遍会移除父函数中剩余的死代码。

4.3 成本模型

与许多编译器优化一样，SBCR 的有效性取决于成本模型。在本文中，我们使用由三个值参数化的成本模型（I、P、C），定义如下：

- I：实现概述切片 S 的指令数量；
- P：表示 S 的函数的参数数量；

- C： S 在目标程序中的出现次数。如果该切片被外提，然后与其他相似的概述切片合并，则该数字近似于对外提函数的调用次数。

基于这些参数，优化可以将 S 提取到单独的函数中，也可以将其保留在原处。如果切片保留在原处，则不会对代码造成任何修改。在我们的实现中，只有当（$3 < I \leq 20,\ P \leq 1,\ C \geq 10$）时，才会提取切片 S。为确定这些阈值，我们以 LLVM 测试套件为训练数据，在（$I = \{10, 20, 40, 80, 160\},\ 0 \leq P \leq 20,\ C = \{10, 20, 40, 80, 160, 320, 640\}$）空间内进行了搜索。第 5.1 节将进一步介绍这一搜索。

5 评估

本节通过实验评估第 4 节介绍的切片方法，该方法实现为一个树外 LLVM pass。本文分析以下研究问题：

研究问题1： SBCR 的理想成本模型参数是什么？

研究问题2： 幂等切片提取能够实现多大程度的代码尺寸缩减？与先前的工作相比如何？

研究问题3： 切片提取对基准测试运行时间有何影响？与先前的工作相比如何？

RQ4： 切片提取会带来多少编译开销？与先前的工作相比如何？

RQ5： 切片提取算法的渐近行为如何？

RQ6： 所提优化各个阶段分别需要多长时间？

RQ7： 将 Rocha 等人 [15] 的代码合并、LLVM 提取器和 SBCR 结合起来，是否能带来累积收益？

基准测试和尺寸指标 我们在 LLVM 测试套件（https://github.com/llvm/llvm-test-suite）中提供的全部 2007 个基准测试上运行实验。该测试套件包含 134 个单元测试程序、来自 gcc-c-torture、C 和 C++集合的 1,548 个回归测试程序、来自 MiBench、PolyBench、TSVC 等基准测试套件的 307 个程序，以及 18 个来自实际应用程序的程序。我们通过两种方式报告这些程序的尺寸：程序中间表示里的 LLVM 指令数量，以及可执行文件 .text 段的大小。我们确认，对于整个测试套件，我们的实现生成的二进制文件均能正确运行。

基线方法 下文中，我们将本文提出的技术称为 SBCR（基于切片的代码体积缩减）。我们将其与先前工作中的两种基线方法进行比较：

FMSA： Rocha 等人 [15] 于 2020 年提出的技术，它使用序列比对来识别公共指令，并将其合并到单一控制流路径中。

IROutliner： LLVM 中提供的一种标准通道，用于识别基本块内重复指令的序列，并将其提取为公共函数。

硬件和软件 实验在一台专用服务器上进行，该服务器配备 AMD 中央处理器（Ryzen Threadripper 7970X），拥有 32核，主频为 4 GHz，随机存取存储器容量为 128 GiB。操作系统为 Ubuntu 24.04.3 LTS（GNU/Linux 6.8.0-85-generic x86 64）。实验在多线程设置下运行；不过，除了加快评估速度外，这一点不会对结果产生任何影响。

编译流水线 为隔离每个优化遍的影响，我们分别在应用和未应用目标遍的情况下，使用 -Os 标志编译每个程序，并测量各项指标。由于 LLVM 测试套件默认不支持此功能，我们修改了其构建系统，创建了一个构建后流水线，流程如图7所示。该流水线首先使用链接时优化（LTO）编译程序，将位码嵌入可执行文件。然后，我们使用 objcopy 提取位码，通过规范化遍（mem2reg、lcssa）对其进行规范化，应用目标优化，最后使用 -Os 重新编译转换后的位码，以收集指标。

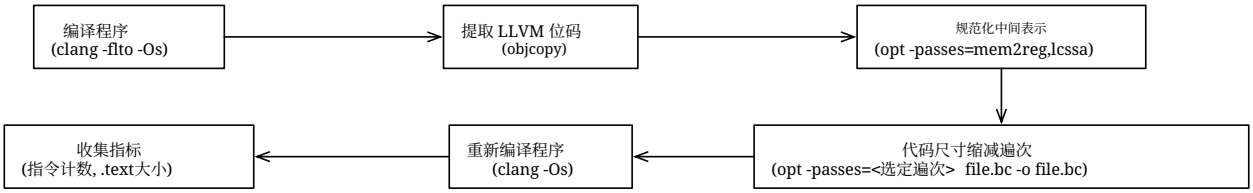


图7：用于评估不同代码大小缩减技术的编译流水线。请注意，代码尺寸缩减方法在链接时应用于包含多源基准测试所有模块的单个位码文件。本节报告的每一种变体都基于 clang -Os。

结果摘要 实验结果汇总于表1。对于每个优化遍，该表报告了受变换影响的程序总数，以及变化的总体几何平均值。几何平均值中的正百分比表示某项指标增加，负百分比表示减少。请注意，如果我们沿这四个不同维度考虑优化：指令计数、.text 段大小、执行时间和编译时间，那么本节评估的三种变换均无法同时缩减全部指标。然而，如果我们聚焦于特定基准测试，则可以看到其中任一指标都有显著改进，我们将在本节其余部分进行分析。

表1：结果摘要。**受影响数量**：在2007个程序中受变换影响的程序数量。**几何平均值：** 相对变化的几何平均值。负百分比表示该指标减少，正百分比表示增加。

	SBCR		函数合并		IROutliner	
指标	**受影响数量**	**几何平均值**	**受影响数量**	**几何平均值**	**受影响数量**	**几何平均值**
指令计数	142	-0.24%	183	-0.35%	207	-0.65%
.text大小	133	0.11%	190	0.39%	160	-0.19%
执行时间	161	0.06%	177	0.25%	160	-0.09%
编译时间	303	4.22%	241	2.06%	258	2.48%

本节其余部分将逐一分析提出的研究问题。表1总结了核心发现。我们评估得出的主要结论如下：

研究问题1： 最有效的 SBCR 配置所概述的切片最多有一个参数，包含少于 20 条 LLVM 指令，并且在程序中至少出现 10 次。

研究问题2： SBCR 和 FMSA 均不能在所有基准测试中普遍缩减 .text 大小；然而，仅考虑对其有益的程序时，两种方法的平均缩减幅度均接近 -10%。

研究问题3： 在运行时执行中不会引入统计显著的性能开销 .
SBCR

RQ4： SBCR 平均使总体编译时间增加约 4%。

RQ5： 在实践中，当以 LLVM IR 指令数衡量程序大小时，SBCR 的执行时间会随程序大小线性增长。

RQ6： T SBCR 中计算开销最大的组件是切片识别阶段 .

RQ7： 结合这三种代码大小缩减技术，所取得的缩减效果优于单独应用任何一种技术。

5.1 研究问题1：成本模型

如第4.3节所述，是否应将切片外提，由一个三参数成本模型来指导。为确定有助于缩减代码大小的有效参数值，我们以 LLVM 测试套件作为评测语料库，在一个有界设计空间内进行了穷举搜索。此次搜索包含 735 次编译运行，并改变以下阈值：最大参数数量（$0 \leq P \leq 20$）、切片中的最大指令数（$I \in \{10, 20, 40, 80, 160\}$），以及切片出现次数的最小值（$C \in \{10, 20, 40, 80, 160, 320, 640\}$）。

讨论。图 8 展示了在改变允许的函数参数数量时，代码大小缩减的几何平均值。结果表明，当切片的输入集非常小时，缩减效果最佳——最好最多只包含一个变量。这些变量将成为外提函数的形式参数。

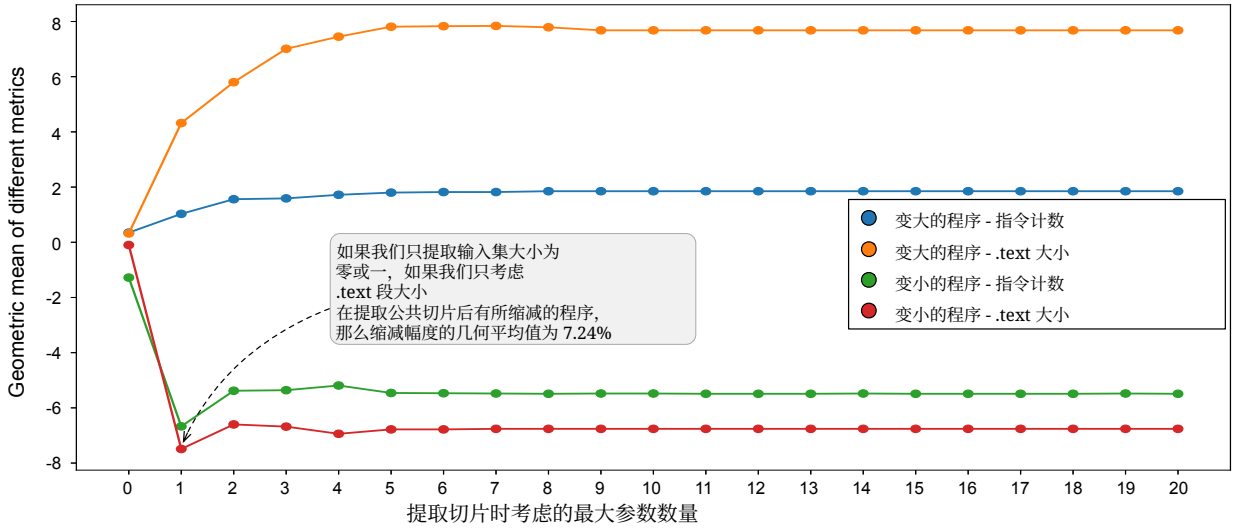


图8：切片参数最大数量对代码大小指标几何平均值的影响。

尽管参数 P 的最优选择取决于 I （指令数）和C （调用次数）的阈值，但我们的评估始终倾向于 $P \leq 1$的配置。函数参数的数量会显著影响函数的序言和尾声是否会带来可观的额外开销。测得的指令计数指标最佳降幅出现在以下情形：概述切片最多包含$I = 20$ 条指令，且在程序中至少出现 $C = 10$ 次。

5.2 研究问题2：代码大小缩减

本节研究 SBCR（以及先前的工作）对 LLVM 测试套件的影响。该方法使用图7所示的流水线编译每个程序，并记录其 .text 段大小和指令数（指令计数指标）。

讨论 如果不加限制地采用基于切片的方法，那么在2007个程序中，几何平均值会略微增加0.11%。然而，如果仅将其用于能够减少代码的程序（同时考量指令计数和 .text大小指标），那么它会使23个程序的 .text 段总体减少，几何平均减少幅度为 -8.39%，指令计数则减少 -9.96%

指标）。如果仅分析只降低指令计数指标的程序子集，那么29个程序的几何平均减少幅度为 -7.24%。请注意，尽管这三种方法旨在减少代码，但出于不同原因，它们也可能导致代码增加：

- 基于切片的方法和基于序列比对的函数合并，可能会因控制流指令（即条件分支和无条件分支）的重复而导致代码增大。尽管提取切片会对父函数进行死代码消除，但分支可能仍未被移除，因为这些分支的目标是包含不属于原始切片的指令的基本块。因此，这些分支会同时保留在父函数和外提切片中。

- LLVM IR 提取器仅针对不含分支的指令序列。然而，将复制参数转换为外提函数所需的额外指令，以及随后将其返回值复制回父函数所需的指令，都可能导致代码增大。

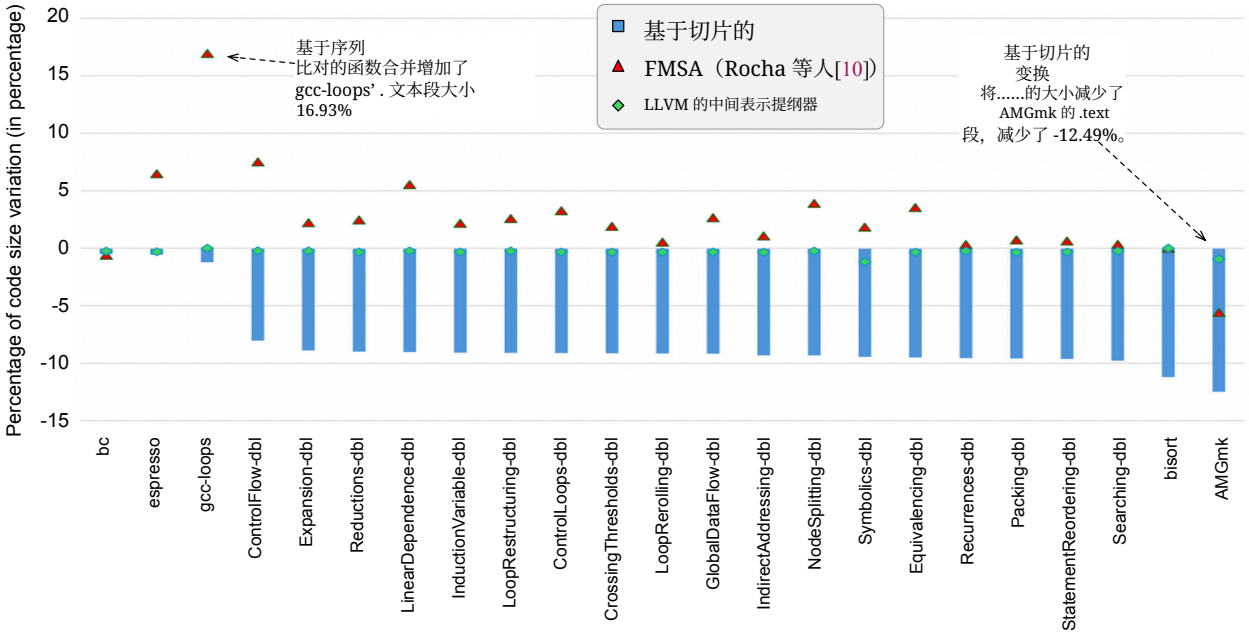


图 9：在基于切片的技术实现最大百分比缩减的 23 个基准测试上，对不同方法进行比较。

没有任何一种技术能涵盖其他所有技术在基于切片的技术能够缩减代码的情况下，其表现显著优于基线方法。图 9 展示了在我们的方法同时缩减指令计数和 .text 大小的 23 个程序中的表现。在这些情况下，我们的技术（蓝色）实现了大幅缩减，而 IROutliner（绿色）的缩减幅度很小，FMSA（红色）则经常导致代码增大。例如，在 IndirectAddressing-dbl.test 中，SBCR 实现了 9.32% 的缩减。IROutliner 的缩减幅度为 0.35%，而 FMSA 则使代码增大了 1.07%。

图 9 展示了 SBCR 能够找到其他算法遗漏的优化后代码中的冗余；不过，也会出现相反的情况。图 10 比较了在我们的方法同时增大了 .text 大小和 LLVM 中间表示指令计数的 57 个程序中，各方法的影响。在 ldecod.test 中，这一权衡的例子十分明显：基于切片的缩减使代码大小增加了 1.97%。而 IROutliner 实现了小幅缩减（-0.53%），FMSA 则实现了显著缩减（-2.5%）。

深入剖析最坏情况基准测试表 2 展示了一个基准测试 activate sps 的详细情况，在该测试中，SBCR 导致了不可忽略的_代码体积增长。SBCR 将 activate sps 的代码体积从 3007 字节增加到 3052 字节（1.50%）。_增长的原因是 SBCR 创建了两个较小的切片函数，每个函数都被调用两次。

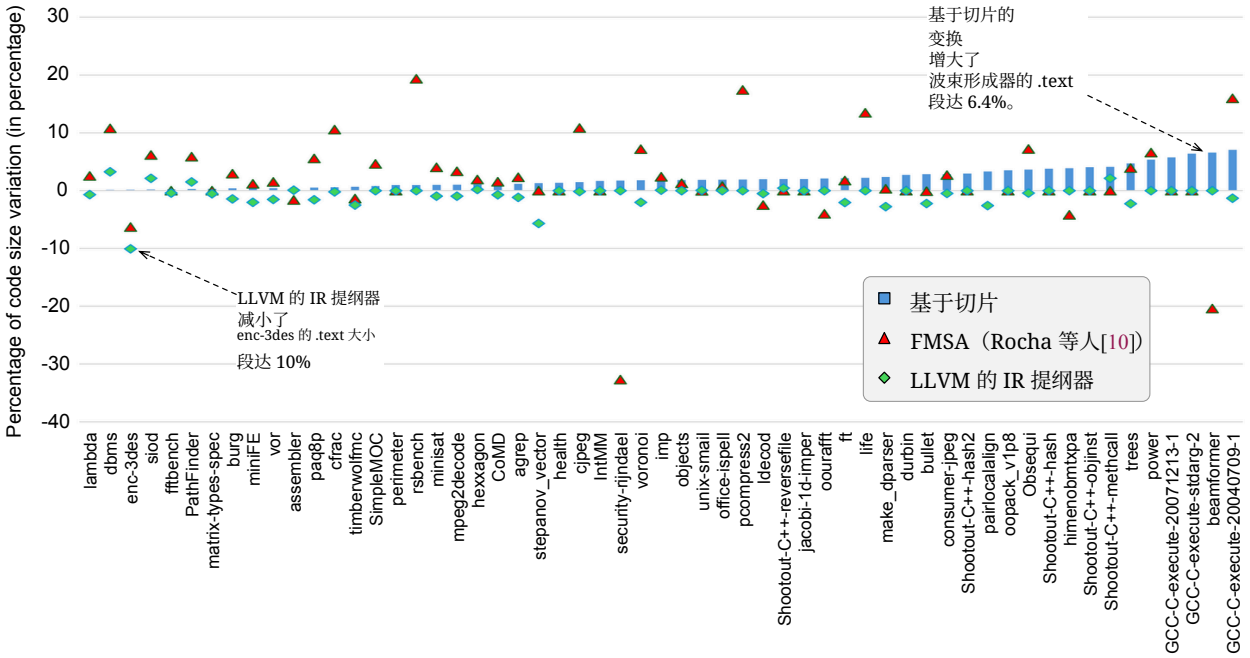


图 10：在基于切片技术取得最大百分比增幅的 57 个基准测试中，不同方法的比较。

这些细粒度、非重复模式带来的开销超过了节省的空间。IROutliner 也将同一个函数的大小增加到 3200 字节（6.42%）。它找到了两个较大的重复区域（174 和 180 字节；共调用 9 次），但调用点复杂度使其无法实现净缩减。相比之下，FMSA 将该函数缩减至 2963 字节（-1.46%）。它用对六个合并函数的 24 次调用替换了原有代码（大小为 211–1145 字节）。在此情形下，粗粒度、重复变换摊薄了开销，并带来了最佳缩减效果。

表 2：activate sps 在各遍次中的代码大小影响比较。

Pass	最终大小（B）	Δ（B）	函数数	新函数大小（B）	# 调用	粒度／重复性
原始版本	3007	0	0	–	–	基线
SBCR	3052	3052 − 3007	2	9, 9	2	细粒度／低
IROutliner	3200	3200 − 3007	2	174, 180	9	中等 / 适中
FMSA	2963	2963 − 3007	6	211, 514, 303, 314, 515, 1145	24	粗粒度 / 高

5.3 研究问题3：运行时间

本节讨论的三种优化主要是为缩减代码尺寸而设计的，而非为了提升性能。因此，它们对执行时间的影响通常有限：在大多数经过变换的程序中，我们没有观察到具有统计显著性的差异。不过，根据所应用的变换，个别基准测试确实表现出显著的加速或性能退化。本节将详细分析这些差异。

讨论。 在所有基准测试中，SBCR 对执行时间的影响在统计上并不显著，几何平均运行时间变化为 0.06%（表1）。即使将观察范围限制在经 SBCR 实际变换的程序中，绝大多数程序的性能差异也无法测量。在性能下降的情况下，SBCR 使 14 个程序的运行时间增加了 4.48%。这一增长是由函数调用开销造成的。类似地，FMSA 和 IROutliner 分别使 6 个和 29 个程序的运行时间增加了 19.96% 和 6.67%。

我们还观察到变换后的基准测试性能有所提升的情况。SBCR 在 7 个基准测试中将执行时间缩短了 3.39%，而 FMSA 在 6 个基准测试中缩短了 15.16%，IROutliner 则在 27 个基准测试中缩短了 8.34%。我们认为，这些提升是由于指令缓存局部性更好，正如本节后文所讨论的。

指令缓存性能分析 我们分析了 Global- DataFlow-dbl 的硬件性能计数器。该基准测试的执行时间、指令数和 .text大小降幅最大。将 -Os 基线与 SBCR 版本进行比较后发现，缓存未命中率有所降低，我们认为这是指令局部性和缓存利用率提升所致。在基线配置中，硬件计数器记录到 6027 次分支加载和 2165 次分支加载未命中，同时记录到 1736 次 L1 instruction cache 加载和 1163 次未命中。应用 SBCR 后，这些数值变为 5661 次分支加载、1870 次分支加载未命中，以及 1011 次 L1 instruction cache 加载和 600 次未命中。GlobalDataFlow-dbl 是 SBCR 的理想案例，因为该优化提升了指令和数据局部性。L1数据缓存未命中率从 66.99% 降至 59.35%。控制流也变得更可预测（分支未命中率从 35.92% 降至 33.03% ），表明内存效率得到提升，且没有造成明显的控制流惩罚。

5.4 RQ4：编译时间

如第 4 节所定义，基于切片的代码尺寸缩减是一种开销相对较高的变换：它需要对程序中的每条二元指令执行反向依赖遍历，随后将切片提取为独立函数，并合并等价函数。尽管如此，本节表明，与完整编译流水线的总体开销相比，相关开销仍然不大。

讨论。 平均而言，SBCR 会使编译时间增加 4.22%；该数值是在编译时间发生变化（无论增加还是减少）的 303 个程序上计算出的几何平均值。图11展示了这一基准测试集合中编译时间的变化情况。其余 1704 个程序的编译时间均未出现统计上显著的变化。我们比较的另外两种基线方法也会带来编译开销，但程度较轻。对于编译时间发生变化的 241 个程序，FMSA 会使编译时间增加 2.06%（仅考虑此项优化；也就是说，这并非整个集合
（图11所分析的基准测试中）。反过来，IROutliner 会使 258 个受影响程序的编译时间增加 2.48%。

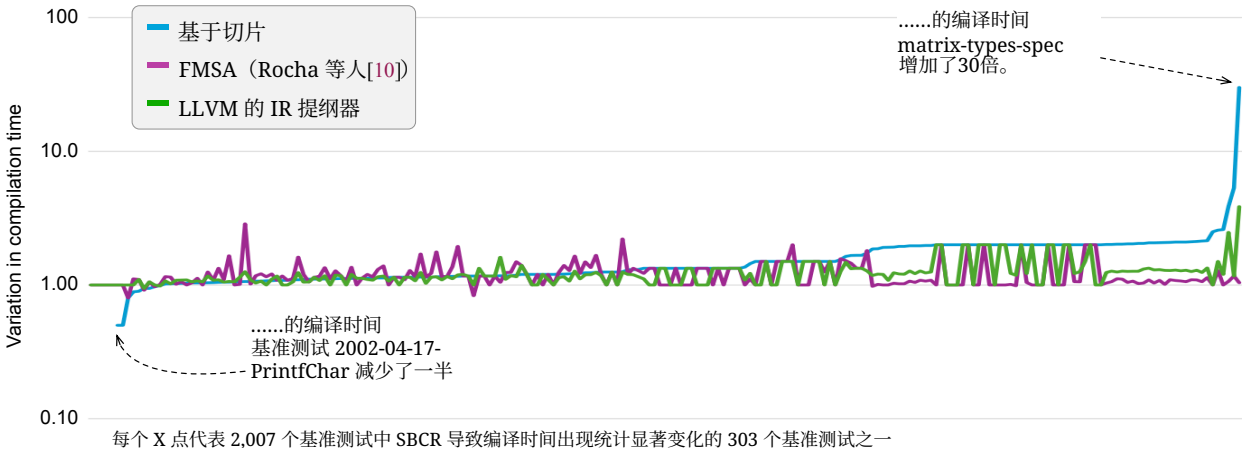


图11：编译时间变化。

SBCR 的主要开销在于提取，这需要将代码克隆到新函数中，并替换父函数中的引用。相比之下，构建 GSA 形式和执行反向依赖关系发现只占运行时间的一小部分。第 5.6 节分析了这些阶段。有趣的是，在某些基准测试中，尽管增加了额外的 SBCR 处理，编译时间仍有所改善。在代码尺寸缩减具有统计显著性的九个基准测试中，编译时间的几何平均值降低了

–19.81%。这些改善源于 SBCR 生成的 LLVM 指令数量减少：由于大多数 LLVM 分析和变换的运行时间都与程序大小成正比，因此缩减中间表示可以加快后续遍次。

5.5 RQ5：渐近行为

定理 4.3 指出，SBCR 的最坏情况为 $O(N^2)$，其中 N 是程序中的 LLVM IR指令数。然而，这一上界在实践中很少出现。本节提供证据，表明 SBCR 在真实世界代码上呈现近线性行为。为此，我们选取了测试套件中规模最大的 100 个程序，并测量了每种变换的指令数与绝对执行时间之间的皮尔逊相关系数。

讨论。 图 12 展示了所评估的三种代码大小缩减技术中，编译时间随程序大小变化的回归线。在所有情况下，编译时间与指令数的相关性都接近 1.0：SBCR 为 0.84，FMSA 为 0.94，IROutliner 为 0.92。图 12(a)–(c) 中的散点图直观地印证了这一趋势，数据点与拟合的线性曲线紧密吻合。尽管 SBCR 在理论上具有二次复杂度，但若干实际特性使其表现接近线性：

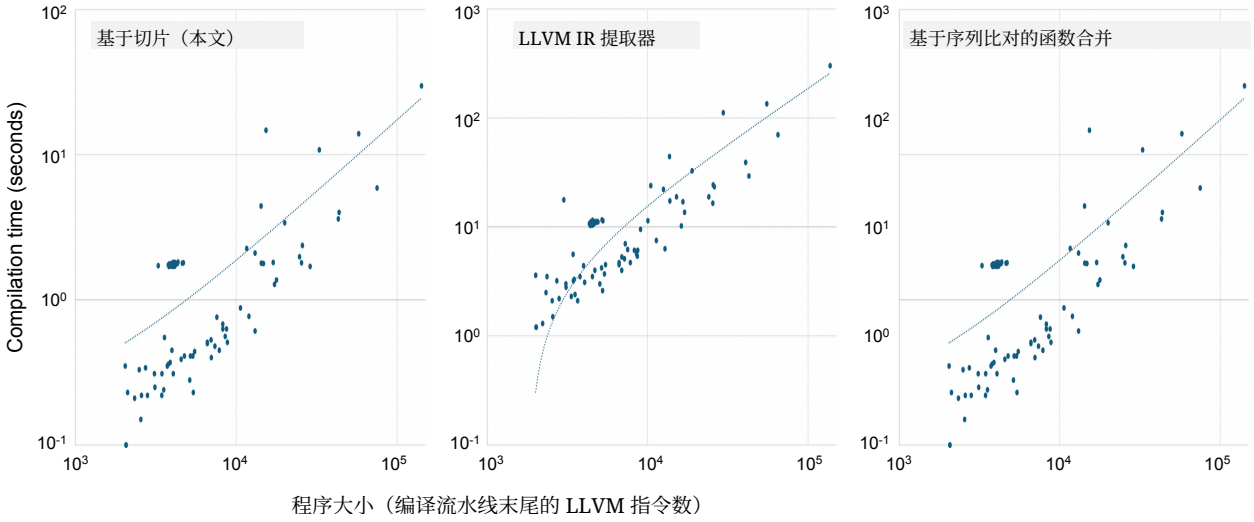


图 12：编译时间与程序大小之间的相关性。编译时间仅包括代码大小缩减处理过程的执行时间。程序大小按图 7 所示流水线末端的指令数衡量。每个点代表一个基准测试。坐标轴采用对数刻度。

- **大多数切片都很小。** 反向依赖遍历（第 3 节 3.2）通常只需访问少量指令即可到达切片准则。因此，对于大多数切片，算法的这一部分的复杂度为 $O(1)$。

- **满足成本模型的切片很少。** 只有少数候选项符合提取和合并的条件，因此与 LLVM 的 mergefunc pass 的交互实际上是常数级的；最坏情况下也只与被接受的切片数量成正比，而这个数量通常很少。

剩余的主要组成部分是 GSA 构建，其复杂度与程序规模准线性相关，这与图 12(a) 中观察到的线性相关性一致。

5.6 RQ6：各阶段耗时

本节将分解 SBCR 的各个变换步骤，并测量给定输入程序中每个阶段所耗费的时间。提醒读者，正如第 4 节所述，SBCR 编译遍由四个阶段组成：提取、

合并、删除指令和简化。轮廓化阶段 包含构造 GSA形式、识别程序切片、检查切片是否适合提取，以及将切片提取到新函数中的过程。合并 阶段负责合并所有相似的外提函数。删除指令 阶段负责从原始函数中删除那些移入新合并函数后变得多余的指令。最后，使用 LLVM 的 simplifycfg 优化遍简化所有经过修改的函数。

讨论 为开展这项研究，我们在所有 2007 个受测程序上运行了 SBCR，并收集了各阶段执行时间所占的百分比。随后，我们计算了所有程序中这些百分比的几何平均值。结果见表4，其中概述了 SBCR 的主要阶段；表3则详细列出了大纲阶段的各项数据。

表3：轮廓化子阶段的计时器。

Name	墙钟时间
切片识别阶段	42.389%
GSA构建阶段	34.350%
canOutline阶段	32.581%
函数轮廓化阶段	11.541%

Table 4：SBCR阶段（大纲包括 Table 3)。

Name	墙钟时间
大纲阶段	48.434%
合并阶段	22.885%
移除指令阶段	18.744%
简化阶段	18.383%

列名称标识正在测量的阶段，而列墙钟时间 表示实际经过的时间，包括任何等待或同步延迟。百分比表示每个阶段对总体编译时间的相对贡献。根据表3和表4中的结果，我们得出结论：大纲阶段 是最耗时的阶段。此外，在分解大纲阶段时，SBCR的切片识别 和*GSA* 构造 子阶段花费大量时间分析指令的依赖关系，并创建 ϕ-函数的门映射。后一项分析用于识别前一项分析所使用的所有数据和控制依赖。

5.7 RQ7：组合结果

我们进行了一项实验，以评估组合使用 SBCR、IROutliner 和 FMSA 是否比单独运行其中任一优化实现更大的代码尺寸缩减。为此，我们使用图7中的流水线，在整个 LLVM 测试套件（2007个程序）上评估了这三种遍次的全部六种排列。随后，我们选出表现最佳的排列，并将其结果与单独应用各项优化的结果进行比较。

讨论。 表5总结了所评估的各个遍序列在每项指标上的几何平均值。表现最佳的排序是 IROutliner $\rightarrow$ SBCR $\rightarrow$ FMSA。该流水线在 86 个基准测试中缩减了 .text大小。在这些基准测试中，指令数的几何平均缩减幅度为 -14.43%，优于单独使用这三个遍次中的任意一个。值得注意的是，如果仅比较 .text大小的缩减，FMSA 的平均缩减幅度略高；然而，它覆盖的程序子集要小得多（34 个基准测试）。

表5：组合优化的效果。括号中的数字（第一行）表示 .text大小有所缩减的程序数量。各列的平均值均针对这一子集计算。

指标名称	IROutliner-SBCR-FMSA	IROutliner	SBCR	FMSA
.text大小	-9.68% (86)	-4.30% (118)	-7.24% (29)	-9.75% (34)
指令计数	-14.43%	-6.67%	-7.93%	-7.11%
执行时间	3.38%	-0.49%	4.55%	-0.05%
编译时间	39.85%	15.84%	65.10%	16.04%

6 相关工作

这项工作涉及编程语言领域文献中的多个方向，包括经典的程序分析与表示（例如 GSA 构造），以及代码尺寸缩减方面的最新进展。本节将回顾这些领域的主要贡献。

程序切片。 程序切片是一项历史悠久的研究领域，起源于 Weiser 的开创性工作 [24]。如示例 2 所示，我们对幂等切片 的定义不同于传统的程序切片：我们不提取子程序，而是构造引用透明函数。因此，切片识别与提取基于 SSA 值而非程序语句进行，正如第 3 节所解释的那样。

从这个意义上说，与我们最相关的工作来自 Guimar˜aes 和 Pereira [10], ，他们使用幂等切片的一种变体，将值调用语义转换为名调用语义。然而，正如第 2.4 节所述，切片的定义和底层算法都不相同。他们的方法依赖 Ferrante 等人 [9]提出的控制依赖定义，并使用 Rodrigues 等人 [17]的稀疏算法在 SSA 图上识别控制依赖。这种方法可能无法识别影响切片结果的某些分支。我们的技术利用 GSA形式的显式依赖结构，避免了这些局限。

GSA 构造。 门控静态单赋值形式由 Ottenstein 等人提出 [13]„ 其工作建立在 Ferrante 等人 [9] 的早期研究以及逐渐形成的 SSA 形式定义之上 [1, 5, 18]。第 3.1 节采用的构造算法源自 Tu 和 Padua [22]；不过，我们的实现遵循 Herklotz 等人 [11]较新的形式化定义。Herklotz 等人为 GSA 形式程序提供了形式化的操作语义，并描述了如何在 ϕ-函数中引入门。我们选择遵循 Herklotz 等人的表述，因为 Tu 和 Padua 并未以我们的基于切片的方法所要求的方式，明确对 η-函数和 μ-函数进行建模。尽管如此，此前的文献均未探索将 GSA 形式作为切片的基础。据我们所知，本研究首次将 GSA 形式用作切片提取的系统化基础。

代码尺寸缩减。 通过消除冗余来缩减代码大小已得到广泛研究，并推动了大量近期研究。由于指令数不会发生统计波动，代码尺寸缩减已成为随机编译器优化的基石 [7, 19, 2]。然而，我们的方法完全是确定性的。不过，以往大多数确定性指令外提技术都着重于发现共有的连续指令序列 [4]。相比之下，基于切片的代码尺寸缩减能够识别包含分支或循环的非连续区域（如图2所示）。

HyFM [16] 和 FMSA [15] 等技术也能够提取非顺序代码。不过，它们都无法合并同一函数内的冗余代码。LLVM IROutliner [14] 可以做到这一点，但它并非基于区域，因此无法提取跨越多个基本块的代码。正如第 5 节所示，SBCR 既不是 FMSA 的子集，也不是其超集；对 IROutliner 也同样如此。与 IROutliner 一样，它既能在过程内提取，也能跨过程提取；但与这两种先前方法不同，它是 基于区域的。通过确定一个切片准则并纳入该准则所依赖的所有指令，SBCR 能够提取表达能力更强、语义更连贯的代码区域，即使这些区域跨越控制流边界也不例外。

7 结论

本文提出了幂等后向切片 这一概念，并介绍了一种在门控 SSA 形式中识别这类切片的实用算法。我们认为，幂等切片代表了一种新颖且表达能力强的冗余单元：引用透明计算，可以在不改变程序行为的情况下提取、复制和调用。基于这一洞见，我们提出了 SBCR，这是一种保语义变换，可将幂等切片提取为单入口函数，从而实现精确且安全的代码尺寸缩减。我们在 LLVM 测试套件中的 2,000 多个程序上进行了评估，结果表明：

- SBCR 实现了具有竞争力的代码尺寸缩减，尤其是在存在细粒度冗余的情况下。

- 没有一种技术在所有情况下都占优：SBCR、IROutliner 和 FMSA 能够发现不同的优化机会。

- 尽管理论上的复杂度为二次方，SBCR 在实践中呈现出近线性的扩展性，因此适用于实际的编译流水线。

我们认为，这项工作是探索将幂等切片作为通用冗余消除抽象的第一步。今后面临的一项关键挑战，是在激进应用变换时避免增加代码大小。这一问题促使我们持续改进收益性启发式和成本模型，或将其与剖面引导优化相结合。这些方向留待未来工作探索。

数据可用性声明

本文讨论的所有思想均已实现，并以 GPL 3.0 许可证公开发布于 https://github.com/lac-dcc/Daedalus。第 5 节报告的实验使用了 LLVM 17，选择该版本是为了兼容 FMSA 的补丁、IROutliner 以及我们的 pass。我们扩展了 LLVM 的 mergefunc pass，使其能够在我们的 pass 中合并函数。LLVM 17 的特定构建源代码可在 https://github.com/Casperento/llvm-project/tree/merge-functions-pass 获取。实验通过为本研究开发的 shell 脚本执行，这些脚本已公开发布于 https://github.com/Casperento/daedalus-dbg-toolkit。我们提供了 Docker 镜像以复现所有实验。*Dockerfile* 位于 https://github.com/lac-dcc/Daedalus/tree/main/artifact/docker。

致谢

本项目获得了 FAPEMIG（资助编号 APQ-00440-23）、CNPq（资助编号 #444127/2024-0）、CAPES（PrInt）和谷歌的支持；谷歌资助了 Rafael Alvarenga 的部分奖学金。我们感谢 Xinliang (David) Li 和 Victor Lee 为促成谷歌赞助所付出的努力。

引用

[1] B. Alpern、M. N. Wegman 和 F. K. Zadeck。《检测程序中变量的相等性》。载于*POPL*，第 1–11 页，美国纽约州纽约市，1988 年。计算机协会。ISBN 0897912527。doi：10.1145/73560.73561。URL https://doi.org/10.1145/73560.73561。

[2] Jordi Armengol-Estap´e、Jackson Woodruff、Chris Cummins 和 Michael F. P. O'Boyle。《Slade：面向优化后汇编代码的便携式小型语言模型反编译器》。载于*CGO*，第 67–80 页，英国爱丁堡，2024 年。IEEE 出版社。ISBN 9798350395099。doi：10.1109/CGO57630.2024.10444788。URL https://doi.org/10.1109/CGO57630.2024.10444788。

[3] Sandrine Blazy、Andre Maroneze 和 David Pichardie。程序切片的验证。发表于*CPP*，第 109–117 页，纽约，纽约州，美国，2015 年。计算机协会。ISBN 9781450332965。doi：10.1145/2676724.2693169。URL https://doi.org/10.1145/2676724.2693169。

[4] Milind Chabbi、Jin Lin 和 Raj Barik。生产级 iOS 移动应用代码尺寸优化的实践经验。发表于*CGO*，第 363–366 页，线上活动，韩国，2021 年。IEEE 出版社。ISBN 9781728186139。doi：10.1109/CGO51591.2021.9370306。URL https://doi.org/10.1109/CGO51591. 2021.9370306。

[5] Ron Cytron、Andy Lowry 和 F. Kenneth Zadeck。高级语言中控制结构的代码移动。载于*POPL*，第70–85页，美国纽约州纽约市，1986年。计算机协会。ISBN 9781450373470。doi: 10.1145/512644.512651。URL https://doi.org/10.1145/512644.512651。

[6] Ron Cytron、Jeanne Ferrante、Barry K. Rosen、Mark N. Wegman 和 F. Kenneth Zadeck。高效计算静态单赋值形式和控制依赖图。*ACM*《编程语言与系统汇刊》，13(4):451–490，1991年10月。ISSN 0164-0925，1558-4593。doi: 10.1145/115372.115320。URL https://dl.acm.org/doi/10.1145/115372.115320。[7] Anderson Faustino da Silva、Bruno Conde Kind、Jos´e Wesley de Souza Magalh˜aes、Jerˆonimo Nunes Rocha、Breno Campos Ferreira Guimar˜aes 和 Fernando Magno Quint˜ao Pereira。Anghabench：包含一百万个可编译C语言基准测试的套件，用于代码尺寸缩减。载于*CGO*，第378–390页，线上活动，韩国，2021年。IEEE 出版社。ISBN 9781728186139。doi: 10.1109/CGO51591.2021.9370322。URL https://doi.org/10.1109/CGO51591.2021.9370322。[8] Rafael Alvarenga de Azevedo。幂等后向切片：一种基于GSA的代码尺寸缩减方法。硕士论文，米纳斯吉拉斯联邦大学，2025年。[9] Jeanne Ferrante、Karl J. Ottenstein 和 Joe D. Warren。程序依赖图及其在优化中的应用。*ACM*《编程语言与系统汇刊》，9(3):319–349，1987年。ISSN 0164-0925。doi: 10.1145/24039.24041。[10] Breno Campos Ferreira Guimar˜aes 和 Fernando Magno Quint˜ao Pereira。为惰性求值而进行的惰性求值：自动将值调用转换为按需调用。载于第*32*届*ACM SIGPLAN*编译器构造国际会议论文集，第239–249页，加拿大魁北克省蒙特利尔，2023年2月。ACM。ISBN 9798400700880。doi: 10.1145/3578360.3580270。URL https://dl.acm.org/doi/ 10.1145/3578360.3580270。[11] Yann Herklotz、Delphine Demange 和 Sandrine Blazy。门控静态单赋值的机械化语义。载于第*12*届*ACM SIGPLAN*认证程序与证明国际会议论文集，第182–196页，美国马萨诸塞州波士顿，2023年1月。ACM。ISBN 9798400700262。doi: 10.1145/ 3573105.3575681。URL https://dl.acm.org/doi/10.1145/3573105.3575681。[12] Markus M¨uller-Olm 和 Helmut Seidl。并行程序的最优切片。载于*STOC*，STOC '01，第647–656页，美国纽约州纽约市，2001年。计算机协会。ISBN 1581133499。doi: 10.1145/380752.380864。URL https://doi.org/10.1145/380752.380864。[13] Karl J. Ottenstein、Robert A. Ballance 和 Arthur B. MacCabe。程序依赖网：一种支持对命令式语言进行控制、数据和需求驱动解释的表示。载于*ACM SIGPLAN 1990*年编程语言设计与实现会议论文集 - *PLDI '90*，第257–271页，美国纽约州怀特普莱恩斯，1990年。ACM出版社。ISBN 9780897913645。doi: 10.1145/93542.93578。URL http://portal.acm.org/citation.cfm?doid=93542.93578。[14] River Riddle。用于缩减代码大小的过程间IR提取。载于*LLVM*开发者会议，第9页，美国佐治亚州亚特兰大，2017年10月。LLVM开发者会议。URL https://llvm.org/devmtg/2017-10/slides/ Riddle-Interprocedural%20IR%20Outlining%20For%20Code%20Size.pdf。幻灯片。[15] Rodrigo C. O. Rocha、Pavlos Petoumenos、Zheng Wang、Murray Cole 和 Hugh Leather。SSA形式中的有效函数合并。载于第*41*届*ACM SIGPLAN*编程语言设计与实现会议论文集，第854–868页，英国伦敦，2020年6月。ACM。ISBN 9781450376136。doi: 10.1145/3385412.3386030。URL https://dl.acm.org/doi/10.1145/3385412. 3386030。[16] Rodrigo C. O. Rocha、Pavlos Petoumenos、Zheng Wang、Murray Cole、Kim Hazelwood 和 Hugh Leather。HyFM：免费的函数合并。载于第*22*届*ACM SIGPLAN/SIGBED*嵌入式系统语言、编译器与工具国际会议论文集，第110–121页，线上加拿大，2021年6月。ACM。ISBN 9781450384728。doi: 10.1145/3461648.3463852。URL https: //dl.acm.org/doi/10.1145/3461648.3463852。

[17] Bruno Rodrigues、Fernando Magno Quint˜ao Pereira 和 Diego F. Aranha。隐式流的稀疏表示及其在侧信道检测中的应用。载于第*25*届国际会议论文集

编译器构造会议，第110–120页，西班牙巴塞罗那，2016年3月。ACM。ISBN 9781450342414。doi: 10.1145/2892208.2892230。URL https://dl.acm.org/doi/10.1145/2892208. 2892230.[18] B. K. Rosen、M. N. Wegman 和 F. K. Zadeck。全局值编号与冗余计算。载于*POPL*，第12–27页，美国纽约州纽约市，1988年。计算机协会。ISBN 0897912527。doi: 10.1145/73560.73562。URL https://doi.org/10.1145/73560/73562.[19] Anderson Faustino da Silva、Bernardo N. B. de Lima 和 Fernando Magno Quint˜ao Pereira。探索用于代码尺寸缩减的优化序列空间：见解与工具。载于*CC*，第47–58页，美国纽约州纽约市，2021年。计算机协会。ISBN 9781450383257。doi: 10.1145/3446804.3446849。URL https://doi.org/10.1145/3446804.3446849.[20] Vugranam C. Sreedhar、Roy Dz-Ching Ju、David M. Gillies 和 Vatsa Santhanam。将程序转换出静态单赋值形式。载于第六届国际静态分析研讨会论文集，SAS '99，第194–210页，柏林、海德堡，1999年。施普林格出版社。ISBN 3540664599.[21] Frank Tip。程序切片技术综述。技术报告，IBM，1994年。[22] Peng Tu 和 David Padua。门控函数的高效构建与放置。载于*ACM SIGPLAN 1995*年编程语言设计与实现会议，第47–55页，美国加利福尼亚州拉霍亚，1995年6月。ACM。ISBN 9780897916974。doi: 10.1145/207110.207115。URL https://dl.acm.org/doi/10.1145/207110.207115.[23] Gregor Wagner、Andreas Gal 和 Michael Franz。通过冷代码移除和乐观式部分程序加载“精简”Java虚拟机。科学计算程序，76(11):1037–1053，2011年11月。ISSN 0167-6423。doi: 10.1016/j.scico.2010.04.008。URL https://doi.org/10.1016/j.scico. 2010.04.008.[24] Mark Weiser。程序切片。载于*ICSE*，第439–449页，美国加利福尼亚州圣迭戈，1981年。IEEE 出版社。ISBN 0897911466。