循环局部性的完全符号分析

利用虚拟重用推断实际性能

YIFAN ZHU，美国罗切斯特大学YEKAI PAN，
美国罗切斯特大学CHEN DING，美国罗切斯特
大学YANGHUI WU，美国罗切斯特大学

arXiv:2603.10196v1 [cs.PL] 10 Mar 2026

本文提出了一种新的局部性理论及其编译器支持。该理论完全采用符号形式，将局部性推导为多项式；编译器分析则支持仿射循环嵌套。该方法以二次表达式和倒数表达式推导缓存性能伸缩，其通用性和精确度均优于经验性扩展规则。

在包含 41 个科学计算内核和张量运算的基准测试套件上进行评估时，编译器平均需要 41 秒来推导局部性多项式。推导完成后，预测任意给定输入规模和缓存配置下的缓存未命中次数所需时间不到 1 毫秒。在所有测试中——无论是否采用循环融合——数据移动预测的准确率均为 99.6%，对比基准为模拟的组相联 L1 数据缓存。

1 引言

按照 Denning 的定义，局部性是指随着时间推移，内存引用倾向于集中在地址空间的小范围区域内 [13]。它是程序行为的一项基本属性，也是数据密集型应用性能的决定因素。局部性优化在现代编译器 [1–3, 57], 变换框架 [39, 63], 、图像处理领域特定语言和编译器 [42] 以及张量运算 [27, 30, 48]中至关重要。完整的刻画应通过机器参数和程序参数来量化局部性。

本文提出一种完全符号化的技术，用于分析循环局部性。给定符号化循环边界，我们的方法推导出两类多项式：第一类计算缓存大小，第二类计算未命中计数或未命中率。我们将这些公式称为代数局部性，因为它们是代数表达式，能够捕捉局部性如何随各种参数而变化。例如，对一个 $n \times n$ 矩阵进行单次遍历时，未命中率多项式为 $\frac{1}{b}$，其中 b 是缓存块大小。

新的分析基于一种称为重用间隔（RI）的度量。重用间隔（RI）定义为对同一内存位置两次连续访问之间的时间。我们主要解决两个问题。第一个是关于利用重用间隔多项式推导缓存未命中多项式的理论及其正确性。第二个是仿射循环中重用间隔的编译器分析。形式化推导是迭代进行的，耗时与符号 RI 值的数量呈线性关系。编译器分析包含两个 Passes，并使用整数集合规划。

代数理论中全新且必要的是虚拟重用的引入。传统上，首次触及访问的重用间隔被视为无限。然而，无限重用区间会给符号分析带来问题。引入虚拟重用是为了给首次触及访问赋予有限的 RI 值。

以往的技术是符号化的，但并非代数的。仿射循环可以通过求解整数集合方程 [4, 6, 19, 26, 40] 进行符号分析。与多项式不同，这些整数方程是线性的，不能包含二次项或倒数项。因此，它们无法完全符号化。例如，它们仅为常量而非符号化的缓存块大小建模空间局部性。此外，其求解时间在最坏情况下呈指数级。相比之下，在新理论中，推导具有线性时间开销。

本文的主要贡献如下：

作者联系方式：Yifan Zhu，罗切斯特大学，罗切斯特，纽约，美国；Yekai Pan，罗切斯特大学，罗切斯特，纽约，美国；Chen Ding，罗切斯特大学，罗切斯特，纽约，美国；Yanghui Wu，罗切斯特大学，罗切斯特，纽约，美国。

• **代数局部性理论**，它使用虚拟重用和迭代方法在线性时间内推导缓存多项式。该理论证明了两个形式性质：工作集正确性和 RI 和不变性。

• **仿射循环编译器分析**将 MLIR 的仿射方言转换为参数多胞体，并使用双遍算法计算所有可能的重用间隔长度及其作为多项式表示的计数。

• **实现与评估**，包括对 30 个科学计算内核 [41] 和 11 个常见张量运算 [8], 在循环融合优化前后的分析速度和准确性进行评估，并与缓存模拟和硬件计数器的结果进行比较。

缓存多项式的一种用途是扩展性分析。例如，$\sqrt{2}$ 该规则对于数据库系统（扩展缓冲池大小）、文件系统缓存和 Web 缓存而言众所周知 [22]。缓存多项式展示了仿射循环的缓存扩展性，并且比扩展规则具有更高的精度。第 5.5 节展示了两个缓存扩展性结果，它们都遵循 $\sqrt{2}$ 该规则，但其未命中率相差两倍。

局部性属性决定数据移动，即未命中计数，但并不能完全决定运行时间；运行时间在很大程度上取决于乱序执行和数据预取等延迟隐藏技术。然而，延迟容忍并不会减少数据移动量。此外，我们仅考虑顺序程序，并关注单级缓存的未命中率，其中考虑组相联度和实际硬件缓存。虽然局部性分析能够支持程序优化，但本文重点关注分析而非优化；唯一的例外是，我们对优化前后的分析进行评估，尤其是循环融合。

2 代数局部性理论

本节介绍该理论及其名为虚拟重用的新概念。图 1所示的流程图展示了各小节的结构：无限重复（第 2.2 节）产生虚拟重用，这些虚拟重用用于推导局部性多项式（第 2.1 节和第 2.3 节）以及符号测试（第 2.4 节）。第 2.5 节介绍 LRU 近似。最后，第 2.6

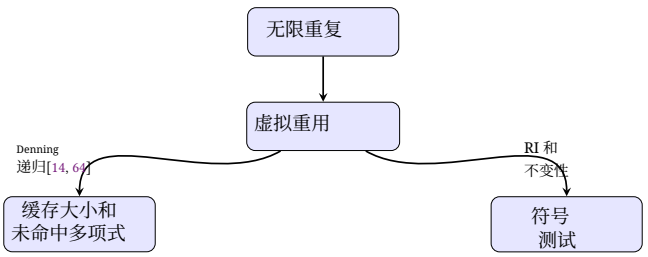


图1. 无限重复创建了用于符号局部性分析的虚拟重用

2.1 背景：重用间隔与 Denning 递归

从存储系统的角度来看，程序执行是一系列内存访问。对同一数据项的每一对连续访问构成一次数据重用。重用间隔（*RI*）是两次重用访问之间的其他访问次数。本文采用逻辑时间，每次访问使时间增加一个单位。两次访问之间的 RI 值是它们逻辑时间之差。

设 $P(ri)$ 为一个 RI 分布，其中 $P(ri = x)$ 是值等于 x 的重用间隔所占的部分。Denning递归 [14] 是一组用于根据 RI 分布计算时间窗口失效率和

平均工作集大小的方程，如下所示：

$$m(x) = P(ri > x)$$
$$s(x+1) = s(x) + m(x)$$

其中 $m(x)$ 是未命中率， $s(x)$ 是工作集大小。这两个方程对所有整数 $x \geq 0$ 均有定义，初始值为 $s(0) = 0$ 和 $m(0) = P(\text{重用间隔} > 0) = 1$。为计算所有结果$m$， s，Denning递归按递增顺序遍历所有重用间隔值 x。每一步中，未命中率等于重用间隔大于 x的重用所占的比例，而工作集大小则增加一个未命中率。

时间窗口失效率 $m(x)$ 是时间的函数。工作集大小是一个平均值，通常为小数。如第2.3节所述，新理论将利用这些值来近似计算 LRU缓存大小和未命中率。

2.2 无限重复与虚拟重用

在程序中，数据访问可能是首次使用某个项，这称为首次触及访问；否则就是数据重用。[1]

首次触及访问在任何缓存中都是冷启动未命中 [23]。然而，对于Denning递归而言，首次触及访问带来了一个根本性困境。如果为首次触及访问赋予无限重用间隔，那么 $P(ri = \infty) > 0$ 会使工作集大小发散至无穷（$s(\infty) = \infty$），从而使分析失去意义。另一方面，如果将首次触及访问完全排除在分析之外，工作集大小虽保持有界，但我们无法计入冷启动缺失。我们将此称为冷启动未命中困境。

代数理论通过引入无限重复解决了这一困境，即程序重复无限次。首次触及访问在第一次运行中仍然如此，但在第二次运行及所有其他重复中则成为重用。我们将这些重用称为虚拟重用。虚拟重用的重用间隔是虚拟 RI。每个数据项的虚拟重用间隔从其在一次运行中的最后一次访问延伸到下一次运行中的下一次访问。如果该数据项在一次运行中未被重用，则该间隔等于程序长度 n。否则，它是上一次运行中最后一次访问与当前运行中第一次访问之间的间隔，最多为 $n-1$。因此，最大可能的虚拟重用间隔为 n。

图 2 展示了该技术。设一个简单程序由五次访问组成。该图展示了程序的第一次运行以及两次重复：第二次运行和第 r次运行。五次访问中的第一次、第二次和第四次是首次运行中的首次触及访问，标为红色。第一次运行之后，它们成为虚拟重用，标为绿色。箭头连接前两次程序运行之间虚拟重用的源与汇。最大的虚拟 RI 值为 5，对应于对‘B’的访问，它是一次运行中的唯一一次访问。

为加以区分，我们将无限重复之前的数据重用称为真实重用。它们的值与份额不会因无限重复而改变。在图2中，它们是每次运行中的第三次和第五次访问（标为蓝色）。

虚拟重用不会直接影响真实重用。然而，要正确建模缓存行为，Denning递归要求所有访问都具有有限的重用间隔值。虚拟重用间隔是一种“加项”。直观而言，将其纳入其中是必要的，因为它们构成了真实重用的上下文，使我们能够正确判断真实重用是否会导致缓存命中或未命中。接下来的两节将先形式化这一正确性性质，然后说明如何将虚拟重用转换回首次访问。

[1]重用的属性可以关联到重用窗口的起点或终点。若关联到前者，则为正向使用；否则为反向重用 Beyls and D’Hollander [6]。本文中，所有重用间隔均为反向重用间隔。

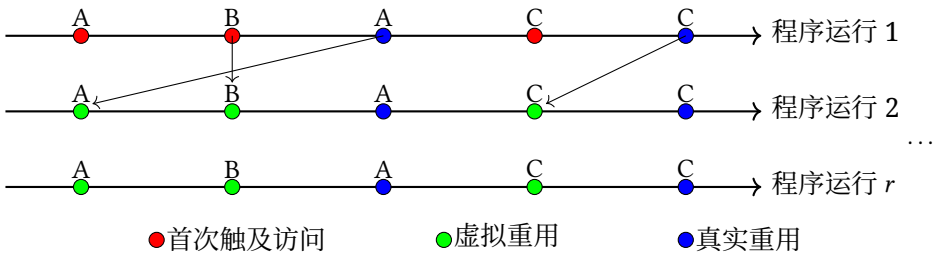


图 2. 无限重复示例，其中首次触及访问（红色）、真实重用（蓝色）和虚拟重用（绿色）

2.3 工作集正确性

Denning递归是作为一种随机性质推导出来的，其前提是存在无穷序列、一个平稳过程，以及第三个性质，即“保证时间平均收敛到随机平均值” [14]。对于确定性序列，Xiang等人 [60] 定义了足迹函数 $fp(x)$，它表示长度为 x 的时间窗口的平均工作集大小。我们结合工作集理论和足迹理论，证明采用虚拟重用时 Denning递归的正确性。

我们证明了无限重复情况下的工作集正确性：Denning递归中计算出的 $c(x)$ 是足迹 $fp(x)$，即确定性序列的平均工作集大小。这意味着，对于任何程序，无限重复情况下的Denning递归都是正确的，且无需对此类情况作出任何随机性假设。

给定一个有限长度序列，Xiang公式会计算足迹$fp(x)$，即所有长度为 x [59] 的窗口的平均工作集大小。设集合 $\{r_i\}$ 为所有重用间隔（RI）的取值，集合 $\{f_d\}, \{l_d\}$ 为所有数据 d 的首次访问时间和末次访问时间。我们稍微改变 Xiang公式的表示方式，将所有边界访问统称为 $\{b_d\}$，其中包括首次和末次访问。每个数据项 d 有两个 b_d。设 n 为访问次数（即轨迹长度）， m为不同数据的数量（即数据的大小）。Xiang公式按如下方式计算足迹

$$fp(x) = m - \frac{\sum_{r_i>x}(ri - x) - \sum_{b_d>x}(b_d - x)}{n - x + 1}$$

其中 $x = 0, \ldots\ldots, n$。我们进一步将重用项足迹定义为[2]

$$rtfp(x) = m - \sum_{i=x+1}^{n} (i - x)P(ri = i)$$

直观地说，$rtfp(c)$ 统计超过缓存大小 c的重用间隔值数量，并按其频率加权。在无限重复下，由于每次访问最终都会成为一次重用，因此该足迹会收敛至零。下述定理将这一性质形式化，并表明工作集大小能够正确计算所有被访问的不同数据项。

定理2.1（工作集正确性）。在无限重复下，*Denning*递归能够正确计算平均工作集大小。

[2]在长度为 n的序列中，可能的最大重用间隔（RI）值为 $n - 1$。在无限重复的情况下，可能的最大值为 n，也就是说，当某个符号在序列中只出现一次时（在重复之前，之后则成为虚拟重用）。图2中的 B 就是一个例子。

证明。设 n 为程序的长度， r 为重复次数。我们首先证明$rtfp(x)$正确性。当 r 趋于无穷时，Xiang公式中的边界项具有有限值，可以将其去掉，因为分母 $rn - x + 1$趋于无穷。此外，每个重用间隔（RI）的值都有限，且至多为 n，其贡献恰好为 $\frac{1}{n}$。因此，重用项足迹是正确的，因为它收敛于 Xiang足迹。此外，Denning递归等价于重用项足迹，即 $s(x) = fp(x)$。我们用归纳法证明这一点。在基础情形中， $s(0) = fp(0) = 0$ （另见下一个引理）。假设 $s(x) = fp(x)$，可以直接验证$fp(x+1) - fp(x) = s(x+1) - s(x) = P(ri > x)$。结合$rtfp(x)$的正确性，我们得到

$$s(x) = rtfp(x) = \lim_{r\to\infty} fp(x)$$

□

在无限重复下，无论使用 Denning递归还是使用足迹计算，平均工作集大小都相同。换言之，在有限长度程序上使用 Denning递归时，虚拟重用保证了其正确性。

由于篇幅所限，此处省略证明；我们还可以证明，当工作集（缓存）大小等于或超过数据大小时，未命中率会降至 0。读者可能会质疑，在实际执行中，冷启动缺失是固有的，不可能为零。接下来，我们将考虑 LRU缓存和冷启动缺失。

2.4 重用间隔和不变性与符号测试

根据工作集正确性（定理 2.1），将时间尺度参数取为 $x = 0$，我们可以推导出以下引理。当 $x = 0$时，零长度窗口不包含任何数据访问，因此工作集大小或足迹为零。我们将其称为零足迹引理。

引理 2.2（零足迹）。在无限重复下，$rtfp(0) = m - \sum_{ri=0}^{n} ri\ P(ri) = 0$，其中 m 是数据大小，$P(ri)$ 是所有 n 个不同重用间隔值 ri 所占的份额。

重新排列各项后，我们得到 $\sum_{ri=0}^{n} ri\ P(ri) = m$。如果将 n RI 值和 n 重用区间占比视为向量，那么左侧就是它们的点积。我们将该方程称为 *RI* 和不变性。该不变性表明：值向量与比例向量的点积必须等于数据大小，或 $\mathbf{v} \cdot \mathbf{p} = m$。

该不变性提供了一种符号重用间隔测试，用于检查符号 RI 值或重用区间占比是否存在错误。根据点积规则：**所有乘积项必须相互抵消**。对于无限重复下的任何程序，该检查都必须通过；否则，某个 RI 值或其重用区间占比必然存在错误。

符号重用间隔测试并不是 RI 正确性的证明 。它检查一组符号项是否必须全部相互抵消。它可以排除单项错误，但多个错误可能相互抵消，仍然通过测试。从统计角度看，随着符号 RI 值数量的增加，该测试能够提供更高的正确性保障。程序越复杂，其包含的 RI 值就越多，发生错误的可能性也越大。尽管并不完美，无论分析是手动完成还是由编译器自动完成，该测试都能提供一种实用的保护措施。

2.5 LRU 近似

我们的工作使用平均工作集大小 $c(x)$近似 LRU 缓存大小。

假设 1 (LRU 近似)。 LRU 缓存的未命中率可以通过以下方式近似为工作集缓存的未命中率：

$$mr(c(x)) \approx m(s(x))$$

其中 $c(x) = s(x)$。

由于 $s(x)$ 是分数，近似得到的 LRU缓存大小 $c(x)$ 也可能是分数。要理解分数形式的缓存大小，请注意， $s(x)$ 是单调的，并且每一步最多增加一（$s(x+1) - s(x) \leq 1$对于 $x > 0$）。因此，每个整数缓存大小 k 要么恰好等于某个 x 对应的 $s(x)$，要么介于两个相邻的值 $s(x)$ 和 $s(x+1)$ 之间，且这两个值的差小于一。根据单调性，整数缓存大小 k 的未命中率可以用周围分数缓存大小对应的未命中率来界定。因此，该近似实际上计算的是连续缓存大小对应的未命中率，从而可以插值得到任意整数值。

许多以往的技术都使用最近最少使用近似，包括程序分析中的时间到重用距离转换 [45], 使用硬件计数器的 Featherlight RD [52], StatStack [17], 存储缓存的平均驱逐时间（AET） [38] （如第6节所述）。这些方法在大规模轨迹上通常表现出很高的准确性。与这些研究不同，我们将这种近似用于循环内核。

移动悬崖误差。Chen等人研究了循环内核 [10]。他们发现了一种主要的误差类型及其成因。如果内核包含阶段，即多个循环嵌套，预测往往不准确。这种误差表现为“移动悬崖”：将预测结果与最近最少使用模拟进行比较时，预测的未命中率下降出现得更早。下降前后的未命中率都能得到正确预测，但预测的缓存大小偏小。

冷启动缺失。为恢复实际未命中率，我们“撤销”无限重复的影响。Denning递归会同时得出缓存大小和未命中率。计算所得的缓存大小是正确的，因为它涵盖了所有不同的数据项，无论这些数据项是通过真实重用还是虚拟重用访问的。然而，未命中率将首次访问视为虚拟重用（即命中，仿佛它们在上一次迭代中已被访问）。实际上，单次程序执行中的首次访问应当是冷未命中 [23]。因此，我们通过将虚拟重用命中还原为未命中来调整未命中率。归因于虚拟重用的命中部分，代表原始单次执行程序的冷启动未命中率。

2.6 推导示例：朴素矩阵乘法

本节以朴素矩阵乘法为具体示例，说明如何推导符号化 RI 分布，并通过 Denning递归将其转换为缓存未命中多项式。

矩阵乘法是科学计算中的基本运算，也是用于局部性分析的一个直观示例。我们考虑用于计算两个 $n \times n$ 矩阵 A、 B 和 C乘积的标准三重嵌套循环。为简化分析，我们假设每个缓存块存储 8 个数据元素（块大小 $b = 8$）。

```
for (int i = 0; i <n; i++) for (int j = 0; j <
n; j++) for (int k = 0; k <n; k++) C[i][j
] += A[i][k]* B[k][j];
```

每次最内层循环迭代都建模为按 C语言、A、B、C语言顺序进行的四次内存访问：首先读取值 $C[i][j]$，然后访问 $A[i][k]$和 $B[k][j]$，最后将 $C[i][j]$写回。这符合朴素实现的典型内存行为，总计产生$4n^3$ 次访问。我们采用无限重复下的虚拟 RI 值，以确保每次访问都关联一个 RI，如第 2.2 节所述，其中也包括对应冷未命中或首次访问某个数据块的访问。使用所有符号 RI 值（包括虚拟 RI）构建完整的代数失效率表后，我们会在报告最终未命中率时移除对应虚拟 RI 的份额。这样调整后，报告的未命中率便能反映所有缓存未命中（包括冷未命中）。

表 1. 朴素矩阵乘法的分析 s的朴素矩阵乘法（*包含 虚拟重用间隔）

	RI	P (ri)	m (ri)	冷缺失率	c（ri）
0	–	–	1	–	0
1	1	$\frac{1}{4}$	$\frac{3}{4}$	–	1
2	3	$\frac{1}{4} - \frac{1}{32n}$	$\frac{1}{2} + \frac{1}{32n}$	–	3
3	4	$\frac{7}{32}$	$\frac{9}{32} + \frac{1}{32n}$	–	5
4	$4n - 28$	$\frac{1}{32} - \frac{1}{32n}$	$\frac{1}{4} + \frac{1}{16n}$	–	$\frac{9n}{8} - \frac{47}{8} - \frac{31}{32n}$
5	$4n$	$\frac{7}{32}$	$\frac{1}{32} + \frac{1}{16n}$	–	$\frac{9n}{8} + \frac{9}{8} + \frac{25}{32n}$
6*	$4n^2 - 28n$	$\frac{1}{32}$	$\frac{3}{32n}$	$\frac{1}{32n}$	$\frac{n^2}{8} + \frac{3n}{8} - \frac{7}{8} + \frac{25}{32n}$
7*	$4n^3 - 4n^2 + 4n - 28$	$\frac{3}{32n}$	$\frac{3}{32n}$	$\frac{2}{32n}$	$\frac{3n^2}{8} - \frac{n}{8} + \frac{9}{8} - \frac{31}{32n}$
8*	$4n^3 - 32n + 3$	$\frac{1}{32n}$	$\frac{3}{32n}$	$\frac{3}{32n}$	$\frac{3n^2}{8}$

RI 分布及其推导出的缓存未命中多项式见表 1。表中以 $*$ 标示虚拟重用的行索引。它们的 RI 区间占比之和即为冷启动未命中率。

RI 和不变性。表格的第二列和第三列1 列出了八个重用间隔值及其对应的份额。根据 RI 和不变性，它们的点积必须等于数据总量，形式化地说，$\mathbf{v} \cdot \mathbf{p} = m$，其中重用间隔值（$\mathbf{v}$）和份额（$\mathbf{p}$）都是 n 中的多项式。

计算这个点积会得到一个包含近 20 项的多项式，其中大多数是非常数项。求和后，所有常数项之和为零，所有线性项相互抵消，所有倒数项也同样消失。唯一保留下来的是二次项，它们的总和为 $\frac{3n^2}{8}$ ——恰好等于三个 $n \times n$ 矩阵中的数据块总数（每块 8 个元素）。因此，RI 和不变性成立，测试通过。

3 编译器设计与实现

我们的框架利用 MLIR [29] 作为灵活的中间表示桥接器。借助 MLIR Affine 方言，我们的符号局部性编译器可以集成到现有工作流中，例如 Polygeist 流水线 [34]，而无需更改前端或后端。这种设计使我们能够改进各种使用 MLIR 基础设施的编程语言和工具中的局部性分析与优化。

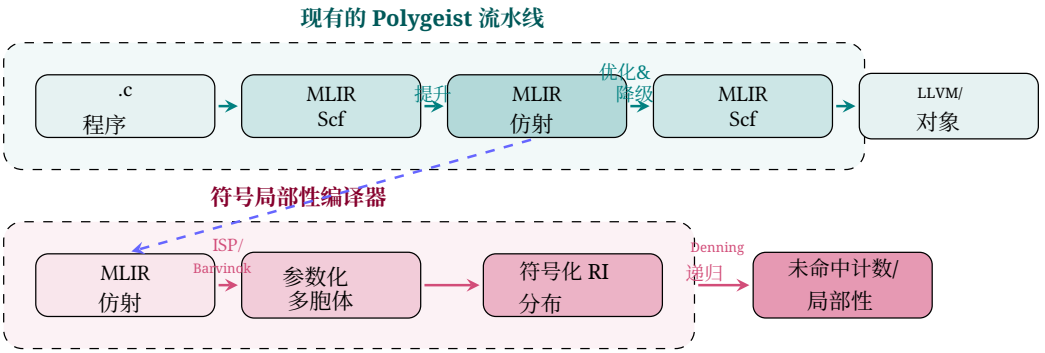


图3. 展示现有 Polygeist 流水线与我们的符号局部性编译器的工作流程图，二者通过 MLIR仿射表示相互连接。

3.1 仿射程序定义

我们将以下程序结构定义为分析目标，它是从 MLIR 的 仿射方言简化而来。我们的框架不允许在仿射表达式中将符号用作系数。表2分两部分展示了语法：左列定义核心程序构造（循环、条件语句、块和内存访问），右列定义仿射表达式和约束集的辅助构造。这里 $\mathcal{S}$ 是符号集，数组 用于标识不相交的内存对象，常量 表示整数字面量，而 $ivar = i_{level}$ 表示给定嵌套层次上的循环索引变量。

表 2. 仿射程序语法

核心语法	辅助定义
程序 = （S，语句） 语句 =循环 | 块 | 如果 | 访问 循环 =**对于** 表达式 **至** 表达式 **步长** 常量 **执行** 语句 **结束循环** 块 = 语句 （；语句）+ 如果 =**如果** 集合 **那么** 语句 **否则** 语句 **结束如果** 访问 = **访问** 数组[表达式(，表达式)∗]	原子 =常量 | 归纳变量 | $s \in \mathcal{S}$ 表达式 =原子 | 表达式 ± 表达式 | 常量 ×表达式 $\mid \lceil \frac{expr}{常量} \rceil \mid \lfloor \frac{expr}{常量} \rfloor$ 约束 =表达式 = **0** | 表达式 < **0** 合取 =约束 (∧ 约束)∗ 集合 =合取 (∨ 合取)∗

3.2 时间戳与内存访问的仿射表示

为了分析重用间隔，我们需要获取访问时间戳集合。这与迭代空间不同，因为时间戳与每次内存访问相关联，并不一定描述循环索引变量的取值。我们展示一种将仿射程序转换为整数向量空间的转换方法，并在图 4中概述该算法的形式语义，并在图5中给出一个具体示例。

我们定义一种操作 $[\![\cdot]\!](\mathrm{M};\ell)$，用于将程序表示 $p \in \mathrm{P}$ 转换到目标空间 $\mathrm{T} \subseteq \mathbb{Z}^n$，其中包含某个 n，并以 M 为上下文，且 ℓ。转换中会用到以下概念：

- M 将索引变量（按循环层级）映射到向量分量，其中 $|\mathrm{M}|$ 等于作用域内索引变量的数量。它会将步进循环索引规范化到连续迭代空间。
- ℓ 统计截至当前构造所使用的时间戳向量维度；进入循环或语句块时递增，以区分内存访问。
- L（s）表示空间 s 的向量维度。
- $x|_{\mathcal{M}}$ 表示在表达式 x 中用 M 替换索引变量（当 x为集合时，则替换约束表达式中的索引变量）。
- $a \cup^{\uparrow} b$和 $a \cap^{\uparrow} b$表示标准的集合并集与交集，当 $\mathrm{L}(a) = \mathrm{L}(b)$ 时；否则，通过添加不受约束的尾随维度，将低维空间提升。我们可以使用相应的 $\sqcup, \sqcap$ 变体来说明对不相交子集的运算。

图 4 归纳定义了转换算法。基本情况涉及内存访问，此时算法返回一个无约束集合。在其他情况下，算法递归地应用于子语句，使空间维度对齐，并添加父级施加的约束以细化空间。

$$[\![\cdot]\!](\mathcal{M};\ell) : \mathcal{P} \to \bigcup_{n\in\mathbb{N}} \mathbf{Pow}(\mathbb{Z}^n)$$

$$[\![(\mathcal{S}, s)]\!](\mathcal{M};\ell) = [\![s]\!](\mathcal{M};\ell)$$

$$[\![\textbf{access } A\ [e_0, ..., e_n]]\!](\mathcal{M};\ell) = \mathbb{Z}^{\ell}$$

$$[\![\textbf{对于}l\textbf{到}u\textbf{，步长为}c\textbf{，执行}b\textbf{结束循环}]\!](\mathcal{M};\ \ell) = \textbf{令}\mathcal{M}' = \mathcal{M} \cup \{|\mathcal{M}| \mapsto (l\ |_{\mathcal{M}} + c \times v[\ell])\};\ \ s = [\![b]\!](\mathcal{M}';\ \ell+1)\textbf{在}\ \ s\cap\uparrow\left\{v \in \mathbb{Z}^{\mathcal{L}(s)} :\ \ (v[\ell] \geq 0) \wedge \left(v[\ell] < \left\lceil \frac{u\ |_{\mathcal{M}} - l\ |_{\mathcal{M}}}{c} \right\rceil\right)\right\}[\![\textbf{如果}$$

$$[\![s_0; s_1; ...; s_n]\!](\mathcal{M};\ell) = \textbf{let}\ \overset{n}{\underset{i=0}{\forall}} s_i' = [\![s_i]\!](\mathcal{M};\ell+1)\ ;\mathcal{L}^* = \max_{i=0\ldots n} \mathcal{L}(s_i)$$
$$\textbf{in} \bigsqcup_{i=0}^{n} \uparrow \left(s_i' \cap \uparrow \left\{ v \in \mathbb{Z}^{\mathcal{L}^*} : (v[\ell] = i) \wedge \left(\bigwedge_{j=\mathcal{L}(s_i')}^{\mathcal{L}^*-1} v[j] = 0 \right) \right\} \right)$$

$$c\textbf{则}\ t\textbf{否则}\ e\textbf{结束条件}](\mathcal{M};\ \ell) = \textbf{令}\ t' = [\![t]\!](\mathcal{M};\ell);\ \ e' = [\![e]\!](\mathcal{M};\ \ell);\ \mathcal{L}^* = \max\{\mathcal{L}(t'),\ \mathcal{L}(e'\ )\}$$
$$\textbf{in} \left(t' \cap \uparrow \left\{ v \in \mathbb{Z}^{\mathcal{L}^*} : c|_{\mathcal{M}} \wedge \left(\bigwedge_{j=\mathcal{L}(t')}^{\mathcal{L}^*-1} v[j] = 0 \right) \right\} \right)$$
$$\sqcup \left(e' \cap \uparrow \left\{ v \in \mathbb{Z}^{\mathcal{L}^*} : \neg c|_{\mathcal{M}} \wedge \left(\bigwedge_{j=\mathcal{L}(e')}^{\mathcal{L}^*-1} v[j] = 0 \right) \right\} \right)$$

图 4. 时间戳空间构造的形式语义

时间戳空间按字典序排列。可以在原始程序 p 的内存访问轨迹与时间戳空间 $\mathrm{T} = [\![p]\!](\emptyset,\ 0)$ 之间建立保序双射。时间戳空间的大小与程序访问轨迹的长度相匹配。

示例 *3.1*。 考虑图 5 左侧所示的程序。我们使用 s_i 表示第 i 行的单条语句，并使用 b_i 表示从第 i 行开始的整个代码块。右侧的表总结了每个 s_i 的局部输入上下文和输出，以及 b_i。值得注意的是， s_2、 s_5、 s_8 和 s_9 都是叶语句。顶层结果 s_1 由这些内部结果逐步推导而来。上下文 $1 \mapsto d_0 + 4d_2 - 512$ 中的第二个映射 $\mathcal{M}$ 在 s_5 此处未使用，但在后续章节构建访问映射时将变得十分重要。

1: **for**0 **to**2026 **step**1 **do**2: **访问** $A[i_0]$; 3: **if** $i_0 > 1024$**then**4: **for** i_0 - 512 **to** $i_0 + 512$ **step** 4 **do**5: **访问** $B[i_0, i_1]$ 6: **end for**7: **else**8: **访问** $A[i_0 + 512]$; 9: **访问** $C[2i_0 + 1]$10: **end if**11: **end for**

语句	ℓ	结果
s_1	0	$d_0 \in [0, 2026)\ \ d_1 = 0,\ \ d_2 = 0$
		$\vee\ d_0 \in [0, 1024)\ \ d_1 = 1,\ \ d_2 \in [0, 256)$
		$\vee\ d_0 \in [1024, 2026)\ \ d_1 = 1,\ \ d_2 \in [0, 2)$
b_2	1	$d_1 = 0,\ \ d_2 = 0$
		$\vee\ d_0 < 1024,\ \ d_1 = 1,\ \ d_2 \in [0, 256)$
		$\vee\ d_0 \geq 1024, d_1 = 1, d_2 \in [0, 2)$
s_2	2	$\mathbb{Z}^2$
s_3	2	$d_0 < 1024, d_2 \in [0, 256)$
		$\vee\ d_0 \geq 1024, d_2 \in [0, 2)$
s_4	2	$d_2 \in [0, 256)$
s_5	3	$\mathbb{Z}^3$
b_8	2	$d_2 \in [0, 2)$
s_8	3	$\mathbb{Z}^3$
s_9	3	$\mathbb{Z}^3$

图5. 时间戳空间推导示例（符号的解释见示例 3.1）。

构建时间戳空间后，我们附加一个仿射访问映射，将每个时间戳与其访问的数据项关联起来。设 R_p^* 为程序 p中所有数组的最大秩。访问映射 $\mathrm{A}p : \mathrm{T} \to \mathbb{Z}^{\mathcal{R}_p^*+1}$ 将每个时间戳 $t \in \mathrm{T}$ 映射到一个数据向量 $m \in \mathbb{Z}^{\mathcal{R}_p^*+1}$，其中第一个分量编码数组标识符，后续 r 分量表示其索引。

可以对这些索引应用额外的仿射变换 F，以对布局变换或块级重用进行建模。例如，如果将数组元素沿最后一个维度按大小为 B 的块分组，则以下变换可调整访问粒度：

$$\mathcal{F}_{\mathcal{B}}([x_0, \ldots, x_{r-1}, x_r]) = [x_0, \ldots, x_{r-1}, \lfloor x_r/\mathcal{B} \rfloor].$$

在转换过程中，访问映射与时间戳空间共享大部分操作，但对于访问语句，其值域会被显式定义：

$$[\![\textbf{access}\ A[e_0, \ldots, e_n]]\!](\mathcal{M}; \ell; \mathcal{F}) = \{v \mapsto \mathcal{F}([A, 0, \ldots, 0, e_0|_{\mathcal{M}}, \ldots, e_n|_{\mathcal{M}}]) : v \in \mathbb{Z}^{\ell}\}.$$

这种构建通过加入能够捕捉数据访问行为的仿射映射，有效扩展了时间戳空间。

3.3 将 RI 分布计数为分段准多项式

我们将重用间隔分布表述为一个计数问题，并通过 Barvinok 分解和重新参数化求解。我们首先介绍整数集合规划和 Barvinok 计数，然后推导符号化 RI 表示并计算完整分布。

3.3.1 背景：整数集合规划与 *Barvinok* 计数。整数集合规划提供了操作仿射集合和关系的代数框架。我们使用整数集合库（ISL） [49] 处理仿射结构，并使用 Barvinok 库 [50] 计算基数。

ISL 映射通过仿射变换表示定义域与陪域之间的多值函数。当映射 R 的定义域和陪域相同 X 时，它定义了一个二元关系 R：X 其中 $(x,\ x') \in \mathrm{R} \Longleftrightarrow\ x' \in R(x)$。我们使用以下操作（在仿射整数集合和映射下均封闭）： A^{-1}（逆）， $A \succ B$（将 $a \in A$ 与所有更小的 $b \in B$相连的字典序）， $A \succeq B$， $A \prec B$， $A \preceq B$（类似的序），**字典序最大值** A（每个像的字典序最大值）， $A \cap B$（交集）， $A \circ B$（复合）。

Barvinok 方法用于计算形式为 $\{x \in \mathbb{Q}^d \mid Ax \geq Bp + c\}$ 的参数多面体中的整数点，其中 $A \in \mathbb{Z}^{m \times d}$， $B \in \mathbb{Z}^{m \times n}$， $c \in \mathbb{Z}^m$。它输出**分段准多项式**，其中每一段都将一个区域与一个准多项式相关联（准多项式通过向上取整除法和取模运算扩展了普通多项式），用于计算该区域中的整数点。

定义 *3.2*（向后重用间隔）。令 $\mathrm{T}[t]$ 表示在时间戳 t 访问的数据项。定义 $t' = \max\{s \mid 0 \leq s < t, \mathrm{T}[s] = \mathrm{T}[t]\}$ 为最近一次访问该数据项的先前时间戳。时间 t 处的向后重用间隔为 $ri_t := t - t' = \left|\left\{ i \,\middle|\, t' < i \leq t \right\}\right|$.

若不存在这样的 t'，则重用间隔$_t := \infty$（冷缺失）。我们将对应的集合 $\left\{ i \,\middle|\, t' < i \leq t \right\}$ 称为 t处的**ri 集**，记为**ri 集**(t)，其中重用间隔$_t = |$**ri 集**$(t)|$。

3.3.2 符号*RI*分析。 我们通过纤维划分，将仿射关系取交集来构造**ri 集** 。单值映射 $A : X \to Y$ 导出**纤维关系** $K_A := \{(x,\ x') \in X \times X \mid A(x) = A(x')\} = A^{-1} \circ A$，它将 X 划分为像相同的等价类（纤维）。将 K_A 与另一个二元关系 R 取交集，可得到同时满足这两个约束的时间戳。

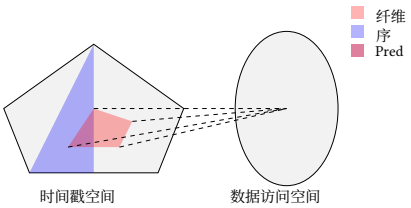


图 6. **prev** 映射的构建示意图。

设 T 为时间戳空间，A : T → D 将每个时间戳映射到其访问的数据项。前驱集为

$$\mathbf{prev}(\mathcal{A},\mathcal{T}) = \left(\mathcal{A}^{-1} \circ \mathcal{A}\right) \cap (\mathcal{T} \succ \mathcal{T}),$$

将纤维关系与字典序相交（见图6）。随后，**ri 集**为

$$\mathbf{ri\text{-}set}(\mathcal{A},\mathcal{T}) = \left(\mathbf{lexmax}\ \mathbf{prev}(\mathcal{A},\mathcal{T}) \circ (\mathcal{T} \prec \mathcal{T})\right) \cap (\mathcal{T} \succeq \mathcal{T}).$$

Since **ri 集**是仿射映射，因此我们通过 Barvinok 计算每个 RI 值，得到 $ri_t = |\mathbf{ri\ 集}(t)|$.

3.3.3 通过重新参数化重新统计 *RI* 分布。将所有 RI 值计算为准多项式还不够；我们还必须统计每个值的出现次数。对于符号参数，枚举所有 RI 值并不可行。但并非毫无办法——我们可以利用定义域的分段结构高效地进行计数，在每个仿射区域内汇总出现次数，而不是逐个遍历时间戳。

算法 1 接受分段准多项式输入 $\{(S_i, \mathrm{QP}_i)\}$，其中 S_i 是时间戳的子集，QP_i 是其 RI 公式。由于不相交子集可能具有相同的 RI 值，我们按值对它们分组并求和其基数。该算法分两个阶段运行：(1) 重新参数化（第 4–13 行）：对于维度向量为 x、参数向量为 p的约束 $Ax \geq Bp + c$，我们将 QP_i 中出现的任何维度 x_i 从 x 移至 p，从而得到 x' 和 p'。变换后的约束 $A'x' \geq B'p' + c$ 保留多面体结构，并对准多项式进行规范化，以便在映射 T中按 RI 值对子集分组。(2) 计数（第 14–17 行）：我们计算每个分组子集的基数 c_q，以获得出现次数，并输出分布对 (S_j, c_j)，其计数为符号计数 c_j。

3.3.4 无限重复。为对无限重复建模，我们添加一个重复维度 $r \in [0, R)$，其中 R 是一个表示迭代次数的符号参数。这使时间戳空间和 RI 计数都随 R 线性增长。为计算 $R \to \infty$ 时的极限重用区间占比，我们应用洛必达法则：由于分子（RI 计数）和分母（总访问次数）都是关于 R 的多项式，该极限等于它们的首项系数之比。这样无需显式计算极限，就能高效地以符号方式计算无限重复复用间隔占比。

4 重用间隔分布推导的难度

在确立算法框架后，我们现在先考察 RI 分布推导固有的计算复杂度，再评估我们的实现。这一分析说明了为什么即便是高效工具，在某些程序类别上也会面临根本性的限制。

我们证明，为仿射程序推导 RI 分布在计算上是困难的。虽然人们可能会认为，只有控制流复杂的程序才难以处理，但我们表明，即使面对结构出奇简单的程序，这个问题仍然难以解决：无分支[3] 具有多项式有界内存的嵌套循环。

[3]“无分支”是指不存在 if–else 语句。

算法 1 计算 RI 分布的重新参数化

要求： 分段准多项式 $\{(S_i,\ QP_i)\}_{i=1}^{n}$，其中 S_i 是一个域子集，QP_i 是一个准多项式
确保： 分布 $Dist$ 为 $\{(S_j,\ c_j)\}$，其中 c_j 是基数公式

```
初始化映射 T，每个条目的默认值为 ∅
对于 每个 i ∈ {1, ..., n} 执行
    令 D_i, P_i 为 S_i 的维度和参数域
    D' ← ∅, P' ← P_i
    对每个 d ∈ D_i 执行
        如果 d ∈ 变量(QP_i) 则
            P' ← P' ∪ {d}                          ▷ 将维度移至参数
        else
            D' ← D' ∪ {d}
        结束 if
    结束 for
    令 S'_i 为 S_i 在 (D', P') 上的重投影
    令 QP'_i 为 S'_i 上的归一化 QP_i
    T[QP'_i] ← T[QP'_i] ∪ S'_i                     ▷ 按 RI 值对各子集分组
结束循环
分布 ← ∅
对于 每个 q ∈ 键(T) 执行
    计算 c_q ← 基数(T[q])                          ▷ 统计 RI 值的出现次数 q
    分布 ← Dist ∪ {(T[q], c_q)}
结束 for
返回 Dist
```

我们的归约利用了一项基本性质：访问一个内存位置会拦截原始轨迹，将较长的重用间隔分割成较短的间隔。这种拦截机制使我们能够将计算问题编码到 RI 分布中：满足赋值会诱导出策略性内存访问，从而产生较短的 RI；而不可满足的公式只会产生较长的间隔。

我们通过将 3-SAT 归约到重用间隔查询来证明其困难性。对于一个 3-SAT 公式

$$\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m,$$

其中 $C_i = l_{i1} \vee l_{i2} \vee l_{i3}$，且每个文字均取自变量 $x_k (k \in [n])$，我们将每个子句编码为

$$\left\lceil \frac{v_{i1} + v_{i2} + v_{i3}}{3} \right\rceil, \qquad v_{ij} := \begin{cases} x_k & l_{ij} = x_k, \\ 1 - x_k & l_{ij} = \neg x_k. \end{cases}$$

当子句被满足时该值为 1，否则为 0。由于 $m \le O(n^3)$，我们只需要多项式规模内存。

定理 4.1. 判定多项式有界内存的无分支仿射程序中是否出现特定的重用间隔值，是 *NP* 完全问题。

证明。 我们使用图 7a中的程序，将 3-SAT 归约为重用间隔检测。嵌套循环会枚举所有 2^n 真值赋值。若 ϕ 可满足，则至少有一次迭代会计算 $\sum C_i = m$，因此程序会在循环内访问 $A[m]$。这会截断 轨迹，从而产生重用间隔值 $< 2^n + 1$。若不可满足，则每次迭代都会访问不同的位置，因此边界访问之间唯一的重用间隔恰好为 $2^n + 1$。因此，重用间隔检测是 NP 完全问题。 □

定理 4.2。 在无分支仿射程序中，计算特定重用间隔值的出现次数是 *#P* 完全的。

```
访问 (A[m]);
for (int i1 = 0; i1 <= 1; i1++)
  对于 (整数 i2 = 0; i2 <= 1; i2++)
    ...
      for (int in = 0; in <= 1; 在 ++)
        访问 (A[C语言1 + C语言2 + ... + C语言m]);
访问 (A[m]);
```

(a) NP完全性归约中使用的程序。

```
for (int i1 = 0; i1 <= 1; i1++)
  for (int i2 = 0; i2 <= 1; i2++)
    ...
      对于 (整数 in = 0; in <= 1; 在++) {
        访问 (A[m]);
        访问 (A[C语言1 + C语言2 + ... + C语言m]);
        访问 (A[m + 1]);
      }
```

(b) #P 完全性归约中使用的程序。

图 7. 困难性归约中并列展示的程序。

证明。 我们使用图中的程序将 #3-SAT 归约为 RI 计数7b。对于每个满足赋值（$\sum C_i = m$），对 $A[m]$ 的两次连续访问恰好产生一个值为 1的重用间隔。对于不满足的赋值，三次访问都触及不同地址，因此不会产生值为 1的重用间隔。因此，值为 1 的重用间隔数量等于满足赋值的数量，从而证明了 #P 完全性。 □

这些结果表明，推导完整的重用间隔分布在本质上十分困难——即使对于具有多项式规模内存的无分支仿射程序也是如此。轨迹拦截机制具有足够的表达能力，可以编码 NP 困难和 #P 完全问题，这意味着困难性源于循环与访问结构本身的表达能力，而非控制流复杂度或无界内存。

如第 5 节所示，尽管一般问题是 NP 困难的，我们的编译器仍能实现良好的实际性能，因为：（1）仿射结构会产生低维多面体；（2）Barvinok 算法在这类情形下表现良好；（3）大多数基准测试中不同的重用间隔值很少。困难性主要影响循环维数较高或访问模式经过对抗性构造的极端情况。

5 评估

本节评估新理论及其编译器实现的时间开销，以及对最近最少使用缓存未命中率的近似准确度。

5.1 实验设置

我们使用两个基准测试套件评估我们的方法：包含 11 个 Einsum 循环内核的集合 [8]以及来自 Polybench 套件的 30 个程序 [41]。Einsum 循环常用于机器学习内核和量子电路（张量网络）模拟 [16, 20, 25, 28, 54, 55]。我们省略了 Einsum 基准中的 17 个示例内核中的 6 个，因为它们的数据重用模式很容易分析，例如仅涉及缓存分块重用。按照 [20]中的建议，我们没有使用完整循环，而是将 Einsum 循环拆分，并缓存反复使用的中间结果。输入程序按照第3节所述进行编译和分析。除了原始代码外，我们先应用循环融合变换，再进行分析。我们使用`mlir-opt` 来应用`affine-loop-fusion`通道 [31]。它专门处理以 MLIR 仿射方言表示的循环。由于早期版本的融合成本模型中存在一个已知错误，我们要求使用 LLVM 21 或更高版本。

我们使用 Cachegrind [47] 进行缓存模拟。执行时间数据采集自一台 64 核 AMD EPYC 7773X 处理器。硬件计数器实验在一台 20 核 Nvidia GB10[4] 上运行。

[4]我们将测试程序绑定到 GB10 性能最强的 Cortex-X925 核心。

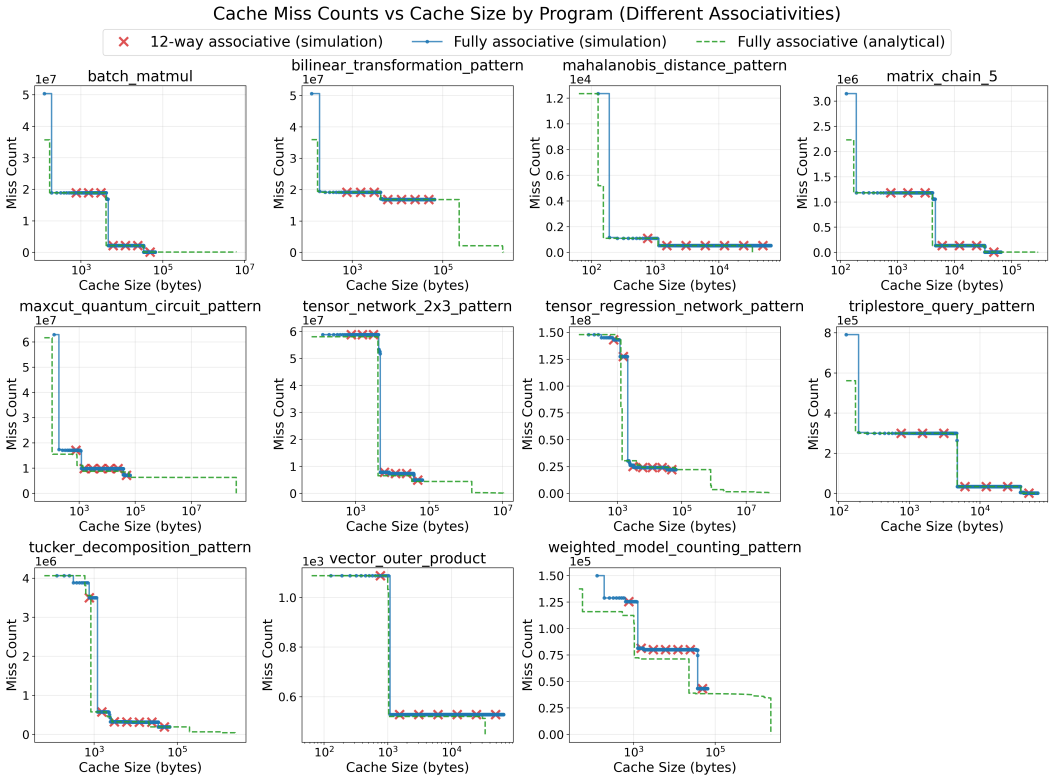


图 8. 张量收缩内核未命中计数曲线

所有程序均使用 -O3 标志编译，以减少栈溢出及其他可消除的内存访问。

我们在所有程序中手动应用数组填充，以避免冲突未命中。最内层（连续）维度被填充至 $8p$，其中 8 是以数据元素数量计的缓存块大小，而 p 是使 $8p$ 大于或等于原始数组维度的最小素数。这满足最后一维为块大小倍数的对齐要求。对于除最外层之外的所有外层维度，我们向上取整到大于或等于该维度大小的最近素数。

5.2 未命中计数与未命中率精度

我们首先将预测与 Cachegrind 的模拟进行比较。图 8 通过未命中率曲线展示了不同缓存大小下的比较。对于 Einsum 循环，预测未命中率曲线与模拟的全相联缓存相匹配，尤其是在缓存大小超过一百字节时，11 项测试中有 10 项如此。例外是 加权_模型_计数_模式，它表现出移动悬崖误差，即它正确预测了未命中率，但低估了下降发生时的缓存大小（见第 2.5节）。

由于篇幅限制，我们对30项Polybench测试使用三种缓存大小，即6KB、12KB和48KB，并将结果一并展示于图9中。对于每种缓存大小，我们比较全相联与12路组相联度。对于每种缓存配置，表中给出两种未命中率：

模拟与预测。最后四列展示了三种缓存大小下平均值的比较。

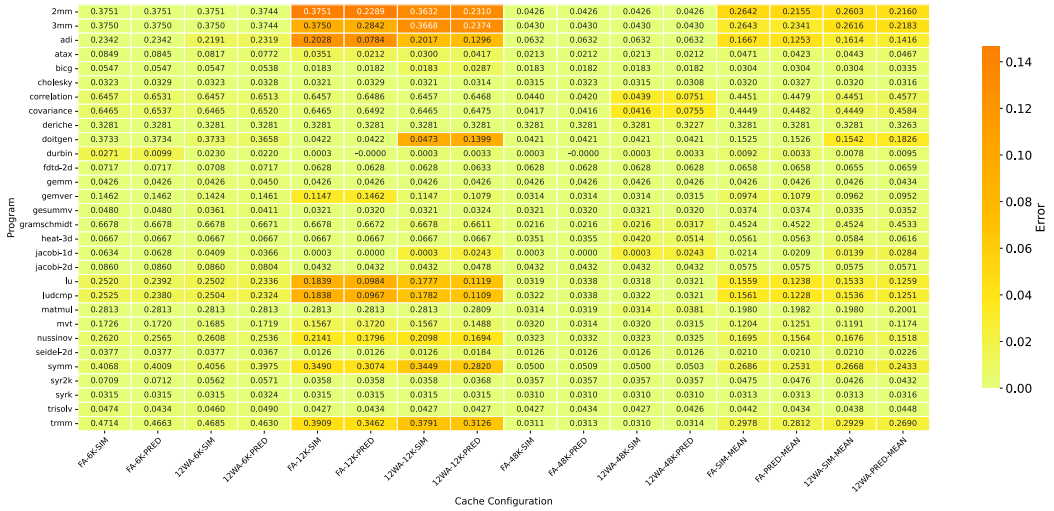


图 9. 不同缓存配置下，30 个 Polybench 基准测试的预测误差。各列分别表示不同大小的全相联（FA）和 12 路组相联（12WA）缓存。单元格颜色反映预测误差的大小，单元格中的数字表示模拟未命中率。

表格根据预测误差着色，该误差是预测未命中次数与模拟未命中次数之差的绝对值除以总访问次数。全部 180 次比较中的最大误差为 14%，对应右侧色条的最高值。大多数比较（60%）的误差小于 1%，其中多数（45%）小于 0.1%。只有 13 次比较（占测试的 7.2%）的误差达到或超过 5%，且都发生在缓存大小为 12KB 时。

对三种缓存大小取平均后可以看到，全相联情况下的预测非常准确：12 次测试的误差小于 0.01%，9 次测试的误差介于 0.01% 和 1% 之间，其余 9 次测试的误差介于 1% 和 5% 之间。全部 30 次测试的平均误差为 1.1%。相比之下，平均未命中率为 16%。

组相联缓存。 组冲突会显著增加未命中计数。在一个 12 路组相联缓存中，如果数组访问的步长与数组的连续维度对齐，使其与 12 有公因数，那么数组数据的映射可能会产生偏差，只利用少量缓存组。

我们的填充方案（第 5.1 节）有效消除了冲突未命中，使组相联缓存的表现与全相联缓存相同。图 8 中 11 个 Einsum 测试的未命中计数曲线展示了这一点：组相联缓存的未命中率（以 × 标记）落在全相联缓存的未命中率曲线上。

我们在三种大小的组相联缓存上测试了 Polybench，结果以颜色编码显示在图 9 中，最后两列显示了三种缓存大小的平均值。总体而言，平均未命中率相近：12 路组相联缓存为 15.6%，全相联缓存为 16.1%。准确度也相近：10 个测试的误差低于 0.01%，12 个测试的误差介于 0.01% 和 1% 之间，其余 9 个测试的误差介于 1% 和 5% 之间。平均误差为 1.3%。

人们已经开发出统计技术来对组相联缓存建模。Smith [46] 提出了利用重用距离估算组相联缓存中冲突误差的经典公式，该公式已用于后来的工具中，包括 HPCToolkit [33]。还有另外两种模型：Set-RD [36, 44] 和

映射足迹 [32]。对于规则的循环代码以及现代处理器采用的高相联度，我们发现，数组填充足以使其性能与全相联缓存相近。

循环融合优化。 循环融合已被广泛研究，例如在 [11, 15] 中。我们在 MLIR 的仿射循环融合之后测试我们的程序，该融合具有两种融合策略[5]：生产者-消费者融合和兄弟循环融合。生产者-消费者融合侧重于合并这样的循环对：生产者循环写入某一内存引用，而消费者循环随后读取该引用。兄弟循环融合处理的循环对不一定存在数据依赖，但具有相同的迭代。我们对生产者-消费者融合使用 LLVM 21 中附带的默认计算开销阈值（30%）。

图 10 中超过 86% 的误差小于 2%，并且除 `lu` 和 `ludcmp` 外，所有误差均小于 4%。在这两个程序中，循环边界依赖于归纳变量。我们还注意到另一个案例 `adi`。在循环融合之前，由于存在多个循环嵌套，它的移动悬崖误差高达 12%。循环融合之后，预测完全准确。

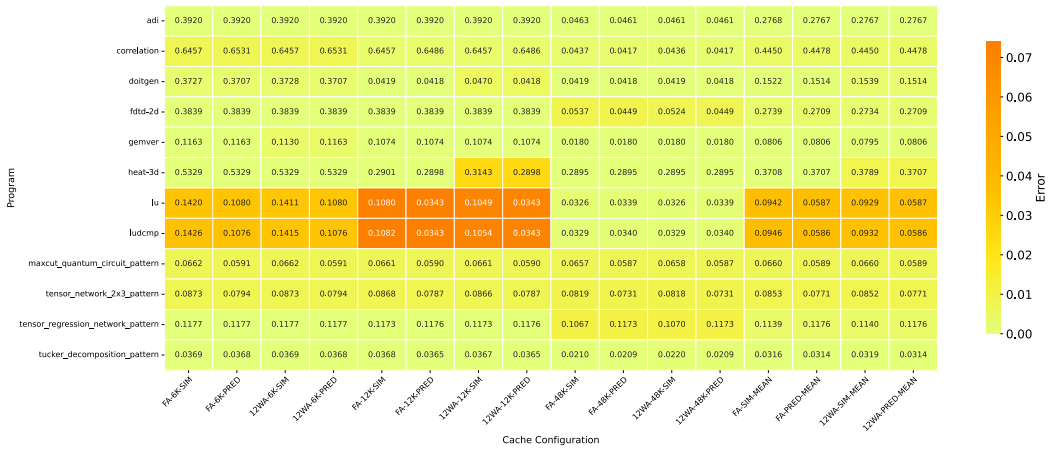


图 10. 在不同缓存配置下，受 MLIR 仿射循环融合影响的 12 个程序中的预测误差。各列表示不同大小的全相联（FA）和 12 路组相联（12WA）缓存。单元格颜色反映预测误差的大小，而每个单元格中的数字表示模拟未命中率值。

48KB 12 路组相联缓存。 这种特定的缓存配置采用 64 字节块（8 个双精度浮点数），见于英特尔酷睿处理器第 10 代至第 14 代的性能核心中，涵盖 Sunny Cove（冰湖处理器）、Golden Cove 和 Redwood Cove（Alder Lake 和 Raptor Lake）。此外，AMD 的 Zen5 核心也为其 L1 数据缓存采用相同的组相联度和容量。实际的缓存替换策略并未公开。我们的模拟采用 LRU（第 5.4 节将展示实际的未命中计数）。

比较结果如图9 和10所示。循环融合前，平均未命中率为 4.1%，预测误差为 0.5%。循环融合后（12 项测试），平均未命中率为 7%，误差为 0.4%，除一项外，误差均低于 1%。在此缓存配置的全部 41 项测试中，平均误差为 0.43%。

[5]Des相关描述总结自 MLIR 文档（https://mlir.llvm.org/docs/Passes/#-affine-loop-fusion ）

数据移动预测准确率。按照标准定义，准确率为 $x\%$ 意味着 $x\%$ 的内存访问被正确预测为命中或未命中。对于数据移动分析，需要关注总体数量，而不必考虑命中和未命中的顺序。因此，我们将访问轨迹转换为命中与未命中序列，然后重新排列，使所有命中都排在未命中之前。给定预测的未命中率后，我们按照该比率生成先命中、后未命中的预测序列。我们将数据移动预测准确率定义为重新排列后的序列与预测序列中匹配的元素（包括命中和未命中）所占的比例。当未命中率误差为 0.4% 时，预测结果与重新排列后的序列有 99.6% 相符。因此，数据移动预测的准确率为 99.6%。

在我们的测试中，未命中率可能相差数个数量级。当平均准确度定义在未命中率或命中率上时，可能因某一次测试而变得不公平，例如相对准确度较低，但对应的未命中率非常小。相比之下，数据移动准确度在所有测试中都以相同方式进行归一化，而不论其命中率或未命中率如何。

虚拟重用的实际效果。虚拟重用对准确度至关重要。在一项早期且不完整的评估中，我们对 23 个 PolyBench 内核在 32KB 全相联缓存上与模拟结果进行了对照测试；若不考虑虚拟重用，预测误差在最坏情况下高达 19.88%，平均为 2.15%。引入虚拟重用后，最大误差降至 1.53%，平均误差降至 0.18%——准确度提升超过十倍。

5.3 分析与预测成本

图 11 展示了 EPYC 7773X 推导缓存多项式并将其用于预测所需的时间。它将前者标记为 构建，将后者标记为预测。

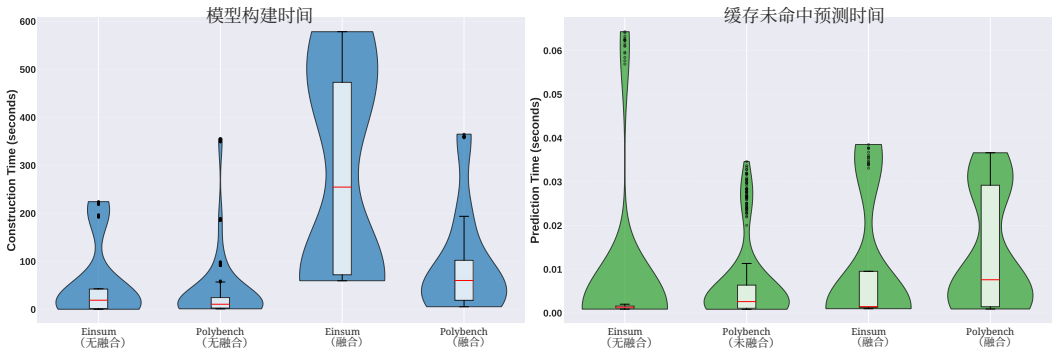


图11. 模型构建和预测阶段的时间分布小提琴图。两个阶段的尺度差异显著，因此分别绘制以便清晰呈现。

在推导方面，大多数未进行融合的程序运行时间不到50秒，其中位数约为65秒。maxcut_quantum_network_pattern（嵌套深度为12）和tensor_regression_network_pattern（包含11层嵌套循环）的推导时间较长，这是由于其时间戳空间维度较高。进行循环融合的程序通常构建时间会增加，因为生产者-消费者融合可能会使程序变得更加复杂。

推导出多项式后，预测便可瞬时完成——所有程序都能在 65毫秒内完成。预测时间与程序本身的复杂度无关，但与符号项的数量相关。我们观察到，循环融合对预测时间的影响极小。

5.4 与硬件计数器的比较

我们通过将模拟结果与 Nvidia GB10 的 Cortex-X925 上的硬件性能计数器进行比较，验证了预测方法。我们使用 ARM PMU 事件收集L1D_缓存_重填和L1D_缓存的数据。Cortex-X925 配备 64KiB 四路组相联 L1d 缓存。为减少小型工作负载中硬件计数器固有的测量噪声，我们将分析范围限定为重填次数超过 600,000 的测试用例。所有实验均使用第 5.1节中所述的填充配置。

图 12 表明模拟结果与硬件计数器测量结果高度吻合。虽然 `fdtd-2d`、相关性和 协方差的相对差异略高，但绝对差异仍然很小。在 `heat-3d` 中，由于移动悬崖误差（第 2.5 节），预测值低于模拟值，但模拟结果与硬件计数器结果相符。在其他所有情况下，与模拟结果和硬件测量值相比，我们的预测都很准确。

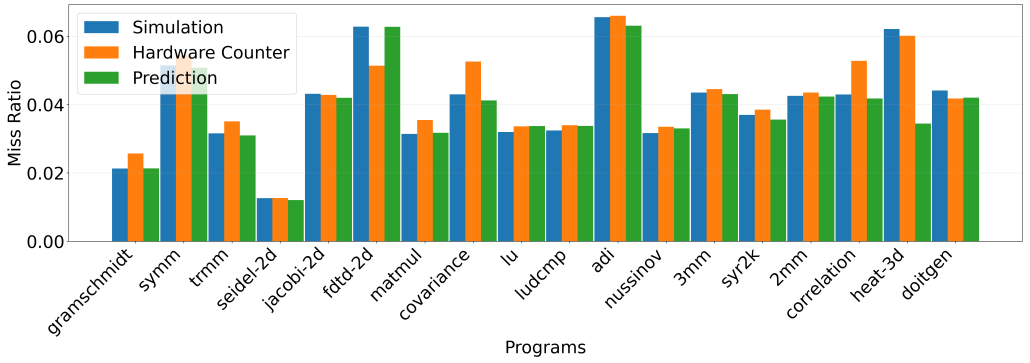


图12. 将硬件计数器结果与预测和模拟结果进行比较

5.5 缓存最小-最大缩放

在分析缓存需求时，一个广为人知的经验法则是 $\sqrt{2}$ 规则，该规则指出如果问题规模翻倍，就需要将缓存大小乘以 $\sqrt{2}$，才能维持相同的缓存命中率[22]。这一规则已通过数据库系统（缩放缓冲池大小）、文件系统缓存和 Web 缓存中的各种工作负载得到实证验证。

缓存多项式可用于推导缓存缩放，例如 $\sqrt{2}$ 规则。我们将缓存最小-最大缩放 定义为：将最大未命中率限制在给定阈值以下所需的最小缓存大小。对于一个程序，其最小-最大伸缩给出了可确保未命中率始终不超过指定上界的最小缓存大小。

表 3 展示了朴素矩阵乘法的缩放情况。两列数据分别显示左侧的最大未命中率和右侧的最小缓存大小。它们摘自表 1，其中缓存大小和未命中率多项式直接复制于此，以给出最小-最大伸缩。由于缓存大小必须为整数，因此从表1复制常数项时，我们取其上整值。

顶行显示起点：没有缓存，也没有命中。仅用 1 块，我们将最大缺失率限定为 0.75。当我们将缓存大小增加到 3 块、再增加到 4 块时，未命中率最多降至 $\frac{1}{2} + \frac{1}{32} \approx 0.53$，随后降至 10/32（0.31）。当所需缓存大小随 n线性增长时，

表 3。C语言 朴素矩阵乘法的缓存最小-最大伸缩 法

索引	最小缓存大小	最大未命中率
0	0	1
1	1	$\frac{3}{4}$
2	3	$\frac{1}{2} + \frac{1}{32n}$
3	4	$\frac{9}{32} + \frac{1}{32n}$
4	$\frac{9n}{8} - 9$	$\frac{1}{4} + \frac{1}{16n}$
5	$\frac{9n}{8} + 2$	$\frac{1}{32} + \frac{1}{16n}$
6	$\frac{n^2}{8} + \frac{3n}{8} - 2$	$\frac{3}{32n}$

未命中率的界限可低至 3/32（0.09）。这就是 √2 规则。最后，最小缓存大小恰好为 $\frac{n^2}{8} + \frac{3n}{8} - 2$，从而捕获所有数据重用，也就是说，所有缓存未命中都是冷启动缺失。

第 4 行和第 5 行都按 √2 规则缩放。为保持最大未命中率，缓存大小随 n 线性增长，而数据大小则随 n 呈二次增长。数据大小每翻一倍，最小缓存大小就增加 √2。

与 √2 规则相比，最小-最大伸缩能够精确给出缓存大小和未命中率。其精确性体现出重要差异。首先，未命中率可能不唯一。例如，第 4 行和第 5 行都按相同的 √2 规则缩放，但未命中率相差一倍。其次，未命中率可能不是常量。第 5 行的未命中率为 $\frac{1}{32} + \frac{1}{16n}$；随着 n 从 1 增至无穷大，未命中率从 $\frac{3}{32}$ 降至 $\frac{1}{31}$，减少为原来的三分之一。

6 相关工作

基于踪迹的局部性刻画。 计算局部性的迭代方法最初源自工作集理论，旨在处理无穷随机过程。Denning 和 Schwartz [14]提出了三个假设：(1) 序列是无限的；(2) 底层随机机制是平稳的；(3) 随着事件之间的间隔变大，这些事件会变得不相关。工作集理论计算工作集大小的极限。例如，非重复引用的页面最多只会被引用有限次，因此不会对工作集大小的极限作出贡献。正如我们所说的冷缺失困境（第 2.2 节），在程序分析中，这一假设并不成立，因为存在非零比例的无限重用间隔。

对于有限长度的执行踪迹，高阶局部性理论（HOTL）[60] 将足迹定义为所有相同长度窗口的平均工作集大小（对于所有窗口长度，均可在线性时间内计算）。首次触及访问的影响包含在由Xiang公式 [59],计算的足迹中，该公式使用首次访问时间和末次访问时间，但不使用无限重用区间。

实践中最快的技术基于重用间隔，因为重用间隔可以被高效采样 [5,53, 60]。Yuan 等人 [64, §3.3] 表明，这些模型在数学上等价于Denning递归。其中包括由Shen-Shaw 公式实现的时间到重用距离转换 [45] 及其在编程工具SLO [7] 中的应用，以及使用硬件计数器的 Featherlight RD [52], 、StatStack [17],，还有基于平均驱逐时间（AET）的存储缓存 [24, 38, 61], 与多级缓存 [62]分析。虽然Shen-Shaw 公式和StatStack基于统计推理，而AET基于缓存驱逐的动力学模型，但它们本质上都是Denning递归的变体。

问题规模扩展。 通过对若干程序输入的剖析结果，一种常见方法是外推并预测其他程序输入的影响。此类技术已同时针对程序分析 [18, 33, 66] 和体系结构或缓存扩展分析 [58] 而开发。剖析重用距离会产生很高的运行时开销，该开销可通过近似 [9, 37, 56, 66],采样 [43, 65]和并行化 [12, 35, 43]加以降低。这些技术直接对最近最少使用进行建模，并且不存在移动悬崖误差。由于是数据驱动的，它们并不局限于循环代码。

仿射循环分析。 Harper 等人 [21] 提出了一种分析方法，用于对组相联缓存和全相联缓存进行建模。他们针对的是循环边界为常量且具有仿射数组引用的完全嵌套循环。他们将每个数组引用对应的物理内存位置（加上数组之间的偏移量）表示为仿射表达式，并使用模运算确定内存访问如何映射到缓存组，通过一种重叠度量来刻画冲突，该度量统计竞争每个缓存组的缓存行数量。他们的方法能够准确预测，通常与模拟结果的误差在 15% 以内。相比之下，我们的方法在达到同等准确度的同时，还能处理符号化循环边界、非完全嵌套循环、分支以及可变的缓存块大小。

先前的技术 [4, 6, 19, 26, 40, 51] 表明，对于某些仿射程序，重用距离（LRU 栈距离）的计算问题可以转化为一个计数问题，即计算参数多面体并集或交集中的整数点。连续访问的范围可以建模为仿射约束集合，而重用距离值则可以通过计算该集合的基数得到。这项工作也将整数集合用于仿射循环，但其目的在于计算重用间隔，而非研究缓存行为。我们已经证明该问题是 NP 困难的，因此除非 P=NP，否则不存在多项式时间解。我们的解决方案采用与先前技术类似的方法来识别数据重用。对于一次数据重用，只需将重用窗口的长度作为其重用间隔即可轻松求得其 RI。

先前关于 PolyCache [4] 的工作报告了各 Polybench 内核的运行时性能。在其系统上，平均时间为 76.2 秒，中位数为 37.0 秒，最短为 1 秒（jacobi-1D、seidel），最长为 323 秒（symm）。相比之下，我们的编译器推导平均耗时 65 秒，预测耗时不到 0.1 秒。PolyCache 需要数值循环边界、缓存大小和缓存块大小。它并不像我们的方法那样具有符号化推导阶段。

最后，先前的技术并未以多项式形式表达局部性。表 3 中的最小-最大伸缩首次展示了通过二次函数和倒数函数自动推导缓存性能伸缩。与依赖诸如 $\sqrt{2}$ 容量扩展规则 [22], 的传统方法不同，最小-最大伸缩能够捕获精确的未命中率，而该未命中率本质上可以是非常量的。

7 总结

本文提出代数局部性，这是一种从符号化 RI 分布推导缓存大小与未命中多项式的理论。它使用无限重复，并将首次触及访问转换为虚拟重用，从而得到完整的 RI 分布。随后应用 Denning递归，可生成闭式表达式形式的缓存大小与缺失率多项式，将局部性刻画为程序参数与缓存参数的代数函数。此外，我们为仿射 MLIR 方言构建了一个编译器，它将循环嵌套转换为参数多面体，并利用整数集方法与 Barvinok 分解计算符号化 RI 分布。

针对 PolyBench 和 Einsum 套件中的 41 个科学计算内核和张量运算，在循环融合前，为所有测试推导缓存多项式平均需要 41 秒；为 12 个融合内核推导则需要 224 秒。推导完成后，预测任意程序输入和缓存配置下的缓存未命中情况

耗时不到 1 毫秒。该方法在典型的 L1D 组相联缓存上的平均准确率达到 99.6%。此外，论文还展示了首个以二次表达式和倒数表达式符号化表示缓存性能伸缩的编译器分析。仿射循环的最小-最大伸缩完全精确，不同于传统的扩展规则，例如 $\sqrt{2}$ 规则；该规则在数据库系统（缓冲池大小伸缩）、文件系统缓存和 Web 缓存中广为人知。

致谢

作者谨向 Jack Cashman 致谢，感谢他早期参与本项目；并感谢 Sree Pai 帮助完善论文的呈现。

参考文献

[1] Aravind Acharya、Uday Bondhugula 和 Albert Cohen。2018年。无需整数线性规划的多面体自动变换。载于《ACM SIGPLAN 编程语言设计与实现会议论文集》，Jeffrey S. Foster 和 Dan Grossman（编）。ACM，529–542页。[2] Alfred V. Aho、Monica S. Lam、Ravi Sethi 和 Jeffrey D. Ullman。2006年。《编译器：原理、技术与工具》（第2版）。Addison-Wesley。[3] Randy Allen 和 Ken Kennedy。2001年。《面向现代架构的优化编译器：基于依赖关系的方法》。Morgan Kaufmann Publishers。[4] Wenlei Bao、Sriram Krishnamoorthy、Louis-Noël Pouchet 和 P. Sadayappan。2018年。仿射程序缓存行为的解析建模。《ACM程序设计语言汇刊》2，POPL（2018年），32:1–32:26。[5] Kristof Beyls 和 Erik D‘Hollander。2004年。通过定位低局部性重用实现平台无关的缓存优化。载于《国际计算科学会议论文集》，M. Bubak、G.D. van Albada、P.M.A. Sloot 和 J.J. Dongarra（编），第3038卷。Springer-Verlag Heidelberg，Krakow，448–455页。[6] Kristof Beyls 和 Erik H. D’Hollander。2005年。生成缓存提示以提高程序效率。《系统架构期刊》51，4（2005年），223–250页。[7] Kristof Beyls 和 Erik H. D’Hollander。2006年。通过重用路径分析发现提升局部性的重构。载于《高性能计算与通信会议论文集》。Springer。《计算机科学讲义》，第4208卷。220–229页。[8] Mark Blacher、Christoph Staudt、Julien Klaus、Maurice Wenig、Niklas Merk、Alexander Breuer、Max Engel、Sören Laue 和 Joachim Giesen。2024年。Einsum基准：助力新一代张量执行引擎的开发。载于《神经信息处理系统进展》第三十八届数据集与基准赛道。https://openreview.net/forum.id=tllpLtt14h [9] Calin Cascaval 和 David A. Padua。2003年。使用栈距离估算缓存缺失和局部性。载于《超级计算国际会议论文集》。150–159页。[10] Dong Chen、Chen Ding、Fangzhou Liu、Benjamin Reber、Wesley Smith 和 Pengcheng Li。2021年。统一租约与LRU缓存：分析与评估。载于 ISMM ’21：2021年ACM SIGPLAN内存管理国际研讨会，线上活动，加拿大，2021年6月22日，Zhenlin Wang 和 Tobias Wrigstad（编）。ACM，15–27页。[11] D. Cociorva、G. Baumgartner、C. Lam、P. Sadayappan、J. Ramanujam、M. Nooijen、D. E. Bernholdt 和 R. Harrison。2002年。一类电子结构计算的时空权衡优化。载于《ACM SIGPLAN 编程语言设计与实现会议论文集》。德国柏林。[12] Huimin Cui、Qing Yi、Jingling Xue、Lei Wang、Yang Yang 和 Xiaobing Feng。2012年。一种面向GPU的高度并行重用距离分析算法。载于《国际并行与分布式处理研讨会论文集》。

[13] 2021 53, 6 2021 113 1 113 36 10 1145 3399709

[14] 1972 15, 3 1972

Peter J. Denning。。工作集分析。《ACM计算调查》（），：–：。doi:。/ Peter J. Denning 和 Stuart C. Schwartz。。工作集模型的性质。《ACM通讯》（），191–198页。[15] C. Ding 和 K. Kennedy。2004年。通过编译器增强全局缓存重用来提高有效带宽。《并行与分布式计算期刊》64，1（2004年），108–134页。[16] Aditya Dubey、Zeki Zeybek 和 Peter Schmelcher。2025年。使用密度矩阵重整化群算法，通过树张量网络模拟量子电路。《物理评论B》

112 2025 104312 10 10 1103 64 4 5（9月），。

[17] 2010

期。doi:。/ hd-q z David Eklov 和 Erik Hagersten。。StatStack：高效的LRU缓存建模。载于《IEEE系统与软件性能分析国际研讨会论文集》。55–65页。[18] Changpeng Fang、Steve Carr、Soner Önder 和 Zhenlin Wang。2005年。基于指令的内存距离分析及其应用。载于《并行体系结构与编译技术国际会议论文集》。27–37页。

[19] S. Ghosh、M. Martonosi 和 S. Malik。1999年。缓存缺失方程：用于分析和调优内存行为的编译器框架。《ACM 编程语言与系统汇刊》21，4（1999年）。[20] Johnnie Gray 和 Garnet Kin-Lic Chan。2024年。对任意几何结构的张量网络进行超优化近似缩并。《物理评论X》

14 2024 011009 10 1103 14 011009 （），。doi: 。/PhysRevX.。J.S.

[21] Harper、D.J. Kerbyson 和 G.R. Nudd。。组相联缓存行为的解析建模。《IEEE计算机汇刊》

48, 10 1999 1009 1024 10 1109 12 805152 （），–。doi: 。/。Allan Hartstein、Vijayalakshmi

[22] Srinivasan、Thomas R. Puzak 和 Philip G. Emma。2008年。关于缓存失效行为的本质：它是 √2吗？《指令级并行期刊》10（2008年）。[23] M. D. Hill。1987年。高速缓存与指令缓冲区性能的若干方面。博士论文。加利福尼亚大学伯克利分校。[24] Xiameng Hu、Xiaolin Wang、Lan Zhou、Yingwei Luo、Zhenlin Wang、Chen Ding 和 Chencheng Ye。2018年。存储缓存的快速失效率曲线建模。《ACM《存储汇刊》》

14, 2 2018 12 1 12 34 10 1145 3185751 （），：–：。

[25] doi: 。/ Cameron Ibrahim、Danylo Lykov、Zichang He、Yuri Alexeev 和 Ilya Safro。。构建用于张量网络量子电路模拟的最优缩并树。载于《2022年IEEE高性能极限计算大会》（HPEC）。IEEE。doi:

10 1109 5821 2022 9926353 . /HPEC . . William Kelly、Vadim Maslov、William Pugh、

[26] Evan Rosser 和 Tatiana Shpeisman。1999年。面向并行性的循环分块。载于《并行计算语言与编译器研讨会论文集》（《计算机科学讲义》，第1656卷）。施普林格，柏林，海德堡，295–308页。[27] Fredrik Kjolstad、Shoaib Kamil、Stephen Chou、David Lugato 和 Saman P. Amarasinghe。2017年。张量代数编译器。《ACM程序设计语言汇刊》1，OOPSLA（2017年），77:1–77:29。http://tensor-compiler.org/codegen.html [28] Arthur M. Krause、Paulo C. Santos、Arthur F. Lorenzon 和 Philippe O. A. Navaux。2024年。HBPB：应用重用距离主动提升缓存效率。《并行与分布式计算期刊》188（2024年），104919。doi:10.1016/j.jp dc.2024.104919[29] Chris Lattner、Mehdi Amini、Uday Bondhugula、Albert Cohen、Andy Davis、Jacques Pienaar、River Riddle、Tatiana Shpeis man、Nicolas Vasilache 和 Oleksandr Zinenko。2021年。MLIR：面向领域专用计算的可扩展编译器基础设施。载于《2021年IEEE/ACM国际代码生成与优化研讨会论文集》（线上活动，大韩民国）（CGO ’

21 2 14 10 1109 51591 2021 9370308 ）。IEEE出版社，–。

[30] doi: 。/CGO . . Rui Li、Aravind Sukumaran-Rajam、Richard Veras、Tze Meng Low、Fabrice Rastello、Atanas Rountev 和 P. Sadayappan。2019年。张量缩并的缓存解析建模与分块尺寸优化。载于《国际高性能计算、网络、存储与分析大会论文集》（SC），Michela Taufer、Pavan Balaji 和 Antonio J. Peña（编）。ACM，74:1–74:13。[31] LLVM项目。2025年。Passes。https://mlir.llvm.org/docs/Passes/。访问日期：2025年11月11日。[32] Hao Luo、Guoyang Chen、Fangzhou Liu、Pengcheng Li、Chen Ding 和 Xipeng Shen。2018年。缓存相联度和粒度的足迹建模。载于《国际存储器系统研讨会论文集》（MEMSYS）。232–242页。doi:10.1145/3240302.3240419[33] G. Marin 和 J. Mellor-Crummey。2004年。使用参数化模型预测科学应用的跨架构性能。载于《国际计算机系统测量与建模大会论文集》。2–13页。[34] William S. Moses、Lorenzo Chelini、Ruizhe Zhao 和 Oleksandr Zinenko。2021年。Polygeist：将C语言提升为多面体MLIR。载于《ACM并行架构与编译技术国际会议论文集》（线上活动）（PACT ’21）。计算机协会，纽约，美国，12页。[35] Qingpeng Niu、Ja mes Dinan、Qingda Lu 和 P. Sadayappan。2012年。PARDA：一种快速并行重用距离分析算法。载于《国际并行与分布式处理研讨会论文集》。[36] Cedric Nugteren、Gert-Jan van den Braak、Henk Corporaal 和 Henri E. Bal。2014年。一种基于重用距离理论的详细GPU缓存模型。载于《国际高性能计算机体系结构研讨会论文集》。[37] F. Olken。1981年。计算固定空间替换策略成功函数的高效方法。技术报告 LBL-12370。劳伦斯伯克利实验室。[38] Cheng Pan、Xiaolin Wang、Yingwei Luo 和 Zhenlin Wang。2021年。Redis中使用增强型AET实现惩罚与局部性感知的内存分配。《ACM《存储汇刊》》17，2（2021年），15:1–15:45。[39] Tharindu Rusira Patabandi 和 Mary W. Hall。2023年。通过分解变换域高效学习局部性优化。载于《国际编译器构造大会论文集》，Clark Verbrugge、Ondrej Lhoták 和 Xipeng Shen（编）。ACM，37–49页。[40] Arjun Pitchanathan、Kunwar Grover 和 Tobias Grosser。2024年。Falcon：一种可扩展解析缓存模型。《ACM程序设计语言汇刊》。

8, 222 2024 25 10 1145 3656452 PLDI，文章（6月），页数。doi: 。/

[41] [ ] 4 0 Louis-Noël Pouchet。n. d.。PolyBench/C。。http://polybench.sourceforge.net。[42] Jonathan Ragan-Kelley、Connelly Barnes、Andrew Adams、Sylvain Paris、Frédo Durand 和 Saman P. Amarasinghe。2013年。Halide：一种用于图像处理并优化并行性、局部性和重新计算的语言与编译器

流水线。在 ACM SIGPLAN 编程语言设计与实现会议论文集。519–530。doi:10.1145/2491956.2462176[43] Derek L. Schuff、Milind Kulkarni 和 Vijay S. Pai。2010年。通过采样和并行化加速多核重用距离分析。在并行体系结构与编译技术国际会议论文集。53–64。[44] Rathijit Sen 和 David A. Wood。2013年。面向缓存的基于重用的在线模型。在国际计算机系统测量与建模大会论文集。279–292。[45] Xipeng Shen、Jonathan Shaw、Brian Meeker 和 Chen Ding。2007年。使用时间进行局部性近似。在 ACM SIGPLAN-SIGACT 编程语言原理研讨会论文集。55–61。[46] A. J. Smith。1976年。论组相联页映射的有效性及其在主存管理中的应用。在软件工程国际会议论文集。[47] Valgrind [n. d.]。Valgrind 文档，3.2.0 版，2006年6月。http://valgrind.org/docs/manual/index.html。[48] Nicolas Vasilache、Oleksandr Zinenko、Theodoros Theodoridis、Priya Goyal、Zachary DeVito、William S. Moses、Sven Verdoolaege、Andrew Adams 和 Albert Cohen。2020年。下一个 700 个加速层：从网络计算图的数学表达式到自动加速 GPU 内核。ACM 体系结构与代码优化汇刊 16，4（2020），38:1–38:26。[49] Sven Verdoolaege。2010年。isl：面向多面体模型的整数集合库。在《数学软件》- ICMS 2010，Komei Fukuda、Joris Hoeven、Michael Joswig 和 Nobuki Takayama（编）。《计算机科学讲义》，第 6327 卷。施普林格，299–302。[50] Sven Verdoolaege 等。2024年。Barvinok：用于计算参数多面体和非参数多面体中整数点数量的库。https://barvinok.sourceforge.io/。版本 0.41.8，访问日期：2026年3月12日。[51] Sven Verdoolaege、Rachid Seghir、Kristof Beyls、Vincent Loechner 和 Maurice Bruynooghe。2007年。使用 Barvinok 有理函数计算参数多面体中的整数点。Algorithmica 48，1 1 007 2007 37 66 10 00453 006 1231 0（3月），–。doi：。/ s - - -[52] Qingsen Wang、Xu Liu 和 Milind Chabbi。2019年。轻量级重用距离测量。在高性能计算机体系结构国际研讨会论文集。IEEE,

440 453 10 1109 2019 00056 –。

[53] doi：。/HPCA。2015。Wang 等。共享缓存中的最优程序共生。在 IEEE 集群计算与网格国际研讨会论文集。[54] J. Weinberg、M.O. McCracken、E. Strohmaier 和 A. Snavely。2005年。量化高性能计算应用内存访问模式中的局部性。在 SC '05：2005年 ACM/IEEE 超级计算会议论文集。50–50。doi:10.1109/SC.2005.59[55] Greg Weirs、Vikram Dwarkadas、Tomek Plewa、Chris Tomkins 和 Mark Marr-Lyon。2005年。验证 Flash 代码：涡旋主导流。《天体物理学与空间科学》298，1-2（2005），341–346。[56] Jake Wires、Stephen Ingram、Zachary Drudi、Nicholas JA Harvey、Andrew Warfield 和 Coho Data。2014年。使用计数器栈表征存储工作负载。在操作系统设计与实现研讨会论文集。USENIX 协会，335–349。[57] M. J. Wolfe。1996年。并行计算高性能编译器。Addison-Wesley，雷德伍德城，加利福尼亚州。[58] Meng-Ju Wu、Minshu Zhao 和 Donald Yeung。2013年。通过重用距离分析研究多核处理器扩展。在计算机体系结构国际研讨会论文集。499–510。[59] Xiaoya Xiang、Bin Bao、Chen Ding 和 Yaoqing Gao。2011年。共享缓存中程序工作集的线性时间建模。在并行体系结构与编译技术国际会议论文集。350–360。[60] Xiaoya Xiang、Chen Ding、Hao Luo 和 Bin Bao。2013年。HOTL：高阶局部性理论。在编程语言与操作系统体系结构支持国际会议论文集。343–356。[61] Jun Xiao、Yaocheng Xiang、Xiaolin Wang、Yingwei Luo、Andy D. Pimentel 和 Zhenlin Wang。2023年。FLORIA：一种快速且轻量级的缓存性能预测方法。在超级计算国际会议论文集，Kyle A. Gallivan、Efstratios Gallopoulos、Dimitrios S. Nikolopoulos 和 Ramón Beivide（编）。ACM，25–36。[62] Chencheng Ye、Chen Ding、Hao Luo、Jacob Brock、Dong Chen 和 Hai Jin。2017年。缓存互斥与共享：理论与优化。ACM 体系结构与代码优化汇刊

14, 4, 34 1 34 26 10 1145 3134437 ： –: 。

[63] doi：。/ Qing Yi。2012。。POET：用于应用参数化源到源程序变换的脚本语言。《软件：实践与经验》42，6（2012），675–706。[64] Liang Yuan、Chen Ding、Wesley Smith、Peter J. Denning 和 Yunquan Zhang。2019年。局部性的关系理论。ACM 体系结构与代码优化汇刊 16，3（2019），33:1–33:26。[65] Yutao Zhong 和 Wentao Chang。2008年。基于采样的程序局部性近似。在内存管理国际研讨会论文集（美国亚利桑那州图森）。

91 100 10 1145 1375634 1375648 –。doi：。/。

[66] Yutao Zhong、Xipeng Shen 和 Chen Ding。2009。。使用重用距离进行程序局部性分析。ACM 编程语言与系统汇刊 31，6（2009），20:1–20:39。