Java 流分析与优化

Eduardo Rosales[a]、Matteo Basso[a]、Andrea Rosà[a]和 Walter Binder[a]
a 意大利瑞士大学，信息学院，瑞士卢加诺

摘要
Java *Stream API* 在 Java 8 中引入，以便以声明式方式表达数据处理逻辑，通常是类似 map-reduce 的集合和数据集数据转换。Stream API 引入了两个关键抽象：流，即数据源中可用的元素序列；以及流流水线，其中包含一系列操作（例如 映射、过滤、归约），这些操作会在执行时应用于流中的元素。Java 流在 Java 开发者中日益流行，因为它们兼具函数式编程的简洁性，并使数据处理的并行化更加容易。

尽管流有诸多优势，但与依赖命令式代码的数据处理方式相比，流可能引入显著的开销，主要原因是额外的对象分配与回收，以及虚方法调用的使用。因此，开发者需要能够研究流运行时行为的方法，以减少这类抽象开销并优化流处理。遗憾的是，目前缺少能够动态分析流的专用工具，无法专门帮助开发者定位导致应用程序性能下降的问题。

本文探讨流的性能分析与优化。我们提出了一种新颖的性能分析技术，用于以经过的参考周期数衡量流执行的计算，并借此定位对应用程序性能影响显著的问题流。准确率对此至关重要，但插入的插桩代码会导致额外周期的执行，其中一部分会被计入分析结果。为缓解这一问题，我们估算并补偿插入的插桩代码所导致的额外周期。

我们在 StreamProf 中实现了我们的方法。据我们所知，它是首个专为 Java 虚拟机（JVM）设计的流性能分析器。借助 StreamProf，我们发现周期分析能够有效检测问题流，优化这些流可带来显著的性能提升。我们还发现，对支持并行流处理的任务进行准确的性能分析，可以根据线程级别的流相关周期分布诊断负载不均衡。

我们对来自三个不同公开来源的顺序和并行流工作负载进行了评估。评估结果表明，我们的性能分析技术效率高，且能够生成准确的分析结果。此外，我们通过指导对 Renaissance 中两个工作负载进行流相关优化，展示了分析结果的实用价值。我们的优化只需修改少量代码行，即可实现最高达 5×倍的加速比。

近年来的研究对 Java 流进行了广泛研究，重点关注开发者如何使用流以及如何对其进行优化。目前，流优化方法主要依赖静态分析技术，但这些方法忽略了运行时信息，在检测 Java 应用程序执行的所有流方面存在重大局限，或不适用于并行流的分析。了解顺序流和并行流的动态行为及其对应用程序性能的影响，对于帮助用户在使用流时做出更好的决策至关重要。

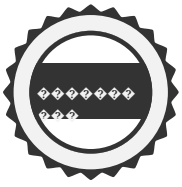
ACM CCS 2012

- **通用与参考 → 度量指标**；**软件及其工程 → 软件性能**；
-

关键词 Java 流、流处理、动态程序分析、性能分析、可操作的性能分析器、精确性能分析、Java 虚拟机

《编程的艺术、科学与工程》

提交日期 2022年12月30日**发表日期** 2023年2月15日**DOI**
10.22152/programming-journal.org/2023/7/10





© Eduardo Rosales、Matteo Basso、Andrea Rosà 和 Walter Binder 本作品采用“CC BY 4.0”许可协议。在《编程的艺术、科学与工程》第7卷第3期，2023年，第10篇文章；39页。

1 简介

自 Java 8 起，*Stream API* [1] 便支持以声明式风格进行数据处理，充分利用函数式编程的简洁性 [2]。此 API 引入了两个关键抽象：流，即数据元素序列的视图；以及流流水线，即由多个阶段的聚合操作（例如 映射、过滤、归约）构成的结构。流中的元素可以来自多种数据源，包括集合、数组、文件、生成器和字符串。执行流时，其流水线中的操作会应用于流中的数据元素 [3]。

如今，流正受到面向 Java 虚拟机 (JVM) [4–6] 的开发者的关注， [7] 因为它们支持多种*MapReduce* [7] 数据转换 [3]。通过采用兼顾软件可扩展性和可维护性的函数式编程风格，流可以用于改进软件设计。通过调用单个方法，流还可以并行执行，从而有望在无需编写任何并行代码的情况下实现加速比 [1]。另一方面，与使用命令式代码进行数据处理相比，流可能会遭受性能损耗，这主要是由于额外的对象分配与回收，以及大量虚方法调用的使用 [8, 9]。

为了减轻与流相关的抽象开销并优化流，开发者需要研究顺序流和并行流执行的运行时行为。据我们所知，已有六项研究致力于帮助开发者优化流。Khatchadourian 等人 [10, 11] 提出了一款 Eclipse [12]插件，帮助开发者更好地编写流代码。Møller 等人 [9] 在字节码级别将顺序流转换为更高效的命令式代码。Basso 等人 [13]使用字节码级转换，专门消除并行流带来的开销。然而，上述方法主要依赖静态分析，忽略了对于定位与流相关的性能问题至关重要的动态信息。Ishizaki 等人 [14]和 Hayashi 等人 [15] 利用 GPU 加速流处理。然而，这两种方法都只能应用于 Stream API 的单个方法，并且需要专有编译器，从而限制了其适用性。

尽管有多种面向 JVM 的分析器能够执行多种分析并针对许多度量指标 [16–21], 但目前还没有分析器能够专门捕获 Java 应用程序所执行的所有顺序流和并行流的运行时行为。总体而言，缺乏能够动态分析流的专用工具，而这类工具可以帮助开发者定位对应用程序性能有显著影响的流，并提供可操作的分析结果[1]，指导流相关优化。我们的目标是填补这一空白，通过处理流的动态分析，帮助开发者定位与流相关的性能缺陷。

贡献。 我们提出了一种新颖的技术，用于分析应用程序在 JVM 上执行的所有流（第 2 节）。我们以参考周期来衡量流所执行的计算，其定义为在标称处理器频率下（不考虑频率缩放）经过的时钟周期，

[1]Mytkowicz等人 [22] 将性能剖析定义为可操作的：如果对其采取行动能够产生预期结果，例如，针对性能剖析所指出的代码片段采取行动可以改善性能。

该时钟周期在一次操作期间收集。分析流处理所耗费的参考周期（以下简称周期）具有多种用途。首先，它可以识别开销较高的流执行，而对这些执行进行优化可能会带来加速比。其次，它可以帮助开发者更好地评估流并行化的潜在收益（判断顺序流计算是否具有足够高的开销，能够从并行执行中受益）。第三，它可以检测影响并行流处理的负载不均衡。最后，它有助于检测仅执行少量周期的大量小型流（例如通过flatMap [23]创建的流），从而为开发者发现优化机会提供线索（例如使用mapMulti [23]移除这类小型流）。

虽然准确率对我们的目标至关重要，但用于周期分析的插桩代码不可避免地会导致额外（不必要）周期的执行，而这些周期会（部分）计入分析结果中。为缓解这一问题，我们会补偿插桩导致的额外周期，即估算并从收集到的分析结果中剔除这些周期。

我们通过StreamProf 实现了我们的方法。据我们所知，它是首个专为 JVM 打造的流性能分析器。我们通过使用StreamProf对业界先进的基准测试套件Renaissance [24],中的两个工作负载进行性能分析，展示了它的实用性：诊断并解决与流相关的性能问题（第 3 节）。我们首先找出可以安全移除、且不会改变工作负载语义的冗余流。随后，我们诊断出影响并行流处理的负载不均衡问题。我们使用StreamProf 生成的可操作的分析结果修复检测到的问题，实现了最高达 5× 倍的加速比。据我们所知，目前还没有其他研究针对Renaissance 套件中的基于流的应用程序进行优化。

我们通过评估三个不同来源中公开提供的工作负载，测量了StreamProf 生成的分析结果（经过补偿后）的准确率，以及分析开销（第 4 节）。评估结果表明，我们的方法效率高，能够生成准确且可操作的分析结果，帮助开发者优化流处理，同时将分析开销控制在可用于 Java 应用程序的水平。

我们的工作面临多项挑战。与相关方法 [9–11,14, 15], 不同，我们的技术旨在检测 Java 应用程序执行的每一个顺序流和并行流。因此，首要挑战在于开发一种插桩，以正确跟踪 Java 类库（JCL）中所有形式的流执行。我们要应对的另一项关键挑战，是降低分析开销，从而实现周期精确流分析。这要求插桩能够高效地收集有意义且数量最少的度量指标，以减少测量扰动。此外，我们还要估算插入的插桩代码导致的额外周期数，并尝试通过扰动补偿技术将其从分析结果中消除。

我们还讨论了本文方法的局限性以及评估中的效度威胁（第5节），回顾了相关工作（第6节），并在最后给出结论（第7节）。

2 StreamProf

本文介绍了我们在 JVM 上对流进行性能剖析的方法。首先，我们介绍性能分析模型，该模型描述了我们这项技术所针对的关键实体和事件（第 2.1 节）。随后，我们详细阐述了对流执行进行性能剖析的方法（第 2.2 节）。

2.1 性能分析模型

列表 1 展示了一个 Java 流的代码示例。在第 2 行，从bookList生成了一个流，也就是一个包含Book类对象的列表。该流水线包含四个操作：过滤，用于筛选畅销书（第 3 行）；flatMap，用于展开[2] bookList 中每本书的作者列表（第 4 行）；以及collect，用于将畅销书作者姓名汇总到一个列表中（第 5 行）。在本小节中，我们将用这个示例来介绍我们的性能分析模型。

列表 1 一个简单的 Java 流代码示例。

```
1 列表<字符串> getTopSellerAuthors(List<Book>bookList) {
2   return bookList.stream()
3     .过滤(Book::isTopSeller)
4     .flatMap(b -> b.getAuthors().流())
5     .collect(Collectors.toList());
6 }
```

```
1 类 Book{
2   ...
3   boolean isTopSeller() {...}
4   List<String> 获取作者() {...}
5   ...
6 }
```

流。 作为一种流，我们将java.util.stream.AbstractPipeline子类型的每个实例都视为流。AbstractPipeline 是表示流及其流水线中操作的抽象类。由于 Java 类库中任何形式的流创建最终都会实例化某个AbstractPipeline子类型，因此我们会对所有这些子类型进行插桩，以确保无论流是如何创建的，都能检测到每次流执行。在 Stream API 中，可以创建一种流，其数据元素可以是对象，也可以是基本类型的值（即仅限int、long或double）。

流执行。 我们的性能分析技术能够检测流执行。调用终端操作即可执行一个流；终端操作会触发流水线的遍历，以返回结果（例如数组）或产生副作用（例如打印所有元素）。流是惰性的，也就是说，在调用终端操作之前不会进行任何处理。一个流最多只能有一个终端操作，而且该操作只能执行一次 [1]。在清单 1 中，collect 是终端操作。终端操作会调用流执行方法，即 Stream API 中（位于 AbstractPipeline 或其子类型中）的一个方法，用于启动流的执行。我们的方法支持无限流（即数据源能够生成无限多个元素的流），并测量流执行期间经过的参考周期，直到执行完成。此外，流还可以是

[2]术语扁平化 是指将多个集合/数组中包含的元素合并到单个数据结构中。

要么是顺序 执行，要么是并行执行。前者由当前线程同步执行，而后者的执行则涉及创建任务。

任务。 执行并行流时，其计算由分叉/合并[3] 任务完成。调用终端操作时会生成一个初始任务，并在需要时分叉出更多任务。这里所说的任务，是指 Stream API 实现中的java.util.concurrent.-CountedCompleter的所有子类型。任务由线程在公共分叉/合并线程池 [26] 中执行，这些线程称为工作线程。分叉/合并线程池实现了工作窃取 [27, 28]。在 Stream API 中，工作线程通过调用方法CountedCompleter.compute的实现来执行任务；我们将此方法称为任务执行方法。执行方法完成后，我们便认为该任务已执行完毕。我们对并行流生成的所有任务进行性能剖析。

包含有状态操作 的流水线（即当前状态可能取决于先前已处理元素状态的中间操作）可能需要缓冲大量数据，从而可能损害并行性能 [1]。我们的方法只能部分衡量数据缓冲对并行流造成的影响，因为我们只测量为执行流而生成的任务所完成的计算，而不考虑操作系统中的线程调度开销（有关我们技术的局限性，请参见第5节）。

嵌套流。 一个流可以嵌套，也就是说，其执行由另一个流触发（并发生在该流执行期间），后者称为外层流。嵌套流总是在外层流执行完毕之前完成。在清单 1 中，flatMap操作使用的流嵌套在由bookList创建的外层流中。外层流本身也可以嵌套，因此可能存在多个嵌套层级。流的嵌套信息有助于开发者更好地评估流并行化的潜在收益，例如，嵌套在并行流中的顺序流可能无法从进一步并行化中获益。此外，检测低效嵌套（例如，大量嵌套流中的每个流都只运行几个周期）可能意味着存在优化机会，因为合并或避免此类流可以提升应用程序性能。具体来说，mapMulti [23] 操作可以使用命令式代码，避免由flatMap等操作引入的低效嵌套。我们的方法能够正确重建流之间的嵌套关系。

参考周期。 与墙钟时间 [29] 和字节码计数 [30], 等其他度量指标相比，参考周期能够精确衡量流所执行的计算，不包括工作线程未被调度到 CPU 核心上执行的时间段，并能计入由未对齐和缓存未命中导致的延迟。对所有流执行进行完整的周期分析，有助于检测与流相关的性能问题（如第 3 节所示）。

位置。 我们将流执行与位置关联起来，也就是终端操作调用者的完全限定名。这些信息会显示每个流在应用程序代码中执行的类和方法。在清单 1 中，

[3]分叉/汇合并行依赖于将工作递归拆分（分叉）为并行执行的任务，等待这些任务完成，然后通常合并（汇合）由分叉任务计算出的结果 [25]。[4]类/接口的完全限定名会在文本中首次出现时显示；之后我们仅使用类/接口名称。

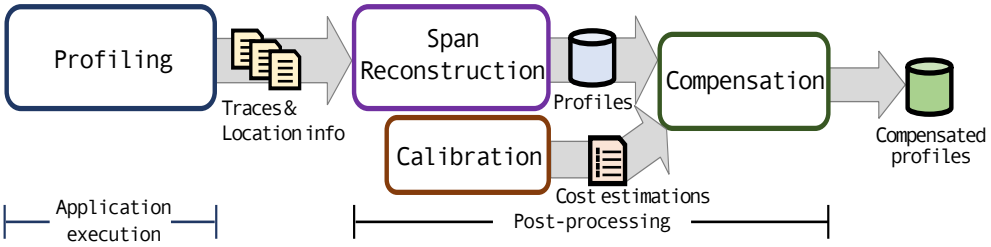


图 1 性能分析方法概览。

外层流的位置是方法 getTopSellerAuthors （省略类名）。位置信息是可操作的性能剖析的关键部分，它能指引开发者找到应用程序代码中优化可能带来最大性能提升的地方。

跨度。 我们围绕跨度这一概念对流执行进行建模，将其定义为线程执行流的时间间隔。每个跨度的开始和结束分别由流执行前后的参考周期读数确定。我们的目标是在目标应用程序执行期间对所有跨度进行性能剖析。

为了收集跨度中经过的周期，我们读取由处理器的性能监控单元提供的每线程虚拟化参考周期计数器。我们将周期$_{begin}$ 和周期$_{end}$ 分别称为在跨度开始和结束时读取的周期值。我们将周期$_{end}$ −周期$_{begin}$ 定义为该跨度的测量周期数。由于每个线程周期计数器按照经过的周期严格单调递增，我们假设在单个线程的范围内，周期$_{begin}$ 和周期$_{end}$ 具有全序关系。此外，一个线程中的跨度不能相互重叠，但一个跨度可以嵌套在另一个跨度内，从而能够检测嵌套流。

一个跨度始终对应单个流，但多个跨度可以关联到同一个并行流。顺序流执行由单个跨度表示，该跨度由流执行方法执行前后的参考周期读数确定。相比之下，并行流执行通常由多个跨度表示（即每个为执行并行流计算而启动的任务对应一个跨度），其中每个跨度由任务执行方法执行前后的参考周期读数确定。同一并行流中各任务执行的计算都会计入该流的测量周期数。

2.2 在 JVM 上对 Streams 进行性能分析

图 1 展示了我们的流性能分析方法。首先，StreamProf 对目标应用程序进行插桩，以便对运行时检测到的所有跨度进行性能分析。StreamProf基于 DiSL [31]，，这是一个基于字节码插桩的 JVM 动态分析框架。DiSL 的一个关键特性是支持对 Java 类库进行完整插桩，Stream API 就实现在其中。得益于 DiSL，StreamProf 能够准确拦截 Java 应用程序执行的每个流。

应用程序执行结束时，StreamProf 会为每个至少执行过一个流的线程生成一条跟踪记录。每条跟踪记录都包含该线程执行的所有流执行跨度的信息。除了逐线程跟踪记录外，StreamProf 还会将位置信息存储在单独的文件中（跟踪记录通过位置索引引用该文件）。

该文件和跟踪记录会在离线状态下进行处理，以尽量减少性能剖析期间的数据处理，从而降低测量扰动。在后处理期间，我们首先执行跨度重建，即综合所有逐线程跟踪记录，生成每个流的性能剖析数据。每个流性能剖析数据中的信息，都是根据与相应流相关的跨度重建而成的。

我们还会执行一个校准 阶段（每个性能分析平台仅执行一次），以获取估算成本，也就是对性能分析跨度所需额外周期数的估算。最后一个阶段会使用这些开销估算，在该阶段中，我们补偿 插桩导致的额外周期数，也就是从每个跨度的测量周期数中扣除这些周期。StreamProf 的输出是性能剖析结果，其中报告了补偿后的周期数、嵌套关系，以及每次流执行的位置，还包括线程级和位置级的聚合统计信息（见第 3 节）。总体而言，我们的性能剖析结果准确报告了目标应用程序执行的所有流的信息。本节余下部分将更详细地介绍各个阶段。

2.2.1 性能分析

性能分析阶段旨在拦截每个跨度的起始和结束，以便将其与相应的测量周期数和位置关联起来，在应用程序执行期间将收集到的数据存储在内存中，并在应用程序执行完毕后持久化这些信息。5

跨度。 顺序流执行由单个线程完成，并由单个跨度表示。这样就能将跨度中的测量周期数直接归因于相应的流，无需使用任何标识符进一步识别顺序流。因此，我们用匿名跨度 来指代顺序流执行中的单个跨度。

并行流通常由多个任务执行，这些任务可由多个工作线程执行，因此需要检测可能关联不同线程的多个跨度。与顺序流不同，在这种情况下，必须将所有跨度归因于相应的并行流，以便汇总各跨度中的测量周期数，从而估算该流执行的全部计算量。根据 Stream API 的设计，并行流执行始终从执行一个原始分叉/合并任务开始；该任务可根据需要派生其他任务。执行原始任务时，我们会为相应的并行流关联一个唯一标识符，即将流 ID 与流水线头部关联起来；该对象表示流水线的初始阶段。在分析同一并行流中某个派生任务对应的跨度时，插桩会将该跨度与流 ID 关联起来（也就是说，由于任务持有对流水线头部的引用，我们会利用它查询流 ID）。因此，我们用命名跨度来指代与并行流执行相关的跨度。

与同一流 ID 关联的所有命名跨度中，恰好有一个跨度对应于流的原始任务的执行。我们将该跨度称为原始跨度，而将同一流中的其他支持跨度 统称为

[5]伪代码形式的插桩逻辑见附录 A。

命名跨度。在对原始跨度进行性能分析时，始终需要生成一个流 ID；而对支持跨度进行性能分析时，只需检索现有的流 ID。如第 2.2.2 节所述，原始跨度还允许识别外层流（如果存在），这是重建流嵌套所必需的。最后，我们指出，我们的技术能够检测每一次流执行，而不受执行该流的线程或分叉/合并线程池配置的影响。此外，我们的技术可以对并行流执行中的所有分叉/合并任务进行性能分析，同时忽略并发使用分叉/合并线程池的其他处理。

位置。为了给每个跨度标注位置，我们会进行专门的分析（在插桩时），以确定所有可能直接调用流执行方法的方法的完全限定名称，并为它们分配唯一的方法 ID。当拦截到跨度开始时，插桩会识别流执行方法的调用方，获取预先计算好的方法 ID，并将其与当前跨度关联。应用程序运行结束时，会转储一个文件，其中包含所有方法 ID 与对应方法名称之间的映射。

跟踪记录生成与转储。插桩依赖于一个跟踪器，即附加到目标应用程序的 JVMTI [32] 代理，负责从本机代码读取周期计数器、存储跨度信息并转储跟踪记录。跨度开始时，插桩会使用跟踪器读取周期*begin*，并将其与位置一起存储为一个事件（跨度开始）。同样，插桩会使用跟踪器读取周期*end*，并将该值存储在第二个事件（跨度结束）中。跟踪器使用*PAPI* [33]（即一个开源 C 库，可提供读取每线程虚拟化计数器的通用接口）来读取事件*PAPI_REF_CYC*。

为降低分析开销，跟踪器维护线程局部缓冲区，将与线程关联的所有跨度信息存储在内存中，从而无需进行同步（分配新缓冲区时除外）。应用程序完成后，跟踪器会转储这些缓冲区，为每个线程生成一条跟踪记录。当预分配的缓冲区已满时，跟踪器会将其转储，并在运行时分配新缓冲区。值得一提的是，第 3 节和第 4 节中分析的工作负载均不需要这样做。最后，每条转储的跟踪记录都包含线程的 ID 和名称，以及所观测到的 跨度开始 和 跨度结束 事件序列。总体而言，为减小测量扰动，我们只存储尽可能少的数据。

2.2.2 跨度重建

我们处理跟踪记录，以计算每流信息，即每个流中的测量周期数总和以及位置。[6] 我们首先逐条处理跟踪记录，为每个线程分别生成一份性能剖析。流的性能剖析是根据跟踪记录中存储的相应跨度开始 和跨度结束 事件重建的。为此，我们为每个线程维护一个栈：每当遇到跨度开始 事件时，就将一个新的跨度对象压入栈中；相应的跨度结束 事件则表示该跨度的处理结束，并将其从栈中弹出。

[6]附录 B 展示了跨度重构算法的伪代码。

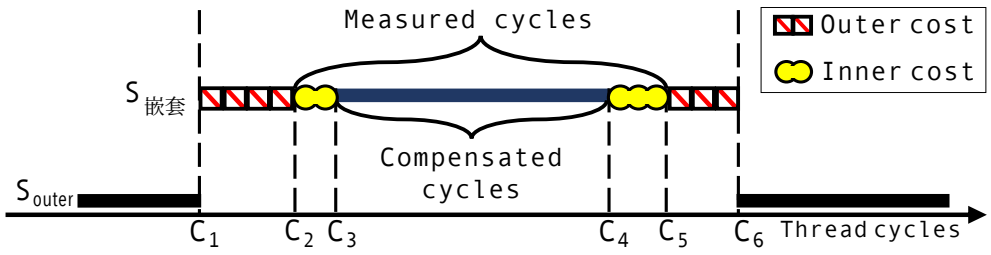


图 2 内部和外部开销的表示。$C_{\#}$ 表示在给定时间从线程周期计数器读取的周期值。C_2 表示S^{nested}_{begin}，而C_5表示S^{nested}_{end}。

对于顺序流执行，测量周期数根据相应的单个匿名跨度计算。相比之下，单次并行流执行中的命名跨度通常分布在多个线程性能分析文件中。因此，我们需要合并这些线程性能分析文件，并将每个命名跨度关联到由插桩分配给该跨度 ID 的并行流执行。合并后，通过汇总其所有命名跨度中的测量周期数，计算并行流执行的测量周期数。每个跨度也会被分配相应的位置。为此，我们会在位置映射文件中查找该跨度的方法 ID。对于并行流，其所有命名跨度都共享同一位置。

栈还可以用于重建跨度之间的嵌套关系。对于匿名跨度，栈可以帮助识别对应的外层跨度（如果存在），即栈顶下方的那个跨度。对于同一次并行流执行中的命名跨度，栈仅能为原始跨度提供有效的外层跨度信息，因为最终的支持跨度表示可能由不同线程执行的任务。因此，对于每次并行流执行，我们使用其原始跨度作为代表，使支持跨度继承该跨度的外层跨度信息。更新外层跨度信息后，我们会计算每个跨度的嵌套层级。如果某个跨度没有外层跨度，则将其嵌套层级设为零；否则，沿外层跨度链递归计算嵌套层级，直到找到最外层跨度。

2.2.3 校准

跨度内测量的周期数是量化流所执行计算的关键指标。不过，StreamProf 插入的插桩代码会导致额外周期的产生，这些周期有一部分会计入每个跨度的测量周期数，从而降低性能分析准确性。我们的目标是从生成的分析结果中去除这些额外周期，使归因于每个跨度的周期数更接近未插桩的应用程序执行所产生的周期数。为此，需要估算这些额外周期，这正是校准阶段的目的。

插桩开销。下文中，我们使用术语 成本 来指代插桩为对某个跨度进行性能剖析而引入的额外周期。我们估算两种类型的成本，如图 2 所示。图中展示了与跨度 S_{nested} 相关的经过的周期，该跨度嵌套在外层跨度 S_{outer}中。x轴表示线程周期计数器的值，该值随着流执行的进行而递增。

对 S_{nested}进行性能剖析时，StreamProf 会执行插桩代码来剖析其开始和结束。假设剖析 S_{nested} 开始处的插桩代码耗时 C_3–C_1 周期（即 C_1 与 C_3之间经过的周期数），而剖析 S_{nested} 结束处的插桩代码耗时 C_6–C_4 周期。在剖析这些事件时，StreamProf 会读取周期计数器两次，以获取 $cycles_{begin}$ 和 $cycles_{end}$。假设 $cycles_{begin}$和 $cycles_{end}$ 的周期值分别为 C_2 和 C_5，。执行 S_{nested} 期间实际经过的周期数为 C_4–C_3。然而，性能分析器测得的周期数为 C_5–C_2，，大于实际周期数。我们将内部开销定义为 (C_3–C_2) +(C_5–C_4)，它表示计入该跨度测量周期数中的开销。请注意，每个跨度都会产生内部开销。

当性能分析器计算 S_{outer} 的周期数时，会减去 S_{nested} 的测量周期数（即 C_5–C_2）。这样做时，由于 S_{nested} 中测得的周期数包含其内部开销，因此 S_{nested} 的内部开销未计入 S_{outer}。然而，S_{nested} 的插桩也会导致在读取周期数$_{begin}$ （C_2–C_1）之前和读取周期数$_{end}$ （C_6–C_5）之后额外执行一些周期。这些周期未计入 S_{nested}，但计入了 S_{outer}。我们将外部开销定义为数量 (C_2–C_1) + (C_6–C_5)，它对应于仅在外层跨度中测得的额外周期数。

外部开销取决于跨度的类型。对匿名跨度进行性能剖析时，外部开销最低，因为无需流 ID。对于命名跨度，外部开销取决于是否需要生成流 ID（即原始跨度），还是只需获取流 ID（即支持跨度）。原始跨度的外部开销高于支持跨度，因为生成新的流 ID 需要进行额外计算。

我们注意到，图 2 表明，与流执行的测量周期数相比，我们的插桩开销通常小得多。不过，在某些情况下，执行插桩代码引入的额外周期可能占主导地位，例如执行仅运行几个周期的嵌套流时。

开销估算。 我们的目标是对内部开销和外部开销进行简单估算，并用常数近似表示。我们分别用 内层成本、*OC_ANON*、*OC_PRIM* 和 *OC_SUPP* 表示内部开销、匿名跨度的外部开销、原始跨度的外部开销以及支持跨度的外部开销的近似值。校准阶段会测量给定平台上的这些常数。为此，我们使用一个跨度生成器，即一个生成成对排列的跨度的 Java 程序，其中一个跨度嵌套在另一个跨度内。嵌套跨度执行期间不会运行任何代码，而在外层跨度执行期间，我们会运行与嵌套跨度依赖跨度类型的开始/结束事件相对应的代码。

要估算外部开销_匿名，在外层跨度期间，仅执行用于分析嵌套匿名跨度开始/结束事件的代码。要估算外部开销_*PRIM*，在外层跨度期间，我们还会执行生成新流 ID 所需的计算。要估算外部开销_*SUPP*，在外层跨度期间，我们会执行检索已有流 ID 所需的计算。在此校准设置下，嵌套跨度的测量周期数对应于内层成本；假设补偿周期数为零（因为嵌套跨度期间未执行任何代码），这一点可从图 2 中看出。我们将嵌套跨度的测量周期数平均值作为内层成本 的估值。然后，根据公式 1 估算不同的外部开销。

$$outer_cost = (S_{nested.cycles_{begin}} - S_{outer.cycles_{begin}}) + (S_{outer.cycles_{end}} - S_{nested.cycles_{end}}) - IC \quad (1)$$

请注意，外部_开销 是一个通用术语，可指三种外部开销中的任意一种。实际计算出的成本取决于生成的跨度类型（例如，如果生成的跨度是匿名跨度，则会计算外部开销_匿名 ）。每项成本的估值均为执行1000 万对跨度所得结果的平均值，并剔除了离群值。

2.2.4 补偿

补偿阶段的目标是计算每个跨度的补偿周期数，如公式 2 所示。

$$compensated_cycles = (measured_cycles - nested_cycles) - \\ (n^{anon} \cdot OC_匿名) - (n^{prim} \cdot OC_PRIM) - (n^{supp} \cdot OC_SUPP) - 内层成本 \quad (2)$$

我们首先从每个外层跨度的测量周期数中减去嵌套周期数（即嵌套跨度所消耗的周期数）（参见图 2）。这样做是为了只计算外层跨度的周期数，因为默认情况下，外层跨度的测量周期数包含嵌套周期数。为此，我们会为每个跨度记录其直接嵌套跨度的信息，包括嵌套周期总数。然后，还要减去所有开销（以常数近似）。这样做是为了消除对执行插入的插桩代码所导致周期数的估算。为此，我们会为每个跨度记录嵌套的匿名跨度（n^{anon}）、原始跨度（n^{prim}）和支持跨度（n^{supp}）的总数，并分别乘以相应的估算外部开销。最后，我们减去每个跨度所承担的内部开销（见第 2.2.3 节）。

3 流的性能分析与优化

在本节中，我们将展示StreamProf 如何帮助开发者优化流。我们首先介绍实验设置（第 3.1 节）。然后，我们介绍对Renaissance 基准测试套件中两个工作负载的优化：首先定位并移除不必要的流执行（第 3.2 节），然后改进负载均衡（第 3.3 节）。最后，我们展示通过优化所获得的加速比（第 3.4 节）。

3.1 实验设置

在本小节中，我们介绍实验设置。

基准测试。 我们的评估目标是Renaissance [24]，这是一个面向 JVM、包含现代并发面向对象工作负载的先进基准测试套件。与DaCapo [34]、ScalaBench [35]、和SPECjvm2008 [36]等此前广为人知的套件相比，Renaissance 包含了大量使用流处理的工作负载。我们使用撰写本文时Renaissance 的最新版本，即 v.0.14.1，该版本于23日、2022年5 月发布。

Renaissance 包含三个用于测试流的工作负载，其中两个是我们优化的对象，即助记符 及其并行版本并行助记符。我们指出，第三个

基于流的工作负载Renaissance，即scrabble，仅用于我们的评估（第 4 节），因为我们发现该工作负载中没有可优化的流相关性能问题。对于实验，我们仅考虑稳定状态，即在运行足够多的预热 迭代后，垃圾收集器人体工程学和动态编译均已稳定的状态。对于助记符 和并行助记符，我们按照Renaissance 的开发者建议，运行16 次预热迭代 [37]。

测试平台。 我们在两台机器 M_1 和 M_2上进行了所有实验。M_1 配备 8 核英特尔 Xeon E5-2680（2.7 GHz）处理器和 128 GB 随机存取存储器。M_2 配备 18 核英特尔 i9-10980XE（3.0 GHz）处理器和 256 GB 随机存取存储器。M_1 和 M_2 运行 Linux，通用内核版本分别为 4.15.0-147 和 5.4.0-89。我们禁用了睿频加速和超线程。CPU 调速器设置为 “ 性能 ”。我们使用 Oracle JDK 17 [38] （构建版本 17.0.3.1）和 PAPI6 （构建版本 6.0.0.1）。

3.2 优化 1：移除不必要的流

我们首先对 助记符进行性能分析和优化。StreamProf 会生成定制热图。7，开发者可以利用这些热图了解流嵌套情况，以及流执行和补偿周期数的分布。在我们的方法中，我们重点关注热图标示的较暗区域，因为减少这些区域中的周期数很可能会提升性能。为便于完成这项任务，StreamProf 还会报告热点位置，即应用程序代码中负责大部分流处理的位置。借助这些信息，我们首先分析最热点位置，也就是流执行次数和总周期数均最高的位置，即 org.renaissance.jdk.streams.MnemonicsCoderWithStream.wordForNum。此方法中执行的流占 助记符所执行全部流（14,532, 738）的90.07%，占其总周期数的76.43%。

我们检查了源代码，发现方法 wordForNum 是在递归处理中执行的。由于递归调用发生在流执行过程中，这就解释了性能分析器检测到的深层嵌套。我们还发现，从方法 wordForNum 触发的流执行始终源自同一个数据源，即在工作负载执行前初始化、此后保持不变的字符串列表。此外，我们发现此类流的执行不依赖当前递归层级的任何输入，因此总是会产生相同的输出。为了优化助记符，可以将这些流移出递归代码，只执行一次。

沿用相同的策略，我们找到了另一个热点位置，即 MnemonicsCoder-WithStream.lambda$encode$9。我们发现，该位置的流执行占 mnemonics 执行的总流数的 3.75% ，却占其总周期数的10.64%。我们发现，其中一些流是由传递给方法 MnemonicsCoderWithStream.encode 的一组数据创建的。此外，该集合初始为空，且从未被修改。我们还确认，由空集合创建的流执行后所产生的输出不会

[7]StreamProf 生成的热图如附录 C 所示。

以任何形式影响mnemonics的输出，这表明我们可以安全地移除这些不必要的流执行。

我们将两项修改都应用于 助记符，移除检测到的有问题的流，并显著减少优化版本中不必要且开销高昂的计算。8 总体而言，减少流执行次数显著提升了性能（见第 3.4 节）。我们的目标是在尽量少改动原始工作负载的前提下进行优化。得益于可操作的分析结果，我们只修改了6 行代码，就优化了 助记符 。

3.3 优化 2：改善负载均衡

我们按照第 3.2 节所述的相同方法优化 par-mne ，针对 MnemonicsCoder.paralleEncode 和 MnemonicsCoder.wordsForNum-Parallel这两个方法。它们分别对应于在 助记符中优化过的方法，并触发并行（而非顺序）流处理。在为优化后的 par-mne生成的分析结果中，我们运行StreamProf 提供的额外分析，以计算负载均衡信息，即参与并行流处理的各工作线程的周期分布。我们使用变异系数9（CV）来分析与流相关的各工作线程周期的变化情况。CV 等于零表示负载完全均衡；CV 越高，负载失衡程度越大。

我们发现，并行助记符 的负载均衡较差。尽管有8 个工作线程在 M_1，上执行并行助记符 ，但只有其中两个工作线程执行了99.99%（分别为79.54% 和20.45</style%）的总周期数，导致变异系数达到2.24。类似地，10 个工作线程在 M_2，上执行并行助记符 ，其中两个工作线程执行了99.99%（分别为80.67% 和19.32%）的总周期数，变异系数为2.55。在两台机器上，总共生成了52 个任务。我们的分析结果指出，工作负载中首次触发并行流执行的位置是MnemonicsCoderWithStream.parallelEncode。我们发现，并行助记符 仅在第一个递归层级执行并行流，而后续递归层级则依赖顺序流。这一行为由一个阈值决定，该阈值会考虑当前递归层级要处理的元素数量。

在分叉/合并线程池中，改善负载均衡的一项关键策略是增加生成的任务数量，这样可以让更多 CPU 核心保持忙碌，提高可扩展性和局部性，并减少 CPU 核心空闲的时间 [39]。然而，任务数量过多可能会因创建、调度和管理任务所带来的开销，抵消并行处理的优势 [40]。为了减少par-mne的负载不均衡，我们对任务粒度进行了优化。

我们提高了上述阈值，使更多递归层级使用并行流。阈值设为20 时（原始阈值为3），效果最佳；在 M_1 和 M_2上共生成226,596 个任务。任务数量增加后，M_1 和 M_2，上的变异系数分别改善至0.60 和0.57 。

[8]优化后的热图如附录 C 所示。[9]变异系数是标准差与均值的比值。

表 1 优化流处理带来的加速比。

基准测试	Ver.	机器	#流	#任务	Time [ms]	加速因子	95% 置信区间
mne	orig	M_1	14, 532, 738		4, 944.72		
		M_2			3, 303.64		
	opt1	M_1	1, 088, 946		1, 200.70	**4.09**	(3.99, 4.19)
		M_2			981.97	**3.36**	(3.32, 3.40)
par-mne	orig	M_1	14, 532, 738	52	4, 419.53		
		M_2			2, 977.83		
	opt1	M_1	1, 088, 946	52	1, 106.22	**3.98**	(3.89, 4.04)
		M_2			905.19	**3.27**	(3.20, 3.33)
	opt2	M_1	1, 088, 946	226, 596	880.09	**5.00**	(4.91, 5.10)
		M_2			764.63	**3.88**	(3.80, 3.96)

总而言之，借助可操作的分析结果，我们优化了par-mnemonics 中的负载均衡。我们的优化只需修改2 行代码（此外还需修改第 3.2 节中报告的6行代码）。我们注意到，要进一步改善变异系数，很可能需要对par-mnemonics进行大规模代码重构，但这超出了我们的工作范围。

3.4 加速比

在本小节中，我们报告了通过优化获得的加速比。

表 1 总结了在 M_1 和 M_2 上，助记符（mne）和并行助记符（par-mne）的原始版本（*orig*）及优化版本（优化版本1 和 优化版本2）中的流处理信息，版本由列 版本区分。首先，我们报告流执行总次数（#流）、任务总数（#任务，仅适用于并行助记符）以及工作负载执行时间（时间）。然后，我们展示优化所实现的加速比[10]。所报告的执行时间和加速比为100 次稳态运行的平均值[11] 。我们报告每个加速比的 95% 置信区间（CI）。所有优化后的工作负载均通过了Renaissance测试框架的验证，表明我们的修改没有改变工作负载输出。[12]

对于所有经过修改的工作负载，我们都观察到优化带来了显著的加速比。首先，移除不必要的流处理（优化版本*1*）使助记符（在 M_1 上最高达4.09× ，在 M_2 上最高达3.36× ）和并行助记符（在 M_1 上最高达3.98× ，在 M_2 上最高达3.27× ）都获得了显著的加速比。其次，除了移除不必要的流之外，改善负载均衡（*opt2*）也使并行助记符获得了显著的性能提升（在 M_1 上最高达5.00× ，在 M_2 上最高达3.88× ）。

总的来说，周期性能分析能够有效检测出存在问题的流（在执行的数百万个流中），对这些流进行优化有望提升应用程序性能。准确分析分叉/合并任务，有助于根据工作线程级别的流相关周期分布来诊断负载不均衡

[10]加速比的计算方式是：原始（未修改）工作负载的执行时间除以优化后工作负载的执行时间。[11]多个工作负载的平均值使用几何平均数计算。[12]我们已联系 Renaissance 的开发者，他们确认了我们的优化既正确又有效。

。此外，StreamProf 生成的可操作的分析结果有助于开发者专注于执行问题流的代码位置。事实上，完成每项优化（即手动检查代码、修改代码并进行全面测试）大约花费了我们两小时。

4 评估

本节介绍对 StreamProf 的评估。我们首先描述实验设置（第 4.1 节）。随后，我们评估 StreamProf 生成的分析结果的准确率（第 4.2 节）及其分析开销（第 4.3 节）。

4.1 评估设置

我们的评估针对来自三个公开可用来源的工作负载。首先，我们针对第 3 节研究的所有工作负载，其中还包括 scrabble，这是一个并行工作负载，执行4,350,761 个流，并分别在 M_1 和 M_2，上生成 32 和 128 个任务。通过纳入 scrabble，我们涵盖了 Renaissance 中所有可用的基于流的工作负载。其次，我们针对来自 JEDI [41, 42]，的工作负载；这是一个由 25个基于流的应用程序组成的基准测试套件，每个应用程序都将一个 *TPC-H* [43] 查询重新实现为使用 Java 流 API 的 Java 实现。最后，我们针对由 OpenJDK 开发者编写、公开发布于 OpenJDK18 开源仓库中的若干基于流的工作负载 [44]（为简洁起见，我们将这些工作负载称为 OpenJDK ）。对于 scrabble，我们使用该套件的测试框架运行 50 次预热迭代（如 Renaissance [37]的开发者所建议）；对于来自 JEDI 和 OpenJDK 的工作负载，我们使用 *Java* 微基准测试框架（JMH） [45]运行 15 次预热迭代。我们在第 3.1 节所述的同一测试平台上进行评估。需要指出的是，我们使用StreamProf 分析了 JEDI 和 OpenJDK 工作负载，并未发现需要优化的流相关性能问题。

我们确定了评估中两个相关的关注维度。第一个维度考察每跨度的平均周期数（CPS）。我们旨在了解不同 CPS 对准确率和分析开销的影响。我们预计，对于 CPS 较低的工作负载，插入的插桩代码所占的相对成本（相对于流处理的成本）高于 CPS 较高的工作负载。因此，我们预计较高的 CPS 会带来更高的准确率和更低的开销。第二个维度考察流处理的执行模式，即顺序执行还是并行执行。我们旨在研究，在并行流执行的情况下，分析结果的准确率或分析开销是否会受到影响。

4.2 准确率

在本小节中，我们评估由 StreamProf 生成的分析结果的准确率。

[13]由于OpenJDK 包含一些非常相似的工作负载（例如，使用方法引用而不是 lambda 表达式），为避免评估产生偏差，我们对每种工作负载仅选取一个具有代表性的实例。

■ **表 2** 插桩的估计成本（以周期计）。*IC*、*OC_ANON*、*OC_PRIM* 和*OC_SUPP*分别表示内部开销、匿名跨度的外部开销、原始跨度的外部开销以及支持跨度的外部开销。

	M_1		M_2	
Cost	**AVG**	**CV**	**AVG**	**CV**
IC	**171.63**	0.24	**130.12**	0.32
OC_ANON	**184.25**	0.24	**132.46**	0.41
OC_PRIM	**212.40**	0.21	**169.82**	0.36
OC_SUPP	**201.17**	0.23	**163.83**	0.44

4.2.1 插桩成本

如第 2 节所述，我们进行校准阶段，将插桩开销估算为常数。表2汇总了在 M_1 和 M_2 上计算出的内层成本以及不同类型外部开销的常数。按照第 2.2.3 节所述的成本估算方法，我们在两台机器上针对每种成本生成1000万对跨度。表中报告了这些跨度的平均成本，以及变异系数。

M_2 上的变异系数高于 M_1 上的变异系数。由于我们是在使用命名跨度的场景中估算成本，因此预期变异会更大，因为 M_2 拥有更多核心（M_1 是8 核心机器，而 M_2 是18 核心机器）。

我们的方法旨在减轻插桩代码执行所带来的扰动。表2中报告的较低数值，得益于采用了高效的插桩来收集最少量的数据，并配合经过精细调优的本机代码（即跟踪器），从而能够高效地收集周期，并在应用程序执行期间将其暂存于内存中。

4.2.2 基线计算

我们的准确性评估旨在比较目标应用程序所有已分析跨度的补偿周期总数（由StreamProf报告），以及同一应用程序在未进行任何性能剖析时消耗的流相关周期，即基线。补偿周期总数与基线之间的差异越小，准确率就越高。我们关注的是所有已分析跨度的补偿周期总数，而非单个跨度，因为据我们所知，目前没有任何方法可以在单跨度级别上获得可靠的基线。我们将在第5节讨论评估中的效度威胁。

精确计算基线对我们的评估至关重要。然而，基线只能通过测量确定，而测量又会受到扰动的影响。为了减少扰动，我们利用了这样一个事实：所有受评估工作负载执行的计算都仅发生在流执行过程中。因此，我们测量工作负载执行期间消耗的总周期数（将其近似视为流相关计算消耗的总周期数），并将其作为基线。

为计算基线，我们使用 PAPI 为每个线程读取周期计数器，但仅读取两次，即在线程执行稳态迭代期间的流处理之前和之后。对于执行顺序流的工作负载，我们仅将 PAPI 附加到触发工作负载执行的线程，即主线程。对于执行

表3 准确性评估结果。

基准测试	Ver.	机器	#匿名跨度	#原始跨度	#支持跨度	总周期数 AVG	CV	CPS	准确率 [%]
mne	orig	M_1	14, 532, 738			13,304,591,663	0.13	915	**87.99**
		M_2				9,869,363,238	0.06	679	**95.18**
	opt1	M_1	1, 088, 946			3,235,842,391	0.02	2,972	**98.99**
		M_2				2,938,993,549	0.02	2,699	**99.54**
par-mne	orig	M_1	14, 532, 736	2	50	14,927,826,366	0.10	1,027	**88.27**
		M_2				11,065,382,118	0.12	761	**95.83**
	opt1	M_1	1, 088, 944	2	50	3,583,085,865	0.02	3,290	**99.68**
		M_2				3,255,485,394	0.01	2,989	**99.66**
	opt2	M_1	1, 074, 785	14, 161	212, 435	4,218,389,373	0.03	3,241	**99.42**
		M_2				4,471,109,416	0.02	3,436	**94.98**
scr	orig	M_1	4, 350, 760	1	31	6,445,643,039	0.03	1,481	**89.97**
		M_2			127	9,359,769,396	0.01	2,151	**97.13**
JEDI		M_1							平均值: **91.55**
		M_2							平均值: **90.34**
OpenJDK		M_1							平均值: **94.82**
		M_2							平均值: **96.73**

并行流的工作负载，我们将 PAPI 同时附加到主线程和参与流处理的工作线程。我们将基线计算为参与流处理的所有线程所经历的总周期数。

测量基线时，每次周期读取都会产生一项成本，该成本会（部分）计入测量结果。不过，这项成本与内部开销同一数量级（即10^2 周期），相对于基线测得的总周期数而言很小，可以忽略不计。

4.2.3 准确度计算

表3汇总了我们对助记符（mne）、并行助记符（par-mne）和 scrabble（scr）在 M_1 和 M_2 上的原始版本（*orig*）及优化版本（*opt*1 和 *opt*2）的准确率和分析开销评估结果，版本由*"Ver."*列区分。对于每项工作负载，我们首先报告匿名跨度（*#Anon.spans*）、原始跨度（*#Prim.spans*）和支持跨度（*#Supp. spans*）的数量（后两项仅针对执行并行流的工作负载）。我们报告经过的总周期数（*Total cycles*）及相应的变异系数（CV），两者均根据100 次基线测量的平均值计算得出。我们还报告每项工作负载的每秒周期数（CPS），其计算方法为总周期数除以跨度总数（四舍五入为整数）。

为估算准确率，我们计算相对误差 *(RE)*，使用100 次运行的平均基线和平均补偿周期数，因此准确率为1 – RE（以百分比表示）。RE 等于0 表示准确率达到最高（100）。

我们还在表格底部报告了JEDI和OpenJDK 工作负载的平均准确率。受篇幅限制，两组工作负载的完整评估结果见附录D。

评估结果显示，在 M_1 （92.72%）和 M_2 （92.97%）上取得的平均准确率相当，这表明我们的方法具有良好的可移植性。对于大多数所研究的工作负载，我们都取得了较高的准确率。

在Renaissance 中，原始助记符 （即 M_1 上为87.99%，M_2上为95.18%）、并行助记符 （即 M_1 上为88.27%，M_2上为95.83%）以及scrabble 的准确率最低（即 M_1 上为89.97%，M_2上为97.13%）。准确率下降的原因可能是这些工作负载的每秒周期数（CPS）较低，因此，与 CPS 较高的工作负载相比，执行插桩代码所付出的相对成本更加显著。事实上，我们发现 CPS 与准确率之间存在正相关关系（皮尔逊相关系数 *PCC*： 0.71；统计范围为两台机器上的所有Renaissance 工作负载）。需要指出的是，CPS 与准确率（以及开销，见第 4.3 节）之间的相关性仅适用于只执行流处理的工作负载。我们还注意到，助记符 （即 M_1 上为0.13 ，M_2上为0.06 ）和并行助记符 （即 M_1 上为0.10 ，M_2上为0.12 ）的总周期数变异系数（CV）表明，这些工作负载本身存在固有的变异性，因此我们的评估也会受到影响。

我们强调，对检测开销进行补偿有助于显著提高性能分析准确性。例如，我们测得，原始分析结果（未进行任何补偿）中，mnemonics 的准确率最低仅为 49.11%，par-mnemonics 的准确率最低为 53.61%，而 scrabble 的准确率最低为 65.94%。如表中所示，我们的补偿技术显著提高了性能分析准确性。缺少补偿可能会导致误导性的分析结果，错误地指出一些优化后对应用程序性能影响不大的流，从而凸显了精心调优插桩以及采用扰动补偿技术的重要性。

我们的方法在 JEDI（91.55% 的 M_1 和 90.34% 的 M_2）和 OpenJDK （94.82% 的 M_1 和 96.73% 的 M_2 ）的工作负载中也都实现了较高的平均准确率。
我们没有观察到执行顺序流的工作负载与额外执行并行流的工作负载之间存在显著的准确率差异，这表明执行模式对准确率没有重大影响。

如表3所示，scrabble 执行的周期数多于采用我们两项优化的par-mnemonics版本，但完成所需的时间更少。这是因为scrabble 实现了近乎完美的负载均衡（在 M_1 上的 CV 为0.03 ，在 M_2 上为0.04 ）。此外，我们发现，在 M_1 和 M_2，上，前述优化版本的par-mnemonics中，工作线程经常会阻塞（由于任务同步）；而scrabble则不存在这一情况。

总体而言，我们的评估显示，在 M_1 和 M_2，上的所有已评估工作负载中，平均准确率为92.85%；这表明StreamProf 适合执行周期精确流分析。我们在第5节讨论补偿技术的局限性。

4.3 性能分析开销

本节讨论StreamProf的分析开销，相关数据如表4所示。
开销根据减速因子计算，即将带有性能分析的工作负载执行时间（表中时间列所报告的数值）除以未进行任何性能分析时工作负载的执行时间。报告的数值为 100 次运行的平均值。我们还给出了95%置信区间。

一般而言，性能分析原始助记符 （即，在 M_1 上为1.55× ，在 M_2上为1.46× ）、并行助记符 （即，1.51× 在 M_1

表4 开销评估结果。

基准测试	Ver.	机器	Time [ms]	开销因子	95% 置信区间
mne	orig	M_1	4,994.72	**1.55**	(1.51, 1.60)
		M_2	3,303.64	**1.46**	(1.44, 1.49)
	opt1	M_1	1,200.70	**1.16**	(1.15, 1.16)
		M_2	981.97	**1.12**	(1.12, 1.13)
par-mne	orig	M_1	4,419.53	**1.51**	(1.47, 1.54)
		M_2	2,977.83	**1.41**	(1.38, 1.45)
	opt1	M_1	1,106.22	**1.13**	(1.13, 1.14)
		M_2	905.19	**1.11**	(1.11, 1.12)
	opt2	M_1	880.09	**1.10**	(1.10, 1.11)
		M_2	764.63	**1.09**	(1.08, 1.09)
scr	orig	M_1	310.84	**1.43**	(1.42, 1.44)
		M_2	187.09	**1.23**	(1.22, 1.23)
JEDI		M_1		平均值：**1.13**	
		M_2		平均值：**1.15**	
OpenJDK		M_1		平均值：**1.07**	
		M_2		平均值：**1.06**	

在 M_2上为1.41× ），以及 scrabble （即，在 M_1 上为1.43× ，在 M_2上为1.23× ）。这种开销可以用这些工作负载的每秒周期数较低来解释。事实上，我们发现每秒周期数与开销呈负相关（PCC： −0.96；统计范围为两台机器上的所有Renaissance工作负载）。

我们的方法在JEDI（M_1 上为1.13×，M_2 上为1.15×）和OpenJDK（M_1 上为1.07×，M_2 上为1.06×）的工作负载中，均实现了较低的平均开销。

此外，我们发现，执行顺序流的工作负载与额外执行并行流的工作负载之间，开销没有显著差异，这表明执行模式对开销影响很小。

总体而言，我们的评估结果表明，在 M_1 和 M_2 上的所有受评估工作负载中，StreamProf 的平均开销因子为1.13×。这一性能分析开销使我们的工具能够投入实际使用（如第 3 节所示）。需要说明的是，我们的评估使用的工作负载仅执行流，因此可视为最坏情况。对于并非只执行流的应用程序（这可能是 Java 应用程序中的常见情况），预计性能分析开销会更低（也就是说，对于应用程序中不执行流的阶段，开销可以忽略不计）。

5 讨论

在此，我们讨论了我们的方法的局限性以及评估中的效度威胁 n.

我们技术的局限性。尽管我们估算并补偿了部分成本，但插入的插桩代码可能会影响即时编译器（JIT）的优化、寄存器压力、缓存局部性和线程调度等。这些额外的扰动来源在实践中极难估算和补偿。我们也没有补偿维护位置信息所需的额外周期，因

为在我们的测量中，这对准确率和开销的影响可以忽略不计（低于1%）。尽管如此，我们的工作提出了一种扰动补偿技术，以提高 JVM 上周期分析的准确率。

使用我们的技术生成的分析结果依赖于平台。底层硬件（例如，使用不同的处理器）或软件（例如，使用不同的 Java 类库）发生变化时，需要重新运行分析，以生成适用于新环境的分析结果。我们还注意到，我们的方法依赖于每线程虚拟化参考周期计数器的可用性。

我们在考虑 OpenJDK [46] 中 Stream API 的实现后，推导出了插桩逻辑。由于 API 的设计，我们的插桩需要针对 Java 类库中的私有类。私有类取决于具体实现，因此，要让我们的插桩适用于不同的 Java 类库，可能需要进行修改，不过我们可以轻松地完成适配。此外，我们已验证，从 Java 版本 8 （引入 Stream API 的版本）到版本17 （撰写本文时最新的长期支持版本），我们的插桩都是正确的。

我们将流定义为 AbstractPipeline 的子类型；该类实现了 Stream API 的顶层接口，即 java.util.stream.BaseStream。这些类为所有流执行方法提供入口和出口，StreamProf 会识别并进行性能剖析。我们排除了仅实现接口BaseStream 而未继承 AbstractPipeline 的实体，因为 BaseStream 不提供此类入口点。因此，如果没有用户提供的信息，就无法定位流执行方法。若缺少流执行方法的信息，StreamProf 便无法定位有问题的流执行。为求完整，作为未来工作，我们计划加入相关方法，以便对实现BaseStream 但未继承AbstractPipeline的类的所有流执行方法进行性能剖析。

我们的技术仅跟踪流执行周期，忽略操作系统中的线程调度开销。因此，我们目前无法测量跨度执行期间发生的线程上下文切换次数。若某个跨度发生的上下文切换次数相对于其周期数而言较高，则可能表明资源使用效率低下。

只要目标应用程序终止，我们的技术就能保证生成完整的流性能剖析数据。对持续运行的应用程序进行性能剖析时，只能根据每次数据转储之前已结束的跨度，重建部分流性能剖析数据。

评估中的效度威胁。我们的准确率评估侧重于所有已分析跨度的补偿周期总数（见第 4 节），因此各个跨度的准确率仍然未知。不幸的是，评估跨度级别的准确率颇具挑战，可能需要专用硬件。

我们的方法所取得的结果在其他平台上可能并不成立。我们在两台不同的机器上开展实验（见第 3 节和第 4 节），以确认检测到的问题（以及取得的加速比）在不同环境下均成立。这表明，本文报告的性能问题并非只存在于某个平台，而是与所研究的工作负载有关。

周期测量结果存在波动。为缓解这一问题，我们对所有评估均进行了100 次运行，并报告平均值，以提供更可靠的估计。

6 相关工作

这里，我们首先讨论与 Java 流的使用和优化相关的研究，然后回顾分析运行于 JVM 上的应用程序多个方面的工作。

Java 流的使用和优化。一些作者研究了实际项目中流的使用情况。Tanaka 等人 [47] 挖掘了100 个软件仓库，以研究 Lambda 表达式、流和java.util.Optional 类的使用情况。Khatchadourian 等人 [5]考察了34 个 Java 项目，以研究 Java 中流的使用情况。这项工作首次尝试专门发现 Stream API 的使用场景和误用情况。Nostas 等人 [6]研究了610 个项目中流的使用情况。这项工作部分复现了 Khatchadourian 等人的研究，考察了更多项目，并验证了此前得出的结果。Rosales 等人 [48] 对132,211 个使用 Stream API 的开源项目进行了大规模研究，基本证实了上述工作的发现。

已有研究 [49] 论证了动态分析用于 Java 流性能剖析的适用性。据我们所知，仅有六项研究关注流的优化。Ishizaki 等人 [14] 修改了 IBM J9 JVM [50] 和 Testarossa 编译器 [51]，以将流代码转换为经过优化的 GPU 代码。Hayashiet 等人 [15] 在此基础上进行了扩展，提出了一种监督式机器学习方法，使运行时能够在 CPU 和 GPU 之间进行选择，从而加速流执行。与我们的工作不同，上述方法需要手动检查代码（并了解 Stream API），主要通过编译器和硬件相关的支持来实现并行流的自动优化。然而，我们的技术并不局限于专有编译器，而是能够优化运行在常用 JVM（例如 HotSpot [52] 和 GraalVM [53]）上的应用程序中的流。此外，这两种方法都只能优化通过终端操作 forEach [54]执行的、基于整数的并行流。相比之下，StreamProf 可帮助开发者优化并行流和顺序流，无论其属于基本类型还是基于对象，并且无论通过何种终端操作执行。不同于上述两种方法，我们还能够在不依赖特殊 CPU/GPU 支持的情况下加速流处理。

Khatchadourian 等人 [10, 11] 引入了一款 Eclipse 插件，帮助开发者更好地编写流代码。正如作者所指出的，他们的方法无法权衡输入规模与开销，评估流的优化效果 [11]。该插件主要依赖静态分析技术，可能无法充分捕捉所分析流的运行时行为。相比之下，我们的技术首次测量了流执行的计算。我们认为，Khatchadourian 等人的工作与我们的工作互为补充，因为该插件可以自动建议安全的流水线重构，从而更好地利用流。

Møller 等人 [9] 利用字节码级转换消除顺序流的抽象开销。然而，这种方法无法分析并行流。Basso 等人 [13] 利用字节码级转换，专门消除并行流的抽象

开销。遗憾的是，这两种方法都忽略了运行时信息，而这些信息对于发现与流相关的性能问题至关重要。我们认为上述方法与我们的工作互为补充，因为我们的分析结果所支持的优化流处理可能会进一步受益于字节码级转换。

JVM 的分析与性能分析器。 </style}多位作者研究了 JVM 上应用程序的动态行为 [17, 18, 20, 55–58]。软件精简技术 [59,60] 虽然可以移除 Java 应用程序中未使用的字段，但第 3.2 节所述的优化针对的是递归代码中实际使用的一个字段和一个参数（通过流处理）。一些方法 [61–63] 专注于调试大数据系统（例如 Apache Spark [64]），这些系统与 Stream API 类似，能够进行类似 MapReduce 的 [7] 数据转换。这些研究针对的是分布式系统，而 Java 流则在单个 JVM 进程中执行。最后，还有许多通用的 JVM 性能分析器 [16, 19, 65–70]。上述研究或工具均未专门涉及 Java 流的周期精确剖析。

7 结论

本文介绍了一种针对 Java 流的新型性能剖析技术。该技术以经过的参考周期为单位，测量顺序流和并行流所执行的计算，准确捕捉流嵌套，并生成便于检测和优化流相关性能问题的可操作性能剖析结果。StreamProf 为我们优化当前最先进基准测试套件 Renaissance 中两个基于流的工作负载提供了指导，使加速比最高达到 5× 倍。

我们的方法补偿了执行插入的插桩代码所导致的额外周期，即我们估算出这些周期并将其移除，从而提高性能分析准确性。我们的评估结果表明，补偿后的分析结果平均准确率为92.85%。与朴素的周期分析相比，我们的补偿技术显著提高了准确率（例如，对于助记符，准确率从未补偿时的49.11%提高到补偿后的87.99%）。

作为未来工作的一部分，我们计划将性能分析器应用于一些开源软件项目，并研究与流相关的优化机会。我们还计划证明该技术对于面向 Java [71–73] 及其他编程语言 [74–76] 的第三方流库的普适性。我们计划测量每个跨度中的线程上下文切换，以帮助揭示流执行中的低效资源使用情况。此外，我们计划支持对流性能剖析数据进行更多可视化，例如使用饼图展示执行并行流的工作线程之间的工作分配。我们还计划将StreamProf 作为开源软件发布。

数据可用性声明 相关工件 [77] 由一个 Docker 镜像组成，可用于生成本文中的表格（不包括附录）。

致谢 本研究得到 Oracle（ERO 项目 1332）和瑞士国家科学基金会（项目 200020_188688）的支持。

A 插桩逻辑

这里，我们详细介绍用于检测第 2.1 节所述流执行的插桩逻辑。清单 2 使用数据结构和基于 Python 的伪代码描述了我们的方法，力求简洁。我们还使用了 DiSL 提供的特殊变量 [31]，下文将对此进行说明。

■ 清单 2 以伪代码表示的插桩逻辑。 1 # 位置 2 3 # 用于维护方法 ID 的变量 4

线程局部变量。 我们的插桩需要存储与当前线程相关的信息。为此，我们利用 DISL 提供的线程局部变量，它可以将信息高效地存储在一个额外的实例字段中，该字段属于表示当前线程的对象。在 DiSL 中，线程局部变量通过@ThreadLocal 注解声明。

合成局部变量。 我们的插桩需要在织入同一方法体的不同代码片段之间传递信息。为此，我们使用 DiSL 提供的合成局部变量，它们被高效地嵌入 Java 调用栈的栈帧中，从而避免堆分配。在 DiSL 中，使用@SyntheticLocal 注解声明合成局部变量。

代码片段。 我们使用 DiSL 支持的代码片段来编写插桩。请注意，每个代码片段前面都有一个注解（例如 @Before）和一个谓词（例如StreamExecutionMethod），两者共同指定插桩代码的插入位置。例如，上述组合会将插桩代码插入每个流执行方法的开头。

B 跨度重构算法

这里，我们详细介绍第 2.1 节所述离线后处理中使用的跨度重建算法。为便于理解，我们在列表 3 中使用基于 Python 的伪代码简洁地描述了我们的方法。

■ 列表 3 以伪代码表示的跨度重建算法。 1 # 数据类型： 2 3 # 表示性能分析器收集的跨度
开
开
int
周
始

```
26 int 嵌套_原始_跨度 27 Span parent = None # 仅在同一线程内有效 28 Span 原始 = None #
此
ID,
me
OC
ret
* C
返
pa
程_
理
表
 =d
始
74
名
```

```
79 case SSB: # 支持跨度开始 80 stack.push(Span(event.cycles, event.streamID,
eve
stac
SE:
emp
par
par
app
else
应线
所有
性能
造函
111
并行
117
行流
spa
find
spa
算外
leve
```

```
132 if span.nesting_level  = -1: 133 if span.outer_span: 134 span.nesting_level = compute_
nes
retu
140
143
146
149
Thr
跟踪
省略
eve
eve
prir
```

C 热图

这里展示 StreamProf 报告的定制热图，以指导第 3 节所述的优化。

图 3 展示了 StreamProf 根据为助记符生成的分析结果制作的两张定制热图。[14] 我们使用定制热图浓缩性能剖析所得的信息（例如，流嵌套信息、补偿周期数、流执行次数以及热点区域的存在情况），帮助开发者分析流处理。

第一个热图（图 3a）对应原始（未修改）版本的执行情况。该热图将流划分为若干组。在 x轴上，根据StreamProf报告的（补偿后）周期数对流进行分组。在 y 轴上，根据流的嵌套层级进行分组。助记符 包含44 个嵌套层级，我们

[14]我们仅展示在 M_1，上针对助记符获得的分析结果，这些结果与在 M_2 上获得的结果非常相似。

将它们按10 个一组（从0开始）汇总，以便于可视化。每个单元格中显示的数字表示给定嵌套层级和周期范围内已执行流的数量。与传统热图不同，单元格的颜色表示该组中所有流的总周期数。颜色越深，该组流所消耗的总周期数越多。

如图 3a 所示，在不同嵌套层级上，有大量流执行对 助记符中性能剖析得到的周期数有显著贡献。在我们的方法中，我们关注热图所示的较深色区域，因为减少这些区域内周期数的优化很可能提升性能。为此，StreamProf 还会补充报告占据大部分流处理的流的位置（如第 3 节所示）。

图 3b 展示了经过优化后的工作负载热图。对比图 3a 和图 3b 可以直观看出，移除两个已分析热点位置中的问题流，避免了优化版本中的不必要且代价高昂的计算。

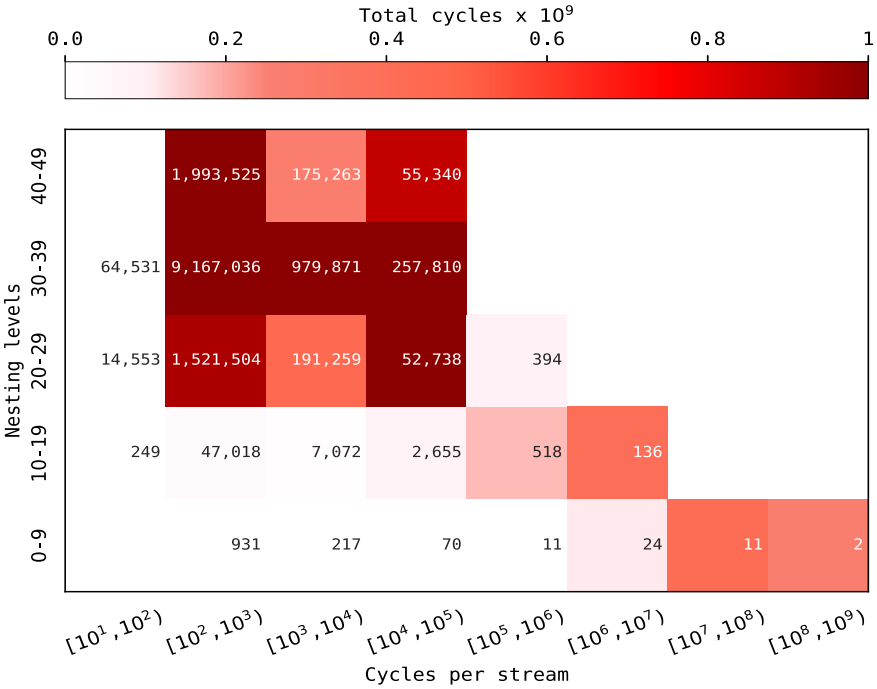
我们注意到，图 3a 和图 3b 所报告的各类别流执行次数略有差异，这是由于我们的实验所涉及的基准测试执行存在波动（如第 5 节所述）。因此，在不同的基准测试运行中，少量流执行可能会被归入不同的类别（通常是相邻类别）。

D 补充评估结果

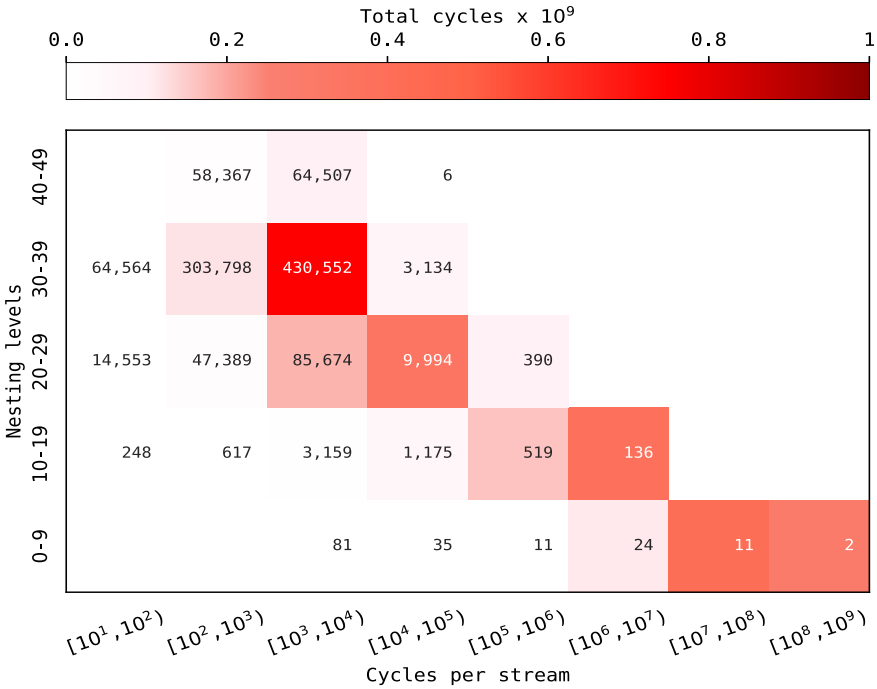
这里展示了第 4 节中讨论的 JEDI 和OpenJDK工作负载的准确率和开销评估完整结果。我们在本附录所有表格的底部报告了平均值。

我们注意到，OpenJDK 的大多数工作负载运行时间都很短，因为它们的设计目的只是测试 Stream API 的特定功能。

由于篇幅限制，我们在表 7 和表 8 中使用简称来标识每个OpenJDK 工作负载。OpenJDK18 开源代码库 [44] 中使用的原始名称如下：w1：AnyMatcher，w2：AllMatcher，w3：Decomposition.saturated，w4：IntegerSum，w5：PrimesSieve，w6：IntegerMax，w7：IntegerDuplicate，w8：PrimesFilter.t100，w9-par：Decomposition.thinktime_par，w10-par：IntegerSum_par，w11-par：IntegerMax_par，w12-par：IntegerDuplicate_par，w13-par：PrimesFilter.t10000_par。



(a)原始版本



(b)优化版本

■ **图3** 助记符中的流执行优化前后对比。每个单元格显示相应类别中已执行流的数量，单元格颜色则对应这些流的总周期数。

表 5 基于 JEDI 工作负载的准确性评估结果。

基准测试	机器	#匿名跨度	#原始跨度	#支持跨度	总周期数 AVG	CV	CPS	准确率 [%]
q1	M_1	2			1,721,311,380	0.01	860,655,690	**92.82**
	M_2	2			2,383,530,808	0.09	1,191,765,404	**93.31**
q1-par	M_1	1	1	31	2,394,566,469	0.01	72,562,620	**90.13**
	M_2	1	1	62	2,809,949,167	0.09	43,905,456	**91.63**
q2	M_1	10			370,840,284	0.02	37,084,028	**92.93**
	M_2	10			491,430,575	0.07	49,143,058	**98.01**
q3	M_1	4			1,801,202,558	0.10	450,300,640	**91.18**
	M_2	4			2,660,667,317	0.10	665,166,829	**92.15**
q3-par	M_1	1	3	93	2,607,513,919	0.04	26,881,587	**88.51**
	M_2	1	3	140	2,981,862,225	0.06	20,707,377	**87.41**
q4	M_1	3			1,235,565,394	0.03	411,855,131	**89.36**
	M_2	3			1,638,539,192	0.09	546,179,731	**83.71**
q5	M_1	7			1,936,425,328	0.01	276,632,190	**95.27**
	M_2	7			2,054,445,608	0.03	293,492,230	**81.71**
q6	M_1	1			567,249,098	0.01	567,249,098	**90.30**
	M_2	1			870,871,942	0.08	870,871,942	**90.45**
q7	M_1	31			1,865,036,543	0.01	60,162,469	**82.09**
	M_2	31			2,476,180,358	0.10	79,876,786	**84.36**
q8	M_1	9			2,161,629,272	0.02	240,181,030	**96.90**
	M_2	9			2,439,844,650	0.03	271,093,850	**96.89**
q9	M_1	7			3,229,724,299	0.06	461,389,186	**97.66**
	M_2	7			3,574,183,258	0.04	510,597,608	**91.36**
q10	M_1	1,685,944			2,513,228,294	0.01	1,491	**83.84**
	M_2	1,685,944			3,156,714,042	0.05	1,872	**84.67**
q11	M_1	7			272,898,418	0.01	38,985,488	**99.49**
	M_2	7			401,778,975	0.08	57,396,996	**99.35**
q12	M_1	3			1,249,998,343	0.01	416,666,114	**89.97**
	M_2	3			1,861,945,683	0.11	620,648,561	**92.91**
q13	M_1	4			2,731,782,055	0.02	682,945,514	**92.42**
	M_2	4			3,001,892,492	0.01	750,473,123	**85.93**
q14	M_1	200,002			1,147,485,490	0.03	5,737	**87.10**
	M_2	200,002			1,434,930,717	0.09	7,175	**86.87**
q15	M_1	6			759,864,247	0.10	126,644,041	**84.65**
	M_2	6			1,115,776,450	0.14	185,962,742	**92.10**
q16	M_1	4			315,863,876	0.04	78,965,969	**98.16**
	M_2	4			387,515,417	0.05	96,878,854	**99.09**
q17	M_1	4			4,917,510,274	0.01	1,229,377,568	**98.59**
	M_2	4			5,356,269,858	0.02	1,339,067,465	**95.39**
q18	M_1	6			3,521,113,897	0.02	586,852,316	**95.21**
	M_2	6			6,800,786,817	0.02	1,133,464,469	**87.52**
q18-par	M_1	2	4	124	5,740,561,404	0.01	44,158,165	**95.16**
	M_2	2	4	202	6,800,786,817	0.02	32,696,090	**97.52**
q19	M_1	214,379			1,505,820,737	0.08	7,024	**88.35**
	M_2	214,379			1,829,283,858	0.06	8,533	**86.41**
q20	M_1	553,216			2,387,540,569	0.01	4,316	**88.78**
	M_2	553,216			3,121,469,875	0.06	5,642	**85.40**
q21	M_1	322,633			5,031,578,786	0.05	15,595	**85.30**
	M_2	322,633			6,503,770,125	0.06	20,158	**82.26**
q22	M_1	5			345,374,176	0.03	69,074,835	**94.49**
	M_2	5			476,944,467	0.05	95,388,893	**92.09**
平均值	M_1							平均值：**91。55**
	M_2							平均值：**90。34**

■ 表 6 JEDI工作负载的开销评估结果。

基准测试	机器	Time [ms]	开销因子	95%置信区间
q1	M_1	639.71	**1.06**	(1.05, 1.08)
	M_2	795.27	**1.06**	(1.05, 1.07)
q1-par	M_1	113.82	**1.07**	(1.06, 1.09)
	M_2	115.71	**1.06**	(1.05, 1.07)
q2	M_1	137.70	**1.07**	(1.06, 1.08)
	M_2	163.99	**1.02**	(1.01, 1.03)
q3	M_1	669.49	**1.18**	(1.17, 1.19)
	M_2	887.74	**1.16**	(1.15, 1.17)
q3-par	M_1	154.05	**1.21**	(1.20, 1.22)
	M_2	146.27	**1.22**	(1.21, 1.23)
q4	M_1	459.69	**1.23**	(1.22, 1.25)
	M_2	546.71	**1.30**	(style id='1'>(1.29, 1.31)
q5	M_1	718.53	**1.03**	(1.01, 1.04)
	M_2	685.46	**1.22**	(1.21, 1.23)
q6	M_1	211.11	**1.09**	(1.07, 1.10)
	M_2	290.61	**1.10**	(1.09, 1.11)
q7	M_1	693.14	**1.28**	(1.27, 1.30)
	M_2	826.27	**1.33**	(1.32, 1.34)
q8	M_1	802.20	**1.01**	(1.01, 1.02)
	M_2	814.05	**1.02**	(1.01, 1.03)
q9	M_1	1198.78	**1.02**	(1.01, 1.03)
	M_2	1192.47	**1.09**	(1.08, 1.11)
q10	M_1	933.67	**1.35**	(1.34, 1.36)
	M_2	1053.26	**1.35**	(1.34, 1.36)
q11	M_1	101.35	**1.04**	(1.03, 1.05)
	M_2	134.15	**1.01**	(1.01, 1.03)
q12	M_1	465.01	**1.13**	(1.12, 1.14)
	M_2	621.46	**1.07**	(1.06, 1.08)
q13	M_1	1013.51	**1.08**	(1.07, 1.09)
	M_2	1001.57	**1.16**	(1.15, 1.17)
q14	M_1	426.88	**1.29**	(1.28, 1.32)
	M_2	478.98	**1.30**	(1.29, 1.31)
q15	M_1	282.71	**1.15**	(1.14, 1.16)
	M_2	372.36	**1.10**	(1.09, 1.11)
q16	M_1	117.30	**1.05**	(1.04, 1.06)
	M_2	129.84	**1.02**	(1.01, 1.03)
q17	M_1	1824.83	**1.03**	(1.02, 1.04)
	M_2	1787.06	**1.14**	(1.13, 1.15)
q18	M_1	1307.56	**1.05**	(1.04, 1.06)
	M_2	496.35	**1.15**	(1.14, 1.17)
q18-par	M_1	480.67	**1.07**	(1.06, 1.08)
	M_2	496.35	**1.05**	(1.04, 1.06)
q19	M_1	559.79	**1.21**	(1.20, 1.23)
	M_2	610.61	**1.21**	(1.20, 1.22)
q20	M_1	886.50	**1.24**	(1.23, 1.25)
	M_2	1041.46	**1.25**	(1.24, 1.27)
q21	M_1	1869.16	**1.24**	(1.23, 1.25)
	M_2	2170.20	**1.25**	(1.24, 1.26)
q22	M_1	128.28	**1.07**	(1.05, 1.09)
	M_2	159.24	**1.11**	(1.10, 1.12)
平均值	M_1		平均值：**1.13**	
	M_2		平均值：**1.15**	

■ **表 7** 基于 OpenJDK 工作负载的准确性评估结果。

基准测试	机器	#匿名跨度	#原始跨度	#支持跨度	总周期数 AVG	CV	CPS	准确率 [%]
w1	M_I	1			879,573	0.04	879,573	**93.15**
	M_2	1			753,280	0.05	753,280	**95.05**
w2	M_I	1			1,242,508	0.01	1,242,508	**94.40**
	M_2	1			1,078,703	0.01	1,078,703	**94.40**
w3	M_I	1			7,240,921	0.01	7,240,921	**98.68**
	M_2	1			6,271,850	0.01	6,271,850	**99.78**
w4	M_I	1			135,089,803	0.01	135,089,803	**98.39**
	M_2	1			152,494,128	0.01	152,494,128	**99.50**
w5	M_I	1			160,450,498	0.10	160,450,498	**99.21**
	M_2	1			151,542,390	0.09	151,542,390	**99.53**
w6	M_I	1			216,617,925	0.01	216,617,925	**91.78**
	M_2	1			197,130,665	0.01	197,130,665	**95.54**
w7	M_I	10,485,761			2,491,458,725	0.13	238	**94.20**
	M_2	10,485,761			2,326,973,493	0.14	222	**92.74**
w8	M_I	1			3,038,972,058	0.01	3,038,972,058	**99.89**
	M_2	1			2,857,904,263	0.01	2,857,904,263	**99.99**
w9-par	M_I		1	31	9,752,180	0.14	304,756	**89.63**
	M_2		1	38	9,280,780	0.12	237,969	**88.07**
w10-par	M_I		1	31	166,038,120	0.07	5,188,691	**86.36**
	M_2		1	31	137,748,443	0.05	4,304,639	**97.85**
w11-par	M_I		1	31	202,450,863	0.04	6,326,589	**93.29**
	M_2		1	31	131,721,768	0.03	4,116,305	**98.44**
w12-par	M_I		1	31	337,715,623	0.05	10,553,613	**93.67**
	M_2		1	31	233,671,930	0.05	7,302,248	**97.03**
w13-par	M_I		1	35	3,041,238,569	0.01	84,478,849	**99.99**
	M_2		1	35	2,850,698,723	0.01	79,186,076	**99.52**
平均值	M_I							平均值：**94.82**
	M_2							平均值：**96.73**

表 8 OpenJDK工作负载的开销评估结果。

基准测试	机器	Time [ms]	开销因子	95%置信区间
w1	M_1	0.32	**1.01**	(1.01, 1.02)
	M_2	0.24	**1.05**	(1.04, 1.06)
w2	M_1	0.45	**1.02**	(1.01, 1.03)
	M_2	0.35	**1.02**	(1.01, 1.04)
w3	M_1	2.68	**1.01**	(1.01, 1.03)
	M_2	2.10	**1.01**	(1.01, 1.02)
w4	M_1	50.23	**1.02**	(1.01, 1.03)
	M_2	50.90	**1.02**	(1.01, 1.03)
w5	M_1	66.43	**1.01**	(1.01, 1.02)
	M_2	50.58	**1.01**	(1.01, 1.02)
w6	M_1	80.38	**1.02**	(1.01, 1.04)
	M_2	65.80	**1.02**	(1.02, 1.03)
w7	M_1	829.77	**1.37**	(1.35, 1.38)
	M_2	769.94	**1.33**	(1.31, 1.34)
w8	M_1	1126.32	**1.01**	(1.01, 1.03)
	M_2	953.05	**1.01**	(1.01, 1.02)
w9-par	M_1	0.75	**1.07**	(1.05, 1.08)
	M_2	0.77	**1.11**	(1.09, 1.12)
w10-par	M_1	8.82	**1.35**	(1.33, 1.36)
	M_2	6.64	**1.22**	(1.21, 1.23)
w11-par	M_1	10.65	**1.03**	(1.03, 1.04)
	M_2	6.30	**1.02**	(1.02, 1.03)
w12-par	M_1	17.87	**1.03**	(1.02, 1.04)
	M_2	11.02	**1.01**	(1.01, 1.02)
w13-par	M_1	238.15	**1.01**	(1.01, 1.03)
	M_2	194.82	**1.01**	(1.01, 1.02)
平均值	M_1		**1.07**	
	M_2		**1.06**	

参考文献

[1] Oracle。《java.util.stream 包》。
h
tt
ps://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/stream/Stream.html。2022年。（访问日期：2022年12月30日）。[2] Richard Bird 和 Philip Wadler。《函数式编程导论》。第1版。GBR：Prentice Hall International (UK) Ltd.，1988。ISBN：0134841891。[3] Joshua Bloch。《Effective Java》。第3版。Addison-Wesley，2018。ISBN：978-0-13-468599- 1。[4] Davood Mazinanian、Ameya Ketkar、Nikolaos Tsantalis 和 Danny Dig。“理解 Java 中 Lambda 表达式的使用”。载于：*Proc. ACM Program.Lang.* 1.OOPSLA（2017年10月），第1–31页。ISSN：2475-1421。DOI：10.1145/3133909。[5] Raffi Khatchadourian、Yiming Tang、Mehdi Bagherzadeh 和 Baishakhi Ray。“Java 8 流的使用与误用实证研究”。载于：形式化方法与软件工程国际会议。2020年，第97–118页。DOI：10.1007/978-3-030-45234-6_5。

[6] Joshua Nostas、Juan Pablo Sandoval Alcocer、Diego Elias Costa 和 Alexandre Bergel。“开发者如何使用 Java 流 API？”载于：*ICCSA*。2021年，第323–335页。DOI：10.1007/978-3-030-87007-2_23。[7] Jeffrey Dean 和 Sanjay Ghemawat。“MapReduce：大型集群上的简化数据处理”。载于：*Commun. ACM* 51.1（2008年），第107–113页。DOI：10.1145/1327452.1327492。[8] Aggelos Biboudis、Nick Palladinos 和 Yannis Smaragdakis。《Lambda 之争》。2014年。DOI：10.48550/arXiv.1406.6631。[9] Anders Møller 和 Oskar Haarklou Veileborg。“通过提前程序优化消除 Java 流管道的抽象开销”。载于：*Proc. ACM Program. Lang.* 4.OOPSLA（2020年11月）。DOI：10.1145/3428236。[10] Raffi Khatchadourian、Yiming Tang、Mehdi Bagherzadeh 和 Syed Ahmed。“通过自动重构优化 Java 8 流软件的工具”。载于：*SCAM*。2018年。DOI：10.1109/SCAM.2018.00011。[11] Raffi Khatchadourian、Yiming Tang、Mehdi Bagherzadeh 和 Syed Ahmed。“面向 Java 8 流智能并行化的安全自动重构”。载于：*ICSE*。2019年，第619–630页。DOI：10.1109/ICSE.2019.00072。[12] Eclipse 基金会。《Eclipse 集成开发环境》。http://www.eclipse.org/ide。2021年。（访问日期：2022年12月30日）。[13] Matteo Basso、Filippo Schiavio、Andrea Rosà 和 Walter Binder。“优化并行 Java 流”。载于：*ICECCS*。2022年，第23–32页。DOI：10.1109/ICECCS54210.2022.00012。[14] Kazuaki Ishizaki、Akihiro Hayashi、Gita Koblents 和 Vivek Sarkar。“为 GPU 执行编译和优化 Java 8 程序”。载于：*PACT*。2015年，第419–431页。DOI：10.1109/PACT.2015.46。[15] Akihiro Hayashi、Kazuaki Ishizaki、Gita Koblents 和 Vivek Sarkar。“基于机器学习的运行时 CPU/GPU 选择性能启发式方法”。载于：*PPPJ*。2015年，第27–36页。DOI：10.1145/2807426.2807429。[16] Laksono Adhianto、Sarbani Banerjee、Mike Fagan、Mark Krentel、Gabriel Marin、John Mellor-Crummey 和 Nathan Tallent。“HPCTOOLKIT：用于分析优化并行程序性能的工具”。载于：*Concurr. Comput.:Pract.Exper.*22.6（2010年），第685–701页。ISSN：1532-0626。[17] Florian David、Gael Thomas、Julia Lawall 和 Gilles Muller。“使用 Free-lunch 分析器持续测量临界区压力”。载于：*OOPSLA*。2014年，第291–307页。DOI：10.1145/2714064.2660210。[18] Aibek Sarimbekov、Andreas Sewe、Walter Binder、Philippe Moret 和 Mira Mezini。“JP2：面向 Java 虚拟机的感知调用点的调用上下文分析”。载于：*Sci. Comput. Program.*79（2014年），第146–157页。DOI：10.1016/j.scico.2011.11.003。

[19] Qiming Teng、Haichuan Wang、Xiao Zhong、Peter Sweeney 和 Evelyn Duester- wald。“THOR：面向运行于多核系统上的 Java 应用程序的性能分析工具”。载于：IBM《研究与开发期刊》54.5（2010），4:1–4:17。doi：10.1147/JRD.2010.2058481。[20] Peter Hofer、David Gnedt、Andreas Schörgenhumer 和 Hanspeter Mössenböck。“在虚拟机层面高效跟踪并灵活分析 Java 应用程序中的锁竞争”。载于：ICPE。2016，第 263–274 页。doi：10.1145/2851553.2851559。[21] Andrea Rosà、Eduardo Rosales 和 Walter Binder。“分析并优化 JVM 上的任务粒度”。载于：CGO。2018，第 27–37 页。doi：10.1145/3168828。[22] Todd Mytkowicz、Amer Diwan、Matthias Hauswirth 和 Peter Sweeney。“评估 Java 性能分析器的准确率”。载于：PLDI。2010，第 187–197 页。doi：10.1145/1806596.1806618。[23] Oracle。Stream 接口<V>。
htt
ps://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/stream/Stream.html。2022。（访问日期：2022年12月30日）。[24] Aleksandar Prokopec、Andrea Rosà、David Leopoldseder、Gilles Duboscq、Petr Tůma、Martin Studener、Lubomír Bulej、Yudi Zheng、Alex Villazón、Doug Simon、Thomas Würthinger 和 Walter Binder。“Renaissance：面向 JVM 并行应用程序的基准测试套件”。载于：PLDI。2019，第 31–47 页。doi：10.1145/3314221.3314637。[25] Doug Lea。“Java 分叉/合并框架”。载于：JAVA。2000，第 36–43 页。doi：10.1145/337449.337465。[26] Oracle。ForkJoinPool 类。
h
t
t
p
s://
d
o
c
s.oracle.co
m/en/java/javase/19/docs/api/java.base/java/util/concurrent/ForkJoinPool.html。2022。（访问日期：2022年12月30日）。[27] Warren Burton 和 Ronan Sleep。“在处理器虚拟树上执行函数式程序”。载于：FPCA（1981），第 187–194 页。doi：10.1145/800223.806778。[28] Matteo Frigo、Charles Leiserson 和 Keith Randall。“Cilk-5 多线程语言的实现”。载于：PLDI。1998，第 212–223 页。doi：10.1145/277650.277725。[29] Achille Peternier、Walter Binder、Akira Yokokawa 和 Lydia Chen。“多线程应用程序的并行性分析与墙钟时间预测”。载于：ICPE。2013，第 211–216 页。doi：10.1145/2479871.2479901。[30] Walter Binder、Jarle Hulaas 和 Philippe Moret。“高级 Java 字节码插桩”。载于：PPPJ。2007，第 135–144 页。doi：10.1145/1294325.1294344。[31] Lukáš Marek、Alex Villazón、Yudi Zheng、Danilo Ansaloni、Walter Binder 和 Zhengwei Qi。“DiSL：一种用于字节码插桩的领域特定语言”。载于：AOSD。2012，第 239–250 页。doi：10.1145/2162049.2162077。[32] Oracle。Java 虚拟机工具接口（JVM TI）。
https://docs.oracle.com/javase/8/docs/technotes/guides/jvmti。2022。（访问日期：2022年12月30日）。

[33] ICL。PAPI。http://icl.utk.edu/papi。2021。（访问日期：2022年12月30日）。[34] Stephen Blackburn、Robin Garner、Chris Hoffmann、Asjad Khang、Kathryn McKinley、Rotem Bentzur、Amer Diwan、Daniel Feinberg、Daniel Frampton、Samuel Guyer、Martin Hirzel、Antony Hosking、Maria Jump、Han Lee、Eliot Moss、Aashish Phansalkar、Darko Stefanović、Thomas VanDrunen、Daniel von Dincklage 和 Ben Wiedermann。“DaCapo 基准测试：Java 基准测试的开发与分析”。载于：OOPSLA。2006，第 169–190 页。doi：10.1145/1167515.1167488。[35] Andreas Sewe、Mira Mezini、Aibek Sarimbekov 和 Walter Binder。“Da Capo Con Scala：面向 Java 虚拟机的 Scala 基准测试套件的设计与分析”。载于：OOPSLA。2011，第 657–676 页。doi：10.1145/2048066.2048118。[36] 标准性能评估公司。SPECjvm2008。https://www.spec.org/jvm2008。2020。（访问日期：2022年12月30日）。[37] Renaissance 套件。文档概览。https://renaissance.dev/docs。2019。（访问日期：2022年12月30日）。[38] Oracle。JDK 17 文档。https://docs.oracle.com/en/java/javase/17。2022。（访问日期：2022年12月30日）。[39] Doug Lea。Java 并发编程。第二版：设计原则与模式。第 2 版。Addison-Wesley Professional，1999年。isbn：978-0-20-131009-2。[40] Oracle。ForkJoinTask 类<V>。
h
t
t
p
s://
d
o
c
s.
oracle.com/en/java/javase/19/docs/api/java.base/java/util/concurrent/ForkJoinTask.html。2022。（访问日期：2022年12月30日）。[41] Marzio Lunghi 和 Filippo Schiavio。Java 中声明式与命令式查询的评估。https://github.com/usi-dag/JEDI。2022。（访问日期：2022年12月30日）。[42] Filippo Schiavio、Andrea Rosà 和 Walter Binder。“借助 S2S 将 SQL 转换为 Stream：Java 流 API 的自动基准测试生成器”。载于：GPCE。2022，第 179–186 页。doi：10.1145/3564719.3568699。[43] TPC。TPC-H 版本 2 和版本 3。https://www.tpc.org/tpch。2022。（访问日期：2022年12月30日）。[44] OpenJDK 社区。OpenJDK/JDK18u。
h
t
t
p
s://
github.com/openjdk/jdk18u/tree/master/test/micro/org/openjdk/bench/java/util/stream。2019。（访问日期：2022年12月30日）。[45] OpenJDK。JMH - Java 微基准测试框架。http://openjdk.java.net/projects/code-tools/jmh。2020。（访问日期：2022年12月30日）。[46] Oracle。OpenJDK。http://openjdk.java.net。2022。（访问日期：2022年12月30日）。[47] Tanaka Hiroto、Matsumoto Shinsuke 和 Kusumoto Shinji。“Java 函数式惯用法现状研究”。载于：IEICE Transactions on Information and Systems E102.D.12（2019），第 2414–2422 页。doi：10.1587/transinf.2019MPP0002。

[48] Eduardo Rosales、Andrea Rosà、Matteo Basso、Alex Villazón、Adriana Orellana、Ángel Zenteno、Jhon Rivero 和 Walter Binder。《真实环境中的 Java 流特征分析》。载于：ICECCS。2022年，第 143–152 页。doi：10.1109/ICECCS54210.2022. 00025。[49] Eduardo Rosales、Andrea Rosà 和 Walter Binder。《Java 虚拟机上的流性能分析》。载于： <Programming>。2020，第 27–30 页。doi：10.1145/3397537. 3397565。[50] Colin Renouf。《面向 Java 6 的 IBM J9 Java 虚拟机》。载于：Pro IBM WebSphere Application Server 7 Internals。Apress，2009年，第 15–34 页。[51] Nikola Grcevski、Allan Kielstra、Kevin Stoodley、Mark Stoodley 和 Vijay Sundaresan。《面向服务器和中间件应用的 Java 即时编译器与虚拟机改进》。载于：VM'04：第 3 届虚拟机研究与技术研讨会论文集——第 3 卷。USENIX 协会，2004年，第 12–12 页。[52] Oracle。《Java SE HotSpot 概览》。https://www.oracle.com/java/technologies/ javase-jsp.html。2022年。（访问日期：2022年12月30日）。[53] OpenJDK：Graal 项目。Graal。http://openjdk.java.net/projects/graal。2022年。（访问日期：2022年12月30日）。[54] Oracle。IntStream<T>接口。https://docs.oracle.com/en/java/javase/19/ docs / api / java . base / java /util/ stream /IntStream . html。2022年。（访问日期：2022年12月30日）。[55] Bruno Dufour、Karel Driesen、Laurie Hendren 和 Clark Verbrugge。《Java 动态度量指标》。载于：OOPSLA。2003年，第 149–168 页。doi：10.1145/949305.949320。[56] Tomas Kalibera、Matthew Mole、Richard Jones 和 Jan Vitek。《理解 DaCapo 中并发性的黑盒方法》。载于：OOPSLA。2012年，第 335–354 页。doi：10.1145/2384616.2384641。[57] Andrea Rosà、Eduardo Rosales 和 Walter Binder。《Java 虚拟机上的任务粒度分析与优化》。载于：ACM Trans. Program. Lang. Syst. 41.3（2019年7月）。doi：10.1145/3338497。[58] Matteo Basso、Eduardo Rosales、Filippo Schiavio、Andrea Rosà 和 Walter Binder。《Java 虚拟机上的精确 Fork-Join 性能分析》。载于：Euro-Par。2022年，第 35–50 页。doi：10.1007/978-3-031-12597-3_3。[59] Frank Tip、Chris Laffra、Peter Sweeney 和 David Streeter。《使用 Java 应用提取器的实践经验》。载于：SIGPLAN Not. 34.10（1999年），292–305。doi：10.1145/320384.320414。[60] Bobby Bruce、Tianyi Zhang、Jaspreet Arora、Guoqing Harry Xu 和 Miryung Kim。《JShrink：深入研究现代 Java 应用的精简冗余代码》。载于：ESEC/FSE。2020年，第 135–146 页。doi：10.1145/3368089.3409738。[61] Muhammad Ali Gulzar、Xueyuan Han、Matteo Interlandi、Shaghayegh Mardani、Sai Deep Tetali、Tyson Condie、Todd Millstein 和 Miryung Kim。《大数据分析的交互式调试》。载于：HotCloud。2016年，第 19–25 页。doi：10.1145/ 3183440.3190334。

[62] Muhammad Ali Gulzar、Matteo Interlandi、Tyson Condie 和 Miryung Kim。《使用 BigDebug 在 Spark 中调试大数据分析》。载于：SIGMOD。2017年，第 1627–1630 页。doi：10.1145/3035918.3058737。[63] Muhammad Ali Gulzar、Siman Wang 和 Miryung Kim。《BigSift：在数据密集型可扩展计算中自动调试大数据分析》。载于：ESEC/FSE。2018年，第 863–866 页。doi：10.1145/3236024.3264586。[64] Matei Zaharia、Mosharaf Chowdhury、Tathagata Das、Ankur Dave、Justin Ma、Murphy McCauley、Michael Franklin、Scott Shenker 和 Ion Stoica。《弹性分布式数据集：内存中集群计算的容错抽象》。载于：NSDI'12：第 9 届 USENIX 网络系统设计与实现会议论文集。USENIX 协会，2012年，2:1–2:14。[65] Oracle。《HPROF：堆/CPU 性能分析工具》。http://docs.oracle.com/javase/8/docs/technotes/samples/hprof.html。2016年。（访问日期：2022年12月30日）。[66] ej-technologies。JProfiler。https://www.ej-technologies.com/products/jprofiler/ overview.html。2021年。（访问日期：2022年12月30日）。[67] YourKit GmbH。YourKit。https://www.yourkit.com。2021年。（访问日期：2022年12月30日）。[68] Oracle。JDK Mission Control。https://www.oracle.com/java/technologies/jdk- mission-control.html。2022年。（访问日期：2022年12月30日）。[69] Sameer Shende 和 Allen Malony。《Tau 并行性能系统》。载于：Int. J. High Perform. Comput. Appl. 20.2（2006年），287–311。doi：10 . 1177/ 1094342006064482。[70] 英特尔。Intel VTune Amplifier。https://software.intel.com/en- us/intel- vtune-amplifier-xe。2021年。（访问日期：2022年12月30日）。[71] 谷歌。类 com.google.common.collect.Streams。https://guava.dev/releases/ snapshot - jre / api / docs / com / google / common / collect / Streams . html。2022年。（访问日期：2022年12月30日）。[72] jOOQ。jOOQ/jOOL。https://github.com/jOOQ/jOOL。2020年。（访问日期：2022年12月30日）。[73] Tagir Valeev。StreamEx。https://github.com/amaembo/streamex。2022年。（访问日期：2022年12月30日）。[74] Paul Chiusano 和 Rnar Bjarnason。《Scala 函数式编程》。第 1 版。美国：Manning Publications Co.，2014。isbn：1617290653。[75] Apache Groovy 项目。java.util.stream 包。https : //docs . groovy- lang. org/latest/html/groovy-jdk/java/util/stream/package- summary.html。2022年。（访问日期：2022年12月30日）。[76] Kotlin 基金会。kotlin.streams 包。https://kotlinlang.org/api/latest/ jvm/stdlib/kotlin.streams/。2022年。（访问日期：2022年12月30日）。[77] Eduardo Rosales、Matteo Basso、Andrea Rosà、Mariano Marciello 和 Walter Binder。《Java 流分析与优化（工件）》。载于： <Programming>。2023年。doi：10.5281/zenodo.7557341。

关于作者

Eduardo Rosales 是意大利瑞士大学信息学院（USI，瑞士）的博士后研究员。他拥有安第斯大学（哥伦比亚）的理学硕士学位。他的研究兴趣包括集群/网格/云计算、程序分析以及并发与并行编程。您可以通过以下地址联系他：rosale@usi.ch

Matteo Basso是意大利瑞士大学信息学院（USI，瑞士）的博士生。他拥有意大利瑞士大学（USI，瑞士）与米兰比可卡大学（意大利）联合授予的理学硕士学位。他的研究兴趣包括编译器、托管语言与运行时，以及代码并行化。您可以通过以下地址联系他：matteo.basso@usi. ch

Andrea Rosà 是意大利瑞士大学（USI，瑞士）信息学院的博士后研究员。他拥有 USI 授予的博士学位、教授资格认证，以及米兰理工大学（意大利）授予的理学硕士学位。他的研究兴趣包括并发与并行、程序分析、编译器、虚拟机和大数据计算。您可以通过以下地址联系他：andrea.rosa@usi.ch

Walter Binder 是意大利瑞士大学（USI）瑞士信息学院的正教授。他在奥地利维也纳工业大学获得理学硕士和博士学位，并取得 Venia Docendi 授课资格。加入 USI 之前，他曾在瑞士洛桑联邦理工学院人工智能实验室从事博士后研究。他的主要研究兴趣包括程序分析、并发与并行编程以及虚拟机。请通过以下邮箱联系他：walter.binder@usi.ch