

这是我论文的主体部分翻译为中文后的结果。所以，可能会比较正式一点，行文也没有那么自然。

1. 背景知识

本章介绍本论文涉及的理论与实际工具基础。我们首先介绍多面体分析的核心概念，然后展示它们如何在缓存建模语境下应用。随后我们讨论软件工程的相关方面，包括所用工具、需求分析和项目的起点。

1.1. 多面体分析

多面体分析聚焦于**嵌套的仿射循环**。这些循环的循环变量称作**归纳变量**，它们从某个下界迭代至上界，每一步按固定常数递增。典型的仿射循环形式如下：

```
for i in lower..upper, step c
    ...
```

在多面体分析中，循环界必须为其外围循环归纳变量的**仿射表达式**。变量 $x_1, x_2, \dots$ 的仿射表达式即它们的线性组合加一个常数项，即 $a_1x_1 + a_2x_2 + \dots + c$ ，其中 a_i 为编译期常数。

循环迭代由其外围循环归纳变量的取值唯一标识。这些值组成一个向量，称为**归纳向量**，按从最外层到最内层排序。它的取值范围称为**迭代域**，记为 $\mathcal{D}$ 。

作为例子，考虑如下循环嵌套：

```
for i in 0..10:
    for j in 0..i:
        a[i][j] *= 2    # S
```

语句 S 的归纳向量可表达为 $\mathbf{x} = (i, j)$ ，其迭代域为 $\mathcal{D} = \{(i, j) \mid 0 \leq i \leq 10, 0 \leq j \leq i\}$ 。

在不同迭代中，语句 S 访问不同的内存地址。为加以区分，我们用 $S(\mathbf{x})$ 表示在第 $\mathbf{x} = (i, j)$ 次迭代中执行的 S 实例，用 $S[\mathbf{x}]$ 表示 S 在迭代 $\mathbf{x}$ 中访问的地址。在上面的例子中， $S[\mathbf{x}] = \&a[i][j]$ 。

1.1.1. 计算依赖关系

考虑缓存行为时，我们将语句 $S(\mathbf{x})$ 视为依赖于先前语句 $T(\mathbf{y})$ ，当且仅当二者访问同一地址¹。现代缓存通常为 write-back 模式，因此我们不关心 S 和 T 是读还是写，因为任一操作都会把被访问数据引入缓存。

形式上，当以下条件成立时， $S(\mathbf{x})$ 依赖 $T(\mathbf{y})$ ：

1. $\mathbf{x}$ 和 $\mathbf{y}$ 分别位于迭代域 $\mathcal{D}_S$ 和 $\mathcal{D}_T$ 内；
2. $S[\mathbf{x}] = T[\mathbf{y}]$ ；
3. $T(\mathbf{y})$ 在 $S(\mathbf{x})$ 之前执行。

当 S 和 T 位于同一循环嵌套中，第三个条件等价于 $\mathbf{x} >_{\text{lex}} \mathbf{y}$ ，其中 $>_{\text{lex}}$ 是字典序比较。然而当 S 和 T 共享全部外围循环，但不包含彼此时，其含义便不再等价于 $\mathbf{x} >_{\text{lex}} \mathbf{y}$ 。复杂情形在 Section 2.4.1 中详细说明。

该定义将 $S(\mathbf{x})$ 与所有访问同一地址的 $T(\mathbf{y})$ 相关联，但对我们使用场景而言，只需取最近的一个。该实例为字典序最大的 $T(\mathbf{y})$ ，记为 $\text{deps}(S(\mathbf{x}))$ 。

$S(\mathbf{x})$ 也可能不依赖任何 $T(\mathbf{y})$ 。此时我们认为 $\text{deps}(S(\mathbf{x}))$ 未定义。

¹更严格地说，只需 $S(\mathbf{x})$ 与 $T(\mathbf{y})$ 访问同一缓存行即可。本项目的扩展考虑了这一因素，如 Section 2.4.4 所述。

```

for i in 0..7:
    load a[i] # T
for i in 0..7:
    load b[i]
for j in 6..2:
    load a[j] # S

```

Depend when $j = i$

Figure 1: 依赖关系的例子。

例如，在 Figure 1 中，当且仅当 $j = i$ 时，语句 $S(j)$ 依赖于语句 $T(i)$ ，故 $\text{deps}(S(j)) = T(j)$ 。另一方面， $T(i)$ 不依赖任何其他语句，故 $\text{deps}(T(i))$ 处处未定义。

1.1.2. 复用距离

$S(\mathbf{x})$ 的复用距离是 $\text{deps}(S(\mathbf{x}))$ 与 $S(\mathbf{x})$ 之间的各语句所访问的不同地址个数，记为 $\text{dist}(S(\mathbf{x}))$ 。换言之，若假设每条缓存行只存放一个元素，这就是 $S[\mathbf{x}]$ 与其上次使用之间被逐出的缓存行数量。

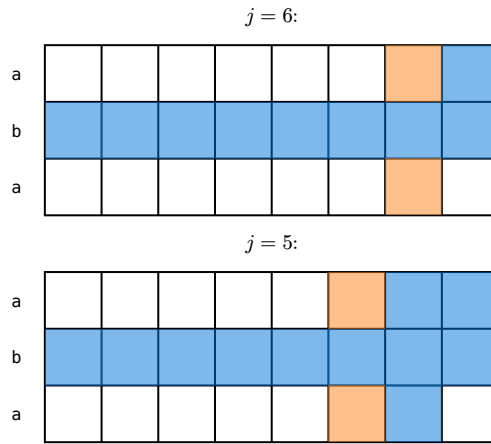


Figure 2: Figure 1 中，不同的 j 访问缓存行的数目。

考虑 Figure 1 中的程序。 $\text{dist}(S(6))$ 的值为 9， $\text{dist}(S(5))$ 为 10。Figure 2 以图形方式展示了该情形，其中橙色方块表示 $\text{load } S(j)$ 及其依赖 $T(j)$ ，蓝色方块代表它们之间访问的缓存行。注意当 $j = 5$ 时，地址 $a[6]$ 被访问两次，但由于我们只关心不同的地址，它只被计算一次。

1.1.3. 缓存缺失计数

我们参考的是缓存模型 Falcon 的设置，假设缓存是全关联的、采用 LRU 替换策略，且每个数组元素占用整条缓存行。我们用 C 表示缓存容量，并逐个考虑每个语句 S 。

S 产生的 compulsory miss 数量，等于 $\text{deps}(S(\mathbf{x}))$ 未定义的归纳向量 $\mathbf{x}$ 的个数。

S 引起的 capacity miss 数量，为满足 $\text{dist}(S(\mathbf{x})) \geq C$ 的 $\mathbf{x}$ 个数。这个条件即 $S[\mathbf{x}]$ 上次访问后，每条缓存行均已被逐出。

为高效统计满足上述约束的 $\mathbf{x}$ ，我们使用 Barvinok 算法，下面将介绍该算法。

1.2. Barvinok 算法

Barvinok 算法 [1] 用于计算多面体内的整点个数。

1.2.1. 多面体

n 维多面体 P 可视为 n 个变量上的不等式集合，每个不等式定义了整个空间 $\mathbb{R}^n$ 的一半。多面体就是所有这些半空间的交集。该表示称为 **H 表示**，其中 ‘H’ 代表 *halfspace* (半空间)。

形式上，多面体 P 的 H 表示由矩阵 A 与向量 $\mathbf{b}$ 定义，满足 $P = \{\mathbf{x} \in \mathbb{R}^n \mid A\mathbf{x} + \mathbf{b} \geq 0\}$ 。除非另有说明，我们只考虑 A 、 $\mathbf{b}$ 、 $\mathbf{x}$ 每个元素均为整数的多面体。

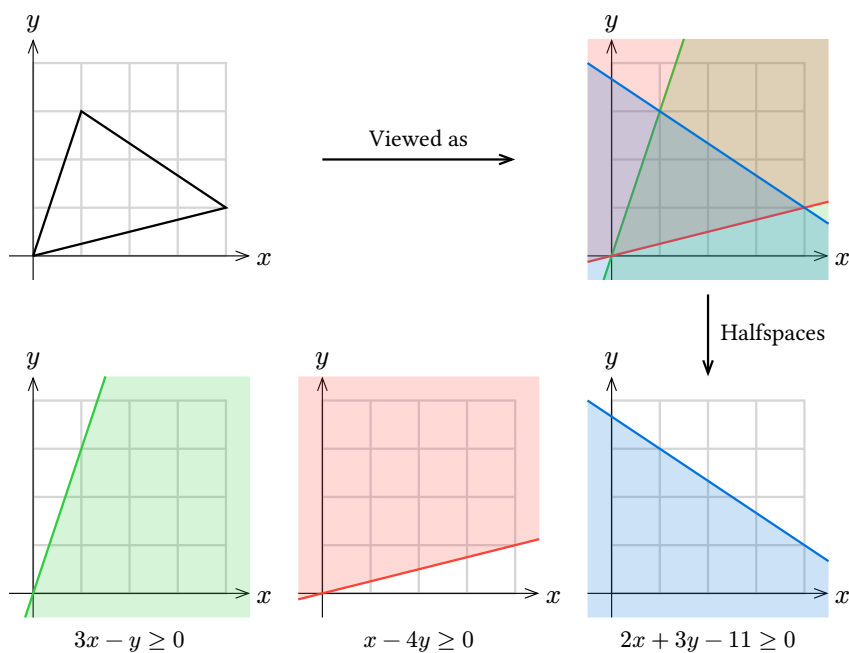


Figure 3: A 2-dimensional polyhedron and its halfspaces.

例如，考虑 Figure 3 所示的三角形，它是一个二维多面体。它可由 3 个不等式表示：

$$\begin{aligned} 3x - y &\geq 0 \\ x - 4y &\geq 0 \\ 2x + 3y - 11 &\geq 0 \end{aligned}$$

各颜色的部分表示由每个不等式标出的半空间。

在每个 n 维多面体的顶点 v 处，至少有 n 个不等式变为等式。称这些不等式在 v 处**活跃**。在上面的例子中，前两个不等式在顶点 $(0, 0)$ 处活跃。

对多面体 P 及其顶点 v ， v 处活跃的不等式构成另一个多面体，事实上这是一个锥。我们称其为 v 处的**切锥**。

1.2.2. 锥

锥是多面体的特殊情形。它们由从原点射出的射线构成。

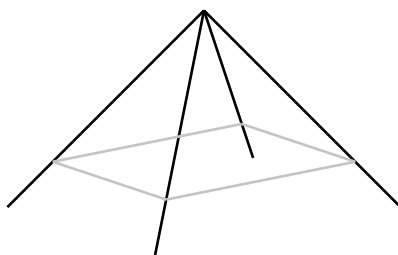


Figure 4: An example of a 3-dimensional cone (black part).

形式上，锥 C 可用一组向量 $\{\mathbf{v}_i\}$ 描述，其中 $C = \{\sum_i \lambda_i \mathbf{v}_i \mid \lambda_i \geq 0\}$ 。这些向量 $\mathbf{v}_i$ 称为**生成元**。该描述方法称为锥的**V 表示**。

锥 C 的生成元可组成一个矩阵，每个生成元占一列：

$$V = [\mathbf{v}_1 \quad \mathbf{v}_2 \quad \dots \quad \mathbf{v}_n]$$

当 V 为方阵且 V 的行列式为 ± 1 时，称锥 C 是**素模**的。

H 表示下的 $C^\circ = \{\mathbf{x} \mid V\mathbf{x} \geq 0\}$ 称为 C 的**对偶**。

1.2.3. 算法概述

Figure 5 展示了 Barvinok 算法的总体结构。为获得图中多面体的整数计数 10，上述步骤需对其他所有切锥重复执行。

图中绿色箭头指代我们在 FPL 中实现的步骤。红色箭头指代当前实现中已修复的 bug，黑色箭头指代早已完备、我们未加修改即使用其功能。

该算法仅适用于满维多面体。对于低维多面体 P ，需要一个预处理步骤将其投影（project）到具有相同整数点数量的满维多面体 Q 。

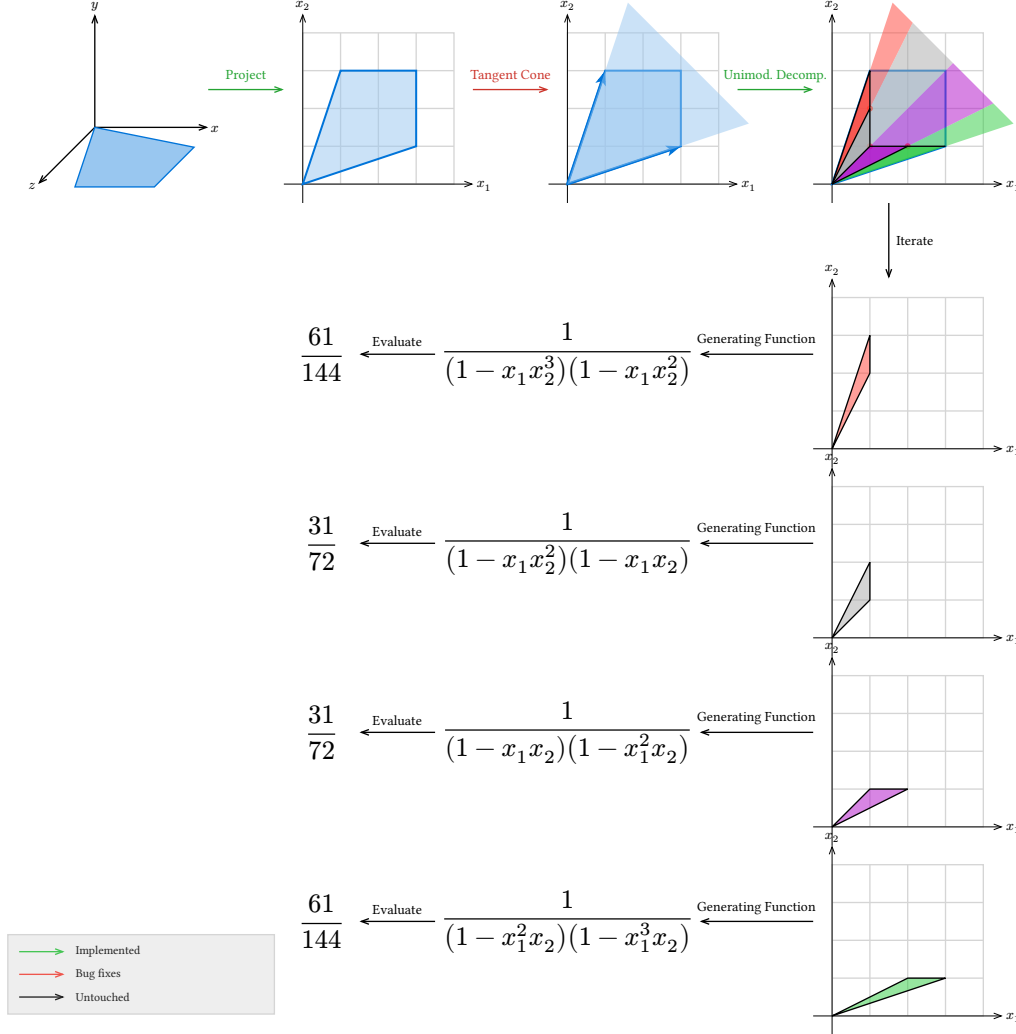


Figure 5: An overview of Barvinok's algorithm.

Barvinok 算法首先识别 Q 每个顶点 $\mathbf{u} = (u_1, \dots, u_n)$ 处的切锥 C 。然后它将 C° 分解为素模锥 U_i 的带符号求和。Figure 5 则直接对原始锥 C 进行分解，这同样是合法的，但此时分解中产生的低维锥同样需要加以考虑 [2]。

对于每个具有 m 个生成元 $\mathbf{v}_1, \dots, \mathbf{v}_m$ 的 n 维素模锥 U_i ，算法随后推导一个含 n 个变量 $x_1, \dots, x_n$ 的有理函数，称为**生成函数**。该函数形式为

$$\frac{x_i^{u_i}}{\prod_{j=1}^m (1 - \prod_{i=1}^n x_i^{\mathbf{v}_{ji}})}$$

最后，算法求该函数在 $\mathbf{x} = 1$ 处的极限，并将各结果求和以得到整数计数。

1.3. Fast Presburger Library

我们介绍项目中使用的数学库，即 Fast Presburger Library [3]，简称 FPL。

FPL 处理**整数关系**，即 Presburger 算术中仿射表达式的合取。等价地，整数关系也可视为整数多面体 $\{\mathbf{x} \mid A\mathbf{x} + \mathbf{b} = 0, C\mathbf{x} + \mathbf{d} \geq 0\}$ 的 H 表示。

FPL 还支持 **Presburger 关系**，即整数关系的有限并。该并的每个分量称为一个**析取分量**。

FPL 提供如下原语：

- 并与交；
- 集合之差；
- 空判定。

该库还提供各种辅助数学工具，如整数矩阵的规范形计算。后续各节将突出说明我们在实现中使用的这些 FPL 过程。

整数关系上的运算基于 Simplex 算法 [4] 的一种变体实现。通过开放对 Simplex 对象的运算，FPL 还提供了整数线性规划的功能。具体而言，FPL 可计算 $\mathbf{k} \cdot \mathbf{x} + b$ 的极小值和极大值，其中 $\mathbf{x}$ 属于整数关系 P 。

1.3.1. 变量类型

FPL 区分四种变量类型：

- **Domain**（定义域）变量，表示关系的输入；
- **Range**（值域）变量，表示关系的输出；
- **Symbol**（符号）变量，表示未知但固定的符号常数；
- **Local**（局部）变量，被存在量词修饰。

这种区分促使我们在库中将整数关系视为**关系**而非**多面体**。该设计使语义解释更为清晰，并便于关系的复合，后者在我们缓存分析 pass 中频繁使用。

例如，给定符号常数 N ，函数 $y = \lfloor \frac{x}{2} \rfloor + N$ 可表示为如下关系：

$$\{(x, y) \in \mathbb{Z}^2 \mid \exists e \in \mathbb{Z}. y = e + N, x \leq 2e \leq x + 1\}$$

在该关系中， x 是 domain 变量， y 是 range 变量， N 是 symbol 变量， e 是局部变量。该例子也展示了通过引入额外局部变量表示整数常系数除法的方法。

对 Presburger 关系，局部变量被限制在对其各析取分量中。一个析取分量无法访问在另一个析取分量中定义的局部变量。

1.3.2. 与原设计之偏离

FPL 的原始设计 [3] 采用**精度渐进策略**。计算从 16 位整数开始，发现溢出时提升到 64 位，再提升到任意精度整数。这种方式对缓存更友好，并能在常见情形下使用 SIMD 指令。

然而，MLIR 内 FPL 的上游实现并不包含此功能。该库中以任意精度整数类型 `DynamicAPIInt` 表示所有整数。这一设计选择由两个原因驱动：

- 精度切换依赖 C++ 异常，而 LLVM 项目禁用了异常。
- SIMD intrinsics 不适合跨平台的可移植代码库。

因此，MLIR 版本的 FPL 优化较少，性能可能劣于 *isl*。但由于该库与 MLIR 深度集成，它仍然是我们项目的理想选择。

1.4. Clang IR

项目基于 MLIR (*Multi-Level Intermediate Representation*) 实现，它是一种编译基础设施框架，支持自定义定义不同形式的中间表示 (IR)。这些 IR 称为 *dialect* (方言)。

我们聚焦 Clang IR——一种为 clang C++ 编译器设计的 MLIR 方言，常简记为 `cir`。Clang IR 项目还提供前端，将 clang 生成的抽象语法树 (AST) 转换为 `cir` 形式的 MLIR 模块。

1.4.1. 结构特征

Clang IR 是静态单赋值 (SSA) 形式, 且具有结构化控制流。

与 LLVM IR 以基本块间分支表示控制流、依赖分析恢复高级构造不同, Clang IR 通过携带**区域**的运算显式编码构造。例如, while 循环的典型形式为

```
cir.while %cond {  
  loop body  
}
```

一个 region 是基本块的容器, 一个基本块包含一个运算列表。运算也可以拥有嵌套的 region。这种相互递归的设计在 Clang IR 中构成树形结构, 其根是 `cir.module` 运算。

显式控制流运算简化了我们高度依赖循环结构的分析, 因为不再需要支配分析来恢复循环结构。

1.4.2. φ 节点

在 SSA 中, φ 节点用于在控制流汇合点选择被执行的前驱块所提供的值。它们通常作为伪指令放置在汇合块开头。

在 MLIR 大多数方言中, φ 节点的结果以**基本块参数**的形式编码, φ 节点的操作则被看作结束各前驱块的 branch 指令的操作。

然而 Clang IR 目前未采用上述任一种 φ 节点形式。对局部变量的读写被翻译为对栈地址的 load 和 store。因此, 值合并通过内存状态隐式表示, 而非在 SSA 中显式表示。

这使得识别循环的归纳变量更加困难, 也使 Clang IR 需要预处理 pass, 其中之一是分析常见的 for 循环模式。我们将在 Section 2.3.1 中更详细地讨论 IR 模式。

1.5. 需求分析

按项目提案, 本项目只有一个交付物: 一个静态分析 pass, 使用多面体模型估计仿射程序的缓存缺失。

1.5.1. 问题范围

我们的分析仅限于仿射程序, 遵循多面体分析中的标准定义:

- 循环界在编译期已知;
- 数组下标是归纳变量的仿射表达式;
- 控制流仅依赖归纳变量, 不依赖运行期数据。

这些限制确保程序可用多面体模型表示。

1.5.2. 实现策略

基于 Section 1.1.3 中描述的缓存建模算法, 项目可分为两部分: 实现多面体缓存模型, 以及完成 Barvinok 算法。

虽然 Barvinok 算法在概念上独立于缓存模型, 但它依赖我们的缓存模型生成测试用例, 以验证其正确性并检查边界情形的覆盖度。基于上述理由, 两个组件以交替方式开发, 缓存模型优先。

为指引开发, 我们构建了如图 Figure 6 所示的依赖图, 蓝色节点对应模型各阶段, 绿色节点对应 Barvinok 算法的组件。Section 2.5 详细解释了每个组件。

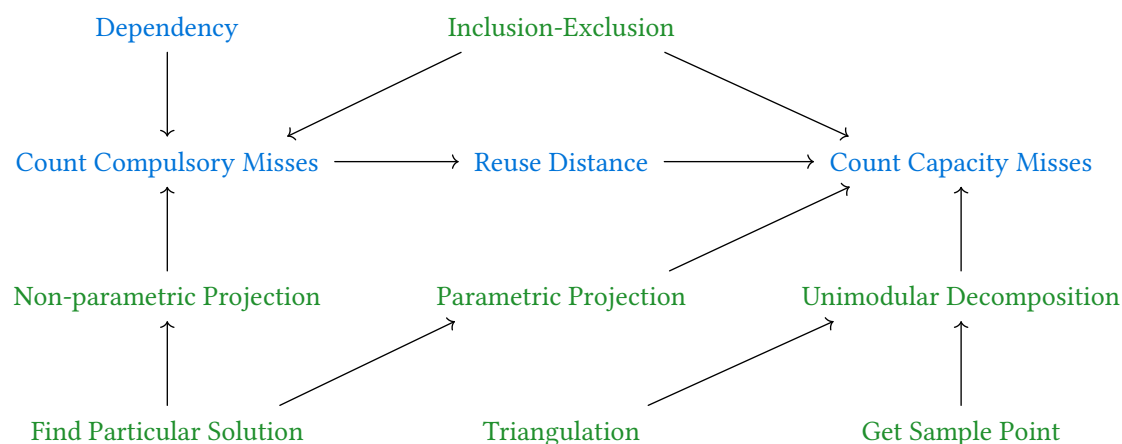


Figure 6: Dependency graph.

我们的开发推进方式为反复选择需求已满足的节点，类似图的拓扑排序。由于缓存模型阶段为计数算法提供测试用例，故被优先。

1.5.3. 验证策略

正确性以增量方式检查。每个缓存模型阶段实现后，部分完成的分析被应用于 6 个小测试程序。中间结果随后与手工计算或缓存模拟器得到的期望结果比较。

实现完成后，系统在另外 28 个程序上评估，以检验正确性、精度与效率。评估细节见 Section 3。

1.5.4. 库与工具

调试与测试高度依赖 LLVM 基础设施。特别是 LLVM 的言机制被广泛使用，用于验证数学过程的前置条件与后置条件。Barvinok 算法及 FPL 中现有方法的单元测试与回归测试使用 `llvm-lit`——LLVM 内的测试工具——来开展。

由于带调试信息的 LLVM 编译内存与时间开销大，GNU 调试工具 `gdb` 仅偶尔使用。

静态分析工具如 `clang-tidy` 和 `clangd` 用于代码 lint 与维护代码质量。版本管理使用 `git`，仓库定期备份到我们 GitHub 上的 Clang IR fork。

1.6. 起点

项目开始之前，我已熟悉 C++、编译器内部机制、MLIR 框架及 Clang IR 仓库。我在 Part IB 团队项目中积累的 Clang IR 经验大有裨益。暑假期间，我研读了多面体分析涉及的数学知识，并学习了 Barvinok 算法。

项目基于 Clang IR git commit `e4003cd3a263`。该 commit 于 2024 年 11 月 10 日合并，但我此前未为本项目编写任何代码，直到 2025 年 10 月。项目期间未再从主分支 pull。

2. 具体实现

2.1. 工作流程概述

我们的分析基于 Clang IR 项目——它既是 C 前端，也是一个 MLIR 方言。其工作流程总结在 Figure 7 中。

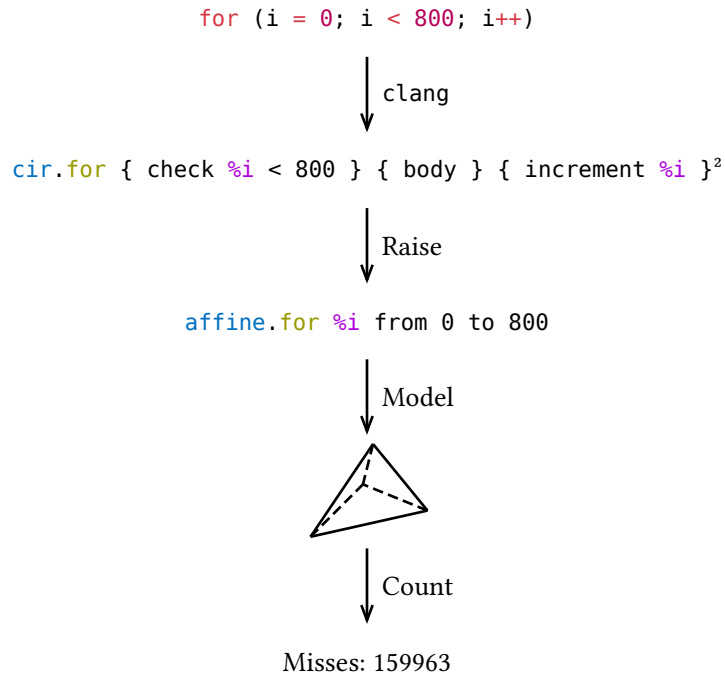


Figure 7: Workflow.

我们从 Clang IR 出发，将其部分提升为 Affine IR。该步骤识别归纳变量及其迭代域，为后续分析做准备。

在提升后的 IR 中，我们将导致缓存缺失的数组下标建模为多面体中的整数点。这通过分析循环内部的数组访问运算实现：计算下标的符号表示，并构造 Presburger 关系。

最终，我们对多面体应用 Barvinok 算法计数其中的整数点，从而得到估计的缓存缺失次数。

2.2. 仓库概述

Figure 8 概述了我们对 Clang IR 仓库的修改。为简洁起见，图中省略了开发用 shell 脚本和小于十行的微小改动。新引入的仓库文件在图中以蓝色高亮。

我们的修改与上一节描述的三阶段工作流相绑定。代码结构按阶段清晰分离，各阶段位于独立的源文件中。

²Abbreviated. See Section 2.3.1 for full details.

clang/include/clang/CIR/Dialect		
FalconPasses.td	+47	-0
clang/include/clang/CIR/Dialect/IR		
CIRFalconAttrs.td	+31	-0
CIRFalconOps.td	+28	-0
clang/lib/CIR/Dialect/IR		
CIRDialect.cpp	+30	-2
clang/lib/CIR/Dialect/Transforms		
ConstFold.cpp	+120	-0
clang/lib/CIR/Dialect/Transforms/Falcon		
ComputeDeps.cpp	+1945	-0
FalconUtilities.h	+31	-0
FindOffset.cpp	+235	-0
PassDetail.h	+30	-0
Profiler.h	+113	-0
RaiseToAffine.cpp	+260	-0
RevertLICM.cpp	+48	-0
mlir/include/mlir/Analysis/Presburger		
Barvinok.h	+27	-2
IntegerRelation.h	+14	-6
Matrix.h	+34	-1
QuasiPolynomial.h	+17	-3
mlir/include/mlir/Support		
Timing.h	+19	-19
mlir/lib/Analysis/Presburger		
Barvinok.cpp	+910	-58
IntegerRelation.cpp	+59	-9
Matrix.cpp	+122	-5
PWMAFunction.cpp	+14	-0
PresburgerRelation.cpp	+23	-9
QuasiPolynomial.cpp	+161	-7
mlir/unittests/Analysis/Presburger		
BarvinokTest.cpp	+58	-1
IntegerRelationTest.cpp	+82	-0

Figure 8: Repository overview.

在文件 `RaiseToAffine.cpp` 中，我们将 Clang IR 提升为 Affine IR，如 Section 2.3.1 所述。它识别 C 语言中常见的 for 循环模式，并翻译为仿射循环。同时生成胶水 IR，以将两种不同方言集成到同一 MLIR 模块中。

`RevertLICM.cpp`、`FindOffset.cpp` 和 `ComputeDeps.cpp` 中的 pass 按此顺序执行，以将数组下标间的关系建模为整数多面体。`RevertLICM.cpp` 重构 IR 以简化 `FindOffset.cpp` 的执行，后者用分析所得的下标注释相关运算。最后一个 pass `ComputeDeps.cpp` 将数组下标上的关系翻译为多面体。

计数步骤实现于 `Barvinok.cpp`，它消费建模步骤中的多面体，并给出缓存缺失次数估计。我们基于 MLIR 中现有的未完成工作，在该文件中完成了 Barvinok 算法。

除这些核心组件外，还修改了 Clang IR 仓库中已有的文件，以便将我们的 pass 整合进 MLIR 框架并收集性能数据。

总体而言，我们的仓库体现了需求分析 (Section 1.5) 中识别的模块化分解，并在此基础上扩展以应对项目复杂度。

2.3. 预处理 pass

2.3.1. IR 提升

在这一阶段，我们将 Clang IR 中的仿射循环提升为 MLIR 的 affine 方言，使后续 pass 能够分析循环结构及其归纳变量。

我们关注如下经典 C 循环结构。

```
for (i = a; i < n; i++)
```

在 Clang IR 中，每个全局变量都被赋予一个栈地址（一个 `cir.alloca` 运算）。循环被生成成为 `cir.for` 运算。该循环有三个 region，分别为：`cond` 对应循环条件 `i < n`，`body` 对应循环体，`step` 对应增量 `i++`。初始化语句 `i = a` 被放置在 `cir.for` 运算之前，因为它仅在进入循环前执行一次。

为了提升仿射循环，我们匹配 Figure 9 所示的模式，提取下界 `%a`、上界 `%n` 和归纳变量 `%i`。随后创建具有相同界与相同 body region 的 `affine.for` 运算。

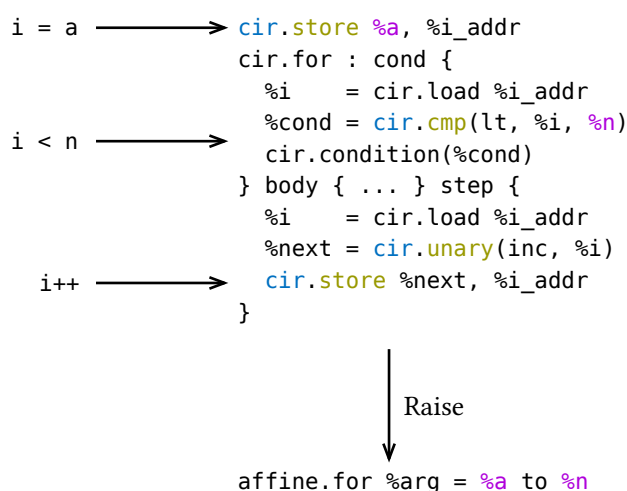


Figure 9: Clang IR and affine dialect for a `for`-loop in C.

一个挑战是 Clang IR 循环变量 `%i` 与 affine 方言循环变量 `%arg` 之间的类型不匹配。Clang IR 拥有自己的类型 `!cir.int<s, 32>`（带符号 32 位整数），而 affine 方言要求使用 MLIR 内置类型 `index` 作为循环变量。因此我们不能直接将 `%i` 的用途替换为 `%arg`。

为此，我们向 Clang IR 扩展了一个类型转换运算 `cir.from_index`，用于将 `index` 类型转换为所需的 Clang IR 整数类型，使两种 IR 能够互操作。类似地，我们引入了 `cir.index_cast` 运算，将 Clang IR 类型转换为 `index`。

我们还处理如下形式的反向循环

```
for (i = a; i >= n; i--)
```

将其规范化为等价的递增形式

```
for (j = n; j < a + 1; j++)
```

并将循环内 `i` 的全部用途替换为 `n + a - j - 1`。

规范化对于 Section 2.4.1 中引入的建模阶段的字典序比较的正确性至关重要。

2.3.2. 发现偏移与基地址

对数组 `b[i][j]` 的访问可分解为一个 *base*（基）`b` 和一个 *offset*（偏移）`(i, j)`。缓存模型需要我们找到二者：基用于区分不同数组，相当于一次基本的别名分析；偏移用于计算依赖关系。数组访问语句的偏移与基在 `FindOffset.cpp` 中的预处理 pass 中被识别。

访问 `b[i][j]` 在 Clang IR 中如下表示，其中多维访问被降为一系列 `cir.get_element` 运算³，每个作用于单一维度。

```
%1 = cir.get_element %b[%i]
%2 = cir.get_element %1[%j]
%result = cir.load %2
```

我们检查 `load` 运算操作数（类似地，`store` 的地址操作数）的定义运算，期望得到一个 `get_element` `%addr[%x]`。我们检查包围该运算的循环嵌套深度 d ，并构造仿射表达式

$$c + \sum a_i x_i$$

其中 c 和 a_i 初始设为零。

系数通过递归沿 `%x` 的 `use-def` 链向上遍历，沿途处理算术运算获得：

- 对于携带常数 v 的 `cir.int`，我们将 c 增加 v ；
- 对于对应 x_i 的值 `%i`，我们将 a_i 增加 1；
- 对于加减运算 `cir.binop` `add`（或 `sub`），我们将两个操作数的仿射表达式相加；
- 对于乘常数 m 的乘法 `cir.binop` `mul`，我们将所有系数与常数乘以 m 。

其他所有表达式都会失败，并被我们的分析 `pass` 拒绝，因为它们已不再仿射。

为找到基，`%addr` 也递归处理。如果 `%addr` 本身由一个 `get_element` 运算定义，我们就递归检查其地址，即沿地址链向上遍历，直到它是 `alloca` 或全局数组为止。

我们在 Figure 10 中给出一个示例以演示该过程。它是从 Polybench 的 `ludcmp.c` 获取的 Clang IR 适配片段。

```
affine.for %i = #0 to #120 {
  %53 = cir.from_index %i
  %55 = cir.sub #119, %53
  affine.for %j = %67 to %11 {
    %68 = cir.from_index %j
    %69 = cir.get_element %A[%55]
    %70 = cir.get_element %69[%68]
  }
}
```

Diagram annotations:

- A purple arrow points from `%A` to `%69` with the label "base: %A".
- A green arrow points from `%55` to `%69` with the label "dim 1: 119 - i".
- A blue arrow points from `%68` to `%70` with the label "dim 2: j".

Figure 10: Worked example of base and offset identification.

我们的分析确定 `%70` 以偏移 $(119 - i, j)$ 访问数组 `A`。基 `A` 通过查看地址操作数 `%69` 以及 `%A`（紫色部分）获得。由于 `get_element` 位于两个嵌套循环内，我们知道偏移是两个归纳变量的仿射函数。第二维的偏移由检查 `%68` 计算得到——它是一个直接引用 `j` 的 `from_index` 运算（蓝色部分）。第一维的偏移类似获得，在到达 `i` 之前额外经过一个 `sub` 表达式（绿色部分）。

该 `pass` 不处理经由任意指针的访问，因为那将需要别名分析，显著增加实现复杂度。

在 `pass` 之前，我们进行常量折叠与内联，以简化 IR 并减少对复杂跨过程分析的需求。

每个访问的基与偏移以附加在每次 `get_element` 运算上的属性形式表示。

对于基，我们为每个 `alloca` 或全局数组分配一个整数作为标识，并将 `array_base` 属性附加到 `get_element` 运算上。该属性是首次引入 Clang IR 的。我们不能直接在属性中存储运算或 SSA 值，因为它们在 MLIR 中不可序列化，会被拒绝。

类似地，我们为数组偏移添加了 `array_offset` 属性。该属性存储仿射映射（affine map），MLIR 原生支持。

³Strictly speaking, Clang IR can generate `cir.ptr_stride` operation as well. The operation is similar to `get_element`, and the process of finding offsets and bases is largely unchanged.

作为例子，pass 之后的 `get_element` 运算可能形如：

```
%17 = cir.get_element %5[%8] {
  #array_base = 1,
  #array_offset = (d0, d1) -> (d0)
}
```

二维 induction vector (d0, d1) 表示该运算嵌套于两个循环中，并以外层循环的归纳变量索引编号为 1 的数组。

2.3.3. 恢复 `get_element` 运算的局部性

如上一节所示，数组访问信息关联在 `get_element` 运算上，偏移仿射映射的维度由该运算所在的循环嵌套决定。

我们的分析期望 `get_element` 运算位于访问语句所在的同一循环嵌套中，以便引用外围循环的全部归纳变量。但并非总是如此。考虑 Figure 11, (a) 所示的 IR 结构：

<p>(a)</p> <pre>for i { %1 = cir.get_element for j { load %1 } load %1 }</pre>	<p>(b)</p> <pre>for i { for j { %1 = cir.get_element load %1 } %2 = cir.get_element load %2 }</pre>
--	---

Figure 11: IR before and after CSE.

load 嵌套于两个循环中，其 induction vector 为 (i, j) ，但其 `get_element` 仅嵌套于一个循环中，在我们的分析中认为其 induction vector 为 j 。维数不匹配违背了我们的假设。

在代码生成中，这种结构通常不会出现。`get_element` 运算紧邻于 load 或 store 之前，如 Figure 11, (b) 所示。该结构产生于内层 `get_element` 运算因循环不变量提取 (LICM) 被提升到循环之外。随后，公共子表达式消除 (CSE) 可能合并这样两个运算，使其与其使用场景进一步分离。

因此，解决该问题的最直接方案是在 Clang IR 中禁用 LICM 与 CSE。考虑到未来与 MLIR 的集成，我们还实现了 RevertLICM pass，通过克隆 `get_element` 运算使过程还原，确保每个 load 都直接由一个 `get_element` 前置。

2.4. 缓存缺失计数分析

2.4.1. 计算必要缺失

为计算必要缺失，我们识别其访问地址此前从未被访问过的语句实例集合。

我们首先识别所有数组访问语句的集合，记为 $\mathcal{L}$ 。当 S 是一个 `cir.load` 或 `cir.store` 运算，且其地址操作数为带有基与偏移属性的 `cir.get_element` 时，称其访问数组。

随后，我们按照语句在程序中出现顺序，给 $\mathcal{L}$ 中每个语句分配一个正整数，称为 *lexical index* (字典序索引)，以便利后续步骤的字典序比较。越早出现的语句获得较小的索引。 $S \in \mathcal{L}$ 的字典序索引记为 i_S 。

我们逐个考虑 $S \in \mathcal{L}$ 中的每个语句。在 Section 1.1.1 中，我们定义了 $S(x)$ 与另一访问运算 $T(y)$ 之间的依赖概念。我们枚举所有 $T \in \mathcal{L}$ ，并构造依赖关系：

$$\text{depend}(S, T) = \{(x, y) \mid S(x) \text{ depends on } T(y)\}$$

如 Section 1.1.1 所述，在我们的缓存模型中， S 依赖 T ，当二者访问同一地址且 T 在 S 之前执行时。

在 Section 1.3 中，我们指出 FPL 仅支持仿射约束。现在我们说明依赖条件确实可以用仿射约束表达。

边界条件以及 $S[\mathbf{x}] = T[\mathbf{y}]$ 的条件是仿射的，因为所有循环界与数组访问都被假设为仿射。

为确保 $S(\mathbf{x})$ 在 $T(\mathbf{y})$ 之后执行，只需要要求 $S(\mathbf{x}) >_{\text{lex}} T(\mathbf{y})$ ，因为所有循环均已如 Section 2.3.1 所述规范化为递增循环。设 S 与 T 的共同循环深度为 d 。当 S 和 T 位于同一循环嵌套中，我们按定义构造字典序比较：

$$S(\mathbf{x}) >_{\text{lex}} T(\mathbf{y}) \iff \bigvee_{j=0}^{d-1} \left(\mathbf{x}_j > \mathbf{y}_j \wedge \bigwedge_{i=0}^{j-1} (\mathbf{x}_i = \mathbf{y}_i) \right)$$

当并非如此时，我们在穷尽所有公共循环后退回字典序。仅当 T 字典序在 S 之前时，我们将上述关系与以下析取式取并：

$$\bigwedge_{i=0}^{d-1} \mathbf{x}_i = \mathbf{y}_i$$

我们需要找到 $\text{deps}(S(\mathbf{x}))$ ——在所有考察过的 T 中， $S(\mathbf{x})$ 最新、即字典序最大的依赖。由于它们可能位于不同深度、不同嵌套的循环中，它们的归纳变量无法直接比较。我们已说明如何比较两个实例 $S(\mathbf{x})$ 与 $T(\mathbf{y})$ 的执行顺序，但要更一般地比较候选，我们必须将向量嵌入一个公共空间，以同时编码字典序与循环结构。

设 n 为所有候选语句 T 中最大嵌套循环深度，并定义一个循环的字典序索引为其内部语句最小字典序索引。假设 $T(\mathbf{y})$ 的字典序索引为 i_T ，且嵌套于索引为 $i_0, \dots, i_m$ 的循环中。我们将 $\mathbf{y} = (y_0, \dots, y_m)$ 提升为

$$\mathbf{z} = \left(i_0, y_0, i_1, y_1, \dots, i_m, y_m, \underbrace{i_T, 0, \dots, i_T, 0}_{n-m \text{ pairs}}, i_T \right)$$

在嵌入中，每一对 i_m, y_m 同时编码了循环身份与对应归纳变量的取值。填充项 $i_T, 0$ 表示没有循环，用于维持不同深度语句之间的正确字典序。最后的 i_T 作为同一循环嵌套内语句的排序标准。

乍看之下，似乎将 $\mathbf{y}$ 提升为以下形式即可：

$$\mathbf{z} = (i_0, y_0, i_1, y_1, \dots, i_m, y_m, 0, 0, \dots, 0, 0, i_T)$$

然而这一变体对非完全嵌套的循环不成立，因为它缺少循环层级之外语句的字典序信息。考虑如下示例：

```
for i:
  for j:
    load a[i]    # T1: index = 1
    load a[j]    # T2: index = 2
```

在此情形下，若在 y_m 之后填充零，我们将得到

$$\mathbf{x}_{T_1} = (1, i, 1, j, 1)$$

$$\mathbf{x}_{T_2} = (1, i, 0, 0, 2)$$

当 i 取相同值时， T_1 应字典序位于 T_2 之前，但提升后的向量却暗示相反。

将全部依赖候选 $T(\mathbf{y})$ 提升进公共空间后，我们使用 FPL 计算字典序最大值以获得 $\text{deps}(S(\mathbf{x}))$ 。最大化过程在 $\mathbf{x}$ 的定义域上是分段的，因此 FPL 返回一组 *pieces*，即二元组 $(D, \mathbf{z}(\mathbf{x}))$ ，其中 $\mathbf{z}(\mathbf{x})$ 是作为 $\mathbf{x}$ 函数表达的提升后最大值， D 是 $\mathbf{z}(\mathbf{x})$ 有效的 $\mathbf{x}$ 取值范围。由于 i_T 唯一，提升过程可逆，我们可将任意 $\mathbf{z}(\mathbf{x})$ 降回 $T(\mathbf{y}(\mathbf{x}))$ 。

程序返回的每个 D 都对应一个存在依赖的区域。将 S 的迭代域减去所有这样的 D ，得到地址此前未被访问过的区域 U 。因此 U 的整数点个数即等于必要缺失数量，可用 Barvinok 算法计算。

2.4.2. 计算复用距离

复用距离 $\text{dist}(S(\mathbf{x}))$ 是严格位于 $\text{deps}(S(\mathbf{x}))$ 与 $S(\mathbf{x})$ 之间被访问的不同缓存行数量。由于 $\text{deps}(S(\mathbf{x}))$ 在 $\mathbf{x}$ 的迭代域上分段定义，我们必须枚举上一节中计算出的全部 piece ($D, \mathbf{z}(\mathbf{x})$) 以及降回的语句 $T(\mathbf{y})$ 。

第一步，我们收集可能在 $S(\mathbf{x})$ 与 $T(\mathbf{y})$ 之间执行的语句，即所有被包含在同时容纳 S 与 T 的最小公共 region 内的数组访问语句。这些语句不仅限于字典序索引介于 i_S 与 i_T 之间的语句，因为位于 S 之后但仍在 S 同区域内执行的语句也可能在 $S(\mathbf{x})$ 与 $T(\mathbf{y})$ 之间执行。对 T 也有类似观察。

每个收集的语句 U 关联一个 induction vector $\mathbf{w}$ 。我们寻找使 U 在 $T(\mathbf{y})$ 之后、 $S(\mathbf{x})$ 之前执行的 $\mathbf{w}$ 取值范围。这些点称为 *reuse instances* (复用实例)。每个这样的 $\mathbf{w}$ 对应 U 的一个在 $S(\mathbf{x})$ 与 $T(\mathbf{y})$ 之间访问内存的实例，当其映射到不同缓存行时，即对复用距离做出贡献。

形式上，

$$\text{reuseInstance}(U, S(\mathbf{x}), T(\mathbf{y})) = \{\mathbf{w} \mid \mathbf{y}(\mathbf{x}) <_{\text{lex}} \mathbf{w} <_{\text{lex}} \mathbf{x} \wedge \mathbf{w} \in \mathcal{D}_{\mathbf{w}}\}$$

这里 $\mathcal{D}_{\mathbf{w}}$ 是 $\mathbf{w}$ 的迭代域， $<_{\text{lex}}$ 表示字典序优先，定义见上一节。

为计算 U 访问的内存位置集合，我们将关系 $\text{reuseInstance}(U, S(\mathbf{x}), T(\mathbf{y}))$ 与 U 的访问函数（一个仿射映射）相复合。然后跨所有 U 取并，得到 $S(\mathbf{x})$ 与 $T(\mathbf{y})$ 之间访问的全部内存位置集合，记为 $\text{total}(S(\mathbf{x}), T(\mathbf{y}))$ 。

若假设缓存行大小 L 为 1，则有

$$\text{dist}(S(\mathbf{x})) = \#\text{total}(S(\mathbf{x}), T(\mathbf{y}))$$

由于 total 关系由仿射等式与不等式构成，可视为多面体的 H 表示。因此我们可用 Barvinok 算法计数其中的整数点。

该多面体对 $\mathbf{x}$ 是参数化的。此时 Barvinok 算法给出的计数不是单一整数，而是关于 $\mathbf{x}$ 的 *quasi-polynomial* (准多项式)，其形式为

$$\sum_{i=0}^n c_i \prod_{j=0}^m \lfloor \mathbf{k}_{ij} \cdot \mathbf{x} + b_{ij} \rfloor$$

其中 c_i 、 $\mathbf{k}_{ij}$ 与 b_{ij} 为算法计算的有理数。该形式对下一节引入的容量缺失计数至关重要。

2.4.3. 计算容量缺失

为获得容量 C 的缓存的容量缺失计数，我们逐个考虑 $S \in \mathcal{L}$ ，并统计满足 $\text{dist}(S(\mathbf{x})) \geq C$ 的 induction vector $\mathbf{x}$ 数量。

当准多项式 $\text{dist}(S(\mathbf{x}))$ 的每个乘积至多含一项时，条件 $\text{dist}(S(\mathbf{x})) \geq C$ 可转化为一个多面体，从而可用 Barvinok 算法高效求解。但更一般的情形下，甚至判定是否存在这样的 $\mathbf{x}$ 就是 NP-hard [5]。因此我们必须枚举 $\mathbf{x} = (x_1, \dots, x_n)$ 以获得精确计数。

我们按维度逐一枚举。给定 $x_1, \dots, x_i$ 的固定值，可借助 Simplex 算法确定 x_{i+1} 的取值范围 $[l, r]$ 。然后遍历其中每个值 $k \in [l, r]$ ，令 x_{i+1} 为 k ，并递归枚举 x_{i+2} 。

我们随时在 PWMA 函数被简化为每个乘积至多一项时停止递归过程，此时调用 Barvinok 算法。出于性能考虑，若剩余变量的可能取值不超过 10 个，则继续枚举而非调用更为复杂的 Barvinok 过程。

2.4.4. 支持可配置的缓存行大小

前述讨论中我们假设每个数组元素占用一整条缓存行。现在我们说明如何支持缓存行大小 $L > 1$ 。

我们以二维数组 $\mathbf{a}[\mathbf{N}][\mathbf{M}]$ 为例。数组访问 $\mathbf{a}[\mathbf{i}][\mathbf{j}]$ 等价于访问地址 $\&\mathbf{a} + \mathbf{i} * \mathbf{M} + \mathbf{j}$ ，其映射到缓存行 $\lfloor \frac{\mathbf{iM} + \mathbf{j}}{L} \rfloor$ 。特别地，当假设 M 是 L 的倍数时，我们只需将最后一维 j 替换为 $\lfloor \frac{j}{L} \rfloor$ 。

然而我们对 $L > 1$ 的支持是有限的。引入取整除法会引入 local 变量，导致生成函数变长，且枚举变得不可避免。即使对小型程序、 $L = 3$ ，我们的实现也无法在 600 秒内终止。

2.5. Barvinok 算法

我们已在 Figure 5 中给出了 Barvinok 算法的概述。本节说明我们实现的细节。

2.5.1. 非参数化投影

我们从一个非参数化的 d 维多面体（既有不等式又有等式）开始：

$$P = \{\mathbf{x} \in \mathbb{Z}^d \mid A\mathbf{x} = \mathbf{c}, D\mathbf{x} \leq \mathbf{f}\}$$

由于 Barvinok 算法作用于满维多面体，因此需要进行 *projection*（投影），以找到一个新的 d' 维多面体 Q ，使其整数点数量与 P 相同，且 Q 填满整个 $\mathbb{Z}^{d'}$ 。这意味着 Q 不能含任何等式。

$$Q = \{\mathbf{y} \in \mathbb{Z}^{d'} \mid D'\mathbf{y} \leq \mathbf{f}'\}$$

但其逆命题不成立：即便 Q 不含等式，也不意味着 Q 是满维的。例如，考虑 $\mathbb{Z}^2$ 中的如下多面体：

$$\begin{aligned} x - y &\geq 0 \\ -x + y &\geq 0 \end{aligned}$$

这显然表明了一个等式 $x = y$ ，故其位于 $\mathbb{Z}^2$ 的一个 1 维子空间内。

我们通过预处理 P 来防止这种情况。对每个不等式 $D_{i,:} \cdot \mathbf{x} \leq \mathbf{f}_i$ ，我们在约束 $A\mathbf{x} = \mathbf{c}$ 与 $D\mathbf{x} \leq \mathbf{f}$ 下计算 $D_{i,:} \cdot \mathbf{x}$ 的最小值。这由 FPL 提供的 Simplex 算法完成。当最小值等于 $\mathbf{f}_i$ 时，我们知道该不等式实为等式。

随后我们设法消除所有等式 $A\mathbf{x} = \mathbf{c}$ 。考虑 A 的 *null space*（零子空间），其定义为

$$\text{null}(A) = \{\mathbf{x} \mid A\mathbf{x} = 0\}$$

它是一个向量空间。我们寻找一个整数基，使其张成向量空间 $\text{null}(A) \cap \mathbb{Z}^m$ ： $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n$ 。它们构成一个矩阵：

$$V = [\mathbf{v}_1 \ \dots \ \mathbf{v}_n]$$

于是，方程组 $A\mathbf{x} = \mathbf{c}$ 的任意解 $\mathbf{x}$ 都可表示为

$$\mathbf{x} = V\mathbf{y} + \mathbf{x}_0$$

其中 $\mathbf{x}_0$ 为一个 *particular solution*（特解），即满足方程组的任意整数解。

$\mathbf{x}$ 与 $\mathbf{y}$ 之间存在双射。此外，当 $\mathbf{y}$ 是整数向量时， $\mathbf{x}$ 亦然。其逆也成立：我们有 $\mathbf{y} = V^{-1}(\mathbf{x} - \mathbf{x}_0)$ 。由于 V 素模而 V^{-1} 因此为整数矩阵，只要 $\mathbf{x}$ 是整数向量， $\mathbf{y}$ 就是整数向量。由于该双射保持整数性，我们可计数向量 $\mathbf{y}$ 而非向量 $\mathbf{x}$ ，计数结果不变。

若我们将多面体 P 内整数点个数记为 $\#P$ ，则有

$$\begin{aligned} \#P &= \#\{\mathbf{x} \in \mathbb{Z}^d \mid A\mathbf{x} = \mathbf{c}, D\mathbf{x} \leq \mathbf{f}\} \\ &= \#\{\mathbf{y} \in \mathbb{Z}^{d'} \mid D(V\mathbf{y} + \mathbf{x}_0) \leq \mathbf{f}\} \\ &= \#\left\{\mathbf{y} \in \mathbb{Z}^{d'} \mid \underbrace{(DV)}_{D'}\mathbf{y} \leq \underbrace{\mathbf{f} - D\mathbf{x}_0}_{\mathbf{f}'}\right\} \\ &\triangleq \#Q \end{aligned}$$

由此，我们获得投影得到的合适 Q ，其中 $D' = DV$ ， $\mathbf{f}' = \mathbf{f} - D\mathbf{x}_0$ 。下面说明如何寻找 $\text{null}(A)$ 的基与特解 $\mathbf{x}_0$ 。

2.5.2. 零子空间的整数基

寻找实值矩阵 $M \in \mathbb{R}^{n \times m}$ 零子空间基时，我们通常将 M 化为 *row-echelon form*（行阶梯形）。对整数矩阵 $A \in \mathbb{Z}^{n \times m}$ ，我们需要一个保持整数性的类似概念。这导出了 *Hermite normal form*（赫米特规范形）与 *unimodular matrices*（素模矩阵）。

当整数矩阵的行列式为 ± 1 时称其为 *unimodular*。素模矩阵的逆也是整数矩阵。这是因为

$$M^{-1} = \frac{\text{adj}(M)}{\det(M)}$$

其中 $\text{adj}(M)$ 是 M 的伴随矩阵（adjugate）。按定义它各元素均为整数：每个元素是 M 的子矩阵行列式。因此 M^{-1} 各元素亦为整数。Figure 12 给出了一个素模矩阵及其逆的实例。

$$M = \begin{bmatrix} -2 & 1 \\ 5 & -2 \end{bmatrix} \quad M^{-1} = \begin{bmatrix} -2 & -1 \\ -5 & -2 \end{bmatrix}$$

Figure 12: A unimodular matrix and its inverse.

整数矩阵 A 的 *Hermite normal form* 是满足如下性质的矩阵 H ：

- 存在一个素模矩阵 U ，使得 $H = UA$ ；
- 每行中第一个非零元素，称为 *pivot*（主元），若存在则为正；
- H 为上三角形，即主元下方的元素为零；
- 主元上方的元素为非负且严格小于主元。

Figure 13 给出了 Hermite normal form 的实例。

FPL 提供计算整数矩阵 M Hermite normal form 的过程。它同时返回 H 与 U 。

$$\begin{array}{c} \text{upper} \\ \text{triangular} \end{array} \begin{array}{c} \text{Pivot} \\ \downarrow \end{array} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 3 & 6 \\ 0 & 0 & 9 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} -1 & -1 & 0 & 0 \\ 3 & 2 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ -1 & 1 & 1 & -1 \end{bmatrix} \begin{bmatrix} 2 & 5 & 8 \\ 3 & 6 & 9 \\ 4 & 7 & 1 \\ 5 & 8 & 2 \end{bmatrix}$$

$\underbrace{\hspace{10em}}_H = \underbrace{\hspace{10em}}_U \underbrace{\hspace{10em}}_A$

Figure 13: An example of Hermite normal form.

对 $n \times m$ 矩阵 A ，当 $m \leq n$ 且 A 列满秩时，零子空间仅含一点，即零向量。故我们考虑 $m > n$ 的情形。

为寻找 $\text{null}(A) \cap \mathbb{Z}^m$ 的基，我们计算 A^T 的 Hermite normal form H 及其对应的素模矩阵 U 。于是 $UA^T = H$ ，故 $AU^T = H^T$ 。

由于 U 可逆，我们可知 $\text{rank}(A) = \text{rank}(H)$ ，记为 r 。由于 H 为上三角形，其末 $m - r$ 行必全为零。于是 U^T 末 $m - r$ 列 $\mathbf{u}_{r+1}, \dots, \mathbf{u}_m$ 为整数，且满足 $A\mathbf{u}_j = H_{:,j}^T = 0$ ，故位于零子空间。向量 $\mathbf{u}_{r+1}, \dots, \mathbf{u}_m$ 彼此线性无关，共 $m - r$ 个，等于零子空间的维数。故它们构成 $\text{null}(A) \cap \mathbb{Z}^m$ 的基。

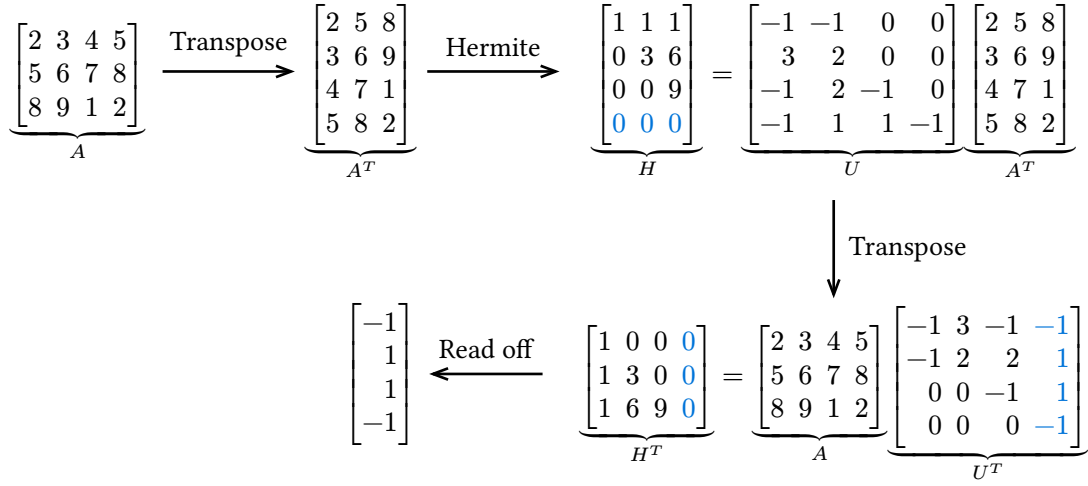


Figure 14: Finding an integer basis of $\text{null}(A)$.

Figure 14 以图形方式展示了该过程。在实际实现中我们只需第一次转置，因为读取 U^T 的列等价于读取 U 的行。

2.5.3. 寻找特解

虽然 Hermite normal form 能解方程组 $A\mathbf{x} = 0$ ，却不足以解 $A\mathbf{x} = \mathbf{c}$ 。因此我们转用 *Smith normal form* (史密斯规范形) 来寻找特解。

整数矩阵 A 的 *Smith normal form* 是一个矩阵 D ，满足如下性质：

- 存在两个素模矩阵 U 与 V ，使得 $D = UAV$ ；
- D 为对角矩阵；
- 对所有 i ，元素 $D_{i,i}$ 整除 $D_{i+1,i+1}$ 。

Figure 15 给出了 Smith normal form 的实例。

$$\xrightarrow{\text{Diagonal}} \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 9 \\ 0 & 0 & 0 \end{bmatrix}}_D = \underbrace{\begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 2 & -1 \\ 1 & 0 & 12 & 10 \\ -1 & 1 & 1 & -1 \end{bmatrix}}_U \underbrace{\begin{bmatrix} 2 & 5 & 8 \\ 3 & 6 & 9 \\ 4 & 7 & 1 \\ 5 & 8 & 2 \end{bmatrix}}_A \underbrace{\begin{bmatrix} 0 & 1 & -2 \\ 0 & 0 & 1 \\ 1 & -4 & 1 \end{bmatrix}}_V$$

Figure 15: An example of Smith normal form.

尽管 FPL 未提供计算 Smith normal form 的过程，但该计算相当标准且与 Hermite normal form 类似。我们将其加入到 FPL。

为寻找 $A\mathbf{x} = \mathbf{c}$ 的特解，我们首先求 Smith normal form $D = UAV$ 。然后方程可改写为

$$(U^{-1}DV^{-1})\mathbf{x} = \mathbf{c}$$

令 $\mathbf{y} = V^{-1}\mathbf{x}$ ，则 $D\mathbf{y} = U\mathbf{c}$ 。由于 D 为对角矩阵 $\text{diag}(d_1, \dots, d_n)$ ，我们可以轻易解出

$$d_i \mathbf{y}_i = (U\mathbf{c})_i$$

当 d_i 与 $(U\mathbf{c})_i$ 均为零时，我们令 $\mathbf{y}_i = 0$ ，任意值均可。

当 $d_i = 0$ 但 $(U\mathbf{c})_i \neq 0$ ，或 d_i 不整除 $(U\mathbf{c})_i$ 时，我们可断言无整数解，此时原多面体的整数计数为零。

则我们的特解为 $\mathbf{x} = V\mathbf{y}$ 。

2.5.4. 参数化投影

一个参数化多面体包含一个未知参数向量 $\mathbf{p}$ ，出现在其定义约束中。我们考虑如下形式的多面体：

$$P = \{\mathbf{x} \mid A\mathbf{x} = B\mathbf{p} + \mathbf{c}, D\mathbf{x} \leq E\mathbf{p} + \mathbf{f}\}$$

投影过程与非参数化情形相同。我们求 $\text{null}(A)$ 的基矩阵 V ，它不涉及参数。然后说明寻找 $A\mathbf{x} = B\mathbf{p} + \mathbf{c}$ 特解 $\mathbf{x}_0(\mathbf{p})$ 的方法。

我们首先按 Section 2.5.3 中的相同步骤进行。这将导出

$$d_i \mathbf{y}_i = (UB\mathbf{p} + U\mathbf{c})_i$$

不同于非参数化情形，此处我们不知道 d_i 是否整除分子。因此我们必须在 $\mathbf{p}$ 上施加约束以保证 $\mathbf{y}_i$ 的整数性。

有两大类约束。

当 $d_i = 0$ 时，这给出一个 *equality*（等式）约束，因为它强制 $\mathbf{p}$ 某些分量之间相等。此时 $\mathbf{y}_i$ 可取任意值，故将其设为零。

当 $d_i \neq 0$ 且它不整除 UB 与 $U\mathbf{c}$ 的每个分量时，这是一个 *modular*（模）约束：即 $U\mathbf{c} - UB\mathbf{p}$ 必须被 d_i 整除。为建模此类约束，我们引入一个 local 变量 e ，并将有效的 $\mathbf{p}$ 集合写为

$$\{\mathbf{p} \mid \exists e \in \mathbb{Z}, UB\mathbf{p} - d_i e + U\mathbf{c} = 0\}$$

$\mathbf{y}_i$ 的解将是 $\frac{UB\mathbf{p} + U\mathbf{c}}{d_i}$ ，因此含有分数系数。由于 FPL 提供任意精度分数与分数矩阵，这对我们的实现不构成问题。

此外，投影过程通过对等式两边乘以分母的最小公倍数（LCM），将此类分数约束转化为整数约束。由于 $\mathbf{p}$ 上的约束保证了 $\mathbf{y}_i$ 的整数性，该方法不会引入那些满足整数约束但不满足分数约束的额外 $\mathbf{y}_i$ 取值。

事实上，所有 $\mathbf{p}$ 上的约束共同构成一个带存在变量的另一个整数关系。该约束关系随后在整个计算中携带，并在 Section 2.5.9 的求和阶段中纳入考量。

2.5.5. 切锥识别

如 Section 1.2.2 所述，*tangent cones*（切锥）由顶点处的活跃不等式构成。我们对 FPL 中识别切锥的过程提出一个修正。

对含 m 个不等式的 n 维多面体，FPL 过程枚举所有 $\binom{m}{n}$ 个 n 个不等式的组合。对每组选择，FPL 随后计算一个使它们全部活跃的点。当该点在多面体内时，即为顶点，故由所选不等式构成的锥是切锥。

Figure 16 展示了该算法在二维多面体上的操作。

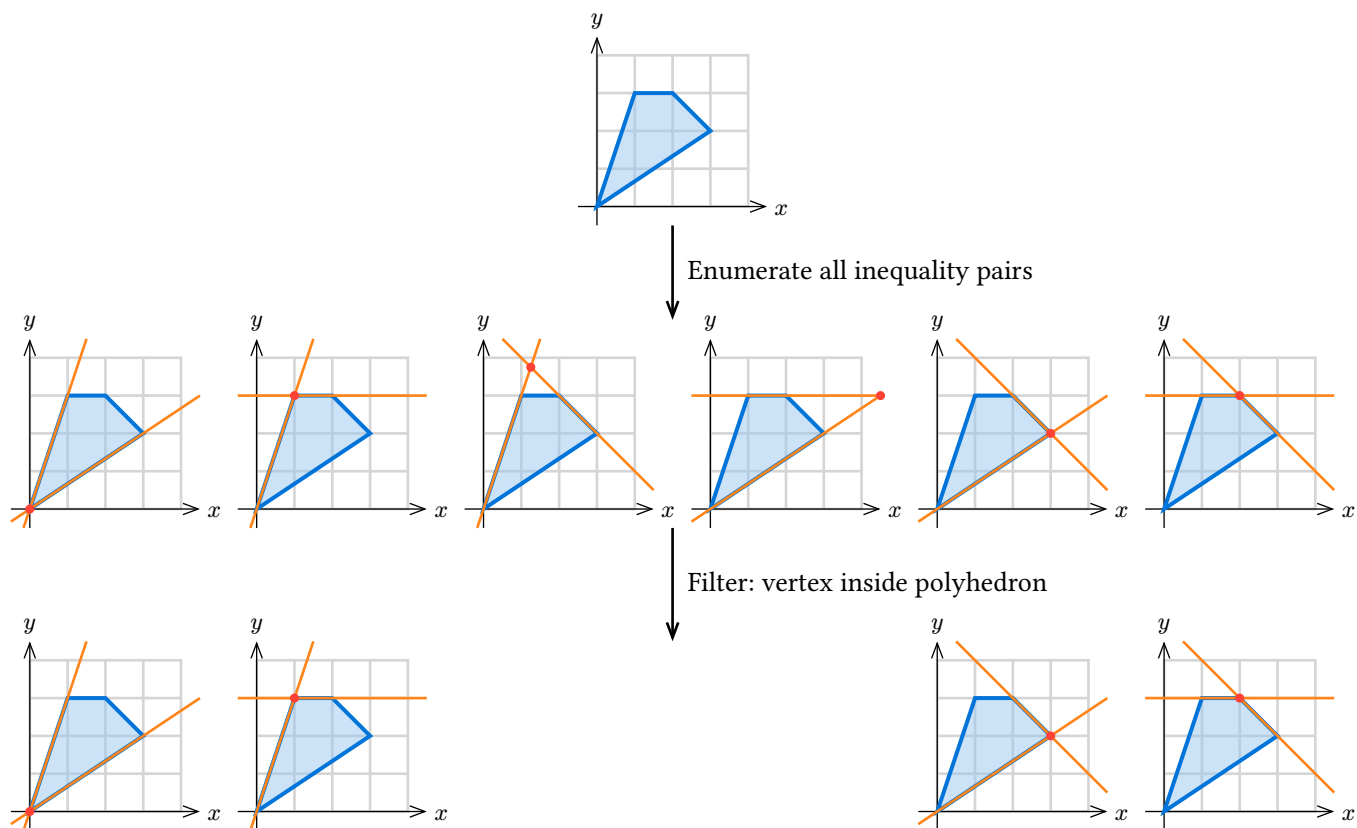


Figure 16: Identification of tangent cones.

然而该过程在超过 n 个不等式在同一顶点处活跃的情形下会失败。以 Figure 17 中的四棱锥为例。

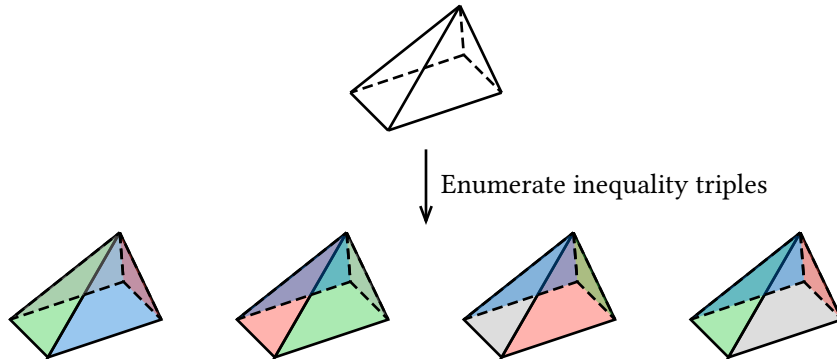


Figure 17: A failure mode of tangent cone identification.

在 Figure 17 中，顶点处会识别出四个切锥。金字塔的蓝色、红色与绿色面分别作为每个锥的活跃不等式三元组被选取。然而由于切锥由顶点处所有活跃不等式构成，顶点处应只有一个切锥，即金字塔本身。

为修复该问题，我们遍历约束并检查每个枚举中是否存在其他活跃不等式。若存在，就将其加入切锥。顶点去重以避免重复处理同一锥。例如，若不去重，Figure 17 中的顶顶点会产生四个相同的切锥，而实际上应只有一个。

形式上，对参数化多面体

$$P = \{\mathbf{x} \mid A\mathbf{x} + B\mathbf{p} + \mathbf{c} \geq 0\}$$

及其顶点 $\mathbf{v} = X\mathbf{p} + \mathbf{y}$ ， $\mathbf{v}$ 位于 P 内的条件是

$$(AX + B)\mathbf{p} + A\mathbf{y} + \mathbf{c} \geq 0$$

要识别 $\mathbf{v}$ 处活跃的不等式，我们寻找 $AX + B$ 第 i 行与 $A\mathbf{y} + \mathbf{c}$ 第 i 项全为零的索引 i 。该做法有效，因为参数空间被期望为满维，故一个不等式要对所有 $\mathbf{p}$ 恒为 0，其系数必须全为零。

约束 $(AX + B)\mathbf{p} + A\mathbf{y} + \mathbf{c} \geq 0$ 在 $\mathbf{p}$ 上构成一个多面体，称为切锥的 *active region*（活跃区域），因为它标出锥有效的（*active*）区域。我们随后将活跃区域与切锥组成的二元组作为该过程的返回结果。

2.5.6. 素模分解

给定 n 维切锥 C ，我们的下一步是将其对偶 C° 分解为素模锥。如 Section 1.2.3 所述，我们分解 C° 而非 C ，以避免在分解中处理低维锥的额外工作。通过分解 C° ，所有低维锥均可舍弃。

设 C° 的生成元为 $V = [\mathbf{v}_1 \dots \mathbf{v}_n]$ 。 C° 可能有超过 n 个生成元，此时需要预处理：见 Section 2.5.7。

我们遵循 [2] 中的算法 5。我们计算一个向量 $\mathbf{z}$ in C° ，使其在范数 L^∞ 意义下短于所有 $\mathbf{v}_i$ ，并将 C° 分解为 n 个锥 $K_1, \dots, K_n$ ，其中 K_i 由用 $\mathbf{z}$ 替换 C° 第 i 个生成元得到。显式地， K_i 的生成元为 $\mathbf{v}_1, \dots, \mathbf{v}_{i-1}, \mathbf{z}, \mathbf{v}_{i+1}, \dots, \mathbf{v}_n$ 。我们对每个 K_i 重复该过程，直到每个生成的锥都是素模锥。

寻找最短向量 $\mathbf{z}$ 时，我们采用 [6] 建议的方法。我们应用 Lenstra-Lenstra-Lovász (LLL) 算法⁴计算 V 的一个约化基，并取最短的行向量作为 $\mathbf{z}$ 。值得注意的是 LLL 并不保证 L^∞ 意义下的精确最短向量，故可能出现 $|\mathbf{z}| > |\mathbf{v}_i|$ 。但在我们的测试用例中这从未发生。

更稳健的做法是在约化基的线性组合上搜索。由于先前启发式方法被证明效果良好，我们的 Barvinok 实现不包含该搜索步骤。

多面体是否参数化不影响上述过程。对偶锥 C° 的生成元由切锥 C 的 H 表示决定，而在参数的活跃区域内它是固定的。

2.5.7. 三角剖分

对具有生成元 $\mathbf{v}_1, \dots, \mathbf{v}_m$ 的 n 维锥 C° 执行素模分解前，我们必须保证 $n = m$ 。若 $m > n$ ，我们须先对 C° 进行 *triangulate*（三角剖分），将其分解为若干含 n 个生成元的锥。

我们实现 Delaunay 三角剖分。首先将每条射线 $\mathbf{v}_i = (v_0, \dots, v_n)$ 提升为 $(n + 1)$ 维点

$$\mathbf{w}_i = \left(v_0, \dots, v_n, \sum_{i=0}^n v_i^2 \right)$$

所有点 $\mathbf{w}_i$ 连同原点，在 $(n + 1)$ 维空间中构成一个新多面体。经过原点且为多面体面（facet）的那些面，必同时经过另外 n 条射线 $\mathbf{w}_k$ ；投影回 n 维后，由该面产生的三角剖分锥即为 $\mathbf{v}_k$ 。这是因为这些锥之间除边界外不相交；此外它们覆盖了整个 C° 。

Figure 18 给出了提升过程的图形示例，尽管对象并非锥。可视化必须使用二维锥，但二维锥从不要求三角剖分。然而该算法可对任意多面体进行三角剖分，不限于锥。

左侧是待三角剖分的多面体，右侧红色点为落在抛物线上的提升点。红色边构成提升后的多面体。

⁴This algorithm is provided by FPL.

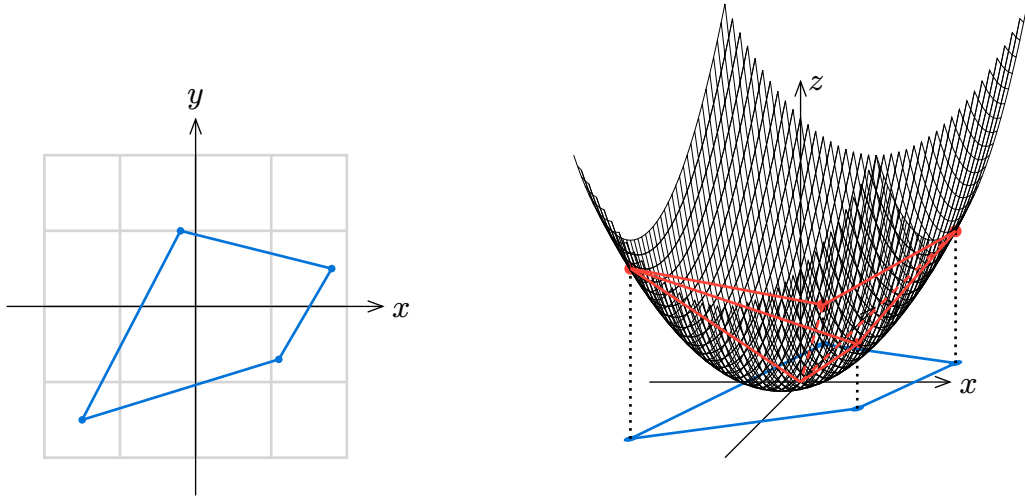


Figure 18: An example of the lifting step in DeLaunay triangulation.

我们说明面的计算方式。我们枚举 $\mathbf{w}_k$ 的所有 n 元子集，计算经过它们及原点的超平面的法向量 $\mathbf{n}$ 。若所有其他点 $\mathbf{w}_j$ ($j \neq k$) 都在该平面的同一侧，即 $\mathbf{w}_j \cdot \mathbf{n}$ 对所有 j 同号，则该超平面就是一个面。否则不是。

这 n 个向量 $\mathbf{w}_k$ 可能线性相关，此时我们提出的方法因不存在唯一的法向量 $\mathbf{n}$ 而失败。为避免此问题，我们对向量加一个微小扰动。向量先乘以一个很大的数⁵，然后向每个向量 $\mathbf{w}_k$ 加上 k 。

存在一种更高效的面查找算法，称为 *double description method*（双描述法）。但我们的项目未实现它，因为三角剖分过程很少被调用，进一步优化的优先级因此较低。

2.5.8. 室分解

如 Section 2.5.5 所述，每个切锥关联一个活跃区域。随后切锥被分解为素模锥，从中计算生成函数，如 Figure 5 所示。

为计算各生成函数在不同参数值下的贡献，我们必须根据活跃区域分割整个参数空间。这由 Clauss-Loechner 算法 [7] 实现，该算法执行 *chamber decomposition*（室分解）。我们发现 FPL 实现中的一个 bug 并修复了。

原始实现可最好地用 FPL 中以下引语概括：

We maintain a list of pairwise disjoint chambers and the generating functions. We iterate over the region list. For each active region R_j (corresponding to the generating function f_j), we examine all the current chambers R_i .

If R_j has a full-dimensional intersection with an existing chamber R_i , then that chamber is replaced by two new ones:

1. the intersection $R_i \cap R_j$, where the generating function is $f_i + f_j$;
2. the difference $R_i - R_j$, where the generating function is f_i .

— comment in Barvinok.cpp

如注释所述，FPL 仅当当前室 R_i 与活跃区域 R_j 有满维交时才分割室，这由 [2] 证明是合理的。然而问题在于，满维性检查在 $\mathbb{Z}^n$ 中执行，而应当在 $\mathbb{R}^n$ 中执行。

该差异最好用例子说明。考虑如下带两个整数参数 N 与 M 的 1 维多面体 $P \subseteq \mathbb{Z}$ ，由 $i \in \mathbb{Z}$ 上的三个不等式定义：

⁵In our test cases, 10^6 would be enough. For a more robust implementation, the magnitude should be adjusted to the maximum element in $\mathbf{w}_k$.

$$2i \leq M \leq 2i + 1, \quad (1)$$

$$2i \geq N + 1, \quad (2)$$

$$2i + N - 2M + 2 \geq 0 \quad (3)$$

由不等式 (1) 与 (2) 可得 $N + 1 \leq M$ 。由 (2) 与 (3) 可得 $M \leq N + \frac{3}{2}$ 。故 (N, M) 的活跃区域为

$$R = \left\{ (N, M) \mid N + 1 \leq M \leq N + \frac{3}{2} \right\}$$

当 M 与 N 为实数时, 关系 R 是满维的。然而由于 FPL 只处理整数关系, 它总是将 M 与 N 视为整数, 并将活跃区域简化为

$$R_{\mathbb{Z}} = \{(N, M) \mid M = N + 1\}$$

关系 $R_{\mathbb{Z}}$ 非满维。由于这一错配, FPL 丢弃了本应考虑的区域。

若我们以有理关系操作替换所有整数关系操作, 问题即可解决。为避免重写整个库以支持有理数, 我们构造一个从有理关系到整数关系的嵌入, 该嵌入保持集合并、交与差。

为将有理关系 $R = \{\mathbf{x} \in \mathbb{Q}^n \mid A\mathbf{x} + \mathbf{b} \geq 0\}$ 表示为整数关系, 我们引入一个额外整数变量 $d \rightarrow +\infty$, 其被解释为公分母。该关系则嵌入为 $R' = \{(\mathbf{x}', d) \in \mathbb{Z}^{n+1} \mid A\mathbf{x}' + \mathbf{b}d \geq 0\}$, 其中 $\mathbf{x}'$ 表示 $d\mathbf{x}$ 。

随后我们像处理普通整数关系一样对其执行差与交运算。将有理数四舍五入为整数的简化只影响常数项, 而不影响 d 的系数 $\mathbf{b}$ 。

室分解后, 我们获得如下形式的室列表:

$$S' = \{(\mathbf{x}', d) \in \mathbb{Z}^{n+1} \mid A\mathbf{x}' + \mathbf{b}d + \mathbf{c} \geq 0\}$$

为将它们转换回整数关系, 我们注意到 $d \rightarrow +\infty$, 故两边除以 d 后只有 $\mathbf{c}_i$ 的符号有意义。

对不等式 $A_{i,:} \cdot \mathbf{x}' + \mathbf{b}_i d + \mathbf{c}_i \geq 0$:

- 若 $\mathbf{c}_i < 0$, 降为 $A_{i,:} \cdot \mathbf{x} + \mathbf{b}_i \geq 1$;
- 若 $\mathbf{c}_i \geq 0$, 降为 $A_{i,:} \cdot \mathbf{x} + \mathbf{b}_i \geq 0$ 。

2.5.9. 求和

我们已讨论了单个参数化多面体 $P = \{\mathbf{x} \mid A\mathbf{x} + B\mathbf{p} + \mathbf{c} \geq 0\}$ 的活跃区域列表及其生成函数的计算方法。然而在我们的缓存缺失计数分析中, 必须计数整数关系内部的整数点, 而整数关系可以是若干多面体的并。因此我们需要 inclusion-exclusion (容斥)。

考虑整数关系 $\bigcup_{i=1}^m P_i$ 。对每个子集 $\mathcal{P} \subseteq \{P_i\}$, 我们计算 $\bigcap \mathcal{P}$ 的活跃室与生成函数列表 $(R_{\mathcal{P}}, f)$, 连同参数约束 $C_{\mathbf{p}}$ 。生成函数 f 的活跃区域即为 $R = R_{\mathcal{P}} \cap C_{\mathbf{p}}$ 。由于多面体的交仍为多面体, 只需应用前面各节所述方法。

我们依赖 FPL 将生成函数 f 在 $\mathbf{x} = 1$ 处求值, 获得关于参数 $\mathbf{p}$ 的、给出整数计数的 quasi-polynomial q 。我们记录每个三元组 $(\mathcal{P}_j, R_j, q_j)$, 其中 q_j 在 $\mathbf{p} \in R_j$ 时有效。

随后我们在有效参数区域 R 列表上对参数空间执行室分解。该过程与 Section 2.5.8 中描述的 Clauss-Loechner 算法类似, 但不可省略低维交集。它产生一组室 C_i , 使得对任意区域 R_j , 每个室要么完全在其中, 要么完全在其外。

对分解后的每个 C_i , 我们遍历所有记录 $(\mathcal{P}_j, R_j, q_j)$ 。现在 C_i 要么 R_j 的子集, 要么与之完全不交。在前者情形, 按容斥原理, 该室的整数计数增加 $(-1)^{|\mathcal{P}_j|+1} q_j$ 。在后者情形, 计数不变。

这完成了参数化 Barvinok 算法, 其结果为参数区域列表以及每个区域内的整数计数函数。

2.6. 优化

2.6.1. 缓存室归属

在 Section 2.5.9 中，我们说明要计算多面体并内部的整数计数，首先需要将参数空间分解为室 C_i ，然后检查每个区域 R_j 是否是它的超集。

超集检查涉及空性测试 (emptiness test)，开销较大。我们可以将结果预计算到分解过程中，从而去除该检查。除了将全域分解为室 C_i ，我们还计算二元组 $(C_i, \mathbf{b}_i)$ ，其中 $\mathbf{b}_i$ 是一个位向量 (bitvector)。这里 $\mathbf{b}_{ij} = 1$ 表示 R_i 是 R_j 的子集， $\mathbf{b}_{ij} = 0$ 表示二者不交。

分解的每一步中，我们枚举活跃区域 R_j ，将每个室 C_i 分裂为 $C' = C_i \cap R_j$ 与 $C'' = C_i - R_j$ 。我们知道 R_j 对前者有贡献而对后者没有，故可将 $\mathbf{b}'_j$ 设为 1、将 $\mathbf{b}''_j$ 置 0，其他条目则从与 C_i 关联的 $\mathbf{b}_i$ 拷贝。

2.6.2. 早期消除空关系

在 Section 2.4.1 中，我们调用 FPL 过程计算一个 presburger relation 的字典序最大值。

该过程首先为每个 disjunct (析取分量) 计算一个分段仿射函数，然后将其并成一个单一函数。两个分段函数的并通过枚举所有 piece 对 (R_i, f_i) 与 (S_j, g_j) 实现。算法随后计算区域 $T = \{\mathbf{x} \mid f_{i(\mathbf{x})} >_{\text{lex}} g_{j(\mathbf{y})}\}$ ，并将区域分为 $R_i \cap T$ 与 $R_i - T$ 。最坏情况下，并后的 piece 数量随析取分量个数呈指数增长。

然而我们注意到，通过差或交构造出的大多数关系实际上是不可行的 (infeasible)。我们在它们出现时就去除这些区域，使其不再参与并操作、不再导致指数行为。对测试用例而言，尽管 piece 数量仍呈指数增长，但经我们的优化后缩放速度显著变缓 (见 Section 3.5.2)。

2.6.3. 并行化

如 [8] 所述，我们的缓存模型分别处理每个 sink (汇) 语句，且该过程是只读的。因此该模型天然并行 (embarrassingly parallel)，我们可借此并行性获得 3.9 倍加速，如 Section 3.5.3 所示。

3. 评估

3.1. 实验设置

我们在 Polybench [9] 上开展测试。Polybench 是由 C 语言编写的 30 个计算程序的集合，它们具有密集的数值核函数 (kernels)、规则的循环结构且体量相对较小，是评估的理想选择。在这些程序中，28 个是 Section 1.5 中所定义的 *affine* 程序 (仿射程序)，其余两个测试用例包含数据依赖的控制流，即 `nussinov.c` 和 `floyd-warshall.c`。其中涉及的控制流实际上是 `min` 或 `max` 函数。尽管理论上可以通过 CSE (公共子表达式消除, common subexpression elimination) 优化将其消除，但这样做需要更复杂的分析，超出了本项目范围。

这些程序具有可配置的数据集大小，下文各节使用了小数据集和大数据集。小尺寸主要用于建立正确性，而大尺寸则用于性能分析。

为了保证开发环境的一致性，我们在 Windows Subsystem for Linux (WSL) 上运行测试，但硬件缓存缺失 (cache miss) 次数计数在原生 Ubuntu 系统上采集，以确保准确性。所有测试均在配备 16 核 CPU 和 512 KB L1 数据缓存的 AMD Ryzen 9 7945HX 上进行。

我们的实现编译为 LLVM 的 release build，采用默认配置。具体而言，编译选项包含 `-O3`，并禁用了 `assertions` 与 `LTO` (链接期优化, link-time optimisation)。

3.2. 正确性

我们将我们的结果与 28 个 Polybench 程序中原始 Falcon [8] 实现的结果进行比较，以证明正确性。参考实现直接以 *affine* 方言 (dialect) 为输入。其测试集已提供这些程序的 MLIR kernel，跳过数组初始化与结果打印。我们相应地修改了输入文件 (C 源代码)。

在一个测试用例 `adi.c` 中，传给 Falcon 的 MLIR 输入包含一次 CSE 优化，将两次 load 合并，从而影响了缓存缺失次数计数。为保证输入等价，我们手动将其“去优化”（de-optimize），使其与在 Clang IR 中生成的形式一致。

对于小数据集，缓存行大小 $L = 1$ 、缓存容量 $C = 64,128$ ；对于大数据集， $C = 524288$ 。我们确认了全部 28 个程序的缓存缺失次数完全一致。这表明在测试所用的仿射程序与缓存参数下具有行为等价性，为我们的实现正确性提供了强有力的经验证据。

当 $L > 1$ 时，我们在一个手工编写的小程序上确认了缺失次数完全一致。然而，即使 Polybench 中体积最小的程序，分析也超过了 600 秒的超时，我们因此无法获得进一步的结果。

3.3. 准确性

我们将我们的模型在 Polybench 程序上的预测与用 Unix 工具 `perf` 测得的缺失次数进行比较，以此衡量模型的准确性。我们使用大数据集，以确保大多数程序的工作集（working set）无法容纳在相对较大的缓存中；否则就不会有容量缺失（capacity miss），相应的模型部分便无法被评估。

当前我们假设每条缓存行包含一个元素。实际上，在我们的设置下每条缓存行为 64 字节，包含测试程序的 8 个元素。为补偿这种不匹配，我们将传给模型的缓存容量输入设置为原来的 8 倍，即 4 MB。这样缓存能容纳的元素个数相同，但空间局部性（spatial locality）将完全丧失。因此我们预期相对误差较大，但能观察到强相关的缩放趋势。

比较结果如 Figure 19 所示，其中 x 轴为我们的静态分析预测值， y 轴为硬件实测值，点线表示 $y = x$ 。左图为对数刻度，右图为线性刻度。

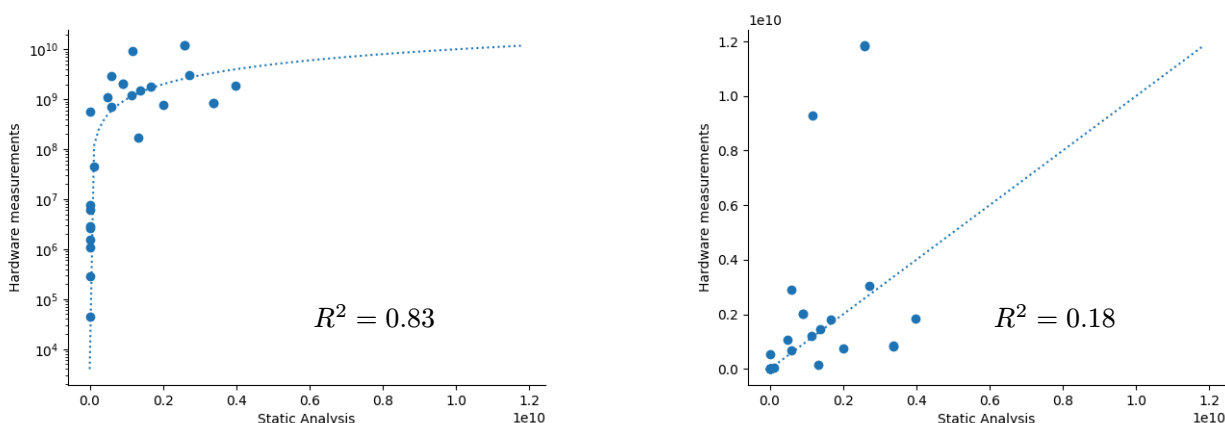


Figure 19: The model predicts the scaling trend of cache misses, but fails to predict the absolute miss counts.

绝对次数几乎完全无相关性，但缩放趋势在很大程度上被我们的模型所捕获。

值得注意的是，线性情况下缺乏相关性部分是由于大离群值（`gemm.c`、`jacobi-2d.c`）主导了误差，因此更有意义的是查看每个单独测试用例的相对预测误差。

3.3.1. 相对误差分析

Figure 20 绘制了我们测试用例相对误差的累积分布函数（CDF）。

预测结果的相对误差几何平均为 62.37%，中位数为 78.08%，表明在 $L = 1$ 的假设下，我们的模型只给出了高度近似的缺失次数。

最大误差在 `gemm.c`（矩阵乘法的朴素实现）上达 673.8%。其工作集大小恰好约为 4 MB，与我们输入的缓存容量相当。对于这种极端情况，模型的任何差异——例如忽略空间局部性——都会对缓存行为造成巨大影响。

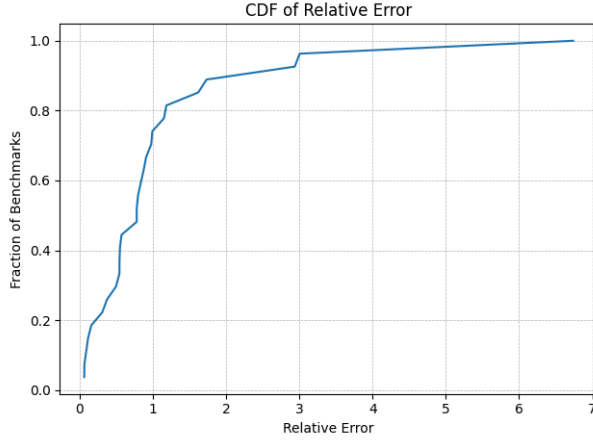


Figure 20: The model's predictions are highly approximate.

3.4. 性能

我们将我们的模型运行时间分别与参考实现 Falcon 以及程序本身的执行时间进行比较，并评估 Section 2.6 中描述的优化所带来的改善。

3.4.1. 运行时间比较

我们模型与 Falcon 的运行时间对比，以及 Polybench 程序在大数据集下的执行时间，如 Figure 21 所示。我们的模型与 Falcon 均以 32 线程运行，这是 `nproc` 的结果，即使用整颗 CPU 的全部能力。

平均而言，我们的实现比实际执行快 1.21 倍，表明其效率足以分析实际程序。

模型比 Falcon 慢 22.1 倍（几何平均）。我们以效率换取了 MLIR 集成：我们将 Falcon 中高度优化的 `isl` 和 `barvinok` 库替换为 FPL 和一个自研的 Barvinok 算法实现。因此我们预期算法层与低层级的优化更少，从而出现这样的减速。

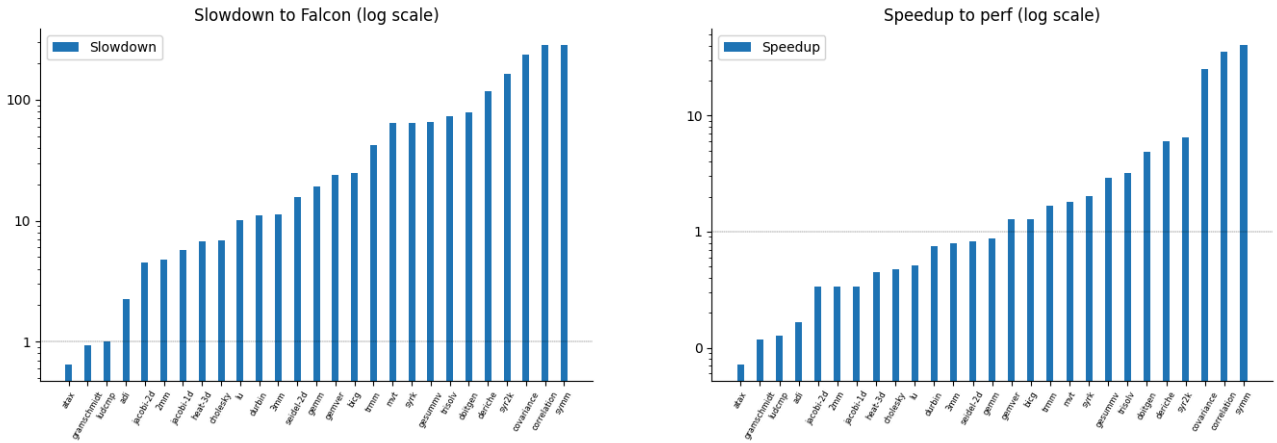


Figure 21: Our analysis is efficient enough but yet has large room for optimisation.

这一减速也提示了未来优化的空间，我们在热点分析中（Section 3.4.2）详细探讨。

3.4.2. 热点分析

本节给出的热点分析反映了当前实现的性能。在此之前，我们已多次进行类似分析，并解决了多个瓶颈。相关细节见 Section 2.6，这些优化的效果在 Section 3.5 中介绍。

如 Section 2.4 所示，我们的算法分为四个部分：计算 $\text{deps}(S(\mathbf{x}))$ 、统计必要缺失（compulsory miss）数量、计算 $\text{dist}(S(\mathbf{x}))$ 、统计容量缺失数量。Figure 22 给出了单线程设置下各部分消耗时间的占比。

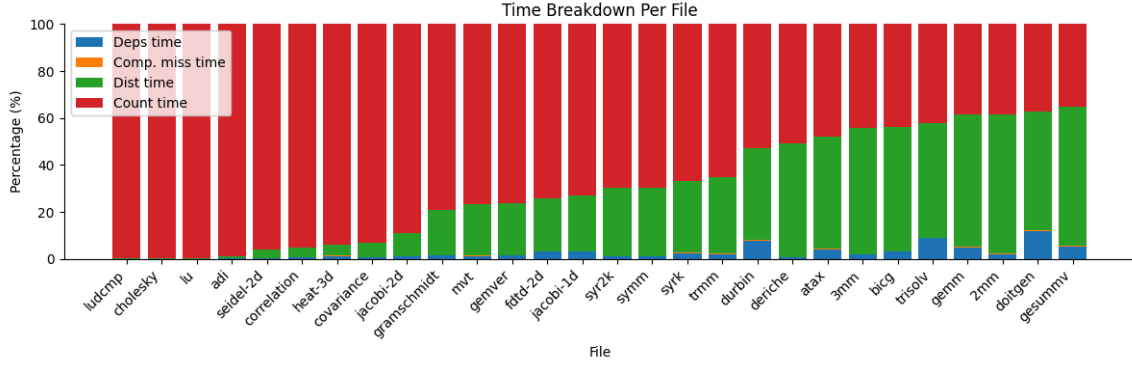


Figure 22: Majority of time is spent in Barvinok's algorithm.

我们注意到容量缺失的最终计数步骤占运行时间的 40%–100%，该步骤对多面体（polyhedra）的并集调用参数化 Barvinok 算法。我们进一步考察该算法的性能。

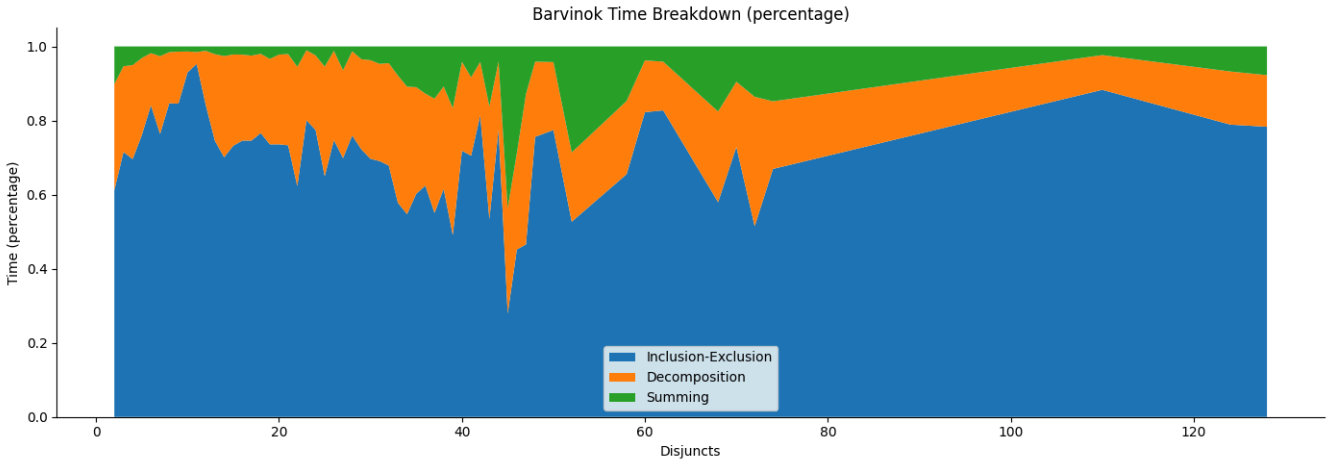


Figure 23: Most time is spent in inclusion-exclusion regardless of disjunct count.

如 Section 2.5.9 所述，该算法由三部分组成：容斥（inclusion-exclusion）、室分解（chamber decomposition）和求和（summation）。我们在 Figure 23 中按不同析取分量（disjunct）数量绘制了各步骤所占时间百分比。我们发现，无论析取分量个数多少，容斥步骤总是最耗时的。

热点分析表明，我们的模型未来的优化努力应聚焦于 Barvinok 算法，尤其是占运行时间主导地位的容斥部分。这并不意外，因为在最坏情况下，其生成结果（records）的数量可随析取分量个数呈指数增长。然而在我们的静态分析中并未遇到如此病态的情形：如 Figure 24 所示，record 数量大约随析取分量个数呈二次增长，表明我们的当前模型能够处理真实世界的程序。该曲线通过 `np.polyfit` 拟合。

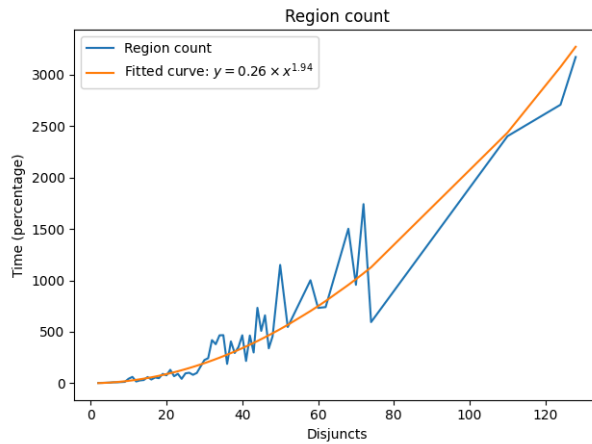


Figure 24: Region counts scale quadratically to number of disjuncts.

一个优化机会是减少传入最终计数步骤的析取分量数量——即在 $S(\mathbf{x})$ 与其依赖 $T(\mathbf{y})$ 之间构建复用实例 (reuse instance) 时，根据易于计算的判据剪除特定语句。由于是二次缩放，该优化将带来可观收益。

3.5. 优化效果

3.5.1. 缓存室归属

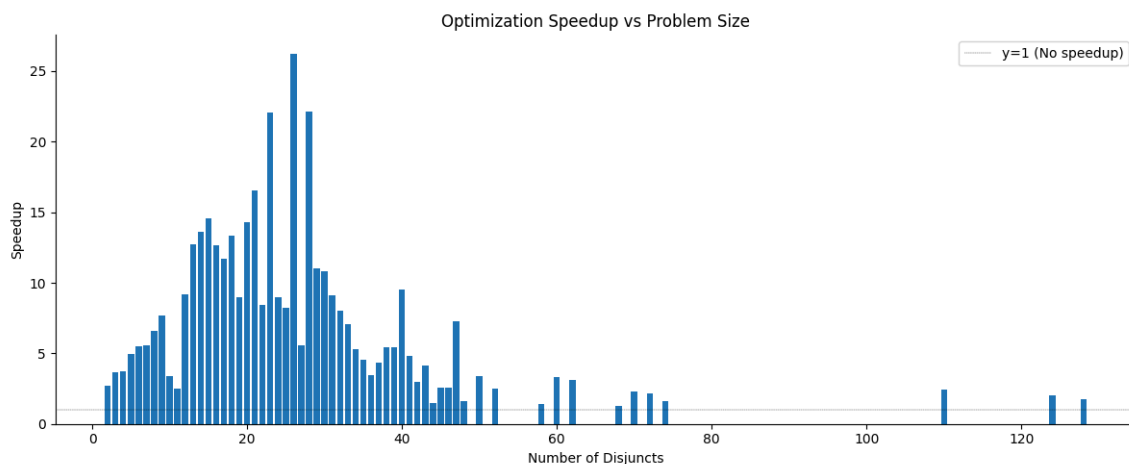


Figure 25: Speedup brought by caching chamber ownership.

按 Section 2.6.1 所述缓存室归属 (chamber ownership)，我们将最终求和步骤加速了 5.32 倍，Barvinok 步骤整体加速了 23% (几何平均)。Figure 25 比较了优化前后求和步骤所用时间。

3.5.2. 早期消除空关系

在并合两个分段仿射函数 (piecewise affine function) 时去除不可行区域，我们缓解了 piece 数量的指数增长，降低了指数的底数。

未优化算法的完整 piece 数量难以获取，因为即使并合 8 个函数，程序也会超时超过 600 秒。经验上，并合 6 个函数后 piece 数量增长到约 16,000。

优化后，如 Figure 26 所示，在 Polybench 上指数增长的底数降至 1.22。具体而言，6 个析取分量时，平均 piece 数量从 16,000 降至 31.5。该优化使我们的分析能够处理 Polybench 中更复杂的 stencil，那里析取分量数量轻松超过 10 个，而待并合的函数数量等于析取分量数量 (见 Section 2.6.2)。

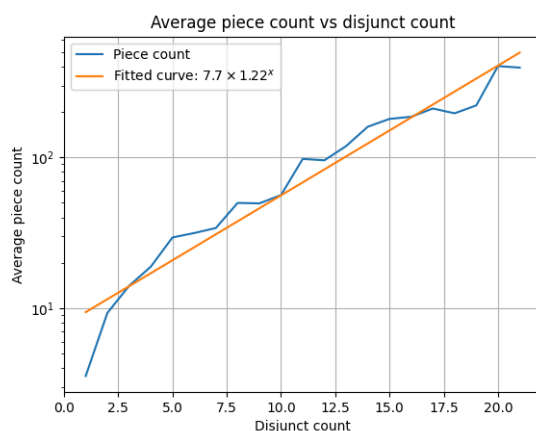


Figure 26: Average piece count scales exponentially with smaller base.

3.5.3. 并行化

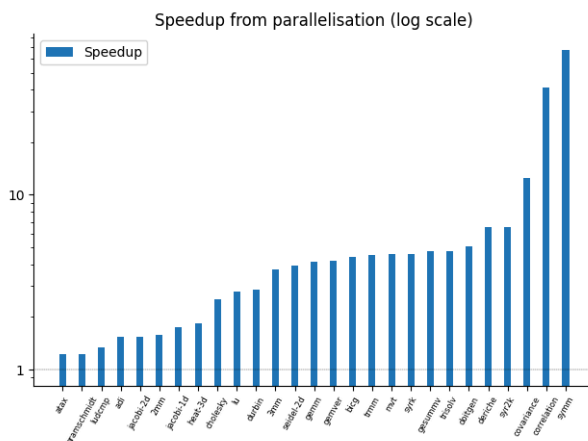


Figure 27: Speedup from parallelisation.

我们利用了任务并行：每个语句的缓存缺失次数计算相互独立。这使我们的测试用例获得 3.9 倍加速（几何平均），如 Figure 27 所示。

由于测试用例以 32 线程运行，如果加速比不足 4 倍就令人意外。原因在于没有 Polybench 程序的数组访问语句超过 32 个，因此程序运行时间由耗时最长的语句决定。对某语句 S ， $\text{dist}(S(\mathbf{x}))$ 的析取分量数量是决定运行时间的主要指标，如 Section 3.4.2 所示，且在不同语句间可能差异巨大。在 stencil 中，单个语句主导运行时间，加速比低；在线性代数核函数中，各语句的析取分量数量相近，因而加速比更大。

3.6. 总结

贯穿本章，我们看到我们的分析能够捕获缓存缺失次数的渐近行为，而且平均而言比在硬件上运行测试程序收集缺失次数更快。我们对分析当前的热点进行了分析，以指导未来优化。我们评估了若干项优化，并展示了该分析扩展到真实世界规模程序的潜力。

Bibliography

- [1] A. Barvinok, “A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed,” in *Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science*, 1993, pp. 566–572. doi: 10.1109/SFCS.1993.366830.
- [2] J. Loera, R. Hemmecke, J. Tauzer, and R. Yoshida, “Effective lattice point counting in rational convex polytopes,” *Journal of Symbolic Computation*, vol. 38, pp. 1273–1302, Oct. 2004, doi: 10.1016/j.jsc.2003.04.003.
- [3] A. Pitchanathan, C. Ulmann, M. Weber, T. Hoefler, and T. Grosser, “FPL: fast Presburger arithmetic through transprecision,” *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, Oct. 2021, doi: 10.1145/3485539.
- [4] D. Detlefs, G. Nelson, and J. B. Saxe, “Simplify: a theorem prover for program checking,” *J. ACM*, vol. 52, no. 3, pp. 365–473, May 2005, doi: 10.1145/1066100.1066102.
- [5] C. H. Papadimitriou, “Computational complexity,” in *Encyclopedia of Computer Science*, GBR: John Wiley, Sons Ltd., 2003.
- [6] L. L. S. A. Grötschel Martin, *Geometric algorithms and combinatorial optimization*. Berlin [u.a.]: Springer, 1988. [Online]. Available: <http://eudml.org/doc/204222>
- [7] P. Clauss and V. Loechner, “Parametric analysis of polyhedral iteration spaces,” in *Proceedings of International Conference on Application Specific Systems, Architectures and Processors: ASAP '96*, 1996, pp. 415–424. doi: 10.1109/ASAP.1996.542833.

- [8] A. Pitchanathan, K. Grover, and T. Grosser, “Falcon: A Scalable Analytical Cache Model,” *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, Jun. 2024, doi: 10.1145/3656452.
- [9] M. J. Reisinger, “PolyBenchC.” Accessed: Feb. 23, 2026. [Online]. Available: <https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1>