[文献阅读04]-Polyhedral AST Generation Is More Than Scanning Polyhedra

论文基本情况：

题目: Polyhedral AST Generation Is More Than Scanning Polyhedra

多面体的抽象语法树生成比代码生成更重要

多面体抽象语法树的生成不仅仅是扫描多面体

作者：Tobias Grosser ,Sven Verdoolaege, Albert Cohen

所在学校： 巴黎高等师范学校 École Normale Supérieure Paris

image-20201026232231209

所在实验室：INRIA 法国国家信息与自动化研究所（或称法国国立计算机及自动化研究院）

INRIA 计算机学科在世界科研机构学科竞争力排行榜中排名全球第七（前十名分别为：麻省理工学院，斯坦福大学，加州大学伯克利分校，德克萨斯大学，伊利诺大学，宾夕法尼亚大学，法国国家信息与自动化研究所，卡耐基.梅隆大学，马里兰大学和加州大学圣迭戈分校）。在 NeurIPS 2019 、ICML 2019 论文入选数全球机构排名中，INRIA 分别排名世界第12、16名。 在Guide2Research 网站发布的 “全球计算机科学和电子领域顶级科学家排名” 中，INRIA有53位计算机/电子工程领域的高h因子学者上榜，和马里兰大学、UCSD、EPFL等大学并列世界第七。在Aminer发布的 “计算机学科学术排行榜" 中，INRIA排名世界第19名，第17、18名分别为UIUC和宾夕法尼亚大学。在苏黎世联邦理工（ETH Zürich）分布式计算实验室发布的 “2018 Ranking of Computer Science and Information Technology Universities and Research Labs” （数据来源： Guide2Research）中，INRIA排名世界第六。

发表期刊：ACM Transactions on Programming Languages and Systems, Vol. 37, No. 4, Article 12, Publication date: July 2015.

程序设计语言和系统的ACM事务处理 卷37，第4条，第12条，发布日期：2015年7月。

摘要：

> 诸如整数多面体之类的抽象数学表示已被证明对精确分析计算内核和表示复杂的循环变换很有用。这种转换依靠抽象语法树（AST）生成器将数学表示形式转换回命令式程序。这种通用AST生成器避免了求助于特定于转换的代码生成器的需要，随着转换变得越来越复杂，开发这种代码生成器可能会非常昂贵，或者在技术上非常困难。现有的AST发生器已经证明了其有效性，但是在更复杂的情况下却受到限制。具体来说， （1）它们不支持或可能无法使用涉及模算术的分段调度或映射来生成复杂转换的控制流； （2）它们对生成专业化的代码提供有限的支持，这些代码公开了紧凑，直线，可矢量化的内核，这些内核具有开发现代硬件的最佳性能所必需的高算术强度； （3）它们不支持内存局部变换（no support for memory layout transformations）； （4）它们无法充分控制AST生成策略，从而无法将其应用于复杂的特定领域优化。
>
> ***我们提出了一种新的***AST生成方法，该方法将经典的多面体扫描扩展到皮尔斯伯格算术(Presburger arithmetic)的全部通用性，包括其中存在的量化变量和分段调度，并介绍了用于检测组件（检测分量）和移动步幅的新优化方法。不局限于控制流的生成，我们公开了从任意分段的拟仿射表达式生成抽象语法树（AST）表达式的功能，从而可以将抽象语法树（AST）生成器用于数据局部变换。我们通过多面体展开，用户控制的版本控制以及根据AST表达式的位置对AST生成进行特殊化的支持来对此进行补充，并通过对所用AST生成策略进行细粒度用户控制来完成这项工作。利用这种AST生成的一般化思想，我们介绍了如何实现复杂的特定域的转换，而无需编写专用的代码生成器，而是依靠参数化到特定问题域的通用AST生成器。

> 类别和主题描述符: D.3.4(编程语言)：处理器；—编译器，优化
>
> 一般术语:算法，性能
>
> 额外的关键字和短语:多面体编译，代码生成，展开，索引集分裂，皮尔斯伯格（Presburger）关系

1. 介绍

针对特定领域或通用编译器的高级优化的发展（开发）在概念上通常可以分为两个部分:高级优化策略的设计和根据该优化策略生成程序代码。在许多情况下，有趣的科学贡献是新的优化策略。然而，在实践中，我们在有效程序代码的生成上付出了巨大的努力。

对于具有仿射（甚至非仿射[Venkat et al。2014]）控制流的循环程序，通常会根据对每个语句实例(即在循环嵌套中动态执行一条语句)进行建模的抽象描述自动生成优化的程序代码。该抽象描述通过对和每个数组元素使用紧凑表示法（例如多面体[Loechner and Wilde 1997]或皮尔斯伯格关系（Presburger relations）[Pugh and Wonnacott 1994]）来分别实现。

> 对于具有仿射控制流的循环程序，通常通过使用紧凑表示(如多面体或预伯格关系)从抽象描述自动生成优化的程序代码，该抽象描述对每个语句实例和每个数组元素进行单独建模。

Pugh和Rosser[1997]、Wonnacott[2002]、Bondhugula等人[2008]、Kong等人[2013]、Zuo等人[2013]和Bandishti等人[2012]的优化是通过修改一个抽象调度来描述的，该调度定义了程序中单个语句实例的执行顺序。根据该调度，使用称为多面体扫描的技术（Ancourt and Irigoin 1991）（重新）生成命令式程序代码，即代码生成[Kelly等。 1995; Bastoul 2004]，或更准确地说，是抽象语法树（AST）生成。解耦优化和程序生成步骤不仅减少了实施一定优化策略所需的时间，而且还加快了对新优化策略的评估[Pouchet等，2007、2008]。此外，如果能够在高度抽象的层次上描述转换就可以开发出复杂的转换[Grosser等 .2014]并依靠AST生成器生成高效的命令式代码。

即使AST发生器有很多好处，但是现有的方法集中在控制流的生成上，仅对生成专用的表达式提供基本支持，并且对代码大小和控制开销的控制有限。这些限制通常会阻止它们的广泛使用。缺少对生成用户提供的AST表达式的支持（例如，描述内存位置）会阻止其应用于数据局部转换或将数据映射到软件托管的缓存上。此外，现有的AST生成器可以根据特定的参数值来生成多个代码版本，以减少控制开销，但是现有的方法本身并不支持专用代码的生成。这对于全部和局部分块的分离特别有用或者用于生成专用的代码，以在迭代空间边界处处理可能不同的条件。但是，必须通过在输入描述中生成每个语句的多个不同副本来实施版本控制。类似地，只有通过复制输入描述中的语句，才能在AST生成期间执行展开。除了概念上不令人满意之外，重复的语句还会引起严重的问题。首先，通过有目的地隐藏语句相同的事实，AST生成器在所有情况下都被迫为其生成重复的代码，从而丢失了复杂表达式中的冗余，并缺少了**分解较冷的部分**的机会。其次，重复的语句增加了命令式控制流生成中涉及的多面体运算的复杂性，支持诸如全部/局部分块分离之类的优化，支持模运算的表达式专门化或简化。如果现在我们希望最小化迭代空间中**较冷部分**（例如，局部分块）的代码大小，则会遇到下一个限制。即使AST生成器对分离语句或控制流专门化（条件提升）中所需的**主动性**提供了基本控制，但在现有方法和工具中，控制级别的粒度还是太粗糙了。此外，不能保证最大的循环嵌套数和生成的最大语句数，这在代码大小是主要关注点的情况下是有问题的，例如使用软件管理的缓存，嵌入式处理器和高级综合功能为许多核心目标生成AST。总体而言，对于复杂的AST生成问题，现有的方法和工具在许多方面尚未成熟。

这项工作提出了一种集成的AST生成方法，除了经典的控制流生成之外，它还允许从任意用户提供的分段仿射表达式生成AST表达式。我们定义了一种细粒度的（详细的）“选项”机制，该机制使用户可以在需要时最大程度地要求专业化，同时保持对代码大小的控制。为了实现积极的专业化，我们允许用户指示AST生成器如何对代码进行版本控制，我们提供了集成的多面体展开工具，并确保AST表达式根据生成位置的上下文进行了专门化生成。这样做是必要的，以正确地对由程序的抽象变换产生的**底层除法**和模运算进行建模，并将这些表达式转换成有效的余数和整数除法，或者转换成现有指令集体系结构和编程语言所提供的低复杂度运算。最后，与现有的多面体扫描工具相比，我们提出了一系列可以提高生成代码的质量优化措施，而且还进行了必要的优化，以覆盖我们的AST生成器的更广泛的应用场景。

我们的贡献如下：

- AST生成方法完全支持皮尔斯伯格（Presburger）关系，包括对分段调度（时间表）的支持，以及支持将索引集拆分表示为只含有调度的转换。
- 根据分段拟仿射表达式在AST中的位置，对生成的AST表达式进行积极的简化，包括检测模运算和除法运算。简化的AST表达式的生成不仅用于从AST生成器中构造循环边界和if条件，还向用户公开，用户可以使用该功能来生成自定义索引表达式和运行时检查。
- 通过细粒度选项控制AST生成，其中包括可用于控制代码大小和确保没有程序语句重复的原子选项。
- 通过多面体展开和用户控制的版本进行**专业化（特殊化）**—特别地，用户可以指定调度空间（例如，完整分块（full tiles））的子集，该子集应与调度空间的其余部分（即部分分块（partial tiles）隔离。
- 改进步幅检测和可重排组件检测的算法。
- AST生成的结构化调度以调度树表示。
- 在高级的特定领域的优化器中进行评估，并与最新的代码生成技术进行比较。

本文的其余内容安排如下。第2部分从总体上概述了我们的新AST生成方法，并提出了新的说明性用例。然后，我们在第3节中介绍理论背景，在第4节中介绍数据结构，在第5节中介绍核心AST生成方法，并在第6节中介绍其对调度树的扩展。在第7节中，我们进行了一系列实验，在第8节中讨论了相关工作，以及第9节展示结论。

2. AST生成的新方法

为了给在这项工作中提出的新的AST生成概念一个定义，我们把它们放在一个复杂的AST生成场景的上下文中。一个这样的理想场景是我们最近在六边形/平行四边形分块上的工作，这是一个针对特定领域的优化，用于为模板迭代计算生成高效的CUDA代码。它是在PPCG之上实现的一个通用的从C到CUDA/OpenCL的转换器，它使用皮尔斯伯格（Presburger）关系来描述计算本身和要应用的程序转换。利用这种抽象的数学描述开发了一种新的涉及复杂的几何形状的分块方案（分块调度）以解决为GPU编译迭代模板时最重要的性能问题，包括共享内存的使用、数据传输的优化、算术强度的增加、多级并行的利用以及避免线程发散。在下面的段落中，我们将说明如何使用新的通用AST生成方法获得高效的代码，而无需开发特定领域或优化的生成器。

CUDA（Compute Unified Device Architecture。统一计算设备架构），是显卡厂商NVIDIA推出的运算平台。 CUDA™是一种由NVIDIA推出的通用并行计算架构，该架构使GPU能够解决复杂的计算问题。 它包含了CUDA指令集架构（ISA）以及GPU内部的并行计算引擎。 开发人员可以使用C语言来为CUDA™架构编写程序，C语言是应用最广泛的一种高级编程语言。所编写出的程序可以在支持CUDA™的处理器上以超高性能运行。CUDA3.0已经开始支持C++和FORTRAN。

OpenCL（全称Open Computing Language，开放运算语言）是第一个面向异构系统通用目的并行编程的开放式、免费标准，也是一个统一的编程环境，便于软件开发人员为高性能计算服务器、桌面计算系统、手持设备编写高效轻便的代码，而且广泛适用于多核心处理器(CPU)、图形处理器(GPU)、Cell类型架构以及数字信号处理器(DSP)等其他并行处理器，在游戏、娱乐、科研、医疗等各种领域都有广阔的发展前景。

当把C代码翻译成CUDA时，我们从由计算语句和循环组成的代码开始。为了简化说明，我们首先在本节中假设该程序由一组完美嵌套的循环，一个外部顺序循环，一组内部并行循环和单个计算语句组成。为了生成用于此计算的CUDA代码，必须获得一组可以顺序启动的内核，并且每个核心都包含两个级别的并行度：粗粒度并行—它将映射到所谓的CUDA线程块，和细粒度并行—它将被映射到所谓的CUDA线程。为了获得这两个级别的并行性，我们将由这些循环枚举的一组独立计算(语句实例)划分为子集(分块)。我们通过计算一个多面体调度来实现这一点，该调度用两组循环来枚举语句实例集:一组枚举分块的外部循环（分块循环）和一组枚举属于某个分块的语句实例的内部循环（点循环）。我们遇到的第一个AST生成问题是，定义分块形状的混合六边形调度将计算分解为多个阶段，并对每个阶段应用了不同的调度。这导致了从中生成AST的分段调度。

下一步，我们将分块和点循环映射到固定数量的线程块和线程。我们从寻找一组平行点循环和一组平行分块循环开始。然后，我们根据线程块和线程的数量来消除每个循环。例如，要将具有n个迭代的点循环映射到一个具有1,024个内核线程的集合，我们将循环除以1,024，以使每1,204个迭代都由同一线程执行。下一步是生成一段CPU代码调度内核加速的实例，该内核代码本身定义特定线程块中特定线程的计算。不会生成枚举线程块和线程集的实际循环，而是CUDA运行时和硬件生成了一组块和线程，并将块和线程ID作为参数提供给每个执行内核代码的线程。为了对此建模，我们首先生成外部循环，然后使用嵌套上下文引入块和线程标识符，最后，考虑到外部C代码的AST生成上下文以及内核和线程标识符的约束，生成可以引用外部CPU代码中的值的内核代码。此信息的利用对于生成高质量代码非常重要。

在生成内核代码时，我们还需要在我们的计算语句中重写所有数组下标。传统上，这是通过从文本上用表达式替换所有对旧归纳变量的引用来完成的，该表达式根据新归纳变量计算旧归纳变量的值。当转换访问A [i + 1]时（其中i现在表示为c0 + 1），经典重写将产生A [（c0 + 1）+1]。使用我们的新方法，我们将表达式i +1本身表示为分段拟仿射表达式，对分段拟仿射表达式进行转换，简化所得表达式，并使用AST生成器从该分段拟仿射表达式生成AST表达式。因此，我们获得了代码A [c0 + 2]。在本示例中，唯一的好处是提高了可读性，因为任何编译器都会将这两个加法符进行常数倍的折叠。但是，总的来说，这个概念要强大得多。它允许根据生成表达式的上下文来对表达式进行特殊化。例如，如果访问A [i == 0？ N-1： i-1]被安排在一个分块中，我们知道i永远不会为0，我们可以简化成A [i-1]。这种简化消除了核心计算中边界条件处理的开销，这种转换对于普通的编译器会丢失上下文信息，并且传统上这种转换需要专门的语句来处理边界和核心计算。使用我们的AST生成方法，一旦将边界计算和核心计算生成为特定AST子树，便会自动对语句进行专门化（简化）。这对于允许用户控制版本的AST生成器来说是很自然的。

生成包括重写的数据访问的基本CUDA代码后，我们可以开始优化代码。一个重要的优化是从使用慢速的“全局内存”切换为使用快速的手动管理的“共享内存”。为此，我们需要更改每个分块的代码，以便在进行实际计算之前，将全局内存中的相关数据复制到共享内存中，最后，将修改后的数据从共享内存中复制回全局内存中。为了在共享内存中执行计算，我们需要调整所有内存访问，以便它们指向新的共享内存数组和相应的位置。如何精确地计算映射不在本工作的范围之内，但是我们如何生成相关的代码是有趣的。我们从映射中导出一组定义新数据位置的分段拟仿射表达式，并为它们生成AST表达式，并依靠AST生成器来确保生成高效的代码。这种方法使我们能够使用可能复杂的映射，而无需编写专门的代码生成例程。为了创建移动数据的代码，我们创建了新的语句，将数据从给定的全局内存位置复制到给定的共享内存位置，反之亦然。如果要复制的数据比线程数多，我们使用模映射将数据位置分配给线程。图1显示了为将数据复制回全局内存而生成的代码。各种有趣的观察都是可能的。首先，我们看到我们的模表达式已映射到C余数运算符%，如果除数是2的幂的常数，并且编译器可以得出除数始终为负数，则它将转换为快速按位运算。仅因为我们具有有关t1值的上下文信息，才可以使用C余数运算符。否则，我们将不得不使用昂贵的floord或intMod表达式处理任意（可能为负）整数，就像处理最新的AST生成器CLooG和CodeGen +一样。其次，我们看到我们生成了一个相当密集的枚举了语句的循环嵌套。由于输入描述中量化变量的存在突出，这本身就很重要（请参见第7.3节）。

```
for (c2 = 0; c2 <= 1; c2 += 1)
  for (c3 = 1; c3 <= 4; c3 += 1)
    for (c4 = max(((t1-c3+130) % 128) + c3 - 2, ((t1+c3+125) % 128) - c3 + 3);
         c4 <= min(((c2+c3) % 2) + c3 + 128, -((c2+c3) % 2) - c3 + 134); c4 += 128)
      if (c3 + c4 >= 7 || (c4 == t1 && c3 + 2 >= t1 && t1 + c3 <= 6
                           && t1 + c3 >= ((t1 + c2 + 2 * c3 + 1) % 2) + 3
                           && t1 + 2 >= ((t1 + c2 + 2 * c3 + 1) % 2) + c3)
          || (c4 == t1 && c3 == 1 && t1 <= 5 && t1 >= 4 && c2 <= 1 && c2 >= 0))
        A[c2][6 * b0 + c3][128 * g7 + c4 - 4] = ...;
```

Fig. 1. Copy code from hybrid hexagonal/parallelogram tiling (a single loop).

尽管如此，我们观察到生成的代码不是非常有效。每次循环迭代执行的计算量很少，并且会判断复杂的条件。可能希望进一步简化条件，但是不幸的是，当移动的五点模板与六边形分块在形状上形成交叉时，修改的数据本身已经是非凸的了。应用另一个级别的模调度使得必要的计算模式会更加复杂，从而难以获得更简单的循环结构。然而，通过在内部三个循环上使用多面体展开，并根据它们被展开的迭代来专门化语句，我们可以移除几乎所有的控制开销，结果如图2所示。代码非常流畅，每个数组下标都专门针对具体位置。我们还可以看到，对于有条件执行的语句，下标根据条件进行了优化，使得剩余操

作完全消失。展开这段代码并不简单，因为它需要在存在多个循环边界和跨步的情况下执行，而且在展开时我们需要支持受保护的指令的生成。最内层的保护指令在GPU上是非常便宜的，因为它们可以作为预测指令来实现。在这个小例子中，这不是很明显。但是，对于实际的分块大小，会有更多的语句共享相同的条件。我们对内核中的计算代码执行类似的展开操作，以确保有足够的指令级并行性可用。

```
A[0][6 * b0 + 1][128 * g7 + (t1 + 125) % 128) - 1] = ...;
A[0][6 * b0 + 2][128 * g7 + (t1 + 127) % 128) - 3] = ...;
if (t1 <= 2 && t1 >= 1)
    A[0][6 * b0 + 2][128 * g7 + t1 + 128] = ...;
A[0][6 * b0 + 3][128 * g7 + (t1 + 127) % 128) - 3] = ...;
if (t1 <= 2 && t1 >= 1)
    A[0][6 * b0 + 3][128 * g7 + t1 + 128] = ...;
A[0][6 * b0 + 4][128 * g7 + (t1 + 125) % 128) - 1] = ...;
A[1][6 * b0 + 1][128 * g7 + (t1 + 126) % 128) - 2] = ...;
A[1][6 * b0 + 2][128 * g7 + (t1 + 126) % 128) - 2] = ...;
if (t1 <= 3 && t1 >= 2)
    A[1][6 * b0 + 2][128 * g7 + t1 + 128] = ...;
A[1][6 * b0 + 3][128 * g7 + (t1 + 126) % 128) - 2] = ...;
if (t1 <= 3 && t1 >= 2)
    A[1][6 * b0 + 3][128 * g7 + t1 + 128] = ...;
A[1][6 * b0 + 4][128 * g7 + (t1 + 126) % 128) - 2] = ...;
```

Fig. 2. Copy code from hybrid hexagonal/parallelogram tiling (unrolled).

代

现在，图2中的代码已接近最佳状态。但是，到目前为止，我们仅看到了一个简化的示例—一个不涉及任何迭代空间边界的分块。如果考虑迭代空间边界，则生成的代码将变得更加复杂。为了确保我们的大多数时间仍然可以使用“接近最佳”的代码，我们使用用户控制的版本控制将核心计算（完整分块）与需要考虑边界条件的分块集合隔离开来（部分分块）。这样做可以使我们获得最大程度的专业化和最佳性能。但是，我们现在不仅专门化和展开核心计算，而且还专门化和展开用来处理边界情况的代码，这增加了所生成代码的大小以及生成它所需的时间。以GPU为目标时，这可能是可以接受的，但对于FPGA 的费用可能会过高。通过使用细粒度的选项来限制边界分块的展开和专门化的数量，可以轻松解决此问题。

总之，将AST生成扩展到控制流的创建之外，使得可以在复杂的场景中使用自动AST生成。即使将现有的AST生成器与变通方法（工作区）结合起来，例如在运行AST生成器之前复制语句都可以用来解决一些以前提到的AST生成问题，但这些变通方法仅适用于某些功能，仅适用于简单的特殊情况，并且通常会抑制其他必要的转换。相反，通过小心地将几个重要的新扩展集成到单个AST生成方法中，我们显著地扩展了自动AST生成的概念，使得它可用于复杂的AST生成场景。我们确保不同的功能不会相互阻碍，但当结合在一起时，会为复杂的AST生成问题提供新的机会和解决方案。因此，我们希望不仅能显著简化AST生成，还能在新的优化场景中使用它。

3.多面体模型

多面体模型是一个强大的抽象，用于分析和转换“足够规则”的程序(部分)。这个模型的关键特征是它是基于实例的。换句话说，每个语句实例(即循环嵌套中语句的每次动态执行)和每个数组元素都通过使用紧凑表示(如多面体或皮尔斯伯格（Presburger）关系)进行单独处理。程序通常用迭代域来表示，包含语句实例；访问关系，将语句实例映射到被访问的数组元素；相关性，相互依赖的相关语句实例；和调度，为语句实例分配执行顺序。

就AST生成而言，最相关的元素是迭代域和调度，其中迭代域描述了需要执行的语句实例，调度描述了它们应该执行的顺序。对于迭代域，我们使用verdolaege[2011]提出的表示法，其中每个语句实例由一个名称(标识语句)和一个整数元组(标识实例)表示。对于每一条语句，迭代域中的实例都用一个皮尔斯伯格（Presburger）公式来描述。我们称这样的集合为普雷斯伯格集合。迭代域的其他表示可以很容易地转换成这样一个命名的预伯格集。

在定义Presburger公式之前，让我们首先考虑仿射表达式，它是由变量、整数常数、符号常数、加法(+)和减法(-)组成的项。整数常数乘法可作为重复加法或减法的语法糖。符号常量有一个固定但未知的值，通常代表问题的大小。准仿射表达式另外允许整数除以整数常数(？/d？).然后，由准仿射表达式、比较(≤)和一阶逻辑运算符:合取(∧)、析取(∩)、否定( )、存在量化(∃)和泛量化(∀).

分段准仿射表达式是一对成对的Presburger集和准仿射表达式的列表。这些集合成对不相交，并且在给定点处的分段仿射表达式的值等于与包含该点的集合关联的拟仿射表达式的值。

来构造预伯格公式分段拟仿射表达式是一组命名的预伯格集和拟仿射表达式。集合是成对不相交的，并且给定点处的分段拟仿射表达式的值等于与包含该点的集合相关联的拟仿射表达式的值。

```
for (int i = 0; i < n; ++i) {
S1: s[i] = 0;
    for (int j = 0; j < i; ++j)
S2:         s[i] = s[i] + a[j][i] * b[j];
S3: b[i] = b[i] - s[i];
}
```

Fig. 3. Example program.

成对命名整数元组上的二元关系可以用类似的方式定义，称为命名前伯格关系。尽管我们将在第6节中使用一个更结构化的时间表表示法，但考虑一下由verdolaege[2011]提出的将时间表表示为命名的Presburger关系的基本情况是有益的。这些命名的Presburger关系将一个整数元组与每个语句实例相关联，由调度表示的执行顺序由这些整数元组的字典顺序给出。考虑图3中的程序。迭代域是

$$\{\mathrm{S1}(i) : 0 \le i < n; \mathrm{S2}(i, j) : 0 \le j < i < n; \mathrm{S3}(i) : 0 \le i < n\}. \tag{1}$$

表达原始执行顺序的一种方式是调度

$$\{\mathrm{S1}(i) \to (i, 0, 0); \mathrm{S2}(i, j) \to (i, 1, j); \mathrm{S3}(i) \to (i, 2, 0)\}. \tag{2}$$

换句话说，语句实例首先根据其实例标识符元组中的第一个(或唯一的)元素进行排序。然后，对于那些具有相同值的语句实例，S1实例在S2实例之前执行，而S2实例又在S3实例之前执行。最后，根据S2实例标识符元组中的第二个元素执行S2的实例。对于其他两个语句，所有实例都已经分开调度，因此最终调度坐标不需要区分实例。因此，可以将其设置为任意值(此处为0)。

可通过进度表获得替代的执行命令

$$\{\mathrm{S1}(i) \to (0, i, 0, 0); \mathrm{S2}(i, j) \to (1, i, 1, j); \mathrm{S3}(i) \to (1, i + 1, 0, 0)\}, \tag{3}$$

在那里S1的所有实例都在S2或S3的任何实例之前被执行。在S2或S3的例子中，S2的第一个维度比S3的单一维度多一个维度的例子被一起执行，S3的例子在S2的例子之前被执行。

AST生成器的目的是构建一个AST，该AST按照由调度分配给迭代域元素的整数元组的字典顺序访问迭代域的元素。该构造使用ISL[Verdolaege 2010]中可用的命名Presburger集和关系上的若干操作，包括关系的域和范围、交集、并集、集差、投影、共享约束(“简单外壳”)、相对于已知约束的集(关系)简化(“要点”)、整数仿射外壳和合并(用单个析取代替析取对，而不引入虚假元素)[Verdolaege 2015]。所有这些

操作通常会改变它们所操作的集合(关系)中包含的元素，但合并除外，它只影响集合(关系)的表示。请注意，本文介绍的AST生成器不使用凸包操作，因为这可能会引入具有大系数的约束。

4.数据结构

在AST生成算法的基本情况下，计划由命名的Presburger关系映射语句实例给出其相对的多维执行时间。从这些多维执行时间中得出生成的AST中的循环。因此，在AST生成期间，更自然地考虑此计划关系的逆，我们将其称为已执行关系，并将执行时间向量映射到应在那些时间执行的语句实例。例如，给定（3）中的计划关系，（初始）执行关系（无域约束）为

$$\{(0, s_1, 0, 0) \to \mathrm{S1}(s_1); (1, s_1, 1, s_2) \to \mathrm{S2}(s_1, s_2); (1, s_1, 0, 0) \to \mathrm{S3}(s_1 - 1)\}. \qquad (4)$$

计划中的深度优先通过级别对应于此已执行关系的输入维度。在每个级别上，已执行关系的域都沿着该维度分解为多个部分，然后依次考虑每个已执行关系。

4.2 股票（库存）

在深度优先遍历过程中离开该级别时，构造与所执行关系的域中的维度相对应的AST节点。但是，有关AST节点的主要信息在第一次输入该级别时已经可用。其中一些信息需要通过遍历进行存储和转发。我们引入库存来收集可用于简化后代AST节点的信息。库存主要跟踪两条信息：已知常量的符号常量条件和外部循环迭代器的当前位置，以及从循环迭代器到调度维度的映射。在深度优先遍历的每个级别上，都会创建一个新的库存，并从较高级别传递过来的库存进行初始化。

条件分为两组：生成的条件和挂起的条件。生成的条件是算法已经确定的条件，这些条件将由AST中的外部节点强制执行。这些通常是外部循环节点上的循环边界。待决条件是可能最终由外部节点强制执行的条件。如果它们被内部AST节点暗示，它们也可能被丢弃。请注意，这种区别仅在构建实际AST节点时才有意义。在AST生成算法的其他方面，我们可以简单地考虑两组约束的组合。

需要从循环迭代器到调度维度的映射，因为与其他AST生成器不同，我们的利用了调度仅指定相对执行顺序这一事实。由于例如AST生成器的缩放或剥离，最终AST中的循环迭代器可能不完全对应于输入时间表中的时间表维度，尽管这些时间表维度仍然可以从时间表的其他部分引用通过选项（请参见第5.6节）或高级计划树节点（请参见第6节）。

4.3 抽象语法树

生成的AST仅包含语法信息，并且已设计为易于翻译为C语言和编译器IR。 AST的每个节点都是以下四种类型之一：if节点，for节点，block节点或user节点。一个if节点具有一个AST表达式作为条件，一个then节点，以及一个else节点。一个for节点具有初始化，条件和增量表达式，以及一个body节点。块节点表示复合语句，并维护节点列表。最后，由用户节点表示的语句表示为AST表达式。

AST表达式本身就是一棵树，内部节点带有运算符，而整数常量或标识符作为叶子。运算符集包含在类似于C的编程语言中找到的标准运算符，还包括更高级的运算符，例如min和max。布尔逻辑运算符和条件运算符（条件？a：b）有两种形式：一种使用短路评估，一种使用热切评估。我们在有关低级编译器的工作中发现了[Grosser等。 2012年]急切地求操作数而不是使用C的短路求值，通常是有好处的，因为它减少了控制开销并简化了循环不变子表达式的提升。

整数除法运算符也有不同的形式，其中之一对应于数学运算“ a / b”。不幸的是，该操作不能直接转换为bin C，因为C中的操作符朝着零[ISO 1999，6.5.5]取整，而不是朝着负无穷大。正确转换为C会涉及符号的条件，这会给某些体系结构（例如GPU设备）带来巨大的额外成本。因此，我们还具有一种整数除法形式，其中已知结果是整数（这样舍入就变得无关紧要），而又有一种已知除数是非负数的形式。用户可以为后一种形式指定首选项，在这种情况下，AST表达式生成器将寻找使用它们的机会（请参阅第5.10节）。同样，余数运算符有两种特殊形式：一种已知被除数为非负数，另一种仅将运算结果与零进行比较。在这些特殊情况下，余数运算符可以转换为C语言中的％运算符。

4.4 带注释的AST

AST节点是在调度的深度优先遍历中访问所有子级之后创建的。但是，AST生成器可能决定不对生成的AST节点中的符号常量需要满足的某些条件进行编码，以使这些条件可以提升到更高的水平。带注释的AST会同时跟踪（纯粹是语法上的）AST本身和此类多面体信息。除了前面描述的条件仍需要更高级别的AST强制执行，带注释的AST还跟踪已经强制执行的条件。然后可以使用后一组条件来简化甚至消除较高水平的库存中的某些待定条件。

5. AST的产生

本节介绍了核心AST生成算法。在6.3节中，我们将看到此算法适用于调度树中的每个频段节点。我们的核心算法源自“Quilleré等人”。该算法具有重大变化，例如**隔离，展开，起吊，部件检测，**移动步幅以及AST表达式的优化生成。

5.1 总览

示例5.1：在深入研究AST生成算法的细节之前，让我们首先将其应用于第3部分的简单示例，在该示例中，我们的算法与标准Quilleré等人的代码基本一致。算法。特别地，让我们考虑（2）的时间表。包括（1）的域约束在内的初始执行关系为

$$\begin{aligned}\{(i,0,0) \to \mathrm{S1}(i) : 0 \le i < n; (i,1,j) \to \mathrm{S2}(i,j) : 0 \le j < i < n; \\ (i,2,0) \to \mathrm{S3}(i) : 0 \le i < n\}.\end{aligned} \tag{5}$$

生成的AST的最外面的循环是从此关系的域中的第一个维度派生的。对该第一维进行投影得出

$$\{(i) : 0 \le i < n\}. \tag{6}$$

在这种简单情况下，我们发现一个由单个析取项描述的集合，并且可以从约束中轻松读取所生成的最外层for循环的上下限。如果投影是由多个析取物描述的，那么我们将必须将它们组合或首先将它们分解为不相交的片段。继续进行第二个维度，原则上，我们将在外部两个维度上找到以下投影，

$$\{(i,t) : 0 \le i < n \wedge 0 \le t \le 2\}, \tag{7}$$

并生成一个循环访问0到2的值。但是，我们可以看到，在此级别上，在S2或S3的任何实例之后都没有对S1的实例进行排序，并且相对于S3，S2的情况类似。因此，我们的组件检测非常简单，并继续按此顺序分别为S1，S2和S3生成AST。请注意，其他AST生成器通过不同的机制来处理这种特殊情况。在每个组件中，第二维都有固定值，因此不需要生成任何循环。分别对于包含S1和S3的组件中的三维尺寸，情况也是如此。包含S2的组件确实会导致一个额外的循环，该循环的处理方式与外部循环相同。

算法3构成AST生成的核心，并在通过调用算法1为下一个时间表维度生成AST节点后，为给定时间表维度为AST节点创建（可能退化的）AST。创建此类for节点的过程为详见5.3节。输入是与调度域中的当前维相对应的单相集。但是，此维度上的实际计划范围可能包含几个分离点。算法2负责将调度域分解（或过分逼近）为不相交的单相干片段，并在每个片段上调用算法3。第5.4节和第5.5节中介绍了分解调度域的不同方法。在示例5.1中，所有相关的计划域都包含一个单独的析取符，因此不需要其他处理。

ALGORITHM 1: Generate Next Schedule Dimension (“next”)

```
Input:
        — stock
        — executed relation
if at inner level then
  | return terminate(stock, executed)
end
list := ()
foreach sorted component do
  | /* detect shifted strides and record mapping in f,see Section 5.8          */
  | (stock, executed, f) := detect shifts(stock, executed)
  | /* project domain of executed on outer level dimension,see Section 5.2     */
  | domain := project(executed, level)
  | if has isolation domain then
  |   | (before, domain, after, other) = split on isolation(domain)
  |   | list′ := base(stock, executed, before, false)
  |   | list′ += base(stock, executed, domain, true)
  |   | list′ += base(stock, executed, after, false)
  |   | list′ += base(stock, executed, other, false)
  | else
  |   | list′ := base(stock, executed, domain, false)
  | end
  | /* undo schedule modification in case of shifted strides,see Section 5.8   */
  | list += transform(list′, f)
end
return list
```

最后，算法1是针对计划空间中每个维度（即已执行关系的域）进行（递归）调用的主要驱动程序。在检测到某些特殊情况后调用算法2。特别是，只要尚未达到内部级别，该算法就会首先按照5.7节中的说明查找组件。在示例5.1中，在第二个计划维中检测到三个组件。在每个组件中，算法都会按照5.8节中的说明检查移位的步幅，如果检测到任何移位的步幅，则可能会修改执行的关系。然后，已执行关系的域将投影到外层尺寸上，如5.2节所述。在示例5.1中，此投影会在外部进度表维度上生成集合{(i) : 0 ≤ i < n}（6）。最后，如果用户指定了需要隔离的时间表空间的一部分（请参阅第5.6节），则该算法将时间表域分为四个部分：隔离部分之前的部分，隔离部分本身，之后的部分以及该部分这是孤立部分无法比拟的。

ALGORITHM 2: Generate Component ("base")

Input:
- stock
- executed relation
- domain: part of schedule domain at current level for which AST should be generated
- isolated: Boolean indicating whether domain refers to isolated part

```
type := generation type(level, isolated)
if type = unroll then
    (bound, n) := find lower bound(domain)
    list := ()
    for 0 ≤ i < n do
        /* select domain elements at offset i from the lower bound */
        domain′ := slice(domain, bound, i)
        domain′ := shared constraints(domain′)
        list += (create(stock, executed, domain′))
    end
    return list
end
if type = separate then
    domain list := separate(domain, executed)
else if type = atomic then
    domain list := (shared constraints(domain))
else
    domain list := make disjoint(domain)
end
list := ()
foreach domain in sorted domain list do
    list += (create(stock, executed, domain))
end
return list
```

ALGORITHM 3: Create for-node ("create")

Input:
- stock
- executed relation
- bounds: single disjunct set describing bounds on current level for which an AST should be generated

```
domain := bounds intersected with domain of executed
stock′ := detect strides(stock, domain)
stock′ := check for single iteration(stock′, bounds)
list := next(stock′, executed)
/* combine a list of annotated ASTs into a single annotated AST, see Section 5.9 */
list := combine(list, stock, stock′)
return construct for loop(bounds, stock, list)
```

当达到内部级别时，在当前执行的关系范围内，进度表不会在语句实例之间做进一步的区分。因此，我们分别为每个语句生成一个AST。通常，执行的关系只会将单个语句实例与给定的调度点相关联，而我们只需创建并返回一个用户节点即可。否则，AST仍然需要遍历不同的实例，并且可以按任何顺序进行。因此，我们使用范围的副本扩展了已执行关系的域，并继续在已执行关系的域中处理新维度，直到再次达到内部级别为止，在这种情况下，已执行关系被保证只有一个语句与给定调度点关联的实例。

5.2 本地调度域约束

在给定级别生成的for循环的上下限是从涉及当前调度维的已执行关系域中的约束中得出的。这些约束还可能涉及其他计划维度，既对应于外部for循环的那些，又对应于内部for循环的那些。考虑到for循环的上下限只能引用外部循环的迭代器，（显然）不能引用内部循环的迭代器，因此我们首先需要将已执行关系的域投影到其第一级维度上。此操作可能会引入其他存在的量化变量，这些变量无法直接在AST表达式中编码。因此，我们需要以某种方式将其删除。

删除现有量化变量的一种方法是执行量词消除。此过程保留了集合的含义，因此确保仅执行具有任何关联语句实例的当前调度维的那些值。另一方面，数量词的消除可能会将计划域分解为几部分。特别地，isl通过应用参数整数编程[Feautrier 1988]计算量化变量的显式，仿射表示来执行量词消除。通常，这会将域分为几个部分，每个部分都有自己的拟仿射表达式。其他量词消除算法导致调度域的类似分解。

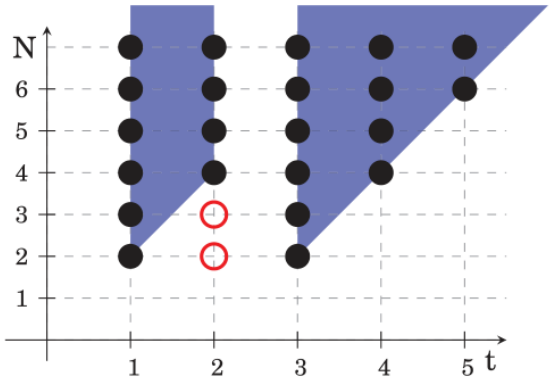

Fig. 4. Quantifier elimination (blue area) and Fourier-Motzkin (black dots + red circles) applied to (8) (black dots).

删除存在的量化变量的另一种方法是对它们执行FourierMotzkin消除。这可能会导致调度域的过度逼近，但是可以保证不会将调度域分解为多个析取项。在我们的AST生成器中，我们采用了第二个选项，以避免第一个选项导致的代码大小扩展。换句话说，我们应用傅立叶-莫兹金消除技术来删除所有尚无明确表示的存在量化变量。请注意，我们仅将可能的过度逼近的约束视为在此级别上生成的，这样，实际进度域上任何不受此过度逼近所满足的约束都将最终在更深层次上得到执行。

示例5.2以这两种方法产生不同结果的情况为例，请考虑以下某些计划域在最外部计划维度上的投影，

$$\{(t) : (\exists \alpha : \alpha \geq -1 + t \wedge 2\alpha \geq 1 + t \wedge \alpha \leq t \wedge 4\alpha \leq N + 2t)\}, \tag{8}$$

其中N是符号常数。对（8）中的集描述应用量词消除会导致

$$\{(t) : (t \geq 3 \wedge 2t \leq 4 + N) \vee (t \leq 2 \wedge t \geq 1 \wedge 2t \leq N)\}. \tag{9}$$

请注意，这是相同的集合，但是它以不同的方式描述，没有存在定量变量。此外，请注意，说明现在包含两个分词。另一方面，对（8）中的α变量进行傅立叶-莫兹金消除

$$\{(t) : 2t \leq 4 + N \wedge N \geq 2 \wedge t \geq 1\}. \tag{10}$$

该集合包含一个额外的元素，该元素不是（8）中的集合的元素。特别是，它包含额外的元素

$$\{(2) : 2 \leq N \leq 3\}. \tag{11}$$

图4以0≤N≤7的黑色显示了（8）中集合的元素，这些与量化符消除导致的（9）中集合描述的元素相同。 Fourier-Motzkin（10）引入的其他元素（如果有）以红色显示。

Using the set in (9) would produce the code

```
for (int c0 = 1; c0 <= min(2, floord(N, 2)); c0 += 1)
  // body
for (int c0 = 3; c0 <= floord(N, 2) + 2; c0 += 1)
  // body
```

whereas using the set in (10) produces the code

```
for (int c0 = 1; c0 <= floord(N, 2) + 2; c0 += 1)
  // body
```

如前所述，isl的量化消除可以通过准仿射表达式来代替量化的变量。在简化集描述的过程中，isl可以执行类似的替换。因此，我们需要考虑的是，即使使用傅立叶-莫兹金消除法消除了存在的量化变量后，调度域的投影也可能涉及这种准仿射表达式。如果这些准仿射表达式中的任何一个依赖于当前水平，则也可以使用傅里叶-莫兹金消除将它们消除，因为循环的下限不能取决于循环迭代器的值。类似地，涉及此类表达式的约束不能用于构造上限，因为它可能无法满足迭代器的某些值，同时又被迭代器的较高值所满足。请注意，此过程还会删除任何步幅信息，但是将如5.3.1节中所述从执行的关系中恢复此信息。

示例5.3考虑一个迭代域

$$\left\{ \mathrm{S}(i) : 3 \left\lfloor \frac{i+1}{3} \right\rfloor \leq i \wedge i \geq 0 \wedge i \leq 3 \right\} \tag{12}$$

有调度：

$$\{ \mathrm{S}(i) \rightarrow (i) \}. \tag{13}$$

计划域及其在最外部（且唯一）级别的投影为

$$\left\{ (i) : 3 \left\lfloor \frac{i+1}{3} \right\rfloor \leq i \wedge i \geq 0 \wedge i \leq 3 \right\}. \tag{14}$$

约束3？（i + 1）/ 3？ ≤i在电流水平上涉及一个拟仿射表达式，因此不能用于构造for循环边界。取而代之的是消除表达式，从而产生调度域

$$\{ (i) : i \geq 0 \wedge i \leq 3 \}. \tag{15}$$

然后在最内层考虑消除的约束，从而产生以下代码：

```
for (int c0 = 0; c0 <= 3; c0 += 1)
  if ((c0 + 1) % 3 >= 1)
    S(c0);
```

请注意，当达到最内层时，我们将无法再执行任何近似运算，而必须对所有剩余的存在的量化变量执行量化器消除。然后，这可能导致生成的语句周围出现析取条件。此外，请注意，Omega库中的量词消除过程与isl使用的过程不同，但它也可能导致域分裂。两者的详细比较超出了本文的范围。目前尚不清楚CodeGen +在哪一点以及在多大程度上应用了量词消除功能，但它似乎是为仅涉及单个存在的量化变量的约束量身定制的。

5.3 节点构建

5.3.1 步幅检测

除了股票和已执行的关系外，用于创建一个for节点的算法3还接受了一个称为bounds的凸集作为额外输入，从中将提取出for-node的界限。第一步是检测跨步，但是此信息无界，因为它不涉及任何依赖当前级别的准仿射表达式。相反，我们使用边界集与已执行关系的域的交集。从结果集的整数仿射外壳[Verdoolaege 2010]中提取步幅。此操作提取集合元素满足的相等约束，并保留（大部分）步幅信息。让

$$h(\mathbf{p}) + u(vi + sf(\boldsymbol{\alpha})) = 0 \tag{16}$$

是这些相等约束之一，其中i是当前计划维，p是符号常量和外部维，α是存在量化的变量。此外，v和s没有公因子，f和h是仿射表达式，因此f的系数没有公因子。如果v为非零且s大于1，则此等式表示非平凡的步幅约束。使用Bézout的身份av + bs = 1，我们可以将（16）重写为

$$ui = -ah(\mathbf{p}) + us(bi - af(\boldsymbol{\alpha})) \tag{17}$$

因此-ah（p）是u的倍数，而i等于o（p）= -ah（p）/ u模s。偏移量o（p）和步幅s存储在库存中。如果一个平凡的步幅有多个相等性，则可以将偏移量和步幅合并，并且总步幅将是步幅中最小公倍数。特别是，如果我们有两个偏移/步幅对

$$i = o_1(\mathbf{p}) + s_1\ f_1(\boldsymbol{\alpha}) \tag{18}$$

$$i = o_2(\mathbf{p}) + s_2\ f_2(\boldsymbol{\alpha}), \tag{19}$$

那么令g为s1和s2的最大公约数，令c和d为cs1 + ds2 = g（贝索特的身份再次成立）。将（18）中的方程乘以t1 = ds2 / g，将（19）中的方程乘以t2 = cs1 / g，我们得到

$$i = (d\,s_2 + c\,s_1)/g\; i = t_1 o_1(\mathbf{p}) + t_2 o_2(\mathbf{p}) + (s_1\,s_2)/g\,(bf_1(\boldsymbol{\alpha}) + af_2(\boldsymbol{\alpha})). \tag{20}$$

换句话说，组合偏移量为t1o1（p）+ t2o2（p），而组合步幅为（s1s2）/ g-s1和s2的最小公倍数。

示例5.4考虑计划域

$$\{(i) : \exists \alpha, \beta : 0 \le i \le 100 \wedge n - i + 6\alpha = 0 \wedge m - i + 10\beta = 0\}, \tag{21}$$

具有n和m个符号常量。对于约束n − i +6α= 0，我们使用（16）表示法，h（p）= n，u = 1，v = -1，s = 6，f（α）=α。我们可以取a = -1和b = 0来找到s1 = 6的o1（p）= n。我们类似地找到o2（p）= m和s2 =10。我们有g = 2和maytakec = 2 andd = -1，导致组合步幅为30，组合偏移为-5n + 6m。要看到此组合偏移量同时满足两个步幅约束，请注意，原始两个步幅约束的组合意味着n-错乘两个倍数。换句话说，边界集（从中消除了存在的量化变量）的形式为

$$\left\{(i) : 0 \le i \le 100 \wedge 2\left\lfloor \frac{m-n}{2} \right\rfloor = m - n\right\}. \tag{22}$$

因此，我们有m−（−5n + 6m）= 5（n− m）实际上是10的倍数。

5.3.2 循环约束

在检测到跨步之后，我们将为当前计划维生成的for循环所强制执行的约束添加到将用于构造后代节点的股票中，以便可以使用这些约束来简化那些后代节点。仅在创建这些后代节点之后，才执行与当前调度维对应的for循环的实际构造。在这一点上，我们将需要区分由for循环的边界强制实施的约束和可能需要由额外的条件约束实施的约束。因此，即使更新对后代节点没有影响，我们在更新存量时也要考虑到这种差异。

为了确定与当前计划维相对应的for循环强制执行哪些约束，我们首先需要知道我们是否还要构建for循环。特别是，基于当前库存中的约束，范围约束和步幅约束（如果有），我们可以确定当前进度表维只能达到单个值。在这种情况下，我们将为循环生成一个特殊的“简并”，我们允许用户将其初始值分配给循环迭代器。注意，该单个值通常将在符号常量和外循环迭代器中指定为分段拟仿射表达式。如果事实证明此表达式由单个准仿射表达式组成，那么我们根本不会生成任何for循环，而是在执行的关系中将当前调度维替换为该单个准仿射表达式。我们将这种情况称为消除循环。仅当用单个拟仿射表达式描述单个值时才执行此替换的原因是，否则，我们将在执行的关系中引入其他析取。在消除的情况下，我们从边界中消除了当前的计划维，并将结果添加到挂起的约束中。在其他情况下，我们将不涉及当前进度表维度的边界中的约束添加到待处理约束中，并将边界中的其余约束以及步幅约束相加（即，进度表维度等于偏移量的事实加上跨度的倍数）生成的约束。

示例5.5考虑计划域

$$\left\{(i) : i \geq 1 \wedge n-1 \leq i \leq n \wedge 4 \left\lfloor \frac{i-2}{4} \right\rfloor = i-2 \right\}, \tag{23}$$

nis是一个符号常量。步幅约束4？（i-2）/ 4？ = i-2没有出现在边界内，但是为了寻找单个值而将其加回去。从这些约束条件中，我们可以看出我获得了4？（n + 2）/ 4？的单个值。 − 2，并且该值表示为单个准仿射表达式。将这个值代入时间表域，我们得到

然后，将这些约束作为防护添加到最内层的带注释的AST。

请注意，实际上也很少消除并没有消除的简并循环。我们仅报告我们对这种情况的处理，以确保完整性，但很可能不是最佳状态。在大多数其余的罕见情况下，实际上可以消除循环，但是我们只是无法得出适当的单个拟仿射表达式。实际上，这些情况以前经常发生，但是大多数情况已经通过改进对单个拟仿射表达的检测得以解决。

在为for-node的主体生成带注释的AST之后，我们知道符号常量和外循环迭代器上的哪些约束由该子树强制实施，并且可以（可选）使用它们来简化挂起的约束。另外，关于生成的约束简化了待处理的约束。然后，将简化的待处理约束与从带注释的AST为主体所悬挂的约束（请参见5.9节）和一组附加的隐含约束进行组合。隐含约束是那些已生成的约束所隐含但它们的AST表达对应项未隐含的约束。特别地，这些是步幅约束所隐含的约束。在简并循环的情况下，这还包括所生成约束所隐含的约束。由于我们允许用户将简并的for循环视为赋值，因此，此上限不强制执行上限（大于或等于下限）的事实（即，甚至存在循环的单个迭代）。分配，因此需要分开考虑。然后，肯定会在当前级别或向上提升到更高级别生成简化的挂起约束，提升约束和隐含约束的组合。从这一点开始，特别是对于尚未从体内悬挂的if条件和for循环边界的条件的AST表达式的构造，因此可以将它们视为生成的约束。

示例5.6作为利用强制约束的效果的示例，请考虑迭代域

$$\{\mathrm{S}(i,j) : 0 \leq i < m \wedge 0 \leq j < n\}, \tag{25}$$

其中和n是带时间表的符号常量

$$\{\mathrm{S}(i,j) \rightarrow (i,j)\}. \tag{26}$$

将进度计划域投影到外部维度上会产生

$$\{(i) : 0 \leq i < m \wedge n \geq 1\}. \tag{27}$$

因此，此级别上的单个待处理约束为n≥1。在内部级别上，相对于库存约束简化的计划域是

$$\{(i,j) : 0 \leq j < n\}. \tag{28}$$

在此内部级别生成的for循环强制执行约束n≥1，因此可以选择使用它简化外部级别的待处理约束。如果我们利用此强制约束，我们将生成代码

```
for (int c0 = 0; c0 < m; c0 += 1)
  for (int c1 = 0; c1 < n; c1 += 1)
    S(c0, c1);
```

否则，待处理的约束将变成if条件，然后我们生成代码

```
if (n >= 1)
  for (int c0 = 0; c0 < m; c0 += 1)
    for (int c1 = 0; c1 < n; c1 += 1)
      S(c0, c1);
```

示例5.7作为跨步约束隐含的约束的示例，请考虑时间表

$$\{\mathrm{S}(t) \to (t) : \exists \alpha : 2t - n = 4\alpha \wedge 0 \leq t \leq 100\}, \tag{29}$$

具有符号常数。步幅检测发现步幅为2，偏移为n / 2。步幅约束（n / 2-t）m o d 2 = 0编码为

$$n - 2t - 4\left\lfloor \frac{n + 2t}{4} \right\rfloor = 0. \tag{30}$$

该约束意味着n是2的倍数。由于此步幅约束已添加到生成的约束中，因此可以在更深的层次上简化此事实。但是，实际生成的for循环并没有暗示它。因此，我们从（30）中消除了t并将结果约束添加到需要在外部级别生成的约束。最终的代码是

```
if (n % 2 == 0)
  for (int c0 = (n / 2) + 2 * floord(-n - 1, 4) + 2; c0 <= 100; c0 += 2)
    S(c0);
```

注意，我们在生成循环初始化时利用了n是2的倍数的事实。

5.3.3 AST表达式（仔细翻译了的）

现在让我们考虑初始化的构造以及从当前调度维度的上下限生成的for循环的条件。由于边界集合可能是调度域的过度逼近，因此在极少数情况下可能不涉及下界和/或上限。如果它们丢失了，那么我们就可以使用参数整数编程[Feautrier 1988]从域集中得出单个分段拟仿射变换的边界。如果此集合没有下限，则会报告错误。如果没有上限，则生成一个无限的for节点。在存在一个或多个下界约束 h(p) + vi ≥ 0并且 v> 0的标准情况下，每个约束都转换为下界 $\ell(\mathbf{p}) = \lceil -h(\mathbf{p})/v \rceil$ 并且将for节点的下限设置为这些下限中的最大值（作为AST表达式）。但是，果循环是跨步的，那么我们需要确保这个下限具有正确的以跨步为模的值。因此，我们首先用 $o(\mathbf{p}) + s\lceil (l(\mathbf{p}) - o(\mathbf{p}))/s \rceil.$

替换下界的每一个 $\ell(\mathbf{p})$

示例5.8考虑迭代域

$$\{\mathrm{S1}(i) : 0 \leq i \leq M; \mathrm{S2}()\}, \tag{31}$$

其中M是符号常量，带有时间表

$$\{\mathrm{S1}(i) \to (i,0); \mathrm{S2}() \to (0,1)\}. \tag{32}$$

假设在外层，我们希望为这两个语句生成一个循环，如5.4节所述，其边界设置为{（i）：i≥0}。此边界集在当前计划维度上没有任何上限，因此我们考虑了计划域

$$\{(0); (i) : 0 \leq i \leq M\} \tag{33}$$

代替。从这个集合中，我们可以得出上限

$$\begin{cases} 0 & \text{if } M \leq 0 \\ M-1 & \text{otherwise.} \end{cases} \tag{34}$$

生成的代码如下：

```
for (int c0 = 0; c0 <= (M <= 0 ? 0 : M); c0 += 1)
{
  if (M >= c0)
    S1(c0);
  if (c0 == 0)
    S2(0);
}
```

示例5.9接例5.4。 （22）中设置的界限在当前调度维度上仅具有一个下限，即i≥0。因此，循环初始化的默认值为max {？ − 0/1？ } =0。但是，此值可能不满足步幅约束，取决于mand n的值。因此，它被替换为

$$-5n + 6m + 30 \left\lceil \frac{5n - 6m}{30} \right\rceil = -5n + 6m + 30 \left\lfloor \frac{5n - 6m + 29}{30} \right\rfloor. \tag{35}$$

取决于用户设置，for-node的上限条件可以构造为循环迭代器与上限值的最小值（类似于下限值）的单个比较，也可以构造为比较的结合，每个比较均直接从上限约束。某些编译器的OpenMP支持可望达到单一的上限，而在FPGA上结合起来更有效[Zuo等。 2013]。最后，将独立约束添加到带有for节点的带注释的AST中。

示例5.10考虑形式的上限

$$M \geq c_3 + 1 \wedge c_1 \geq 3c_3 + 8. \tag{36}$$

如果用户选择了生成单个上限，则会生成形式为c3 = c 3 +1 & &c 1> = 3 * c 3 + 8的约束条件。

5.4 分离

在算法1中计算的调度域可以是任意的Presburger集，在isl中以析取范式表示。但是，在算法3中创建一个for节点需要一个单相集作为输入。然后，中间算法2的职责是用不相交的单相异域的有序序列替换调度域。获得这种单分离域的方法基本上有两种：通过单个分离域对整个域进行近似，或者将域分解为单分离域。第一种选择可能需要在当前构造的for节点的后代中引入其他保护，这可能会导致运行时开销。第二种选择可能导致代码重复，因为同一域的不同实例可能会出现在不同的单相调度域部分中。

对于每个计划维，用户可以指定采用这些选项中的哪个。或者，用户也可以指定应展开计划维度（第5.5节），或者可以不指定选项，在这种情况下，计划域以务实的方式分解为单相分离域的列表。

让我们更详细地考虑两个主要选项。如果指定了“ separate”选项，那么将分别为每个语句计算计划域。这些调度域中的每一个都分解为不相交的单相集，并计算出一个共同的细化。这是Quilleré等人的标准分离方法。算法[Quilleré等。 2000; Bastoul 2004]。在“原子”情况下，使用共享约束。换句话说，分离的约束又被考虑，并且仅保留整个调度域所满足的约束。在某些情况下，此过程可能会导致没有下限和/或上限-我们在5.3节中讨论了这种情况。

示例5.11接例5.8。考虑（33）中外部维度的计划域，此处重复

$$\{(0);(i) : 0 \leq i \leq M\}. \tag{37}$$

在分离的情况下，我们分别考虑每个语句的调度域，即{（0）}和{（i）：0≤i≤M}，然后应用标准分离算法，将这些集合分解为不相交的单个-分离集，导致

$$\{(0) : M \leq -1\}, \quad \{(0) : M \geq 0\} \quad \text{and} \quad \{(i) : 1 \leq i \leq M\}. \tag{38}$$

生成的代码如下：

```
if (M <= -1) {
  S2(0);
} else {
  S1(0);
  S2(0);
  for (int c0 = 1; c0 <= M; c0 += 1)
    S1(c0);
}
```

在原子的情况下，我们考虑由（33）中的析取函数共享的约束。在此示例中，只有一个这样的约束（即，i≥0）。注意，第一个析取项中的等式约束i = 0等于i≥0∧i≤0。在这种情况下，生成的代码如例5.8所示。

5.5 展开

AST生成中的展开工作是通过获取计划域的切片以获得当前计划维的连续值，并对每个切片调用“创建”来进行的。通过构造，调度维在每个切片中具有固定的准仿射值，并且不会创建实际的for-node。有两个因素在展开过程中起着重要作用：步幅检测和最合适的下限的选择。如第5.3.1节所述执行步幅检测。如果发现任何跨度，则在调度域中将其替换（i = o（p）+ si？），以选择下限。

示例5.12如果计划域的格式为

$$\{i : 0 \leq i < 1024 \wedge i \bmod 256 = 0\}, \tag{39}$$

然后将其替换为{i ?: 0≤i？<4}。

下限识别需要单个析取，因此我们再次考虑调度域的共享约束，尽管在这种情况下，我们也允许约束的恒定移动。对于每个下限约束h(p) + vi ≥ 0 w i t h v > 0，我们计算i+1的最大值？—h(p)/v？在调度域上。如果最大值存在并且具有值n，那么我们知道我们可以用i =？—h(p)/v？0 ≤ t < n的+ t .我们取这种n最小的下界.如果找不到合适的下界，那么我们报告一个错误.

示例5.13对于计划域

$$\{i : 0 \leq i < 1000 \wedge N \leq i < N + 4\}, \tag{40}$$

我们宁愿下限N（包含4个切片）胜过下限0（包含1,000个切片）。

5.6 隔离

算法1中的隔离由用户指定的选项控制。如果设置了该选项，则该选项描述了计划域的一部分，该部分应与计划域的其他部分隔离。典型的用例是从部分切片中分离出完整的切片，或者从序言和结尾中分离出一个可矢量化的循环。可以为隔离部分和调度域的其余部分分别指定原子/分离/展开选项。对于任何给定级别，隔离域首先被投影到第一级维度上，如第5.2节所述。特别地，内部维度被投影出来，并且所有存在的量化变量和所有涉及当前维度的准仿射表达式被消除。我们随后用它的共享约束来替换集合，以确保中心部分是连续的。与当前明细表域的交点产生隔离的中心部分。“之前”部分是通过首先构造一组迭代来获得的，这些迭代在中心部分的某个迭代之前执行，然后减去该中心部分。类似地，“之后”部分是通过首先构造一组迭代来获得的，这些迭代在中心部分的一些迭代之后执行，然后减去中心部分和“之前”部分。“其他”部分是通过从当前调度域中减去之前、中间和之后的部分而获得的，并且由那些与中间部分不可比较的迭代组成。如果为调度域的其余部分指定了任何原子/单独/展开选项，那么它将分别应用于之前、之后和其他部分。

例5.14。假设我们有一个迭代域

$$\{ \mathrm{S}(i) : m \leq i < n \}, \tag{41}$$

其中m和n是符号常数，带有初始时间表

$$\{ \mathrm{S}(i) \to (i) \} \tag{42}$$

并且我们希望将循环分成4份，这是一个可以使后端编译器向量化内部循环的因子（例如，考虑计算或目标向量指令集中使用的数据类型）。我们首先将调度修改为

$$\left\{ \mathrm{S}(i) \to \left(4 \left\lfloor \frac{i}{4} \right\rfloor, i \right) \right\}, \tag{43}$$

那么我们要挑选出那些导致恰好四个迭代的内部循环的迭代。特别是，我们需要确保第一个调度维度t = 4？i/4？属于第二维度的调度域，t + 3也是如此。因此，我们分离出满足以下条件的第一个调度维度的值

$$\{ (t) : m \leq t \wedge t + 3 < n \}. \tag{44}$$

将内部维度投射出来并用其共享约束替换该集合并不会修改该孤立集合。前一部分是

$$\{ (t) : n \geq 4 + m \wedge t \leq m - 1 \}, \tag{45}$$

后面的部分是

$$\{ (t) : n \geq 4 + m \wedge t \geq n - 3 \}, \tag{46}$$

另一部分是

$$\{ (t) : m - 3 \leq t \leq n - 1 \wedge n \leq m + 3 \wedge n \geq m + 1 \}. \tag{47}$$

生成的代码(如下所示)由一个执行序言计算的循环、两个枚举中心部分的循环(现在易于矢量化)和一个形成结尾计算的循环组成。另一部分还有一个额外的循环嵌套，在n和m的值产生一个空的中心部分的情况下执行。在某些情况下，通过将相关迭代作为前后部分的一部分进行枚举，可以避免为另一部分生成专用代码。我们目前没有进行这样的优化。

```
{
  if (n >= m + 4)
    for (int c1 = m; c1 <= 4 * floord(m - 1, 4) + 3; c1 += 1)
      S(c1);
  for (int c0 = 4 * floord(m - 1, 4) + 4; c0 < n - 3; c0 += 4)
    for (int c1 = c0; c1 <= c0 + 3; c1 += 1)
      S(c1);
  if (n >= m + 4 && 4 * floord(n - 1, 4) + 3 >= n) {
    for (int c1 = 4 * floord(n - 1, 4); c1 < n; c1 += 1)
      S(c1);
} else if (m + 3 >= n)
    for (int c0 = 4 * floord(m, 4); c0 < n; c0 += 4)
      for (int c1 = max(m, c0); c1 <= min(n - 1, c0 + 3); c1 += 1)
        S(c1);
}
```

例5.15 我们还简要说明了多维隔离。假设我们有一个迭代域

$$\{\mathrm{S}(i,j) : 0 \le i < n \wedge 0 \le j < m\}, \tag{48}$$

其中n和mare是符号常数，带有初始时间表

$$\{\mathrm{S}(i,j) \to (i,j)\}, \tag{49}$$

我们希望通过平铺来提高寄存器重用。为了实现这种寄存器分块(例如，3 × 4)，我们修改调度以

$$\left\{\mathrm{S}(i,j) \to \left(3\left\lfloor\frac{i}{3}\right\rfloor, 4\left\lfloor\frac{j}{4}\right\rfloor, i, j\right)\right\}. \tag{50}$$

然后，我们希望挑出那些导致两个内部循环和12个迭代的迭代，目的是生成可以完全展开的循环，而不会引入阻碍寄存器使用的条件。为了实现这一点，我们隔离完全位于调度域{(t3，t4) : 0 ≤ t3< n∧ 0 ≤ t4< m}中的调度维度t3= i，t4= j的瓦片。特别是t1= 3的瓦片？t3/3？，t2= 4？t4/4？如果(t1，t2)和n d (t1+2，t2+3)都满足这些约束，则属于调度域。因此，我们分离出满足以下条件的前两个计划维度的值

$$\{(t_1,t_2) : 0 \le t_1 \wedge t_1 + 2 < n \wedge 0 \le t_2 \wedge t_2 + 3 < m\}. \tag{51}$$

本例生成的AST(如下所示)由外层的两个循环嵌套组成。第一个迭代在维度上是完整的，而第二个迭代在这个维度上是部分的。在第一个循环嵌套内的第二层，瓦片被进一步分割成在第二维中完整的和部分的瓦片。请注意，当生成下面的AST时，我们已经在输入中(通过上下文节点，参见第6节)指定nand m已知不是很小(n > 2∧m> 3)。如果没有这条额外的信息，AST发生器将为n和m这样的小值生成额外的代码，对应于隔离的“其他”部分。

```
{
  for (int c0 = 0; c0 < n - 2; c0 += 3) {
    for (int c1 = 0; c1 < m - 3; c1 += 4)
      for (int c2 = c0; c2 <= c0 + 2; c2 += 1)
        for (int c3 = c1; c3 <= c1 + 3; c3 += 1)
          S(c2, c3);
    if ((m - 1) % 4 <= 2)
      for (int c2 = c0; c2 <= c0 + 2; c2 += 1)
        for (int c3 = -((m - 1) % 4) + m - 1; c3 < m; c3 += 1)
          S(c2, c3);
  }
  if ((n - 1) % 3 <= 1)
    for (int c1 = 0; c1 < m; c1 += 4)
      for (int c2 = -((n - 1) % 3) + n - 1; c2 < n; c2 += 1)
        for (int c3 = c1; c3 <= min(m - 1, c1 + 3); c3 += 1)
          S(c2, c3);
}
```

5.7 成分（组件）

5.8 移动步幅检测

5.9 组合带注释的AST

5.10 生成AST表达式

6 调度树

本节描述了我们的调度表示，它推广了其他调度表示，如命名的预伯格关系，并解释了如何调整AST生成算法以采用这样的调度树作为输入。

6.1 动机

多面体编译中使用的调度自然具有树的形式。这显然是代表程序原始执行顺序的调度的情况，因为它们处理循环和复合语句。特别地，复合语句首先执行第一个“宏语句”(它本身可以是循环或复合语句)中的所有语句实例，然后执行下一个宏语句中的所有语句实例。这个顺序可以用一个“序列”节点来表示，该节点表示其子节点应该按顺序执行。循环的执行顺序可以表示为一个仿射函数，它用语句实例来表示循环迭代器(或它的负数)。然后，语句实例按照该仿射函数的递增顺序执行。例如，原始执行顺序(2)的调度树表示如图5所示。

6.2 节点

6.3 AST生成

7. 实验结果

现在，让我们考虑AST生成器的定量方面，评估其与最新技术相比的运行情况。这些实验中使用的isl版本为isl-0.14-368-g23e8573。

7.1 鲁棒性

AST生成器的主要目标之一是，它应该接受任何格式正确的输入并应生成正确的代码。为了测试AST生成器的正确性，我们从CLooG和CodeGen +发行版中收集了输入，并验证了我们为它们各自生成了正确的代码。由于这些工具的输入类型构成了AST生成器接受的输入的严格子集，因此可以轻松地将输入转换为具有三个嵌套节点的调度树：域，上下文和波段。为了验证输出的正确性，我们使用pet [Verdoolaege and Grosser 2012]解析输出，并验证输出执行的语句实例是否与输入指定的语句实例相同，并且它们的执行顺序是否与输入调度相匹配。而且，如果输入调度是单值的（通常是这种情况），那么我们检查每个语句实例是否仅执行一次。我们对控制输出形状的选项的各种设置执行此测试。尽管我们知道可以检查不同版本的CLooG是否产生等效输出的工作[Verdoolaege等。 2012]，我们尚不了解先前的工作，无法系统地验证所生成的AST是否与输入时间表匹配。

对从CLooG输入生成的CodeGen +输出进行相同的检查，我们发现对于13个测试案例（在94个案例中），CodeGen +生成的输出包含N / A。这意味着CodeGen +无法根据调度代码中的循环迭代器来表达语句实例，这可能是由于CodeGen +将调度应用为预处理步骤并在调度域上进行了AST生成，从而阻止了语句的恢复非计划时间表的情况下。我们还发现了一个测试用例（较旧），其中所生成的代码包含在循环之外的循环迭代器上进行迭代的条件，以及一个测试用例（walters），其中输入中的某些语句实例未出现在所生成的代码中。 3请注意，N / A问题也出现在CodeGen +发行版随附的四个（禁用）测试用例中。一个原始的codegen测试用例（p.delft2）会产生错误“保护条件太复杂而无法处理”。用CodeGen +输入在CLooG上执行类似测试是不可行的，因为较旧版本的CLooG（在我们的增强之前）将不允许在输入中存在定量变量。

```
if (n >= 2)
  for (i = 2; i <= n; i += 2) {
    if (i%4 == 0)
      S0(i);
    if ((i+2)%4 == 0)
      S1(i);
  }
```

(a) CLooG 0.18.1 generated code

```
for (int c0 = 2; c0 < n - 1; c0 += 4) {
  S1(c0);
  S0(c0 + 2);
}
if (n >= 2 && n % 4 >= 2)
  S1(-(n % 4) + n + 2);
```

(b) isl generated code

```
#define intMod(a,b) ((a) >= 0 ? (a) % (b) : (b) - abs((a) % (b)) % (b))
for(i = 2; i <= n; i += 2)
  if (intMod(i,4) == 0)
    S0(i);
  else
    S1(i);
```

(c) CodeGen+ generated code

Fig. 7. Code for example from Chen [2012, Fig. 8(b)] (style edited).

7.2 生成的代码质量

我们通过相关工作中的示例说明了AST生成器的代码质量的改进。图7比较了具有迭代域{S0（i）：∃α：1≤i≤n∧i =4α; S1（i）：∃α：1≤i≤n∧i =4α+ 2}的输入的输出和时间表{S0（i）→（i）; S1（i）→（i）}，这是从Chen [2012，图8（b）]重构而来的示例，迭代空间扩展为负数。注意，由于第5.8节的偏移步幅检测，模运算从isl输出的内部循环中删除。在构造n％4> = 2条件期间，我们利用以下事实：仅当条件n> = 2也成立时才执行主体，如5.10节所述。对于Chen [2012，图8（a）]的输入，isl产生与CodeGen +相同的AST。

```
if (n >= 2)
  for (i = 2; i <= n; i += 2) {
    if (i%4 == 0)
      S0(i);
    if ((i+2)%4 == 0)
      S1(i);
  }
```

(a) CLooG 0.18.1 generated code

```
for (int c0 = 2; c0 < n - 1; c0 += 4) {
  S1(c0);
  S0(c0 + 2);
}
if (n >= 2 && n % 4 >= 2)
  S1(-(n % 4) + n + 2);
```

(b) isl generated code

```
#define intMod(a,b) ((a) >= 0 ? (a) % (b) : (b) - abs((a) % (b)) % (b))
for(i = 2; i <= n; i += 2)
  if (intMod(i,4) == 0)
    S0(i);
  else
    S1(i);
```

(c) CodeGen+ generated code

Fig. 7. Code for example from Chen [2012, Fig. 8(b)] (style edited).

图8（c）说明了第5.7节中组件的检测。该示例是Bastoul [2004，图6]使用的示例。在没有检测到组件的情况下，无需进行某些分离，如CodeGen +的图8（b）和CLooG 0.14.1的图8（a）所示。 Bastoul的代码[2004，图7]与isl生成的代码相似，但是是通过手动方式获得的或使用从未公开的“单一”技术的初步实现方式获得的。进一步说明，对于darte输入，即使完全分离，每个语句在isl输出中也只会出现两次。在CLooG输出中，每个语句出现五次，并被多个模态包围，其中一些是多余的。如第7.1节所述，在此示例中，CodeGen +产生不正确的输出。

```
for(i=1; i<=n-2; i++) {
  S0(i,i);
  S1(i,i);
  for(j=i+1; j<=n-1; j++)
    S1(i,j);
  S1(i,n);
  S2(i,n);
}
S0(n-1,n-1);
S1(n-1,n-1);
S1(n-1,n);
S2(n-1,n);
S0(n,n);
S1(n,n);
S2(n,n);
for (i=n+1; i <= m; i++)
  S3(i,j);
```

(a) CLooG 0.14.1

```
for(i=1; i<=m; i++) {
  if(i>=n +1) {
    S2(i,n);
  } else {
    S0(i,i);
    S1(i,i);
    if (i>=n)
      S2 (i,i);
  }
  for(j=i+1; j<=n-1; j++)
    S0(i,j);
  if(n >= i+1) {
    S0(i,n);
    S2(i,n);
  }
}
```

(b) CodeGen+

```
for (int c0=1;c0<=n;c0+=1) {
  S0(c0, c0);
  for (int c1=c0;c1<=n;c1+=1)
    S1(c0, c1);
  S2(c0, n);
}
for (int c0=n+1;c0<=m;c0+=1)
  S2(c0, n);
```

(c) isl codegen

Fig. 8. Code for youcefn taken from Bastoul [2004, Fig. 6] (style edited).

尽管刚刚介绍的C代码片段已经使读者理解了某些代码属性，但是在目标上执行的指令序列可能与我们看到的C代码明显不同，并且通常在很大程度上取决于所使用的目标编译器。为了了解常规C编译器对AST的理解和优化程度，我们分析了生成的代码的大小和运行此代码时执行的指令数，然后将其与生成的汇编代码进行分析相结合。对于本实验，我们使用选项-O3 -std = gnu99 -march = corei7-avx -mtune = corei7-avx -fno-inline，使用clang 3.6，gcc 4.9.1和icc 15.0.0编译代码。在其他编译中，我们禁用向量化（-fno-vectorize，-fno-tree-vectorize，n -d-no-vec）和循环展开（-fno-unroll-loops，-

unroll = 0）。生成的代码的大小以其所在函数的大小来衡量，这是通过在目标文件上调用nm -S获得的。动态指令的数量是使用Valgrind的callgrind工具测量的，该工具是一种概要分析工具，可跟踪代码的执行情况并为每个功能计算执行的指令数。在测量动态指令数时，我们的目标不是精确的周期性能预测，而是希望了解不同的编译器及其优化而不会迷失在目标特定细节中。

表I列出了我们从图7和图8中的代码获得的代码大小和指令数。作为语句，我们在图7中使用A [x] ++和B [x] ++，并使用A [x图8的] [y] ++，B [x] [y] ++和C [x] [y] ++。对于图7中的代码，我们看到使用clang生成的代码大于CLooGandCodeGen +生成的代码归因于循环结语isl生成。 Clang可以利用我是肯定的知识来仅保留CodeGen +的intMod指令的肯定分支，但无法删除CLooG和CodeGen +在循环体中引入的if条件。类似地，gcc还简化了intMod指令，并将条件控制流留在了循环中。由于isl生成的代码不评估循环主体中的条件，因此与CodeGen +和CLooG生成的代码（355至506个动态指令）相比，它执行的指令明显少得多（约190条指令）。 clang和gcc都不会展开或矢量化任何生成的代码；另一方面，icc对代码进行矢量化和展开。如果我们都禁用了两者，则icc无法成功消除intMod（）的负分支，也不会将余数计算转换为按位和（即，将i%（2n）转换为i＆（2n-1））。结果，与运行为icl生成的AST的代码icc派生时执行的192条指令相比，CodeGen +和CLooG的icc生成的代码不仅明显更大，而且还需要878甚至1105条动态指令。万一我们允许展开，icc会成功消除intModbranch，但仍无法将余数按位表示，并且gcc和clang在所有重要情况下都适用。启用自动矢量化时，icc仅针对isl生成的AST导出矢量代码。向量化会导致代码长度大幅增加（240字节至1,104字节），但会稍微减少指令数。动态指令数量的减少是不可能的，因为我们测试用例的内存访问模式已被限制，并且AVX上的内存访问需要额外的标量加载或混洗指令。允许收集和分散指令的指令集可能会显示出向量化带来的更多可见优势。对于包含更复杂语句的循环内核也是如此，其中内存访问复杂性的增加具有较低的相对影响。

我们在表1中从youcefn获得的性能结果表明，在禁用展开和向量化的情况下，isl AST生成器在所有编译器中均具有明显的代码大小优势，而动态指令数几乎没有差异。当允许循环展开时，icc是唯一利用循环展开的编译器，因此减少了所有AST的动态指令数量，并且对isl和CodeGen +的影响最大。值得注意的是，isl生成的代码现在仅执行16598条指令，仅需要784个字节的代码即可。当另外启用向量化时，所有编译器的动态指令数都将大大减少：使用icc编译的isl生成的代码可产生10,489条动态指令，这与使用icc编译的CLooG产生的10,383条指令非常接近，而其大小仅为1,008个字节，后者为2960个字节（对于类似的动态指令计数，为代码大小的三分之一）。

总结了此代码质量研究，isl可以执行控制流优化，从而显着提高性能，并补充展示并行性和局部性的循环嵌套转换。这些优化不是由普通的C编译器执行的，相反，其中一些启用了进一步的编译器优化，例如自动矢量化。通常，isl生成的代码也要小得多。注意，对于给定的代码大小预算，后者可能会为进一步展开和向量化打开机会。

7.3 模映射和现有量化变量

我们的算法旨在为任何Presburger关系生成一个有效的AST。第7.1节没有充分涵盖处理现有量化变量的一个领域。这可能是从全局内存到共享内存的模映射的结果，也可能是从一个完整的迭代空间到一组线程标识符的映射的结果。事实上，在存在量化变量的情况下生成有效的AST需要特别注意模块化算法，消除整数除法的余数，或者尽可能简化它们。

我们的算法旨在为任何Presburger关系生成有效且高效的AST。处理存在的量化变量是第7.1节未充分涵盖的领域。这些可能是由于从全局内存到共享内存的模映射，或者是从完整的迭代空间到一组线程标识符的模映射。确实，要在存在定量变量的情况下生成有效的AST，需要特别注意模块化算术，消除整数除法的余数或尽可能简化它们。我们从简单的模运算{S [i]→[i]：i = n mod 128}（在n≥0的情况下）开始，以验证是否可以完全检测到模运算。由于较旧版本的CLooG（我们的增强功能之前）不允许存在量化的变量，因此在本节中我们不将其与之进行比较。对于isl和CodeGen +，图9显示isl使用带有余数运算的单个语句，而CodeGen +生成循环。由于调用了intMod和额外的控制流开销，使用循环的效率非常低。但是，可以通过观察表达式n％128是所有周围循环的不变量来优化它，编译器的循环不变代码运动遍历应该可以访问它。两个稍微复杂一点的示例是从一组迭代到一组线程t1的映射，其中t≤t1 <128。第一个映射是一对一映射{S [i]→[i]：7≤i≤ 134∧i mod 128 = t1}，isl再次转换为一条指令；第二个是映射{S [i]→[i]：7≤i≤130∧i mod 128 = t1}，该映射将124个迭代映射到128个线程。 isl将此映射降低为单

个条件语句。对于两种情况，CodeGen +都会生成完整的循环嵌套。有趣的是，所有先前显示的循环都是仅一次迭代的简并循环。 CodeGen +无法检测到这些循环，而isl设计为始终识别简并循环（请参见5.3节）。

```
// Simple
S(n % 128);

// Shifted
S(((t1 + 121) % 128) + 7);

// Conditional
if ((t1 + 121) % 128 <= 123)
  S(((t1 + 125) % 128) + 3);
```

(a) isl

```
// Simple
for(i = intMod(n,128); i <= 127; i += 128)
  S(i);

// Shifted
for(i = 7+intMod(t1-7,128); i <= 134; i += 128)
  S(i);

// Conditional
for(i = 7+intMod(t1-7,128); i <= 130; i += 128)
  S(i);
```

(b) CodeGen+

Fig. 9. Modulo conditions (examples not supported by CLooG).

对于先前的测试用例，由于计划中的单个模运算，因此仅引入了单个存在的量化变量。对于更复杂的用例，例如已经包含模表达式的访问函数的模映射或嵌套模映射，通常可能会引入多个存在的量化变量。第一个测试用例{S [i]→[i]：∃（α，β：i =2α+3β∧0≤α<3∧0≤β∧0≤i <8）}涉及到a中两个存在的量化变量单一平等。 CodeGen +在这里中止，因为消息防护条件过于复杂而无法处理。在图10中，我们看到isl能够生成有效的代码（有关详细信息，请参见5.2节），可以对其进行展开，以提高效率并更好地理解所执行的计算。下一个测试用例是{S [i，j]→[i，j]：∃（α，β：0≤i≤1∧t1 = j +128α∧0≤j +2β<128∧510≤t2 +2β ≤514∧0≤2β-t2≤5）}，与图1中的示例相比有所减少。CodeGen +中止，无法立即生成多个通配GEQ防护。 isl要么生成具有多个循环边界的循环，要么（如果展开）一组条件语句。由于Chen [2012]没有讨论如何处理现有量化变量，因此CodeGen +的支持范围尚不清楚。在检查CodeGen +的源代码时，我们发现了几个代码路径，每个约束都需要一个单独存在的量化变量。 isl对每个约束中存在的量化变量的数量没有限制（请参见第5.2节）。

```
// Two e.q. variables
for (int c0 = 0; c0 <= 7; c0 += 1)
  if (2 * (2 * c0 / 3) >= c0)
    S(c0);

// Multiple bounds
for (int c0 = 0; c0 <= 1; c0 += 1)
  for (int c1 = max(t1 - 384, t2 - 514);
       c1 < t1 - 255; c1 += 1)
    if (c1 + 256 == t1 ||
        (t1 >= 126 && t2 <= 255 && c1 + 384 == t1) ||
        (t2 == 256 && c1 + 384 == t1))
      S(c0, c1);
```

(a) isl

```
// Two e.q. variables
S(0); S(2); S(3);
S(4); S(5); S(6); S(7);

// Multiple bounds
if (t1 >= 126)
  S(0, t1 - 384);
S(0, t1 - 256);
if (t1 >= 126)
  S(1, t1 - 384);
S(1, t1 - 256);
```

(b) isl unrolled

Fig. 10. Existentially quantified variables (examples not supported by CLooG/CodeGen+).

Table II. Code Size and Dynamic Instruction Count for Figure 9 and Figure 10

		Code Size (bytes)			Inst. Count		
		clang	gcc	icc	clang	gcc	icc
Modulo—simple	CodeGen+	100	66	400	15	5	42
	isl	38	28	32	9	8	8
	isl (unsigned mod)	**12**	**11**	**16**	**3**	**3**	**3**
Modulo—shifted	CodeGen+	116	145	416	14	15	43
	isl	45	33	48	10	9	8
	isl (unsigned mod)	**16**	**16**	**32**	**4**	**4**	**4**
Modulo—conditional	CodeGen+	116	145	416	14	15	43
	isl	78	63	80	19	18	18
	isl (unsigned mod)	**29**	**29**	**32**	**8**	**8**	**8**
Two e.q. variables	isl	49	49	64	8	8	8
	isl unrolled	49	49	**48**	8	8	**6**
Multiple bounds	isl	260	367	160	1,966	2,120	3,392
	isl unrolled	**140**	**150**	**112**	**20**	**21**	**20**

使用7.2节中介绍的方法，我们再次为前面显示的例子分析不同编译器生成的目标代码。对于不同的取模条件，表II中的结果显示了代码大小，尤其是动态指令计数相对于CodeGen+的优势，icc生成的代码对CodeGen+来说非常糟糕。一个例外是条件模的代码，由于isl生成的代码评估两个模表达式:一个在条件中，一个在语句本身，因此CodeGen+使用的动态指令略少(clang为14，而clang为19)。有趣的是，单个模表达式非常昂贵，尽管它使用的是2的幂除数，应该降低到有效的按位"与"运算。事实上，当使用有符号类型时，没有一个被测试的编译器有足够的信息来执行这种优化。正如我们在这个例子中知道的，余数的被除数总是非负的，我们可以通过手动将模被除数转换成无符号整数类型来传递必要的信息。isl用非负被除数标记余数指令，甚至主动形成它们(见第5.10节)，这样，如果需要，可以自动插入相关的转换。从结果中可以清楚地看出，关于被除数积极性的知识允许所有编译器优化取模条件，使得所有编译器和所有取模测试用例生成的isl代码现在都明显更小，并且比CodeGen+生成的代码需要更少的动态指令。

具有两个存在性量化变量的内核导致所有编译器的目标代码几乎相同，不管它是否展开。由于循环的大小是静态已知的并且非常小，所有编译器都会自动展开循环。相比之下，"多边界"测试用例中更复杂的循环不会被任何编译器展开，而是生成一段具有大量动态指令的复杂代码。通过利用AST生成器中可用的上下文信息，我们可以将代码展开成一个有效的(部分预测的)语句序列。这减少了代码大小，并导致动态指令数量的大幅减少。

总的来说，isl在利用现有的量化变量从一般的普雷斯伯格关系生成更有效的目标代码方面取得了显著的进展。这种能力使得一边的复杂循环转换的探索和另一边的有效控制流的(重新)生成之间有更好的解耦。isl通过利用最先进的模块化算术优化来实现这一点，通过特别关注操作数的符号来补充这些优化，并主动形成具有非负红利的表达式。isl还利用整数运算来识别不需要循环的情况，简化了紧凑控制流的生成。最后，虽然对于具有恒定行程和小指令数的循环，循环展开可以由C编译器执行，但是由于额外的可用上下文信息，更复杂循环的部分展开优选地由AST生成器执行。这在这些isl实验中得到了进一步的证实和推动。

7.4 索引集拆分

传统上，当通过改变单个语句实例的执行顺序来优化循环程序时，不同实例的顺序由一个调度来描述，该调度为每个语句包含一个向语句实例分配执行时间的准仿射表达式。对于某些变换，这样一个单一的仿射表达式是不够表达的，而是有必要对迭代空间的不同子集使用不同的仿射表达式。需要这种分段调度的优化被称为索引集分裂[Griebl等人，2000]变换。最近的两项工作，混合六边形镶嵌Grosser等人，2014年周期性模板计算的时间镶嵌[Bondhugula等人，2014年]，都必须产生具有分裂索引集的时间表。为了重新生成高效的控制流，这两项工作都得益于我们的AST生成器对分段调度的本地支持。在周期性模板工作的情况下，这个控制流允许对swim进行时间分片，这是一个大型的真实应用程序，是

SPEC 2000基准套件的一部分。据报道，基于索引集拆分的切片方案的使用导致了显著的性能改进，而使用手动转换是无法实现的，因为手动转换被认为过于复杂且难以调试。

图11显示了一个简单的1D模板代码，类似于swim内核中的循环。这个内核的迭代域是

```
#define mod(a, b) ((a) < 0 ? (a)+(b) : (a) >= (b) ? (a) - (b) : (a))

for (t = 0; t < N; t++)
  for (i = 0; i < 2N; i += 1)
S:  A[(t+1)%2][i] = A[t%2][mod(i+1, 2N)] + A[t%2][mod(i-1, 2N)]
```

Fig. 11. Original stencil code with wrapping dependences.

数据相关性是

$$S(t,i) \rightarrow \begin{cases} S(t+1,i+1) & \text{if } i < 2N-1 \\ S(t+1,i+1-2N) & \text{if } i = 2N-1 \end{cases}$$

$$S(t,i) \rightarrow \begin{cases} S(t+1,i-1) & \text{if } i > 0 \\ S(t+1,i-1+2N) & \text{if } i = 0. \end{cases}$$

现在，为了利用沿t维度的重用，有必要将代码平铺在t的多次迭代中。不幸的是，即使大多数数据依赖具有短的依赖距离((1，1)；(1，1))，也有一些具有更长的相关距离((1，1-2N)；(1，1 + 2N))在迭代空间边界处。后一种相关性阻碍了这种时间分片的应用。为了解决这个问题，可以使用以下分段调度:

$$S(t,i) \rightarrow \begin{cases} (t,i,0) & \text{if } i < N \\ (t,2N-i-1,1) & \text{if } i \geq N. \end{cases}$$

该调度将迭代空间分成两部分，并反转第二部分，使得相关迭代可以彼此靠近移动，并且相关距离缩短到(1，0，1)；(1,0,−1);(1,−1,0);(1,1,0).我们可以直接使用这个时间表来生成图12中的代码，或者，现在缩短了依赖关系，将其用作进一步切片的基础。

有了我们的AST生成器本身支持的索引集分割，就有可能生成这样的代码，而不必被迫引入新的虚拟语句，这些语句对分配了不同调度的迭代空间子集进行建模。尽管这种预处理是可能的，但它需要在调度优化器中采取一些AST生成步骤，从而打破了抽象障碍。虚拟语句的使用也向AST生成器隐藏了这样一个事实，即调度的不同部分执行相同的语句，这意味着没有复制这些语句的内在原因。相反，可以指示AST生成器避免对很少执行的代码路径进行代码复制(当使用隔离时，在部分之前或之后)，并使用代码复制来最大化计算中心部分的性能。

7.5 乐观转换的运行时防护

在常规C编译器的环境中执行循环嵌套优化时，输入程序通常不会提供足够的静态信息来验证感兴趣的转换的有效性。然而，仍然可以通过乐观地假设所需的程序属性并仅在乐观假设可以在运行时验证的情况下执行转换后的代码来应用这种转换，如果该运行时验证失败，则返回到代码的未优化版本。然后，第5.10节中的AST生成工具可用于为收集的条件打印有效的代码，以保护乐观转换后的代码。

这种AST生成特性的一个例子可以在LLVM的Polly循环优化器中找到[Grosser等人，2012]。Polly很大程度上依赖于用户提供的表达式的AST生成，这允许Polly方便地从建模为整数集的约束或从给定的仿射表达式中导出必要的运行时检查。波莉使用这个工具来提供一个“假设跟踪”框架，在这个框架中，在波莉的上下文中执行的分析和转换收集他们所采用的假设。

然后，Polly将这些假设收集在一个整数集中，利用isl来简化这个集合，最后使用呈现的AST生成器来导出AST表达式，该表达式在运行时验证收集的假设。Polly目前使用这个框架来跟踪关于多维数组的假设。它假设对固定大小数组的访问总是保持在界限内。对于1D阵列的多项式访问函数(例如，A [i + n * j]至A[])，Polly旨在恢复多维访问形状和相应的下标表达式[Grosser等人，2015]，因为这种转换通常将多项式相关性分析问题转化为通常更容易解决的线性问题。Polly还利用工具来构建AST表达式来处理指针别名，它乐观地假设没有别名。为了生成证明不存在混叠的运行时检查，Polly导出循环程序中的数组访问集，并为每个数组基指针计算在循环程序执行期间作为参数准仿射表达式访问的字典最小和最大(可能是多维的)下标向量，然后使用所呈现的AST生成器将其转换为AST表达式。

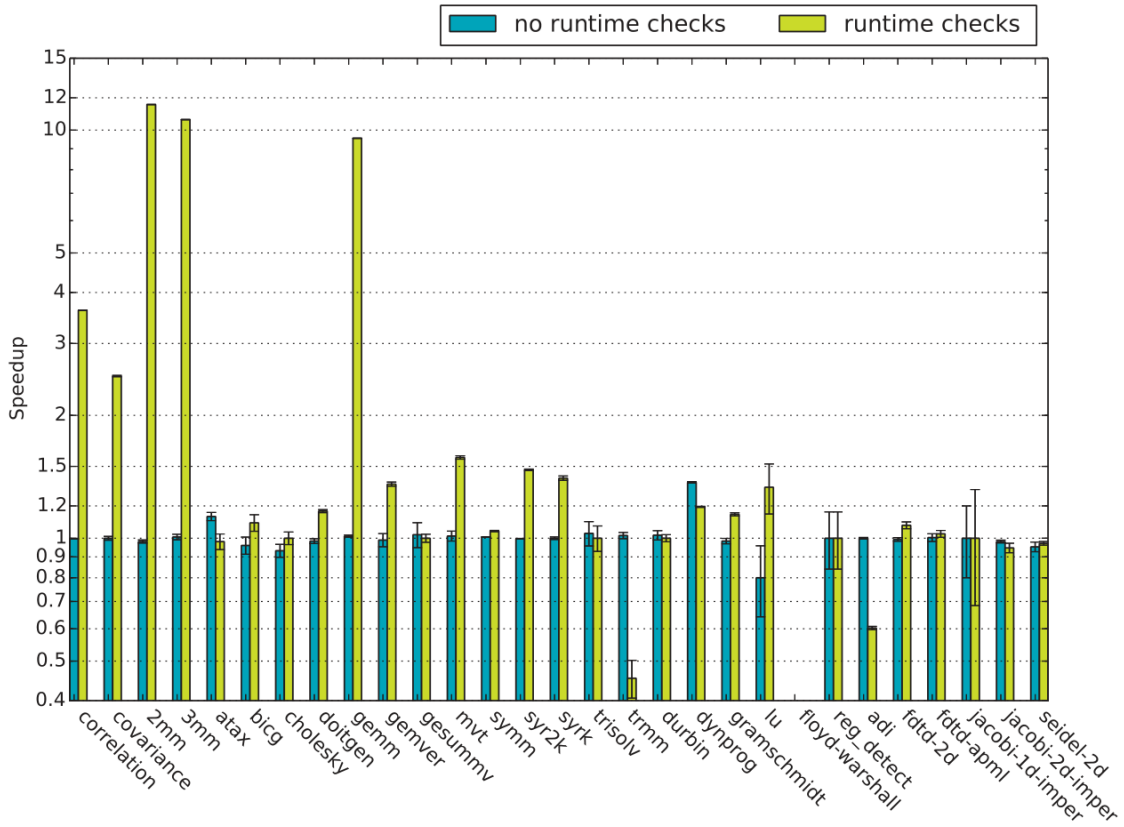

Fig. 13. Speedup of Polly with/without runtime checks in comparison to `clang -O3` on PolyBench 3.2.

为了让读者了解运行时防护器的发射所实现的额外优化，我们使用Claung和Polly (r236223)编译PolyBench 3.2，一次使用普通的Claung-O3，然后使用Claung和Polly，不支持运行时检查的生成，然后再次使用启用运行时检查生成的Claung和Polly。生成的二进制文件在英特尔至强E5430处理器上运行单线程。我们在图13中说明了结果(10次运行的中间值)。在图中，lu、jacobi1d-imper和reg_detect显示了非常高的方差，因为默认运行时间非常短，因此在后续分析中被忽略。

对于大多数基准测试来说，没有启用运行时检查生成的Polly产生了与clang -O3相同的性能，这主要是由于Polly不能静态地对循环内核建模，因此没有应用任何转换。当使用运行时检查时，29个基准中的6个比clang -O3产生50%以上的加速，另外3个基准产生至少20%的加速。波利的改进是由数据局部性转换(主要是平铺)引起的。平台之间的绝对加速可能有很大差异，主要取决于正确的时间表和良好的切片大小的选择，并且特别高，因为clang不执行任何循环切片。还有几个测试案例，我们发出的运行时检查启用了进一步的转换，但是波利的试探法选择的转换与clang -O3或没有运行时检查生成的波利相比降低了性能。

我们强调，我们的贡献不在于找到最佳代码转换，甚至不在于找到假设跟踪框架的定义。然而，给出的例子和数字很好地说明了什么样的优化可以通过我们的AST生成器对AST表达式生成的支持来实现。有不同的方法来导出运行时检查，并且不总是需要使用多面体AST生成器来发出它们。然而，对于更复杂和(可能是部分冗余的)条件的生成，使用基于整数集的框架来收集这些条件，并使用AST生成器来直接为它们生成代码，这在Polly中表现良好。基于快速原型生成器的方法还使得未来更依赖于上下文的优化可直接用于任意仿射调度的快速原型生成，以及用户提供的条件和表达式的生成。

7.6 AST生成策略的性能含义

为了理解我们新的AST生成策略的性能含义，我们分析了它们对生成代码运行时的影响。我们通过分析一个完整的端到端领域特定的编译器来确保一个现实的场景。作为一个编译器，我们选择第2节中介绍的模板编译器。我们提醒读者，这个编译器是基于通用编译器PPCG。为了创建针对模板计算领域优化的代码，通用执行调度的计算被替换为针对模板计算领域专门优化的混合六边形/平行四边形执行调度的计算。除了特定于领域的时间表，唯一的其他特定于领域的部分是我们的AST生成器的参数化，以从部分瓦片中隔离(第5.6节)完整的瓦片，以及展开(第5.5节)计算和输入输出代码。AST表达式生成(第5.10节)用于专门化语句的访问功能，例如在展开或分离之后。

Table III. AST Generation Strategy-Based Performance (GFLOPS)

	Heat 2D		Heat 3D	
AST Generation Options	GFLOPS	Speedup	GFLOPS	Speedup
a: no options enabled	2.1	1.0x	5.5	1.0x
b: all optimizations enabled	28.4	13.4x	21.0	3.8x
c_1: all, except full/partial separation	21.5	10.2x	19.7	3.6x
c_2: all, except IO unrolling	5.1	2.4x	10.5	1.9x
c_3: all, except compute unrolling	15.9	7.5x	11.3	2.1x
c_4: all, except modulo detection	29.1	13.7x	18.3	3.3x

我们使用CUDA编译工具5.5.0在NVIDIA NVS 5200M GPU上执行评估，使用热2D和热3D模板作为基准，并报告10个样本的中值结果。由于性能结果显示了2D和3D模板之间的巨大差异，我们选择了两个基准来涵盖最常见的维度。我们将自己限制在单一类型的模板上，因为不同类型模板之间的一般趋势没有变化到足以为这一分析提供额外的见解。关于不同硬件和不同模板类型的进一步性能结果，我们请读者参考格罗斯等人[2014]。表三显示了我们的分析结果。我们在a中看到，没有启用进一步专门化的正常AST一代产生非常低的性能，在2D情况下只有2.1 GFLOPS，在3D情况下只有5.5 GFLOPS，而在b中，我们启用了所有优化，在2D情况下获得28.4 GFLOPS，在3D情况下获得21.0 GFLOPS，在2D情况下加速13.4倍，在3D情况下加速3.8倍。为了更好地理解这种加速的来源，我们单独禁用了某些优化。

在c1中，我们禁用了完全/部分图块分离，这将热2D的性能降低了24%，热3D的性能降低了6%。2D的较大变化是由于全瓷砖的百分比较高。在3D中，已经有大量的时间花费在部分切片上，因此加速完整切片执行的优化不太明显。在C2和c3中，我们看到对于热3D，禁用IO展开或计算展开会降低大约50%的性能。对于2D的情况，禁用计算代码的展开也会降低44%的性能，更重要的是，如果不展开输入输出代码，会损失80%以上的性能。这种巨大的性能差异是由于展开后增加的ILP和通过展开实现的简化(参见图2)。在c4中，我们看到在没有模检测的情况下，热3D的性能降低了13%，令人惊讶的是，在2D，性能略微提高了2%。热2D的增加是由于循环不变代码运动引起的寄存器溢出，这也是由于模检测后代码更简单而成为可能。允许nvcc，t h e NVIDIA编译器，使用更多的寄存器来防止寄存器溢出，模数检测再次变得有益，热2D的峰值性能达到29.4 GFLOPS。总的来说，我们看到仅仅使用多面体扫描生成控制流远远不足以生成高性能的GPU代码。取而代之的是，多面体展开和全部和部分瓦片的特殊化对于获得具有竞争性能的代码是非常重要的。事实上，与几乎未优化的代码相比，我们实现了较大的加速，这可能不会让读者感到惊讶。然而，我们要注意的是，这种优化目前在任何其他AST生成器中都不可用，即使其中一些可以在额外的预处理或后处理的帮助下执行，让所有优化在AST生成器内仔细交互也是有益的。它实现了相关优化的紧密集成，例如，模检测可以访问关于当前展开的循环的信息。此外，能够在不影响程序进度的情况下逐渐增加优化，从而不影响程序的正确性，这在开发新的基于AST的工具时非常有用。

7.7 生成时间

虽然我们主要关注的是易用性和生成的AST的质量，但为了完整起见，我们也报告了一些AST的生成时间。对于这个实验，我们取用CLooG分发的64个测试用例(CLooG 0.14.1和CodeGen+都可以处理的测试用例)并合计AST生成总时间。对于CLooG 0.14.1(在我们增强之前，使用PolyLib作为后端)，我们使用固定大小的整数计算获得0.3s，对于任意精度的整数获得1.0s。对于CLooG 0.18.1(包括我们的一些增强和使用isl进行任意精度整数的集合运算)，我们得到0.9s，对于CodeGen+，我们得到3.1s，对于isl，得到1。我们将CLooG的时间差归因于我们还没有实现Vasilache等人的一些启发式方法，并且我们在优化方面更加积极，从而产生了更好的输出代码。

7.8 生成代码性能

很难比较不同AST生成器生成的代码的性能。首先，我们接受作为输入的时间表的类型比CodeGen+和旧版本的CLooG支持的要通用得多。为了能够进行比较，我们因此用相对简单的时间表进行了实验。特别是，我们采用了每一个PolyBench基准[Pouchet 2012]，应用了isl调度器(冥王星调度器的变体[Bondhugula等人，2008])，平铺了最外层的可平铺循环，并生成了CPU代码。对于我们的AST生成器，我们在磁贴循环带上设置原子选项，在最里面的带上设置单独的选项。对于CodeGen+，我们使用默认选项，本质上是在最里面的循环上进行分离。我们还修改了CodeGen+的源代码，以生成对二进制最小和最大宏的嵌套调用，而不是对n进制宏的单一调用。

比较不同AST生成器生成的代码性能的另一个问题是，生成的AST或源代码仍然需要由编译器编译，使用不同的编译器会导致显著不同的性能结果。因此，我们使用三种不同的编译器编译生成的代码:gcc 4.9.2带有选项-O3 -march=native，clang 3.5带有选项-O3 -march=native，a n d icc 13.1.0带有选项-fp-model strict -O3 -xHost。实验是在AMD皓龙6164 HE处理器上进行的。对于每个基准测试和每个编译器，我们使用了标准的PolyBench性能度量，即运行实验五次，去掉最快和最慢的执行时间，取剩余三次执行时间的平均值。

为了使比较尽可能公平，我们进行了以下调整。AST生成问题的上下文——即符号常数上的已知约束——由isl和CodeGen+进行不同的处理。而在我们的AST生成器中，上下文只是用来简化生成的AST，在CodeGen+中，上下文约束可能会出现在AST中。默认上下文由pet[Verdolaege和Grosser 2012]提取，包含从整数类型派生的符号常数的边界。在这种情况下，CodeGen+将在for循环的下限中使用符号常数的下限，这进一步暴露了一个b u g i n CodeGen+将使用正常数替换负常数，导致循环不被执行。对非负符号常数的限制避免了这个错误，但是符号常数的上限仍然会被用作for循环的上限，导致执行时间长几个数量级。因此，我们决定从CodeGen+输入中删除上下文。另一方面，我们发现一些编译器，尤其是gcc，对最小和最大宏的实现方式非常敏感。特别是，平均来说，当这些宏只计算它们的参数一次时，gcc会产生更快的代码，极端情况高达两倍。由于某种未知的原因，这种影响在isl生成的代码上比在CodeGen+生成的代码上更明显。在我们的实验中，我们使用宏，这些宏只计算它们的参数一次，使用gcc扩展，这些扩展也被clang和icc支持。

为了使比较尽可能公平，我们进行了以下调整。AST生成问题的上下文——即符号常数上的已知约束——由isl和CodeGen+进行不同的处理。而在我们的AST生成器中，上下文只是用来简化生成的AST，在CodeGen+中，上下文约束可能会出现在AST中。默认上下文由pet[Verdolaege和Grosser 2012]提取，包含从整数类型派生的符号常数的边界。在这种情况下，CodeGen+将在for循环的下限中使用符号常数的下限，这进一步暴露了一个b u g i n CodeGen+将使用正常数替换负常数，导致循环不被执行。对非负符号常数的限制避免了这个错误，但是符号常数的上限仍然会被用作for循环的上限，导致执行时间长几个数量级。因此，我们决定从CodeGen+输入中删除上下文。另一方面，我们发现一些编译器，尤其是gcc，对最小和最大宏的实现方式非常敏感。特别是，平均来说，当这些宏只计算它们的参数一次时，gcc会产生更快的代码，极端情况高达两倍。由于某种未知的原因，这种影响在isl生成的代码上比在CodeGen+生成的代码上更明显。在我们的实验中，我们使用宏，这些宏只计算它们的参数一次，使用gcc扩展，这些扩展也被clang和icc支持。

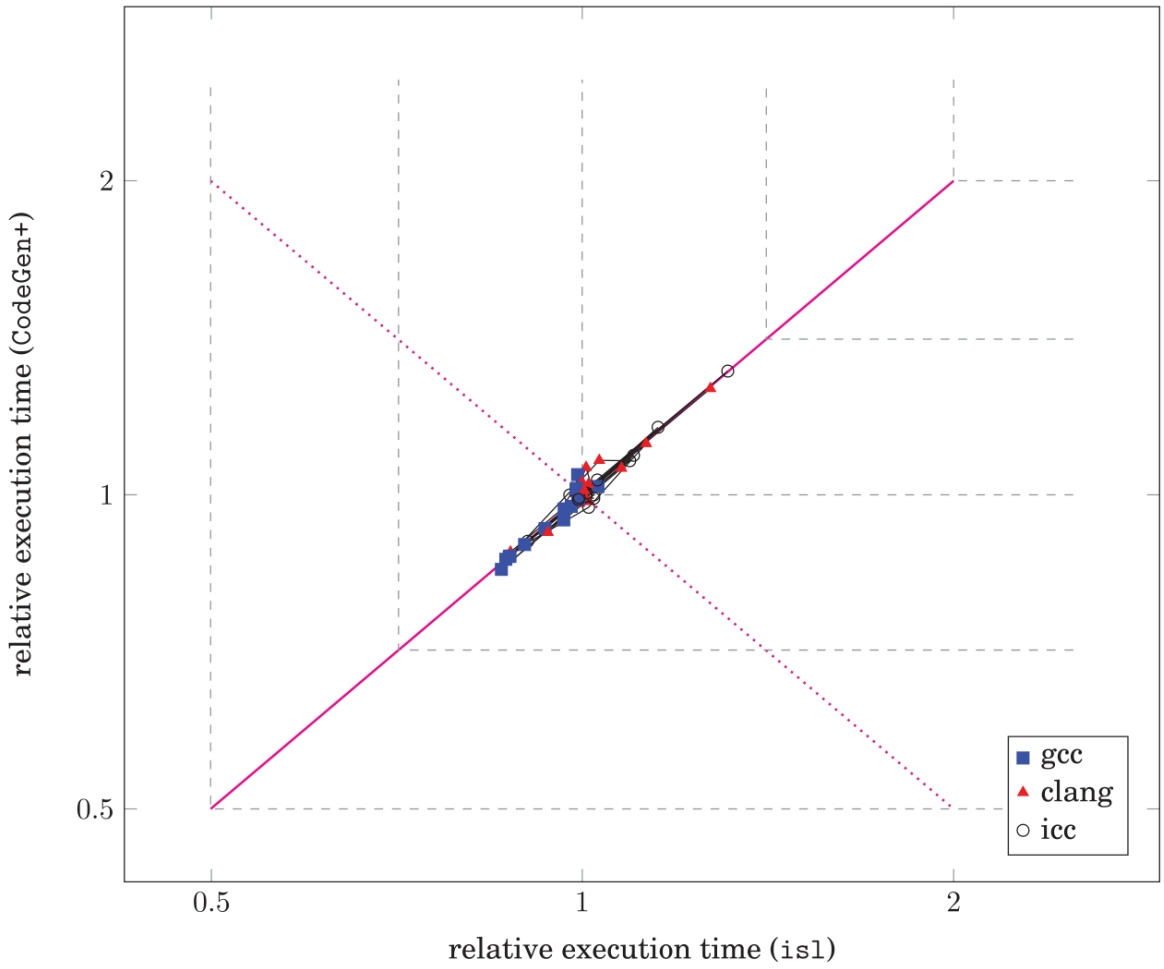

Fig. 14. Relative execution time—benchmarks with consistent performance.

结果如图14和15所示。图14显示了16个基准的性能结果，其中isl和CodeGen+之间的性能差异小于1.1倍。图15显示了14个基准测试的结果，其中至少一个编译器的差异更大。数据的呈现方式需要一些解释。由于我们主要对相对执行时间感兴趣，所以我们用六个执行时间的几何平均值(两个AST生成器和三个编译器)来划分每个基准的执行时间。对于每个基准测试和每个编译器，我们在相对执行时间绘制一个标记，并将这三个标记连接起来。根据构造，每个结果三角形的中心位于第二条对角线上(点洋红色线)。主对角线(全洋红色线)上方的点是isl生成的代码更快的点。主对角线以下的点是CodeGen+生成的代码更快的点。与主对角线的距离是衡量性能提高的一个指标。类似地，位于第二条对角线下方的点是对应的编译器比三个编译器的平均值产生更快代码的点。

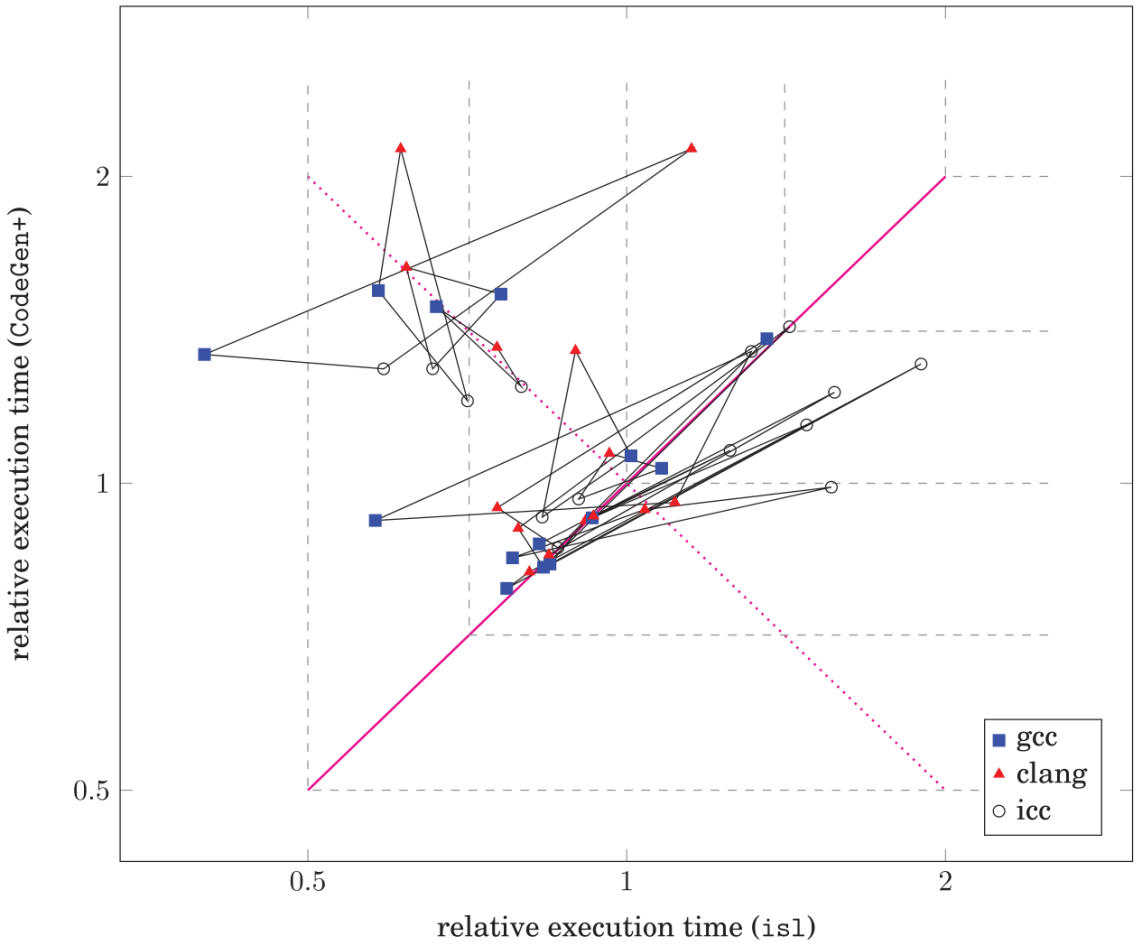

Fig. 15. Relative execution time—outliers.

总的来说，我们看到isl和CodeGen+生成的代码的性能非常相似，编译器之间的差异大于AST生成器之间的差异。因此，只考虑一个编译器可能会导致误导性的结果。例如，有五个基准测试，其中icc在CodeGen+生成的源代码上生成的代码要快得多，但其他编译器为这两个源代码生成的代码要快得多。另一方面，还有一个基准，icc在isl生成的源代码上生成的代码要快得多。还有四个基准测试，其中isl生成的代码在用所有三个编译器编译时性能更好。快速调查表明，这种差异是由于CodeGen+在某些情况下无法检测到步幅，而第5.3.1节的步幅检测能够检测到所有步幅。

虽然这些数字似乎表明isl生成的代码比CodeGen+生成的代码更快，但我们想再次强调，这些结果取决于许多因素，因此我们不想做出任何此类声明。比如，当使用min和max宏对它们的参数求值两次时，gcc编译的代码通常要慢得多。此外，对于isl生成的源代码来说，这种放缓更加明显，导致一些基准测试在CodeGen+生成的源代码上显示出最好的性能(但比我们的数字中的性能更差)。

我们还简要研究了陈[2012]报告的实验，其中CodeGen+生成的代码的性能比CLooG (0.16.3)生成的代码有所提高。我们的AST生成器生成几乎与CodeGen+相同的代码，具有出色的性能。我们确实发现，在使用它们的输入公式的lu情况下，代码生成时间稍微长一些(8.7s vs 2.6s)，在这种情况下，转换被应用于迭代域而不是时间表中，包括重复迭代域以获得展开的效果。使用一个更自然的调度树公式，我们可以在1.4s中生成相同的代码。

8 相关工作

通用AST生成有两种主要方法：一种基于Presburger关系库，着重于提高控制开销[Kelly等。 1995; Chen 2012]，以及一种基于有理多面体并且主要试图消除自上而下的开销的方法[Bastoul 2004; Quilleré等。 2000]。我们的方法可以看作是一种组合，使用了与Bastoul [2004]和Quilleré等人相同的分离算法。 [2000]，但建立在Presburger关系图书馆的顶部。

从历史上看，我们首先将CLooG移植到isl并进行了改进。后来，我们在isl之上构建了一个新的AST生成器。两种原始方法都只允许单个分离的上下文和进度表，而CLooG还不支持现有的量化变量。CodeGen +可以在某些情况下处理此类变量，但由于Chen [2012]并未讨论一般如何处理此类变量，因此支持程度尚不清楚。相比之下，我们的AST生成器支持Presburger算术的全部通用性，包括存在的量化变量和分段计划。

关于生成的AST的质量，isl中尚未实现文献中提出的某些扩展。第5.7节的组成部分与Bastoul [2004年，第二部分]的“单独”程序具有相同的目的。 4.2]。但是，在unisolate过程尝试取消某些分离的情况下，这些组件使我们甚至无法应用分离。而且，从未公开发布过该孤立程序的实现。相反，最新版本的CLooG实现了我们的组件检测。 Vasilache等。 [2006]提出了一些在CLooG中实施的优化方法，以减少AST生成时间。其中一些优化是针对其对调度树的编码而定制的，当将调度表示为显式调度树时，则不需要这些优化。相同的作者还提出了一种消除模态条件的算法，该算法一方面可以看作是5.8节的偏移步幅检测的一般化，但另一方面是基于限制性更强的多面体公式。 Zuo等。 [2013]描述了为高级合成量身定制的CLooG的几种微调，其中只有一些在isl中可用。

凯利等。 [1995]和Chen [2012]，以及Quilleré等。 [2000]和Bastoul [2004]生成了必要的AST表达式，以生成用于扫描迭代空间的控制流，但它们没有公开任何功能来为用户提供的任何分段仿射仿射表达式生成AST表达式。我们还不知道有任何使用AST生成上下文来专门化AST表达式的工作，尤其是当它们在准仿射表达式中出现时优化模运算和除法。 CodeGen +总是生成昂贵的intMod调用，而CLooG仅在将运算符的结果与零进行比较的情况下才引入％运算符。

Vasilache等人已经提出了在AST生成器中进行多面展开的功能（没有提供可用的软件）。 [2006]对于单模进度表的特殊情况，其中具有单个下限和单个上界偏移一个恒定非参数距离的维可以完全展开。在我们的工作中，我们介绍了由任意Presburger映射定义的计划的多面体展开，并支持在存在多个下限的情况下展开，在存在步幅的情况下展开以及使用条件语句对有界，非恒定迭代次数的循环展开。在多面体AST生成器中尚未实现用户控制的迭代空间任意子集的隔离。 Bastoul [2004]使用的自动分隔通常会引入专门的代码版本，但用户只能控制分隔的数量，而不能控制彼此分离的子集。 Ancourt和Irigoin [1991]以及Goumas等人已经讨论了全部/部分平铺作为独立的转换。 [2003]，并结合展开，Jiménez等人。 [2002]。在参数平铺的背景下[Kim等。 2007; Renganarayanan等。 2007; Hartono等。 [2010年]，已针对专用于此用例的AST发生器研究了全部/部分瓦片分离。我们不知道有任何工作使用多面体AST生成器提供的通用隔离功能来执行完全/部分瓦片分离。由于参数平铺技术通常依赖于多面体AST生成器，因此在参数平铺的情况下，相同的隔离技术可能会很有用。

我们不知道有任何工作可提供如此细粒度的可配置性。 CLooG [Bastoul 2004]最初允许对分离进行多维级别的控制，最近又获得了对每个语句的控制。 Chen [2012]允许对控制流的数量进行逐级控制。据我们所知，针对所生成的AST的不同子树的不同AST生成策略是我们工作所独有的。另外，使用户能够执行“原子” AST生成策略以最小化代码大小或强制展开的功能是新的。

9 结论

这项工作大大拓宽了多面体AST产生的范围。它通过将传统的控制流生成扩展到Presburger算术的全部通用性来实现。尤其是，我们为分段仿射计划以及复杂使用现有量化变量的计划提供支持，将AST生成开放给新的应用领域和更复杂的程序优化，并增强其可靠性-可预测地生成高效控制流的能力。我们的工作还通过对步幅和组成部分进行了优化，提高了生成的命令式代码的质量。我们还承认优化问题不仅限于控制流重组，还需要更改数据访问功能：为了支持这种优化，我们提出了从分段准仿射形式生成有效AST表达式的工具。最后，我们改进了用于在生成的代码中恢复除法和模表达式的最新技术，并将这些技术应用于优化索引表达式的优化，这些优化通常出现在显式管理的高速缓存的上下文中。总体而言，我们扩大了通用AST生成器的范围。

但是，要实现达到最高性能的领域或目标特定优化，通常需要对生成的代码进行大量专业化处理。为此，我们允许对AST生成器进行参数化，以在非常普通的多面体设置中执行循环展开和循环迭代器的部分评估。此外，我们介绍了如何分离代码的某些部分，并展示了如何使用这种分离方式为完整和部分图块生成专用代码。通过允许根据用户生成的AST表达式的生成环境进行特殊化，相同的功能也可以用于生成边界条件的特殊代码。由于最大的专业化不一定总是最好的，因此我们可以在细粒度的级别上配置AST生成选项，例如分离，展开和原子执行。每个单独的贡献本身都是有用的，但是只有集成在单个AST

生成器中才能确保它们的无缝交互。如混合六边形/经典平铺显示的那样，生成的AST生成器可以实现复杂的特定于域的优化，并且是开发特定于问题的代码生成器的有力替代方法。