迈向解决最小开销监控的挑战

David Georg Reichelt
d.g.reichelt@lancaster.ac.uk 莱比锡兰卡斯特大学／莱比锡大学 德国萨克森州莱比锡

Stefan Kühne
stefan.kuehne@uni-leipzig.de 莱比锡大学 德国萨克森州莱比锡

Wilhelm Hasselbring
hasselbring@email.uni-kiel.de 基尔大学 德国石勒苏益格-荷尔斯泰因州基尔

arXiv:2304.05688v1 [cs.SE] 12 Apr 2023

摘要

检查软件的性能变化或性能行为，需要对其性能进行测量。这通过探针实现，即插入被检查应用程序中的代码片段，用于获取并处理测量数据。在单个方法中执行这些探针会产生开销，从而降低对调用方法进行性能测量的准确性，并拖慢测量过程。因此，性能测量的一项重要挑战是降低测量开销。

为应对这一挑战，应尽可能降低开销。基于对性能开销来源的分析，我们推导出以下四种优化方案：（1）使用源代码插桩，而非 AspectJ 插桩；（2）减少测量数据；（3）更换队列；（4）聚合测量数据。我们使用 MooBench 基准测试评估这些优化方案的效果。由此，我们表明这些优化方案能够降低监控框架 Kieker 的监控开销。对于 MooBench，每次方法调用的平均执行时长可从 4.77 μs 降至 0.39 μs。

CCS 概念

• **通用与参考** → **性能**；• **软件及其工程** → **软件性能**；软件维护工具。

关键词

软件性能工程，基准测试，性能测量，监控开销

ACM 参考格式：
David Georg Reichelt、Stefan Kühne 和 Wilhelm Hasselbring。2023。迈向解决最小开销监控的挑战。载于*2023* 年 *ACM/SPEC* 国际性能工程会议（*ICPE '23 Companion*）伴随会议论文集，*2023*年*4*月*15*日至*19*日，葡萄牙科英布拉。ACM，纽约州纽约市，美国，8 页。https://doi.org/10.1145/3578245.3584851

出版权已授权给 ACM。ACM 承认，本贡献由国家政府的雇员、承包商或附属人员撰写或共同撰写。因此，政府保留非独占、免版税的权利，可仅出于政府目的出版或复制本文，或允许他人这样做。*ICPE '23 Companion*，*2023*年*4*月*15*日至*19*日，葡萄牙科英布拉 © 2023 版权所有归所有者/作者所有。出版权已授权给 ACM。
ACM ISBN 979-8-4007-0072-9/23/04...$15.00
https://doi.org/10.1145/3578245.3584851

1 引言

对软件的性能变化、性能异常或总体性能行为进行检查，通常需要对性能进行测量。[1] 在生产环境中进行测量称为监控 [21]。[2] 性能监控可以通过两种方式实现：(1)使用插桩，即将测量探针插入源代码中；或(2) 使用采样，即从外部频繁获取应用程序的当前状态。测量探针是用于获取和处理测量数据的代码片段，并且会被插入到受检查的应用程序中。要频繁获取应用程序的当前状态，需要一个合适的技术接口，例如 Java 虚拟机提供了 jstack 工具，用于获取应用程序当前的堆栈。

插桩和采样都会产生开销。在测量性能时，这些开销会降低测量结果本身的准确性，并拖慢测量过程。如果测量准确性下降，测量值的标准差通常会增大。能否检测到微小的性能变化，取决于效应量，即性能变化幅度与标准差之间的关系。因此，检测微小的性能变化尤其需要尽可能降低性能监控开销。在旨在识别微小性能变化时，这一点尤为重要 [17]。因此，性能监控面临的一项重要挑战是降低测量开销。

为应对这一挑战，主要有以下几种方案：（1）优化采样间隔和配置；（2）配置由自适应插桩支持的自适应监控；（3）优化测量过程本身。在这项工作中，我们迈出了应对监控开销最小化挑战的第一步，即降低测量过程本身的开销（方案 3）。我们使用应用程序性能监控框架 Kieker [6] 展示了这些开销降低效果。

基于对 Kieker 中性能开销来源的分析，我们得出以下四项优化：（1）源代码插桩，即对软件的源代码进行插桩，而不是使用 AspectJ；（2）精简数据存储，即仅测量持续时间和方法名称；（3）队列交换，即使用 CircularFifoQueue 替代 LinkedBlockingQueue，

[1]替代方案需要具有同样精细分辨率的数据源，例如，使用现有日志记录中的数据 [11] 需要细粒度的日志语句。[2]如今，生产环境中的详细测量也常被称为可观测性，因为工具营销希望强调所获取数据的全面性以及可能进行的分析。我们仍沿用传统术语监控。

以及（4）聚合写入，即写入聚合后的测量性能数据，而不是每次单独方法调用的测量值。我们采用基准测试的经验标准 [5] 来比较这些开销降低措施的效果，并发现每项优化都能显著降低性能开销。采用所有开销降低措施后，我们可以将每次方法调用的开销从 4.77 μs 降至 0.39μs 。

本文其余部分安排如下：首先，我们介绍 Kieker 框架和 MooBench 基准测试。随后，我们阐述针对 Kieker 探针的优化。接着，我们利用 MooBench 给出开销测量结果。然后，我们将结果与相关工作进行比较。最后，我们进行总结并展望未来。

2 基础理论

在本节中，我们介绍监控框架 Kieker 和监控开销基准测试 MooBench。

2.1 Kieker

多种工具都能够进行性能监控，包括 Kieker、SPASSmeter [3], inspectIT,[3] OpenTelemetry[4] 和 Dynatrace APM。[5] OpenAPM[6] 倡议概述了现有的监控工具及其互操作性。

监控工具的两个核心组件是库 和代理。其中一个或两者都会被添加到软件的执行过程中，以获取监控数据。这些监控数据可以处于基础设施、应用程序或业务层面；在本研究中，我们专注于应用程序 监控。应用程序监控可以通过插桩实现，即向被测系统添加监控代码；也可以通过采样实现，即借助技术接口在指定时间获取测量数据。

在 Java 虚拟机中，采样只能在安全点获取监控数据 [7]。因此，它会停止所有线程的执行，并可能获取不准确的测量数据。为避免这些问题，我们专注于使用插桩进行监控。根据 MooBench 基准测试，与 OpenTelemetry 和inspectIT 相比，Kieker 是监控开销最低的框架 [16]。

图 1 描绘了 Kieker 的测量过程。首先执行插桩，这通常通过 AspectJ 完成。插桩会添加一个监控探针，用于创建监控记录，例如，每次方法调用都会生成一个 OperationExecutionRecord 实例。这些记录包含执行过程的元数据，例如执行顺序索引和执行栈大小，从而能够在之后重建调用树。随后，这些记录会传递到队列中，默认使用 LinkedBlockingQueue。写入器在线程中并行运行，与程序的主线程并行执行。写入器线程会等待新记录，并将记录写入指定位置。默认情况下，写入器是一个 FileWriter，会将监控记录写入硬盘。

[3]https://www.inspectit.rocks/[4]https://opentelemetry.io/ [5] https://www.dynatrace.com/de/platform/application-performance-monitoring/ [6] https://openapm.io/

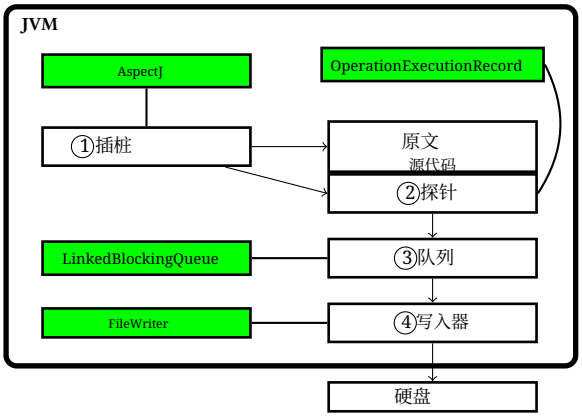

图 1：Kieker 中的监控过程

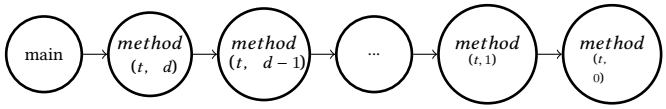

图 2：MooBench 的调用树

2.2 MooBench

MooBench 是一项基准测试，旨在测量监控框架的性能开销 [22]。为了测量性能开销，它会以给定的调用深度 d递归调用一个方法。在叶节点中，它会执行忙等待，直到经过指定的时间 t。为避免性能优化，递归方法调用会返回根方法调用的时间戳。图 2 展示了这一过程的调用树。[7]

由于即时编译、优化以及 Java 虚拟机的垃圾回收，Java 性能测量会受到非确定性的影响。因此，需要复杂的测量过程，并对测量结果进行统计分析 [4]。

因此，MooBench 包含一些脚本，用于自动重复启动虚拟机以及在虚拟机内进行基准测试迭代，从而以符合统计学原理的方式检测性能差异。它最初是为监控 Kieker、inspectIT 和 SpassMETER 而构建的 [3]。对于每个受支持的框架，MooBench 都包含不同测量配置的定义，例如，测量 Kieker 在将数据序列化到硬盘或 TCP 接收器时的性能开销。这样便能够比较这些变体的性能。MooBench 已在不同的执行环境中使用 [10]。最近，它已扩展为支持 OpenTelemetry 的测量 [16]。

3 KIEKER 探针的性能优化

Kieker 监控过程的所有部分都有性能优化空间：(1) 为避免 AspectJ 带来的开销，可以使用**源代码插桩**；(2) 为避免元数据创建

[7]原始数据源代码：https://github.com/kieker-monitoring/moobench/blob/ef9ca00259a8546e6fa9cdda47ac4cca3f116fe5/tools/benchmark/src/main/java/moobench/application/MonitoredClassSimple.java

步骤方面，只能获取创建**DurationRecord** 所需的数据；(3) 为减少队列插入的等待时间，可以使用**Cir-cularFifoQueue**；(4) 为减少数据存储的等待时间，只能存储**聚合数据** 。这些优化的概要如图 4 所示。下文各小节将详细介绍这些优化。

3.1 源代码插桩而非 AspectJ

监控程序测量方法的执行时间需要修改执行的源代码。可以使用插桩库来实现，例如用于 Open-Telemetry[8] 的 ByteBuddy，或 Kieker 中的 AspectJ。[9] Kieker 的 AspectJ 插桩通过 aop.xml 配置。它定义了应使用哪个切面，即使用记录方法执行开始和结束时间的 OperationExecutionRecord，还是仅记录方法执行开始时间的 BeforeOperationRecord。启动应用程序时，通过向 Java 虚拟机传递参数 -javaagent:kieker-1.15.2-aspectj.jar 来启动监控。

使用插桩库会产生开销，因为它会向堆栈跟踪中添加 JoinPointImpl.proceed 调用。通过将监控代码直接插入被监控的应用程序，可以避免这种开销。因此，我们创建了工具 *kieker-source-instrumentation*[10]，它会自动将测量代码添加到所有被调用的方法中。这需要对被监控应用程序的源代码进行两项自动更改：(1) 将所需变量添加到被监控的类中；(2) 将监控源代码添加到所有需要监控的方法中。

第一步是必要的，因为监控至少需要一个 MonitoringController 的引用，用于将创建的监控记录传递到队列；还需要当前使用的 TimeSource，以获取当前时间。这些都是单例；然而，使用 getInstance 动态获取实例会产生开销，应当避免。因此，源代码插桩会为每个被插桩的类创建静态 final 字段。

第二步是将监控源代码插入方法中。这始终需要定义签名、确定方法的开始和结束时间、创建监控记录，并将该记录写入队列。还可以选择获取其他元数据，例如执行栈大小。所有变量都会添加前缀，例如 _kieker，以避免与现有方法名冲突。清单 1 展示了一个带有当前栈大小获取操作的简化插桩示例。

3.2 使用 DurationRecord 替代 OperationExecutionRecord

对于使用 OperationExecutionRecord 进行监控的情况，会存储执行顺序索引、执行栈大小、当前主机名以及当前会话的 ID。这需要存储当前执行次数和当前栈大小。此外，数据会存储在记录中，并写入硬盘。

[8]https://github.com/open-telemetry/opentelemetry-java-instrumentation/blob/main/javaagent-extension-api/build.gradle.kts [9] https://kieker-monitoring.readthedocs.io/en/1.15.2/getting-started/AspectJ-Instrumentation-Example.html#gt-aspectj-instrumentation-example [10] https://github.com/kieker-monitoring/kieker-source-instrumentation

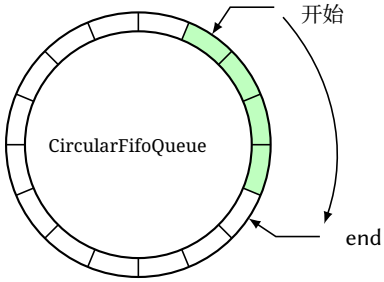

图 3：CircularFifoQueue 结构

因此，我们创建了 DurationRecord，这是一种最精简的 Kieker 记录，仅存储持续时间。为了能够监控这种记录，我们相应地调整了插桩流程。

3.3 使用 CircularFifoQueue 替代 LinkedBlockingQueue

目前，Kieker 使用 LinkedBlockingQueue。它是一个单向链表，并存储着一个容量。当队列中的元素数量超过容量时，每次插入新元素都会使队列阻塞。阻塞会显著降低应用程序线程的执行速度，因此绝不应发生。LinkedBlockingQueue 的一个缺点是，它需要创建新的队列元素，这需要预留空间，并通过设置最后一个队列元素的指针来链接新元素。

使用 CircularFifoQueue 可以提高队列的速度。CircularFifoQueue 是一个用于存储队列元素的环形缓冲区。其结构如图 3 所示：它包含一个指定大小的数组、当前起始元素的索引以及当前结束元素的索引。添加新元素时，会将结束索引处的元素设为该新元素，并将结束索引递增（对队列大小取模）。从队列中取出元素时，会返回起始索引处的元素，将该位置设为 null，并将起始索引递增（对队列大小取模）。因此，在处理队列时无需分配新的内存。Apache Commons Collections 的当前实现不会在队列已满时作出响应，也就是说，如果所有位置都已被占用，新元素会覆盖旧元素。

基础 CircularFifoQueue的另一个问题是，它不能直接用于并行。由于 Kieker 的读取和写入是由不同的线程完成的，这是必要的。因此，我们使用了经过调整的 CircularFifoQueue版本，该版本是同步的。[11] 要使用它，我们使用 Kieker 读取的 Java 属性配置了队列。

3.4 存储聚合数据，而非方法执行记录

向队列中插入数据以及向硬盘写入数据都很耗时。此外，执行时间取决于硬盘的当前状态。因此，我们仅通过获取和存储聚合数据（例如平均值）来减少存储的数据。

[11]同步版 CircularFifoQueue：https://github.com/DaGeRe/KoPeMe/blob/main/kopeme-core/src/main/java/de/dagere/kopeme/collections/SynchronizedCircularFifoQueue.java

列表 1：示例插桩源代码

```
public void myMethod (){ final String _kieker_signature =" public ␣ void ␣ net . kieker . Class .
myMethod ()";[...]final int _kieker_ess ; long _kieker_traceId = _kieker_controlFlowRegistry .
recallThreadLocalTraceId ();[...]if (_kieker_traceId == -1) {_kieker_ess = _kieker_controlFlowRegistry .
recallAndIncrementThreadLocalESS ();[...]} else {_kieker_ess = _kieker_controlFlowRegistry .
recallAndIncrementThreadLocalESS ();[...]} final long _kieker_tin =C0_0. _kieker_TIME_SOURCE . getTime
(); try { // 执行方法原始代码 } finally { final long _kieker_tout =C0_0. _kieker_TIME_SOURCE . getTime ();_
kieker_controller . newMonitoringRecord ( new OperationExecutionRecord (_kieker_signature , _kieker_tin ,
_kieker_tout , ...)); }
```

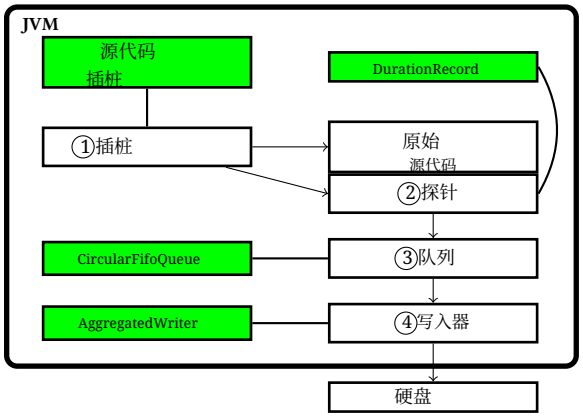

图 4：可能的监控优化

1000 个方法执行时长，而不是每个方法的执行时长。为此，我们向每个被检查的类添加两个计数器，一个记录执行时间总和，另一个记录执行次数。每次迭代时，将当前执行时间加到总和中，并将计数器加一。当计数器达到参数化的执行次数（例如 1000）时，就会创建一条监控记录。此外，总和和计数器都会重置。

这种优化使得无法检查方法调用的细粒度属性，例如异常值的出现频率。因此，只有在需要检查长期性能表现时才能应用这种优化。例如，如果某个方法的平均速度变慢了 2%，则可以使用聚合数据存储来检查。如果每 500 次调用中就出现一次异常值，导致平均速度变慢 0.5%，则无法通过聚合数据存储来检查；这种情况下需要细粒度数据。

4 性能优化的基准测试结果

为了测量性能优化带来的开销，我们使用 MooBench 基准测试，分别单独执行各项优化，并执行它们的组合。本节首先介绍实验设置，随后介绍各项单独测量结果。

4.1 实验设置

对于每项优化，我们都分别在启用和未启用该优化的情况下运行 MooBench。我们使用了调用树深度 10、方法调用次数 2 000 000 和虚拟机启动次数 10，这些都是 MooBench 的默认参数。根据技术上的适用性，我们还比较了不同优化的组合。此外，我们将测量结果与完全未进行插桩时的执行情况进行了比较。

所有测量均在 i7-4770 CPU @ 3.40GHz 上执行，使用的是 OpenJDK 1.8.0_352 中的 JVM，也就是执行时最新发布的 OpenJDK 版本。以下各小节介绍我们的测量结果。除非另有说明，所有数值的单位均为每次 **微秒 (**μs**)MooBench 方法调用**。例如，Kieker 使用 AspectJ 且调用树深度为 10 时，测得的数值为 4.77，这意味着监控 10 个节点耗时 $4.77\mu s$，也就是说，监控单个节点的平均开销为 $0.477\mu s$。可使用以下内容重复这些测量：

	无插桩	AspectJ	源代码插桩
Mean	0.0548	4.7711	2.4169
95 %	± 0.0	± 0.0444	± 0.0038
Q_1	0.0520	4.1890	2.4380
中位数	0.0530	5.0790	2.5730
Q_3	0.0580	5.5030	2.6880

表1：源代码插桩统计

	无说明。	源代码插桩	DurationRecord
Mean	0.0548	2.4169	2.3426
95 %	± 0.0	± 0.0038	± 0.0037
Q_1	0.0520	2.4380	2.1720
中位数	0.0530	2.5730	2.4720
Q_3	0.0580	2.6880	2.6080

表 2：DurationRecord 的统计数据

MooBench 主仓库的 optimizations-all 分支。[12] 我们的测量结果数据集已发布。[13]

4.2 源代码插桩

表 1 展示了未进行插桩、使用 AspectJ 插桩以及进行源代码插桩时的统计数据。由于未插桩时的平均执行时间很短，95% 置信区间的宽度实际上为 0。这表明，AspectJ 插桩大约会使执行时间增加 10 倍。根据方法粒度的不同，这一倍数在生产工作负载中可能更高或更低。使用源代码插桩可使执行时间大约缩短一半，因此是一项值得推荐的优化。不过，只有在源代码可用时才能使用这种方法；因此，在某些情况下，仍有必要使用 -javaagent 注入监控探针。

4.3 DurationRecord

由于 DurationRecord 目前仅适用于源代码插桩，我们比较了针对 Kieker 常规 OperationExecutionRecord 的源代码插桩，以及使用 DurationRecord 的源代码插桩。结果见表 3。结果表明，使用 DurationRecord 在统计上也能显著缩短测量时长，但其效应量远不及源代码插桩所带来的改进。

4.4 CircularFifoQueue

CircularFifoQueue 可与源代码插桩以及 OperationExecutionRecord 或 DurationRecord 一起使用。因此，我们比较了两者的执行时间。结果显示在表 3 中。该表显示，无论使用 OperationExecutionRecord 还是 DurationRecord，CircularFifoQueue 都能带来统计上显著的性能提升。令人惊讶的是，使用 Kieker 的默认 OperationExecutionRecord 时，性能提升高于使用 DurationRecord 时。

[12]https://github.com/kieker-monitoring/moobench/tree/optimizations-all [13] https://doi.org/10.5281/zenodo.7566677

我们认为这是由 Java 虚拟机的内部优化所致；然而，因此不应盲目地将 CircularFifoQueue 与 DurationRecord 结合使用。

	无插桩。	CircularFifoQueue + OperationExecutionR.	CircularFifoQueue + DurationRecord
Mean	0.0548	1.5017	1.6503
95 %	± 0.0	± 0.0047	± 0.0039
Q_1	0.0520	1.2330	1.2670
中位数	0.0530	1.3300	1.4190
Q_3	0.0580	1.4810	1.7420

表 3：CircularFifoQueue 的统计信息

此外，如果加入的元素数量超过队列的容量，CircularFifoQueue 可能会吞掉 元素。因此，我们只建议在特定情况下使用 CircularFifoQueue，并确保不会发生这种情况。下面将介绍另一种降低队列开销的方法：在数据插入队列之前对其进行聚合。

4.5 聚合写入

目前，聚合仅针对源代码插桩实现。此外，由于执行栈大小等元数据可能难以聚合，因此目前只有 DurationRecord 可以直接实现聚合。因此，我们比较了 DurationRecord 的聚合写入和使用情况。也许可以通过将执行栈大小映射到当前数据来存储，从而实现元数据的聚合。这种方法的性能特征将在很大程度上取决于树结构。对此进行研究可以作为未来工作的一部分。

表 4 展示了聚合数据读取的统计信息（始终聚合 1000 次方法调用时）。结果表明，对数据进行聚合也能显著降低监控开销。因此，对于可以使用聚合数据的场景，我们建议采用聚合写入。

	无插桩	DurationRecord	聚合写入
Mean	0.0548	2.3426	0.4014
95 %	± 0.0	± 0.0037	± 0.0003
Q_1	0.0520	2.1720	0.3810
中位数	0.0530	2.4720	0.3860
Q_3	0.0580	2.6080	0.3870

表 4：聚合数据读取的统计数据

4.6 Combination

由于聚合写入减少了创建的监控记录数量，因此将聚合写入与使用 CircularFifoQueue 相结合也是合理的。表5展示了这种方法的统计数据。在这种设置下，测量值之间的差异在统计上并不显著。因此，我们无法判断在这种设置下使用 CircularFifoQueue 是否能降低开销。

	无指令。	聚合写入	组合
Mean	0.0548	0.4014	0.3897
95 %	± 0.0	± 0.0003	± 0.0002
Q_1	0.0520	0.3810	0.3830
中位数	0.0530	0.3860	0.3880
Q_3	0.0580	0.3870	0.3890

表5：聚合数据读取的统计信息

在实际应用场景中，开销不仅与调用树深度为 10 的情况有关。因此，我们测量了这些优化组合在不同调用树规模下的表现。图 5 展示了随着调用树深度增加，开销的变化情况。为便于观察，我们用线连接各个测量点，即使调用树深度是一个具体值。曲线周围的区域标示了 $\mu + \sigma$ 和 $\mu - sigma$ 的范围，其中 μ 为均值，σ 为标准差。我们发现，对于所考察的调用树规模，使用 DurationRecord 仅能略微降低开销。由于即使不使用 CirclarFifoQueue，使用 DurationRecord 也能降低开销，而且在聚合操作调用时无法使用元数据，因此我们继续使用 DurationRecord。总体而言，该图表明，对于规模为 128 的调用树，与使用 AspectJ 进行常规监控相比，所有优化都能显著降低开销。

5 相关工作

相关工作可分为三类：仅测量监控开销的工作、探讨如何降低监控开销的工作，以及减少监控本身的工作。以下将概述这些领域的相关工作。

5.1 测量

Horky 等人 [8] 考察了 Java 中动态监控的开销，即按需插入和移除的监控。他们指出，三种因素会影响被监控方法的执行时间：探针本身的存在、探针执行的操作，以及探针可能引发的代码优化。通过使用 SPECjbb2015TM，他们发现，在真实环境中激活和停用探针都可能使执行速度变慢或变快。

Bara 等人 [1] 开发了用于处理器核心性能监控的硬件支持，并表明性能监控也会增加处理器的能耗。与本文不同，他们关注的是硬件层面的性能监控器。

性能测量的一项主要要求是可重复性。Eichelberger 等人 [2] 研究了如何复现 SPASS-meter 的 MooBench 测量结果。此外，Knoche 和 Eichelberger [9, 10]讨论了如何使用 Raspberry Pi 进行可重复的性能测量。与他们的工作不同，我们展示了在典型台式机上对成熟基准测试进行测量的结果。为便于复现我们的结果，合理的做法是在不同的执行硬件上重复测量，例如在 Raspberry Pi 上。

此外，还存在一些采用不同技术、旨在实现最小开销的测量实现。JPortal [25] 通过使用基于硬件的英特尔处理器跟踪技术，实现了极低的开销。由于这需要对机器码执行所存储的元数据进行解码，因此获得的跟踪记录并不完全正确。不过，他们报告的开销最高为 16.5%。与我们的工作不同，他们使用了不同的工作负载进行基准测试，其目标并不是获得精确的运行时间测量结果，而是获得准确的跟踪记录。

Schardl 等人 [18] 开发了 CSI 框架，该框架能够在编译时插入代码。除其他用途外，这项技术还可用于测量已执行代码的性能。他们通过对 Apache 服务器和 bzip2 数据压缩器进行插桩，并使用适当的基准测试对其进行基准测试，评估了 CSI 框架。他们发现，开销不到程序执行时间的 70%。虽然他们采用了类似的插桩方法，例如在编译时进行插桩，但使用了不同的技术（以 C 编写的软件），并通过基准测试评估了真实场景中的用例。因此，他们的结果无法与我们的结果直接比较。

5.2 监控开销的降低

下文将首先按时间顺序介绍 Kieker 中有关降低开销的研究，随后介绍采用不同插桩技术的相关工作。

为了降低监控开销，Waller 和 Hasselbring[24] 首先研究了使用多核处理器对 Kieker 运行时开销的影响。他们发现，在使用多核处理器系统时，异步监控写入器的开销较低。Waller 等人 [23] 指出，监控开销由插桩开销、数据收集时间以及将数据写入队列所需的时间构成。他们考察了四种可能的优化：（1）内部优化，例如重用方法的 String 定义；（2）使用 Disruptor 框架中的 ArrayBlockingQueue；（3）将 ByteBuffer 实例而非记录发送到监控队列；（4）通过提供精简版 Kieker 项目，减少 Kieker 中使用的源代码。他们发现，前两种优化带来了显著改进，并且从可维护性的角度来看是可行的。优化（3）能够降低开销，但需要为每种记录类型分别定义序列化，因此不再进一步考虑。优化（4）虽能略微降低开销，但也会降低 Kieker 的可维护性和可扩展性，因此同样不再进一步考虑。

最后，Strubel 和 Wulf [20] 进一步降低了监控开销。此前，字符串属性使用带有 id 的注册表记录进行序列化，因此字符串并未被完全序列化。注册表记录的这一转换是在应用程序线程中完成的；将其移至写入器线程后，可以降低代码复杂度以及应用程序线程内部的开销。

由于本研究基于当前的 Kieker 版本，Waller 和 Hasselbring [24], Waller 等人 [23] 以及 Strubel 和 Wulf [20] 提出的可行优化已包含在内。尽管如此，他们的重点是保留架构发现功能，而本研究则侧重于监控方法执行时长，不保留架构发现功能。

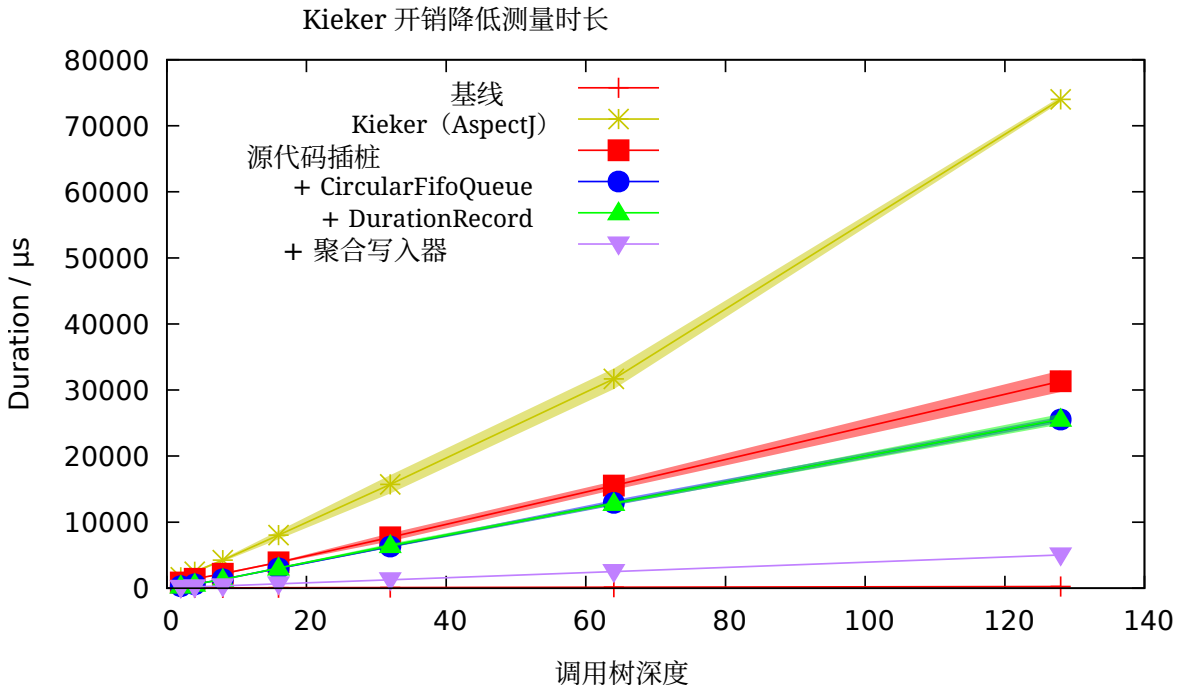

图5：基于调用树大小的开销增长

用于字节码插桩的领域专用语言 DiSL 相较于 AspectJ [12] 支持更细粒度的插桩。由于它依赖于 ASM，其性能特征很可能与 Kieker 默认的 AspectJ 插桩不同。因此，它可以作为我们源代码插桩的替代方案，并可据此进行比较。不过，目前尚无可用的实现，因此我们无法对两者进行比较。

Okanovic等人 [14] 研究了如何使用不同的插桩技术来降低监控开销。因此，他们在监控系统 DProf 中使用 DiSL，而不是 AspectJ。他们发现，在其用例中，使用 DiSL 将监控开销降低了 1.2%。与我们的工作不同，他们使用 DProf 系统评估性能开销。其 1.2% 的性能提升表明，DiSL 很可能无法像我们的源代码插桩那样大幅降低开销。

5.3 监控的减少

为了减少监控本身带来的开销，Popiolek 等人 [15] 研究如何通过减少云基础设施中的性能监控器来降低性能监控的开销。他们使用皮尔逊相关系数对计数器进行聚类，也就是说，确定哪些计数器会同时处于高值或低值。由此，他们将监控开销最多降低了三分之一。类似地，Shang 等人 [19] 也使用测量值的皮尔逊相关系数来确定性能计数器。之后，他们使用 Calinski-Harabasz 停止准则。与这些工作不同，我们并不试图选择监控位置，而是要降低发生监控的方法所带来的监控开销。

Mertz 和 Nunes [13] 通过自适应配置降低开销：在第一步中，他们执行轻量级监控。根据监控结果，他们决定应对源代码的哪些部分进行更详细的监控。与这项工作不同，我们不采用自适应配置，而是尝试通过静态配置降低监控开销。

6 总结与展望

我们介绍了如何降低 Kieker 框架的监控开销。我们通过对监控过程进行四项改进实现了这一点：直接插桩源代码，而不是使用 AspectJ 进行插桩；减少存储的监控数据；使用 CircularFifoQueue 替代 LinkedBlockingQueue；以及使用聚合写入器替代 Kieker 的 FileWriter。

只有在需要检查长期性能的情况下，减少存储的监控数据并使用聚合写入器才是可行的。如果需要检查单个方法调用及其异常值，则只能使用源代码插桩和队列交换。

使用监控开销基准测试 MooBench，我们证明了所有这些优化都能降低监控开销。因此，计划将它们提供给常规监控使用。这需要进行以下调整：(1) 对于 Kieker 源代码插桩，这项工作已通过将代码发布在 Kieker GitHub 组织中完成。(2) 对于 DurationRecord，

将包含监控信息精简记录纳入 Kieker 基础设施的工作目前仍在进行中。由于 Kieker 支持多种语言，纳入此类记录需要先在 Kieker Instrumentation Language 中定义一种基本记录，再据此为每种语言生成相应记录。(3) 是否切换队列将由用户决定。虽然使用 CircularFifoQueue 在统计意义上显著提升了性能，但尚不清楚生产系统是否会生成过多记录，导致队列不堪重负，进而丢失记录。未来的研究可以探讨，在不影响监控开销降低效果的前提下，能否增加额外的完整性检查。(4) 计划将 AggregatedWriter 作为 Kieker 的一部分加入其中，但不将其设为默认项。这样，如果用户使用 DurationRecord，且目标只是测量性能而非执行架构恢复任务，就可以切换到 AggregatedWriter。除了调整 Kieker 本身，还应检查数据聚合在多大程度上能够改善测量质量。虽然我们假设开销降低总能提高测量结果的可区分性，但仍需对此进行验证。

这项工作是迈向最小化监控开销的第一步：除了优化监控过程本身，还应研究以下进一步的优化：（1）应将我们优化后的监控过程与采样进行比较。采样只能在安全点获取堆栈跟踪，但开销很低。因此，将我们优化后的 Kieker 探针与采样进行比较时，应同时比较两者的开销以及检测性能变化的能力。（2）使用自适应监控（如 [13]）：研究表明，停用探针几乎可以将监控开销降至零。为了在保持测量准确性的同时获取可用数据，只在部分时间段激活探针或许是一个可行的方案。此外，我们的方法只能测量性能，无法测量调用树的创建过程，因为其中没有监控元数据。如果需要监控元数据，也可以在同一段源代码中添加多个探针：一个常规探针用于创建 OperationExecutionRecord，另一个探针用于创建 DurationRecord。之后即可使用所需的探针。通过这项工作，以及对采样和自适应监控的考察，可以制定性能监控的最小开销流程。这有望为理解生产系统中软件的底层性能行为奠定基础。

致谢 本研究由德国联邦教育与研究部通过“高效集成性能监控”项目（PermanEnt，BMBF 01IS20032D）资助。

参考文献

[1] Lucian Bara、Oana Boncalo 和 Marius Marcu。2015。用于测量 OpenRISC 处理器性能并估算能耗的硬件支持。载于*2015 IEEE* 第十届周年国际应用计算智能与信息学研讨会。 399 404 10 1109 2015 7208237 –。 https://doi.org/。/SACI.。Holger Eichelberger、Aike Sass 和 Klaus Schmid。。从可复现性问题到改进：一段历程。载于*SSP 2016*，软件技术趋势，第 36 卷。43–45。[2] 2016 [3] Holger Eichelberger 和 Klaus Schmid。2014。Java 程序的灵活资源监控。系统与软件杂志93 (2014)，163–186。https: //doi.org/ 10 1016 2014 02 022 。/ j.jss.。。Andy Georges、Dries Buytaert 和 Lieven Eeckhout。2007。严谨的 Java 性能统计评估。*ACM SIGPLAN* 通讯42，10 (2007)，57–76。https: [4] //doi.org/ 10 1145 1297027 1297033 。/。Wilhelm Hasselbring。。将基准测试作为软件工程研究的实证标准。载于 EASE 2021：软件工程评估与测量。软件工程评估与测量，365–372。https: //doi.org/doi: 10 1145 3463274 3463361 。/。[5] 2021 Wilhelm Hasselbring 和 André van Hoorn。2020。Kieker：面向软件工程研究的监控框架。Software Impacts 5 (2020)，100019。https://doi.org/ 10 1016 2020 100019 。/j.simpa.。。Peter Hofer、David Gnedt 和 Hanspeter Mössenböck。2015。利用部分安全点和增量式栈跟踪进行轻量级 Java 性能分析。载于第六届 ACM/SPEC 国际性能工程会议论文集。75–86。[6] [7] Vojtěch Horký、Jaroslav Kotrč、Peter Libič 和 Petr Tůma。2016。动态 Java 性能监控的开销分析。载于第七届 ACM/SPEC ICPE（荷兰代尔夫特）（ICPE '16）。计算机协会，纽约州纽约市，美国， 275 286 10 1145 2851553 2851569 –。https://doi.org/。/。[8] [9] Holger Knoche 和 Holger Eichelberger。。Raspberry Pi：可复现性能基准测试的平台？Softwaretechnik-Trends 37，3 (2017)，14–16。[10] Holger Knoche 和 Holger Eichelberger。2018。使用 Raspberry Pi 和 Docker 进行可复现的性能实验：经验报告。载于 2018 年 ICPE 会议论文集。305–316。 https://doi.org/10.1145/3184407.3184431[11] Lizhi Liao、Jinfu Chen、Heng Li、Yi Zeng、Weiyi Shang、Catalin Sporea、Andrei Toma 和 Sarah Sajedi。2022。在实际运行中定位基于 Web 系统性能回归的根本原因：经验报告。IEEE 软件工程汇刊 48，12 (2022)，4986–5006。https://doi.org/10. 1109/TSE.2021.3131529[12] Lukáš Marek、Alex Villazón、Yudi Zheng、Danilo Ansaloni、Walter Binder 和 Zhengwei Qi。2012。DiSL：用于字节码插桩的领域专用语言。载于第十一届面向切面的软件开发国际会议。 239 250 10 1145 2162049 2162077 –。 https://doi.org/。/。Jhonny Mertz 和 Ingrid Nunes。。软件监控在实践中的可行性：一种低影响执行跟踪框架。载于第十四届 SEAMS（加拿大魁北克省蒙特利尔）。IEEE 出版社，169–180。https://doi.org/10.1109/SEAMS. 2019.00030[14] Dušan Okanović、Milan Vidaković 和 Zora Konjović。2013。降低性能监控开销。载于 2013 年 IEEE 第十一届智能系统与信息学国际研讨会（SISY）。135–140。 https://doi.org/10.1109/ SISY.2013.6662557[15] Pedro Freire Popiolek、Karina dos Santos Machado 和 Odorico Machado Mendizabal。2021。共享基础设施的低开销性能监控。Expert Systems with Applications 171 (2021)，114558。 https://doi.org/10.1016/j. eswa.2020.114558[16] David Georg Reichelt、Stefan Kühne 和 Wilhelm Hasselbring。2021。OpenTelemetry、inspectIT 和 Kieker 的开销比较。载于 SSP 2021。[17] David Georg Reichelt、Stefan Kühne 和 Willhelm Hasselbring。2019。PeASS：用于识别代码级性能变化的工具。载于第 *33* 10 1109 2019 00123 届 ACM/IEEE ASE 会议论文集。ACM。https://doi.org/。/ASE.。Tao B Schardl、Tyler Denniston、Damon Doucet、Bradley C Kuszmaul、I-Ting Angelina Lee 和 Charles E Leiserson。2017。用于编译器插入式程序插桩的 CSI 框架。《ACM 计算系统测量与分析论文集》 1, 2 2017 1 25 10 1145 3154502 ( )，–。 https://doi.org/。/ Weiyi Shang、Ahmed E Hassan、Mohamed Nasser 和 Parminder Flora。。利用聚类性能计数器回归模型自动检测性能回归。载于第六届 ACM/SPEC 国际性能工程会议论文集。ACM，15–26。https://doi.org/10. 1145/2668930.2688052[20] Hannes Strubel 和 Christian Wulf。2016。重构 Kieker 的监控组件以进一步降低运行时开销。载于 2016 年软件性能研讨会（SSP '16）。[21] Jan Waller。2015。应用程序监控框架的性能基准测试。BoD–按需图书。[22] Jan Waller、Nils Christian Ehmke 和 Wilhelm Hasselbring。2015。将性能基准测试纳入持续集成以实现 DevOps。ACM SIGSOFT 软件工程简报 40，2 (3 2015)，1–4。 https://doi.org/10.1145/ 2735399.2735416[23] Jan Waller、Florian Fittkau 和 Wilhelm Hasselbring。2014。应用程序性能监控：开销降低与可维护性之间的权衡。《软件性能研讨会论文集》（2014）。[24] Jan Waller 和 Wilhelm Hasselbring。2012。不同多核处理器对应用程序级监控运行时开销的影响比较。载于多核软件工程、性能与工具国际会议。Springer，42–53。https://doi.org/10.1007/978-3-642-31202-1 5_[25] Zhiqiang Zuo、Kai Ji、Yifei Wang、Wei Tao、Linzhang Wang、Xuandong Li 和 Guoqing Harry Xu。2021。JPortal：利用 Intel 处理器跟踪对 JVM 程序进行精确、高效的控制流跟踪。载于第 42 届 ACM SIGPLAN ICPLDI 会议论文集。1080–1094。https://doi.org/10.1145/3453483.3454096