arXiv:2601.02218v1 [cs.PL] 5 Jan 2026

MLIR-Smith：一种用于评估编译器流水线的新型随机程序生成器

Berke Ates
beates
@student.ethz.ch 苏黎世
联邦理工学院 瑞士

Theodoros Theodoridis
theodoros.theodoridis@inf.ethz.ch
苏黎世联邦理工学院 瑞士

Filip Dobrosavljević
dofilip@student.ethz.ch
苏黎世联邦理工学院

瑞士

Zhendong Su
zhendong.su@inf.ethz.ch
苏黎世联邦理工学院

瑞士

摘要

编译器对于软件的性能和正确执行至关重要，在各种科学领域都具有普遍意义。尽管如此，用于测试和评估编译器的工具仍明显不足，尤其是在灵活可扩展的多级中间表示（MLIR）环境中。本文针对这一需求展开，提出一种能够适应 MLIR 可扩展性的工具，而这一特性是 Csmith 等以往方法所不具备的。本文介绍了 MLIR-Smith，这是一种专为测试和评估基于 MLIR 的编译器优化而设计的新型随机程序生成器。我们通过对 MLIR、LLVM、DaCe 和 DCIR 开展差分测试，展示了 MLIR-Smith 的实用性，并在这些编译器流水线中发现了多个缺陷。MLIR-Smith 的推出不仅填补了编译器测试领域的空白，也凸显了对这些系统进行全面测试的重要性。本文提供了一种能够生成随机 MLIR 程序的工具，增强了我们评估和改进编译器的能力，并为未来工具的发展铺平道路，有望影响软件测试和质量保证的整体格局。

1 引言

编译器优化通过采用死代码消除、循环展开和公共子表达式消除等多种技术，显著提升了程序的性能 [16]。可以通过使用随机生成的程序进行模糊测试和差分测试，评估这些优化的效果以及编译器的正确性。Yang 等人 [24] 使用 Csmith 展示了这一方法。Csmith 是一款专门用于生成随机 C 程序的工具。

创建多层中间表示（MLIR） [11]项目的目标是降低编译器构造的复杂性。MLIR 是一种统一的基础设施，旨在缓解软件碎片化、提升异构硬件上的编译性能，并大幅——

大幅降低领域特定编译器的开发成本。此外，它还有助于在现有编译器之间建立联系。

Polygeist [15]能够将 C 代码转换为 MLIR 代码。然而，由于其较新，它对随机生成的 C 代码的处理鲁棒性尚不完全可靠。此外，MLIR 能够表示超出

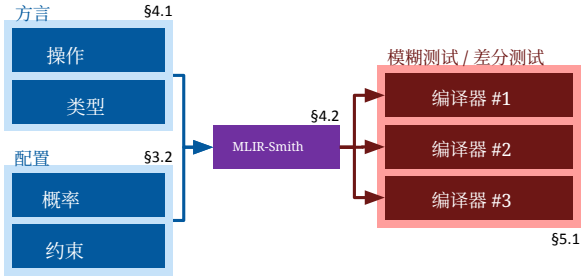


图 1：MLIR-Smith 概览。

C 编程语言指令集范围的操作。这带来了挑战，因为目前还没有专门用于生成随机 MLIR 程序的工具，而这对于全面测试和评估基于 MLIR 的编译器及其优化至关重要。

为应对这一挑战，我们推出了 MLIR-Smith，这是一款面向 MLIR 的新型随机程序生成器。MLIR-Smith 可生成随机 MLIR 程序，从而全面评估基于 MLIR 的编译器的优化能力，并确保编译后代码的正确性和有效性。

我们通过实现一个测试框架来展示 MLIR-Smith 的用途，该框架使用死代码消除（DCE）技术评估 MLIR [11]，LLVM [10],DaCe [3], 和 DCIR [2]的优化能力。DCE 是一种成熟的方法，可用于分析编译器的优化效率，并发现代码优化方面的遗漏机会 [23]。此外，我们还利用 MLIR-Smith 通过差分测试发现 MLIR、LLVM、DaCe 和 DCIR 中的缺陷。

本文的贡献如下：

- 我们设计并实现了 MLIR-Smith，这是一种专为 MLIR 设计、利用操作最小化接口的随机程序生成器。
- 我们解决了 MLIR 测试基础设施当前存在的局限，这些局限妨碍了有效的编译器测试。
- 我们通过一系列评估展示了 MLIR-Smith 的有效性，彰显了其提升编译器研究与测试领域的潜力。

2 背景

我们首先讨论本研究所依托的基础性方法，具体包括 MLIR [11] 和 DaCe [3] 框架、DCIR 流水线 [2], 以及 Csmith [24]。

2.1 MLIR

LLVM [10] 和 Java 虚拟机 [12] 是传统编译器系统的典型例子，它们依赖单一抽象层级。这些框架将程序从一种语言转换为中间表示（中间表示），并在其中完成所有必要的变换、改进和验证。MLIR [11] 旨在通过提供一个支持多个抽象层级共存的框架，简化多种中间表示的创建和互操作。

MLIR 中的中间表示称为方言。方言提供专用操作和数据类型，以及用于从较高层级的中间表示进行降级、并在同级中间表示之间转换的转换点。尽管 MLIR 自带多种标准方言，用户也可以自由创建独特的方言。大多数方言都提供向下一级的降级遍，从而减少编译器基础设施开发中的冗余代码。

前端和后端项目对 MLIR 的广泛采用，凸显了这项技术对编译器的重要性。例如，前端项目 Flang [14] 会将 FORTRAN 代码转换为 MLIR 方言。在后端，Katel 等人 [9] 最近进行了一项性能评估，将 MLIR 与 CUBLAS [17] 进行了比较，以证明 MLIR 能够生成性能媲美知名矩阵乘法库的 GPU 代码。

随着 MLIR 日益普及，对可靠测试工具的需求也大幅增加；这些工具对于确保方言实现正确至关重要。

2.2 DaCe

尽管有许多优化技术旨在改善数据移动，但其中许多技术专为特定硬件系统或单独的硬件配置而设计。与此同时，高级程序需要考虑数据移动，这往往会增加代码复杂度。

DaCe 编程框架通过提供有状态数据流多重图（SDFG）[3] 表示，为这一问题提供了解决方案。这种中间表示的主要目标是理解并改进数据流。DaCe 提供了多个前端，可将 Python、Octave 或 C 代码转换为 SDFG 中间表示。此外，它还为该中间表示提供变换 API，从而有效划分性能工程师与开发者各自的职责。DaCe 已在多个领域得到应用，包括天气和气候模型 [4, 6], 、量子输运模拟中的稀疏线性代数 [25], 、图分析 [3], , 以及深度学习中的全面神经网络优化 [21]。

DaCe 框架的优化方法始于 SDFG 简化遍，它会拓宽纯数据流区域并消除多余的内存分配，这与编译器的 -O1 标志类似。之后会采用自动启发式优化（-O2），有时性能工程师也会手动应用转换。SDFG 中间表示是一种控制流图，以状态机的形式表示，由数据流无环多重图组成。SDFG 区分了计算节点（称为 任务单元）中对数据的使用，以及数据容器和数据移动的使用（由数据流边表示）。因此，中间表示可以直接描述数据移动，甚至可以利用数据依赖来确定执行顺序。只有在无法推断数据流的情况下，才会使用显式控制流。

然而，控制流中心的中间表示中常实现的一些编译遍，例如循环不变量代码外提或通用公共子表达式消除，天生无法用 SDFG 表示。任务块被视为一个原子单元，因此无法检查其内容是否发生变化。此外，一些设计模式（如参数化深度递归或动态指针）可能很容易由其他中间表示处理，却无法在 SDFG 中简洁地表达。

为了实现两端的优化，Ben-Nun 等人 [2] 在 SDFG 与 MLIR 之间搭建了一座桥梁。

2.3 DCIR

在近期的一项进展中，Ben-Nun 等人 [2] 推出了一种新型编译器流水线——数据中心中间表示（DCIR）。DCIR 独特地结合了 MLIR [11] 和 DaCe [3], 的优化能力，从而发掘此前尚未探索的优化可能性。DCIR 通过引入一种新型方言——SDFG 方言来实现这一点，该方言与状态数据流图高度相似。此外，转换编译遍和翻译编译遍在 MLIR 与 DaCe 之间搭建起桥梁。

DCIR 展现出的良好结果，尤其是在特定场景下增强死代码消除（DCE）能力方面的表现，表明其在更广泛的情况下可能带来 substantial 的收益。值得注意的是，作者展示了一个涉及大型数组和嵌套循环的示例，DCIR 能够将其完全消除，而与之比较的编译器 GCC [7], Clang [13], DaCe 和 MLIR 均未能做到这一点。

2.4 Csmith

传统的编译器测试方法在测试用例的覆盖率和多样性方面往往有所欠缺。Csmith [24],是一种 C 随机程序生成器，它通过生成随机且格式良好的 C 程序来解决这一问题。这使得人们能够更全面地测试编译器优化和变换，并发现传统测试方法可能遗漏的编译器错误。Csmith 通过从一组预定义操作和结构中随机选择来生成程序，确保生成的程序语法正确且不存在未定义行为。它结合随机决策与安全检查来实现这一点，确保生成的程序不会包含可能导致非确定性执行的未定义行为。先前的研究 [24]已充分记录 Csmith 在发现编译器错误和评估编译器优化性能方面的效用。它能够生成多样化的测试用例，因此是编译器测试的宝贵工具。然而，Csmith 只能生成 C 程序，这限制了它在支持 C 的编译器和框架中的适用性。

虽然可以将 Csmith 与 Polygeist 等工具结合使用 [15] 来生成 MLIR [11] 代码，但这种方法存在局限性。Polygeist 已展现出翻译科学应用程序的能力，例如 Polybench 套件中的应用程序 [19]。然而，当需要翻译由 Csmith 生成的 C 程序时，Polygeist 的稳定性不足以胜任这一任务。此外，这种方法会将生成的 MLIR 程序的范围限制在 C 代码的范围之内。

为了解决这些局限，并支持对 MLIR 进行更全面的测试，我们提出开发 MLIR-Smith。这是一款受 Csmith 启发、专为 MLIR 设计的工具。

3 方法

本节将介绍 MLIR-Smith 背后的复杂设计决策，并概述 MLIR-Smith 的结构与设计权衡。

3.1 设计目标

MLIR 被设计为一种灵活且可扩展的编译器基础设施，允许纳入新的方言和转换。为 MLIR 框架精心设计的工具应充分利用这种可扩展性。因此，MLIR-Smith 的主要设计目标是成为一种高度可扩展的 MLIR 工具，让开发者能够轻松集成来自新方言的新操作。

MLIR-Smith 的另一项设计目标是满足用户的特定要求，因为用户可能有独特的需求，或倾向于专注测试部分操作。MLIR-Smith 提供了高度可配置性，使用户能够自定义关键方面，例如为操作分配的概率或嵌套块的深度。这种灵活性让用户可以根据自身独特的需求和偏好调整系统。

3.2 结构

MLIR-Smith 的结构旨在与 MLIR 环境无缝集成，并遵循既有的 MLIR 工具和惯例。与 mlir-opt 和 mlir-reduce 等其他 MLIR 工具采用的结构范式一致，MLIR-Smith 的构造方式也让人感到熟悉，便于与其他 MLIR 实用工具协同使用和开发。MLIR-Smith 有效利用了 MLIR 的核心能力，同时引入了新的实用函数以提升易用性。

MLIR-Smith 由两个不同的组件构成，各自在其功能中发挥着关键作用。MLIR-Smith 的核心负责初始化配置并设置初始程序结构，其中包括 MLIR 模块及其 main() 函数。除了这一核心组件，MLIR-Smith 还加入了一个辅助实用类，扩展内部 MLIR 操作构造器的功能，并增强 MLIR-Smith 的整体功能。例如，该实用类提供辅助函数，用于生成具有指定返回类型的基本块。此外，开发者还可以使用辅助函数，对特定类型的操作数进行采样和生成。

与 Csmith 不同，Csmith 已知所生成 C 操作的行为和要求，而 MLIR-Smith 必须自适应地从新的方言中选择并插入可能全新且未记载的操作。MLIR-Smith 并不了解这些操作的语义、约束和要求。因此，MLIR-Smith 将生成操作的责任交给相应方言的开发者，具体做法是将生成过程委托给相应的操作。通过这种方式，我们不仅确保生成的操作符合特定约束，也充分利用了框架固有的可扩展性。为支持注册新操作，MLIR-Smith 采用了一套稳健的机制，借助 MLIR 方言注册表，并通过新定义的特征识别操作。MLIR 中的方言注册表是一个集中式代码仓库，用于存储已注册方言及其相关操作的信息；特征则充当注解，提供有关操作特性的补充信息。

除特征外，MLIR-Smith 还实现了一种名为 GeneratableOpInterface 的接口。该接口为可生成的操作声明函数，从而建立标准化结构。该接口配有一个特征，可简化将操作添加到可生成集合的过程，同时增强搜索能力。为了将新开发的操作无缝集成到程序工作流中，MLIR-Smith 利用方言注册表搜索具有此特定特征的操作。借助基于特征的方法，MLIR-Smith 确保仅考虑适用于程序生成的操作，从而实现有针对性且高效的生成过程。

3.3 设计权衡

为在 MLIR-Smith 中平衡灵活性与效率，我们仔细考虑了多项设计权衡。以下段落将详细介绍这些设计权衡。

不保证终止。 尽管可以生成始终能够终止的随机程序，但 Yang 等人 [24]认为，这会过度限制生成器的表达能力。此外，Yang 等人 [24]指出，始终能够终止的测试无法发现那些错误地终止非终止程序的编译器错误。换言之，如果测试仅限于程序始终终止的情形，那么任何导致非终止程序错误终止的缺陷都不会被发现。因此，我们选择采用这种设计理念，不对程序终止作出保证。为应对可能出现的非终止程序，我们采用了超时机制。

没有真实值。 与 Csmith 类似，MLIR-Smith 所处的领域并不适用“真实值”这一概念。在程序生成和编译器测试的语境中，真实值是指用于对照生成的程序或编译器输出、以判定其正确性的明确标准。然而，由于 MLIR-Smith 生成的程序具有随机且多样的特性，为每个生成的程序确立通用的真实值并不切实际。因此，MLIR-Smith 的设计用途是在差分测试场景中比较不同编译器的输出。尽管存在这一局限，MLIR-Smith 仍可用于测试单个编译器或优化遍中的崩溃问题。

表 1：MLIR-Smith 生成的方言及其操作概述。

方言	已实现的操作
`func`	func.return , func.call
`scf`	scf.if、scf.for、scf.while、scf.condition 和 scf.yield
arith	已实现所有操作。
`math`	已实现所有操作。
内存引用	memref.alloc , 内存引用栈分配, memref.load , memref.store , 内存引用释放和内存引用复制

类型安全。 为确保数组安全性，MLIR-Smith 不允许生成动态大小，并在编译时强制使用静态维度。具体而言，MLIR-Smith 生成的内存引用类型不包含跨步访问模式或仿射映射，且维度最多为三个；同时采用静态维度大小，最大可达 100000。这一经过审慎考虑的设计决策使 MLIR-Smith 能够为每个维度生成处于允许范围内的索引，从而减少与数组边界相关的运行时错误。

4 实现

下面，我们将详细介绍 MLIR-Smith 的实现，包括操作的选择和构建过程。

4.1 支持的操作

由于我们的工作重点是评估 MLIR、LLVM、DaCe 和 DCIR 这四种编译器框架，因此我们专门实现了所有框架均支持的 MLIR 方言子集。这样，我们便能够生成 DCIR 支持的 MLIR 代码，并对这四种编译器框架进行差分测试。表 1 总结了已实现的操作。如表所示，我们实现了部分结构化控制流（*scf*）操作，例如 `scf.for`、`scf.if` 和 `scf.while`，以及算术（*arith*）和 math（*math*）方言中的所有指令。虽然 *arith* 方言实现了基本的整数和浮点运算，但 *math* 方言包含了简单算术运算之外的操作。最后，DCIR 支持内存引用（*memref*）方言的一个子集，该方言提供用于创建和操作多维内存引用的操作。生成的 *memref* 类型受第 §3.3 节所述约束的限制。

4.2 程序生成

由于 Csmith 已证明其在生成多样且具有挑战性的测试用例方面卓有成效，MLIR-Smith 紧密遵循 Csmith 用于生成 C 程序的核心原则。具体而言，随机程序生成过程采用自顶向下方法，从程序的 `main()` 函数 [24]开始。图 2展示了 `mlirSmithMain()` 函数的简化概览，该函数实现了程序生成过程的核心。

第一步会解析 MLIR-Smith 的命令行选项。用户可以指定要使用的配置文件和随机种子，同时还可以为生成的代码选择输出目录。MLIR-Smith 还提供了一个选项，用于转储当前配置。

```
LogicalResult mlir::mlirSmithMain(int argc, char **argv, DialectRegistry &registry) { // 解析命令行选项。 [...]// 加载配置。 [...]// 创建 OpBuilder。 GeneratorOpBuilder builder(&ctx, config); // 插入顶层模块。 ModuleOp module = ModuleOp::build(builder);
builder.setInsertionPointToStart(module.getBody()); // 插入 main 函数。 FunctionType funcType = builder.getFunctionType({}, {I32}); func::FuncOp func = func::FuncOp::build(builder, "main", funcType); // 生成 main 函数体。
builder.setInsertionPointToStart(func.addEntryBlock());
builder.generateBlock(/*ensureTerminator=*/true, /*requiredTypes=*/{I32}); // 输出结果。
module.print(); return success(); }
```

图 2：MLIR-Smith 中代码生成流程的简化概览。

接下来会加载程序生成所用的配置。如果未指定配置文件，MLIR-Smith 将使用默认配置值；为简洁起见，此处省略这些值。此外，还会创建一个 GeneratorOpBuilder 实用类的实例。mlirSmithMain() 使用该实例构建 MLIR 模块和程序的 main() 函数，以此搭建初始程序结构。随后，GeneratorOpBuilder 会生成一个基本块，并将其插入 main() 函数的开头。下面将详细介绍 GeneratorOpBuilder 类。

构建机制。 为了更好地控制操作构建流程，我们引入了 GeneratorOpBuilder 类。该类扩展了 MLIR 生态系统中的 OpBuilder 实用工具，旨在简化操作的构建过程。OpBuilder 在构建过程中有效地管理当前基本块和区域的上下文，并在 MLIR 函数或基本块中维护当前插入点。这使得在正确的位置添加操作变得十分简单。

GeneratorOpBuilder 在此基础上扩展了功能，提供了额外的实用函数，包括采样操作数和填充空基本块的能力。GeneratorOpBuilder 提供的一个关键函数是 generateBlock()，它会在基本块中随机生成操作。如有需要，generateBlock() 会确保存在终结符，并为该终结符提供必要的类型。

通过扩展 OpBuilder 类，GeneratorOpBuilder 可以重写某些函数，例如 create() 函数，以强制执行特定约束。这些约束包括限制嵌套区域的深度，或单个基本块中的操作数量。此外，GeneratorOpBuilder 还能确保插入点位于正确位置，并实现有限的回滚功能。此功能可撤销操作生成失败时所做的更改，从而加强对构建过程的控制。这种设计也为构建器的未来扩展提供了灵活的基础，例如添加更多约束或实用工具。

```
LogicalResult arith::AddIOp::generate(GeneratorOpBuilder &builder) {
  llvm::SmallVector <类型>        possibleTypes    =  getGeneratatableTypes(builder);
  当 (!possibleTypes.empty()){
    unsigned  idx  =  builder.均匀采样(possibleTypes.size()                    - 1);
    类型 resultType  =  possibleTypes [idx ];

    llvm::Optional <值>  lhs   =  builder.sampleValueOfType(resultType);
    llvm::Optional <值>  rhs   =  builder.sampleValueOfType(resultType);

    if  (!lhs.has_value()       ||  !rhs.has_value())       {
      Type  *it  =  llvm::find(可能的类型,              resultType);
      如果(it  !=  possibleTypes.end())
        possibleTypes.erase(it);
      继续;
    }

    操作状态 state(builder.getUnknownLoc(),
                   arith::AddIOp::getOperationName());
    arith::AddIOp::build(builder , state, lhs.value(), rhs.value());
    如果(builder.create(state)  !=  nullptr)
      返回 success();

    类型 *it   =  llvm::find(possibleTypes , resultType);
    如果 (it !=  possibleTypes.end())
      possibleTypes.erase(it);
  }

  返回failure();
}
```

图3：arith.addi生成函数概览。

操作生成。 若要让 MLIR-Smith 将某个操作识别为可生成操作，必须将 GeneratableOpInterface 特征添加到该操作的特征列表中。接下来，每个操作都必须定义两个独立的函数，即 generate() 和 getGeneratableTypes()。generate() 负责构建相应操作的过程，而 getGeneratableTypes() 则返回该操作的生成函数能够生成的类型列表。GeneratorOpBuilder 中实现的多个辅助函数都会用到 getGeneratableTypes()。例如，generateValueOfType() 会使用 getGeneratableTypes() 函数，查找能够生成所需返回类型的操作。另一个使用场景出现在 sampleTypes() 的实现中。它负责从当前基本块位置可用类型的几何分布中进行采样。具体而言，它会利用 getGeneratableTypes() 从可用操作中收集所有唯一的可生成类型。不过，getGeneratableTypes() 的用途不止于 GeneratorOpBuilder；它也可以在 generate() 函数中使用。

图 3 展示了生成 `arith.addi` 操作的代码。如图所示，大部分功能都封装在 GeneratorOpBuilder。此抽象简化了开发者的工作流程，使他们能够轻松地将新方言集成到 MLIR-Smith 中进行测试。虽然 MLIR 支持任意位宽的整数，但我们选择仅支持位宽为 1、8、16、32和64 位的整数。这一决定基于这样的观察：这些是最常用的整数位宽。同样，对于浮点类型，我们选择支持位宽为 16、32和64 位的类型。

在 arith.addi生成函数的每次循环迭代中，都会从可生成类型列表中采样一个结果类型，然后使用 GeneratorOpBuilder 的实用函数 sampleValueOfType() 采样两个相同类型的操作数。需要注意的是，有些操作要求操作数具有特定的值和类型，开发者有责任根据各自的特定要求确保采样到正确的操作数。

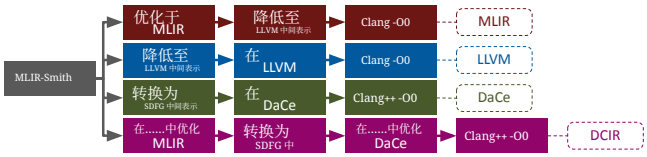


图4：评估中使用的流水线概览。

对于 arith.addi，操作数的类型必须与结果类型一致，且对操作数没有特殊要求。如果操作构建成功，就会将其插入当前位置。但是，如果找不到指定类型的操作数，或者因违反限制而导致操作创建失败，构建过程就会失败。在这种情况下，我们会从可生成类型集合中移除该结果类型，并在新一轮循环迭代中重试。如果所有循环迭代都未能成功构建操作，就会返回失败信号。GeneratorOpBuilder 会在遇到失败时转而处理下一个可用操作。

5 评估

本节展示 MLIR-Smith 作为一种工具的实际效果，它能够发现 MLIR、LLVM、DaCe 和 DCIR 编译器流水线中的缺陷和遗漏的优化，从而证明其在提升这些系统的可靠性和能力方面的价值。

5.1 实验设置

我们的评估使用 MLIR-Smith，并采用表 2 中详述的配置来生成 MLIR 程序。为提高缺陷发现过程的有效性，我们将嵌套区域的深度限制为 4，并将一个基本块中的操作数量限制为 50，这是基于这样一种假设：缺陷往往会在规模相对较小的示例中显现。此外，我们为 `scf.if`、`scf.for` 和 `scf.while` 操作设置了较高的概率，因为预计这些操作更容易暴露缺陷和遗漏的优化。最后，我们停用了 DCIR 不支持的部分 `arith` 和 `math` 操作，以免不支持的操作错误淹没评估结果并可能掩盖重要发现。为简洁起见，此处不列出具体的 `arith` 和`math`操作。

生成的 MLIR 代码会经过四条不同的流水线处理，如图 4 所示。我们会监测编译错误和运行时差异，包括超时、打印输出和退出代码。为了单独评估各框架的优化能力，我们会有策略地在流水线的某些阶段禁用优化（例如使用 `-O0` 标志）。这些流水线的具体信息将在下文中详细介绍。

MLIR 流水线。在 MLIR 流水线中，我们使用 mlir-opt[1] 执行死代码消除（DCE）、公共子表达式消除（CSE）、循环不变代码外提（LICM）、函数内联和规范化。随后，我们将其降低为 LLVM 方言，并使用 mlir-translate[1]生成 LLVM 中间表示。然后使用标志 -O0 -fPIC -march=native 通过 clang[1] 编译代码。

[1]提交：6f29d1adf29820daae9ea7a01ae2588b67735b9e

表2：在对 MLIR、LLVM、DaCe 和 DCIR 编译器流水线进行比较评估时，MLIR-Smith 的配置参数。

参数	值	描述
区域深度限制	4	嵌套区域的最大深度。
块长度	50	一个基本块内操作的最大数量。
默认概率	1	操作的默认概率值。
scf.if	10	scf.if 操作的概率值。
scf.for	10	scf.for 操作的概率值。
scf.while	10	scf.while 操作的概率值。

LLVM 流水线。在 LLVM 流水线中，我们跳过 MLIR 优化，直接将其降低为 LLVM 方言，使用 `mlir-opt`[2] 生成 LLVM 中间表示，使用 `mlir-translate`[2]。然后使用 `opt`[2]，通过 `-O3` 标志进行优化，并使用 `clang`[2] 通过以下标志进行编译：`-O0 -fPIC -march=native`。

DaCe 流水线。在 DaCe 流水线中，我们使用 sdfg-opt[3] 将 MLIR 代码转换为 SDFG 方言，并使用 sdfg-translate[3] 生成 SDFG 中间表示。随后，使用 DaCe 的自动优化器[4] 优化代码，并使用 Clang++-11（版本 11.1.0）和 -O0标志进行编译。由于在 DaCe 中使用较新版本时遇到了兼容性问题，我们选择了 Clang++-11。值得注意的是，Clang 仅用于编译，不用于优化，因此不会影响结果。通过这种方法，我们可以直接评估 DaCe 框架本身的性能和有效性。

DCIR 流水线。在 DCIR 流水线中，我们首先使用 mlir-opt[2]，执行与 MLIR 流水线中相同的优化，然后使用 sdfg-opt[3] 将其转换为 SDFG 方言，并使用 sdfg-translate[3] 生成 SDFG 中间表示。接着，我们使用 DaCe 的自动优化器[4] 进行优化，并使用 Clang++-11（版本 11.1.0）和 -O0标志编译优化后的 SDFG。选择 Clang++-11 的原因与 DaCe 流水线部分所述一致。

5.2 发现的缺陷与遗漏的优化

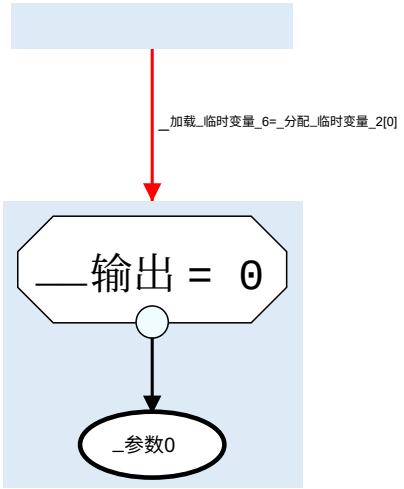
正如 §5.1所述，在整个评估过程中，我们在不同的编译器流水线中遇到了各种缺陷。本小节精选并介绍了其中一些最值得关注的缺陷。我们将全面探讨这些缺陷的特征、导致其被发现的具体情况，以及它们对各自编译器流水线造成的影响。

DaCe 错误。我们的评估过程发现了一个 DaCe 中的死代码消除（DCE）错误，如图 5a 所示的精简 MLIR 程序所展示的那样。尽管成功消除了大多数操作，DaCe 却未能识别出底部状态中 `_load_tmp_6` 仍未被使用，如图 5b 所示。这一疏忽妨碍了边赋值的消除，进而阻止了数组分配的移除。无论多次调用，还是采用不同的优化器配置（例如重新排序简化和自动优化遍），该错误始终存在。有趣的是，移除 `arith.maxsi`

[2]提交：6f29d1adf29820daae9ea7a01ae2588b67735b9e[3]
提交：8734d2c10ecb9078a81ff3ae0b64a774b098aca5[4]提交：4fcb0b5ee90384829ab7a76bde0a07a7a8cbcb4b

```
模块 {
  func。func @main() ->i32 {
    %alloc  = 内存引用.alloc() : 内存引用<0xi32>
    %c0 = arith.常量 0 : index
    %0 = 内存引用.load %alloc[%c0] : 内存引用<0xi32>
    %c0_i32 = arith.常量 0 :          i32
    %1 = arith.maxsi %0, %c0_i32 : i32
    返回     %c0_i32   : i32
  }
}
```

(a) 揭示 DaCe 死代码消除错误的精简 MLIR 程序。



(b) DaCe 优化后的 SDFG。

图 5：DaCe 中死代码消除缺陷的示意图。

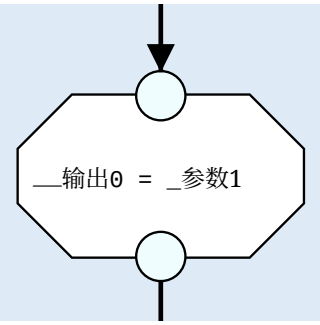
运算后，编译和执行结果均正确。我们的调查发现，sdfg-translate 生成的 SDFG 除了 arith.maxsi 运算外完全相同，这表明错误出在 DaCe 优化器中。我们向 DaCe 开发团队报告了这一错误后，他们成功复现了该问题，并随后确认了问题确实存在。

DCIR 缺陷。在评估过程中，我们发现了 DCIR 中的翻译缺陷，具体涉及针对布尔值的 算术符号扩展 操作。图 5a 展示了导致我们发现这一 DCIR 翻译缺陷的精简版 MLIR 程序。如图 5b所示，DCIR 将 算术符号扩展 操作翻译为单纯地将输入复制到输出。对于输入布尔值为 假的情况，这种翻译是正确的；但对于经过有符号整数扩展的 真 输入值，输出整数的每一位都应为 1。然而，DCIR 错误地将所有有符号整数扩展都视为无符号扩展，导致结果不正确。进一步调查发现，DCIR 对负整数的 算术符号扩展 翻译普遍存在缺陷。我们尝试进一步精简输入的 MLIR 程序，但更简单的版本会导致 `sdfg-opt` 在使用 `sdfg-translate`进行翻译之前执行常量折叠和传播，从而掩盖了该缺陷。我们报告了这一缺陷，DCIR 开发团队随后确认了该问题。

MLIR 遗漏的优化。如 §5.1 所述，我们的评估过程发现了 MLIR 流水线中的一项遗漏优化。当通过 MLIR 流水线处理图 7a所示的 MLIR 程序时，这项被忽略的优化便显现出来。执行生成的程序会导致段错误，我们将其归因于一个庞大且未被消除的

```
模块{
  func。func @main() -> i32 {
    %c0_i8 = arith.常量0: i8
    %0 = arith.extsi %c0_i8 : i8 至 i64
    %1 = arith.乘法%0, %0 : i64
    %2 = arith.cmpi无符号小于或等于, %0, %1 : i64
    %3 = arith.extsi %2 : i1 to i32
    %4 = arith.addi %3, %3 : i32
    return %4 : i32
  }
}
```

(a) 揭示 DCIR 中错误的精简 MLIR 程序。



(b) 生成的 SDFG 中经过转换的 arith.extsi 任务单元。

图 6：DCIR 中转换错误的示意图。

```
模块{
  func。func @main() -> i32 {
    %c0 = arith.常量0: 索引
    %alloc = 内存引用.alloc() : memref <100000x100000xi64>
    %0 = 内存引用.load %alloc[%c0,%c0] : 内存引用 <100000x100000xi64>
    内存引用.store %0, %alloc[%c0,%c0] : 内存引用 <100000x100000xi64>
    内存引用.dealloc %alloc : 内存引用 <100000x100000xi64>
    %c2_i32 = arith.常量2: i32
    返回 %c2_i32: i32
  }
}
```

(a) 展示 MLIR 中遗漏优化的精简 MLIR 程序。

```
主函数:
  压栈指令 %rax
  异或指令 %eax, %eax
  移动指令 %eax, %edi
  64 位立即数移动指令 $80000000000, %rax
  加法指令 %rax,%rdi
  调用指令 内存分配函数@PLT
  64 位移动指令 %rax,%rdi
  64 位移动指令 (%rdi), %rax
  movq %rax, (%rdi)
  调用指令 free@PLT
  移动指令 $2, %eax
  popq %rcx
  retq
```

(b) MLIR 流水线生成的汇编代码。

图 7：MLIR 中未能实现的优化示意图。

输入代码中未使用的内存分配。该内存分配很可能会失败，随后在读取操作期间触发空指针解引用。生成的汇编代码也进一步印证了这一判断（参见图 7b），其中可以清楚地看到，所有内存操作都仍然存在。

值得注意的是，如果从输入代码中移除加载或存储操作，MLIR 流水线便能够消除内存分配。此外，在其他操作中使用读取的值，或存储来自另一操作的值，也会导致内存分配被消除。这一发现表明，尽管单个操作本身并无问题且会被消除，但加载与存储操作之间的依赖关系似乎妨碍了死代码消除，从而导致前述段错误。值得一提的是，所有其他流水线都能够识别未使用的内存，并进而消除内存分配。

总之，对 MLIR-Smith 的评估凸显了其在多个编译器流水线中识别缺陷和遗漏的优化方面的实用性。这些发现进一步表明，全面测试对于提升此类系统的可靠性和能力至关重要。

6 相关工作

本节将探讨在目标或技术上与我们的方法相似的其他方法。

历史概述。 使用随机技术进行编译器测试的历史已有半个多世纪。Boujarwah 和 Saleh [5] 于1997年对这一领域进行了全面探讨。Richard L. Sauder [22] 于1962年率先开展了这一领域的研究，当时他将随机变量应用于程序数据段，以评估 COBOL 编译器的性能。他的工作围绕开发一种“测试数据生成器”展开，该生成器与 COBOL 编译器协同工作。该系统不仅用于生成符合 COBOL 程序数据部中所给描述的数据，还用于建立必要的数据关系，以测试 COBOL 程序的逻辑。

1970年，在 Sauder 的研究基础上，K.V. Hanford [8] 开发了一套测试用例自动生成系统。该系统旨在测试程序产品在接收新的测试用例时能否保持可靠，不出现任何错误。系统的输入是用形式化表示法编写的语法定义。这项任务颇具挑战性，但能深入了解语言的结构，并揭示现有文档中的模糊之处或歧义。Hanford 的系统包含 ECMA Algol、FORTRAN IV 以及 PL/I 的一个主要子集的定义。

1972年，Purdom [20]提出了一种语法引导方法，用于为解析器生成测试句子。在其论文中，Purdom 概述了一种技术，该技术采用高效算法，能够根据上下文无关文法生成一组简洁的短句。此方法确保文法中的每条产生式规则至少使用一次。这些句子有助于测试语法分析程序和调试文法。这些句子还用于测试一些自动生成的简单 LR(1)解析器，展现了该方法在测试 LR(1)解析器方面的有效性。

随机生成与自动化测试。Barany [1] 提出了一种新颖的随机程序生成方法，并将其称为“活性驱动”。该方法自底向上构造随机程序，同时借助结构化数据流分析进行引导，以确保生成器绝不会生成死代码。Barany 的方法以插件形式实现于 Frama-C 框架中。与标准的随机 C 程序生成器 Csmith 相比，Barany 的工具生成的程序能够编译成更多机器码，且指令组合更加复杂。与我们的方法不同，这种方法在避免生成死代码方面尤为有效。

Pałka 等人 [18] 提出了一种用于编译器随机测试的方法，以随机生成的程序作为输入，并比较程序在优化前后的行为。他们专注于生成针对列表的类型化函数，并使用成熟的生产级 Haskell 编译器 Glasgow Haskell 编译器对这些函数进行编译。在约 20,000 次测试后，他们触发了一次优化器故障，并自动将其简化为仅包含少量结构的程序。他们的方法独具特色，在随机生成过程中考虑了语法、作用域规则和类型检查，同时实现了良好的测试数据分布。

最后，Yang 等人 [24] 推出了 Csmith，这是一款能够生成涵盖 C 语言很大一部分特性的程序、同时避开未定义行为和未指明行为的工具。他们结合静态分析和动态检查，以限制未定义行为或未指明行为。借助 Csmith，他们在 GCC [7] 和 LLVM [10]中发现了 325 多个缺陷。他们还提供了一套全面的定性和定量结果，介绍在开源 C 编译器中发现的缺陷。

我们的工作采用了根本不同的方法，因此有别于前面讨论的各种方法。我们并非专注于某种具有预定义操作集合的特定语言，而是设计了一个用于代码生成的可扩展框架。该框架主要旨在为新语言的开发者提供便利，支持模糊测试和差分测试，并特别着眼于 MLIR 方言。

融入高级技术。我们的框架旨在轻松融入高级技术。例如，Theodoridis 等人 [23] 提出的优化标记可以无缝集成到我们的系统中。Theodoridis 等人的贡献尤其重要，因为他们创新地运用了优化标记，从死代码消除的角度帮助识别被忽视的优化。他们的方法在 GCC [7]和 LLVM [10], 62 中发现了 84 个缺陷，其中一些已得到确认或修复。

总而言之，我们的工作标志着对传统方法的重大转变，为编译器测试提供了更灵活、更动态的解决方案。借助框架的可扩展性，我们致力于让开发者更轻松、高效地开展编译器测试和探索新技术。

7 结论

本文介绍了 MLIR-Smith，这是一款用于在 MLIR 框架中生成随机程序的新型工具。本文通过一系列评估展示了该工具的有效性，彰显了其在推动编译器研究与测试领域发展方面的潜力。该工具具有高度的可扩展性和可配置性，使开发者能够轻松集成来自新方言的新操作，并自定义工具的关键方面，以满足特定需求。未来，可通过将更多 MLIR 核心方言纳入 MLIR-Smith 来拓展本工作的范围。此外，操作应能够向 MLIR-Smith 添加各自独有的配置。最后，支持复合类型和无界类型集合将带来显著优势。

A 制品附录

A.1 摘要

本文相关的制品包括 MLIR-Smith 工具和实验环境设置，详情见§5.1。此外，还提供了用于简化实验工作流的脚本，以及§5.1中提及的配置文件。该实验设计为可在标准硬件上运行；为方便起见，我们提供了一个 Docker 容器，以简化设置过程。请注意，所使用的所有软件均可公开获取，并已包含在提供的容器中。

A.2 制品检查清单（元信息）

- **编译：** Clang（版本 17.0.0）、Clang-11（版本 11.1.0）、DaCe（git 修订版本 4fcb0b5）[5]
- **变换：** MLIR 优化器和转换器（git 修订版本 f4dfabd）[6]、LLVM 优化器和编译器（git 修订版本 00a1258）[6]、DCIR 优化器和转换器（git 修订版本 8734 2 [7] d c）二**进制文件：** MLIR-Smith 程序生成器（git 修订版本 f 4 [6] dfabd）**运行时环境：** 在 Ubuntu . 和 . 上进行测试。具体取决于 Docker 安装，[20 04] [22 04] 可能需要根用户权限。
- **指标：** 编译错误、运行时差异、遗漏的死代码消除优化
- **输出：** 生成的代码以及按子文件夹分类的所有指标 • **实验：** 其中包含用于执行实验的多个脚本
- **所需磁盘空间（约）：** 拉取 Docker 容器需要 676 MiB（解压后为 2.1 GiB）
- **完成实验所需时间（约）：** 5 分钟
- **是否公开可用？** 是 • **代码许可证（如果公开可用）：** BSD 三条款许可证
- **是否使用工作流框架？** 否
- **是否已归档（请提供数字对象标识符）？** 否

A.3 描述

A.3.1 如何访问。所有必要文件以及运行这些文件的说明均可在 GitHub 上找到（https://github.com/ Berke-Ates/AST）。

A.3.2 软件依赖项。要构建并运行 Docker 容器，必须在系统上安装 Docker，并确保 Docker 守护进程处于运行状态。

A.4 安装

代码仓库中的 README.md 文件详细介绍了手动安装和 Docker 容器化安装的说明。为方便使用，推荐采用 Docker 容器化安装，可通过以下命令启动：

```
docker pull berkeates/mlir-smith:latest
```

A.5 实验工作流程

在我们的实验中，我们首先使用 MLIR-Smith 生成一个随机的 MLIR 程序。然后按照 §5.1中概述的方式，通过各条流水线处理该程序。在此过程中，我们会监控是否出现编译器错误。如果检测到错误，就会停止执行

[5] https://github.com/Berke-Ates/dace/tree/4 5 [6] 0fcbb
https://github.com/Berke-Ates/llvm-project/tree/f4dfabd [7]
https://github.com/spcl/mlir-dace/tree/8734d2c

并将 MLIR 输入文件及所有相关日志移至名为 comp_err 的单独文件夹。如果所有编译器都成功处理了该程序，我们便执行生成的二进制文件，并比较打印输出和退出代码。如果存在差异，就会再次停止执行，并将所有相关文件移至名为 exe_diff 的文件夹。如果所有编译器流水线的可观察行为一致，我们便检查汇编代码中的外部函数调用，以找出遗漏的死代码优化机会。如果发现至少有一个编译器消除了更多外部函数调用，就将这些文件移至名为 flag_diff 的文件夹。如果所有检查均顺利通过，表明未检测到任何差异，则将这些文件放入常规文件夹。

所有必需的编译器、转换器和代码生成器都已包含在 Docker 容器中，并在其中完成构建。可以使用以下命令启动此容器：

docker run -it berkeates/mlir-smith。

A.6 评估与预期结果

该`diff_test.sh` 脚本位于`scripts` 文件夹中，用于调用 MLIR-Smith 随机生成 MLIR 程序，并通过 §5.1中详述的四条流水线对其进行处理。此脚本需要提供 MLIR-Smith 的路径、输出目录以及可选的配置文件。脚本的执行示例如下：`./scripts/diff_test.sh ./bin/mlir-smith out diff_test.config`。处理结果可在新创建的`./out` 文件夹中查看。

A.7 实验自定义

可以通过编辑主目录中名为 diff_test.config 的文件来修改配置，也可以向 diff_test.sh 脚本提供另一个配置文件。如果要检查特定的输入文件，可以执行 diff_test_file.sh 脚本来进行单次差分测试；该脚本位于 scripts 文件夹中。

参考文献

[1] Gergö Barany。2017。活跃性驱动的随机程序生成。arXiv: 1709 04421 [ ] 10 48550 1709 04421 。cs.PLdoi:./arXiv..Tal Ben-Nun、[2] Berke Ates、Alexandru Calotoiu 和 Torsten Hoefler。2023。连接以控制为中心和以数据为中心的优化。载于 第*21*届*ACM/IEEE* 国际代码生成与优化研讨会论文集(加拿大魁北克省蒙特利尔)*(CGO 2023)*。计算机协会，纽约州纽约市，173 185 10 1145 3579990 3580018 – 。doi:./.Tal Ben-Nun、Johannes de [3] Fine Licht、Alexandros Nikolaos Ziogas、Timo Schneider 和 Torsten Hoefler。2019。有状态数据流多重图：一种面向异构架构性能可移植性的数据中心模型。载于高性能计算、网络、存储与分析国际会议论文集（*SC '19*）。计算机协会，纽约州纽约市。doi: 10 1145 3295500 3356173 ./.Tal Ben-Nun、Linus Groner、Florian [4] Deconinck、Tobias Wicky、Eddie Davis、Johann Dahm、Oliver Elbert、Rhea George、Jeremy McGibbon、Lukas Trümper、Elynn Wu、Oliver Fuhrer、Thomas Schulthess 和 Torsten Hoefler。2022。使用 Python 对天气与气候建模进行高效性能工程。载于高性能计算、网络、存储与分析国际会议论文集（*SC'22* 10 48550 2205 04148 ）。IEEE出版社。doi:./arXiv..A.S. Boujarwah 和 K. Saleh。. 编译器测试用例生成方法：综述与评估。[5] 1997 *Information and Software Technology* 39, 9 (1997), 617 625 10 1016 0950 5849 97 00017 7 – 。doi:./S-()-Johannes de Fine Licht、[6] Andreas Kuster、Tiziano De Matteis、Tal Ben-Nun、Dominic Hofer 和 Torsten Hoefler。2021。StencilFlow：将大型模板程序映射到分布式空间计算系统。载于论文集 2021年IEEE/ACM代码生成与优化国际研讨会（虚拟会议，大韩民国）（CGO '21）。IEEE出版社，315–326。doi:10.1109/CGO51591.2021.9370315[7] 自由软件基金会。2023。GNU编译器套件（GCC）。https://gcc. gnu.org/。版本11.1。[8] K. V. Hanford。1970。测试用例自动生成。IBM系统期刊 9, 4（12月 1970 242 257 10 1147 94 0242 ），– 。doi: . /sj. . Navdeep Katel、Vivek [9] Khandelwal 和 Uday Bondhugula。2021。使用 MLIR 为矩阵乘法生成高性能 GPU 代码：一些初步成果。arXiv: 2108 13191 [ ] 10 48550 2108 13191 . cs.DC doi: . / [10] 2004 arXiv. . Chris Lattner 和 Vikram Adve。. LLVM：用于终身程序分析与转换的编译框架。载于 CGO。美国加利福尼亚州圣何塞，75–88。doi:10.1109/CGO.2004.1281665 [11] Chris Lattner、Mehdi Amini、Uday Bondhugula、Albert Cohen、Andy Davis、Jacques Pienaar、River Riddle、Tatiana Shpeisman、Nicolas Vasilache 和 Oleksandr Zinenko。2021。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代码生成与优化国际研讨会（CGO）。2 14 10 1109 51591 2021 9370308 – 。doi: . /CGO . . [12] 1999 Tim Lindholm 和 Frank Yellin。. Java虚拟机规范（第2版）。Addison-Wesley Longman Publishing Co., Inc.，美国。[13] LLVM项目。 [n. d.]。Clang：LLVM 的 C 语言家族前端。[14] LLVM项目。2022。Flang：FORTRAN 前端。https://github.com/llvm/llvm-project/tree/main/flang。[15] William S. Moses、Lorenzo Chelini、Ruizhe Zhao 和 Oleksandr Zinenko。2021。Polygeist：将 C 提升为多面体MLIR。载于ACM并行体系结构与编译技术国际会议论文集（虚拟会议）（PACT '21）。计算机协会，纽约州纽约市，12页。doi:10.1109/PACT52795.2021.00011[16] Steven S Muchnick。1997。高级编译器设计与实现。Morgan Kaufmann。[17] NVIDIA公司。2022。cuBLAS：NVIDIA GPU 上的基础线性代数。https://developer.nvidia.com/cublas [18] Michał H. Pałka、Koen Claessen、Alejandro Russo 和 John Hughes。2011。通过生成随机Lambda项测试优化编译器。载于第6届软件测试自动化国际研讨会论文集（美国夏威夷州檀香山威基基）（AST '11）。计算机协会，纽约州纽约市，91–97。doi:10.1145/1982595.1982615[19] L. N. Pouchet。2016。PolyBench：多面体基准测试套件。https: // sourceforge.net/projects/polybench [20] Paul Purdom。1972。用于测试解析器的句子生成器。BIT 12, 3（9月 1972 366 375 10 1007 01932308 ），– 。doi: . / [21] BF Oliver Rausch、Tal Ben-Nun、Nikoli Dryden、Andrei Ivanov、Shigang Li 和 Torsten Hoefler。2022。机器学习的数据中心优化框架。载于第36届ACM国际超算会议论文集（虚拟会议）（ICS '22）。计算机协会，纽约州纽约市，文章 36, 13 10 1145 3524059 3532364 页。doi: . / . [22] 1962 Richard L. Sauder。. COBOL 通用测试数据生成器。载于1962年5月1日至3日春季联合计算机会议论文集（美国加利福尼亚州旧金山）（AIEE-IRE '62（春季））。计算机协会，纽约州纽约市， 317 323 10 1145 1460833 1460869 – 。doi: . / . [23] Theodoros Theodoridis、Manuel Rigger 和 Zhendong Su。2022。从死代码消除的视角发现遗漏的优化。载于第27届ACM国际编程语言与操作系统体系结构支持大会论文集（瑞士洛桑）（ASP-LOS '22）。计算机协会，纽约州纽约市，697–709。doi:10.1145/3503222.3507764[24] Xuejun Yang、Yang Chen、Eric Eide 和 John Regehr。2011。发现并理解 C 编译器中的缺陷。载于第32届ACM SIGPLAN编程语言设计与实现大会论文集（美国加利福尼亚州圣何塞）（PLDI '11）。计算机协会，纽约州纽约市， 283 294 10 1145 1993498 1993532 – 。doi: . / . Alexandros [25] Nikolaos Ziogas、Tal Ben-Nun、Guillermo Indalecio Fernández、Timo Schneider、Mathieu Luisier 和 Torsten Hoefler。2019。极大规模第一性原理耗散量子输运模拟的数据中心方法。载于高性能计算、网络、存储与分析国际会议论文集（美国科罗拉多州丹佛）（SC '19）。计算机协会，纽约州纽约市，文章1，13页。doi:10.1145/3295500.3357156