arXiv:2603.14019v1 [cs.PL] 14 Mar 2026

MapReplay：面向 Java HashMap 的基于跟踪的基准测试生成

Filippo Schiavio
filippo.schiavio@usi.ch 意大利语瑞士大学 卢加诺，瑞士

Andrea Rosà
andrea.rosa@usi.ch 意大利语瑞士大学 卢加诺，瑞士

Júnior Löff
loeffj@usi.ch 意大利语瑞士大学 卢加诺，瑞士

Lubomír Bulej
bulej@d3s.mff.cuni.cz 查理大学 布拉格，捷克共和国

Petr Tůma
petr.tuma@d3s.mff.cuni.cz 查理大学 布拉格，捷克共和国

Walter Binder
walter.binder@usi.ch 意大利语瑞士大学，瑞士卢加诺

摘要

基于哈希的映射，尤其是 `java.util.HashMap`，在 Java 应用程序和 JVM 中无处不在，因此其性能至关重要。由于性能取决于操作模式、键分布和扩容行为等因素，评估优化效果颇具挑战。微基准测试运行快速且可重复，但往往过度简化工作负载，无法捕捉真实的使用模式。应用基准测试（例如 DaCapo、Renaissance）能够提供真实的使用场景，但运行成本更高、容易产生波动，而且性能表现受非 `HashMap` 计算主导，因此难以观察与映射相关的性能变化。

为应对这一挑战，我们提出了 MapReplay，这是一种基准测试方法，将应用基准测试的真实性与微基准测试的高效性相结合。MapReplay 跟踪`HashMap` API 的使用情况，生成能够重现相同操作序列、同时忠实重建内部映射状态的重放工作负载 。这使得我们能够在真实的使用模式下，对替代实现进行真实且高效的评估。将 MapReplay 应用于 DaCapo-Chopin 和 Renaissance 后，得到的套件 MapReplayBench 能够重现应用级性能趋势，同时缩短实验时间，并揭示难以通过完整基准测试获得的见解。

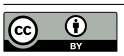
CCS 概念

• **So软件及其工程 → 软件性能** **表现**。

关键词

基准测试、追踪、记录/回放、基于哈希的映射、Java 虚拟机、Java 微基准测试框架

ACM 参考格式：
Filippo Schiavio、Andrea Rosà、Júnior Löff、Lubomír Bulej、Petr Tůma 和 Walter Binder。2026 年。MapReplay：面向 Java HashMap 的基于跟踪的基准测试生成。载于第 *17* 届 *ACM/SPEC* 性能工程国际会议论文集（*ICPE '26*），*2026*年*5*月*4*日至*8*日，意大利佛罗伦萨。ACM，纽约，纽约州，美国，14 页。https://doi.org/10.1145/3777884.3797010



本作品采用知识共享署名 4.0 国际许可协议授权。*ICPE '26*，意大利佛罗伦萨
© 2026 版权所有归作者/所有者所有。ACM ISBN
979-8-4007-2325-4/2026/05 https://doi.org/10.1145/3777884.3797010

1 引言

基于哈希的映射是现代软件系统中应用最广泛的数据结构之一 [18]。数十年来，提升其性能和内存效率一直是活跃的研究课题，涵盖缓存和单指令多数据高效设计 [5, 10, 46]、兼顾存储与持久化的哈希表 [54, 60]，以及广泛部署的库和运行时中的行业驱动型优化 [1,32, 34]。此外，基于哈希的映射是众多计算系统与技术的基础构件，例如数据库引擎和数据管理流水线 [9]、数据驱动压缩与编码方案 [25]，以及依赖精心设计的哈希表的性能关键型运行时和存储系统 [5]。在 Java 中，`java.util.HashMap`[1] 无处不在——它被用于应用代码、标准库，甚至 Java 虚拟机（JVM）内部。哈希映射的性能取决于许多相互关联的因素，包括操作的类型组合及顺序、键哈希码的分布、当前映射大小以及扩容行为。这些依赖关系使得可靠地评估所提出的优化颇具挑战。

微基准测试通常用于在受控条件下评估特定的 `HashMap` 操作。虽然它们能够支持可重复且快速的实验，但其工作负载往往过于简单，无法捕捉真实应用中出现的复杂访问模式。通常，针对不同操作（例如插入、查找或迭代）会编写不同的微基准测试，每个测试都着重考察某一特定性能方面 [18]。然而，在评估潜在优化（例如更改默认初始容量或扩容策略）时，如何权衡一种操作类型上的改进与另一种操作类型上的退化并不明确。因此，单靠微基准测试很少能为是否应在标准库中采用某项优化提供可靠依据。

另一方面，DaCapo Chopin [7, 8] 和 Renaissance [41] 等完整应用基准测试能够展现真实的`HashMap` 行为。然而，它们的测量变异性很大，且运行耗时较长。由于花在`HashMap` 代码上的时间只占总执行时间的一小部分，因此可能需要反复运行很多次——有时基准测试耗时长达数百小时——才能得出有意义的结论，确认性能变化是否具有统计显著性

[1]为简洁起见，以下文本中省略了完全限定名称 .

。此外，这些工作大多耗费在与映射无关的计算上，而这些计算对于评估HashMap 替代方案并不重要。

本文提出一种折中方案，将应用基准测试的代表性与微基准测试的效率和可控性结合起来。我们介绍 MapReplay，这是一款跟踪记录应用对 HashMap 的使用情况，并提供重放工作负载 的工具；该工作负载能够重现相同的映射操作序列，但不包含任何外围应用逻辑。重放工作负载会忠实重建内部映射状态，确保这些状态在 HashMap 的控制流决策方面等效，旨在执行与原始应用中相同的 HashMap 代码路径。这种方法无需付出重新运行整个应用的成本和复杂性，即可在真实世界工作负载下快速、真实且可复现地评估不同的映射实现。此外，即使应用中的映射操作原本只占总执行时间的一小部分，MapReplay 也能从中生成真实的、映射密集型重放工作负载，从而将测量工作集中于最重要的性能方面。要实现忠实的跟踪记录回放，必须在保真度与效率之间取得平衡。MapReplay 的设计既能确保准确重现性能关键的库代码路径，又能尽量减小跟踪记录的大小和回放开销。

借助 MapReplay，我们创建了 MapReplayBench，这是一个针对基于哈希的 Java 映射的基准测试套件。它是将我们的方法应用于两个广泛使用的 Java 应用基准测试套件——DaCapo-Chopin [7, 8]和 Renaissance [41]——而创建的。MapReplayBench 使研究人员和从业者能够分析HashMap以及 JVM 上其他基于哈希的映射实现，在具有代表性的、源自应用的工作负载下的性能。我们通过将 MapReplay 用于评估默认初始容量不同的HashMap 变体，展示了 MapReplay 的可行性。我们使用了三类基准测试：(1) 公开可用的HashMap 微基准测试，(2) 来自 DaCapo Chopin 和 Renaissance 的完整应用基准测试，以及 (3) 我们的回放工作负载。我们表明，MapReplay 提供的性能洞见与映射密集型应用基准测试所得的结果一致，但执行成本更低。此外，由于 MapReplay 会放大应用中的映射相关活动，因此即使回放工作负载源自非映射密集型应用，用户也能观察到性能趋势；而仅使用原始应用程序执行则无法从这些应用中得出明确结论。

本文作出以下贡献：

(1) 我们提出了 MapReplay，这是一种面向 Java HashMap 的新型轨迹驱动基准测试方法。MapReplay 记录应用中对 HashMap操作的使用情况，并提供一种回放工作负载，在等效的内部映射状态下执行相同的操作序列，以覆盖所测试的 HashMap 代码路径。其设计通过紧凑的跟踪记录和轻量级回放，兼顾了保真度——忠实重现性能关键的HashMap 执行路径——与效率。

（2）我们使用 MapReplay 从成熟的 Java 基准测试套件 DaCapo-Chopin [7, 8] 和 Renaissance [41]生成跟踪记录。对于这些跟踪记录，我们的回放工作负载展现出与应用基准测试一致的性能趋势，同时缩短了实验时间。此外，我们还证明，回放工作负载能够揭示使用应用基准测试可能难以或无法获得的性能洞见。

(3) 我们创建了 MapReplayBench，这是一个可直接使用的独立回放工作负载套件，其中包含从 DaCapo-Chopin 和 Renaissance 生成的跟踪记录，以及用于执行这些工作负载的重放基础设施。[2] 借助 MapReplayBench，研究人员和从业者可以在映射密集型且贴近实际的工作负载上评估HashMap 的替代实现。

在介绍 HashMap 的背景知识（第 2 节）后，我们将介绍 MapReplay（第 3 节）、MapReplayBench（第 4 节）的设计与架构，以及我们的评估（第 5 节）。随后，我们将讨论该方法更广泛的影响、优势与局限性（第 6 节）、相关工作（第 7 节），并在第 8 节总结全文。

2 背景：HashMap

HashMap 类是 Java 类库 [15] 中标准的基于哈希的映射实现。在内部，它维护着一个桶表，每个桶包含零个或多个键，即哈希码映射到同一桶索引的键值对。桶表采用延迟分配，也就是说，直到发生第一次插入之前都不会预留内存。添加键值对时，会将键的哈希码与表容量（2 的幂）结合起来，通过按位掩码计算桶索引，其效果等同于将哈希码对表容量取模。

在正常情况下，每个桶都由一个条目链表表示。随着元素不断累积，映射项总数超过由当前容量和负载因子决定的阈值时，表会调整大小，容量翻倍。该阈值确保平均桶长度保持较小，从而维持查找、插入和删除操作的预期常数时间性能。值得注意的是，即使调用了 `clear()`，表也不会缩小，以避免后续插入时产生代价高昂的重新分配。初始容量 和负载因子 参数分别默认为 16 和 0.75，它们控制表的初始大小和调整大小的阈值（即 $16 \cdot 0.75$）。

在少数情况下，许多键发生冲突并进入同一个桶时，相应的链表可能会转换为平衡红黑树，这一过程称为树化 [34]，旨在将查找、插入和删除的性能从线性时间提升至对数时间，从而缓解性能下降。反过来，如果树化桶中的条目数降至阈值以下，它就会转换回链表。这些机制共同确保HashMap兼具平均情况下的常数时间操作和可靠的最坏情况性能保证。实践中，树化极少发生，通常表明哈希函数不合适。

3 设计与架构

本节介绍 MapReplay 的设计与实现原则。MapReplay 是一款面向 HashMap 的基于跟踪记录的基准测试生成工具。我们首先介绍本文使用的术语以及 MapReplay 的整体软件架构，随后阐述我们的高层目标，描述关键设计决策，讨论性能方面的考量，并概述跟踪器的设计与实现。

[2]MapReplay 和 MapReplayBench 可通过 https://github.com/usi-dag/MapReplay 公开获取。

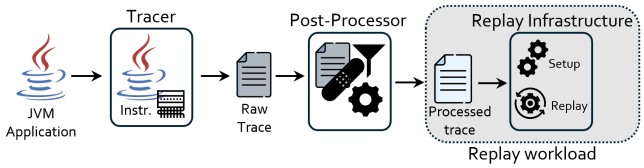


图 1：MapReplay 架构。

3.1 术语和 MapReplay 架构

MapReplay 系统由三个主要组件组成：跟踪器、离线跟踪后处理器和重放基础设施。图 1 展示了它们之间的交互。

跟踪器。 该跟踪器 通过为应用植入检测代码，捕获相关的HashMap 操作及其参数。它记录最少量的信息（例如操作类型、目标映射、键的标识和哈希码），以确保应用执行期间的开销较低。跟踪器的输出是原始跟踪，即一系列与HashMap相关的事件。

离线跟踪后处理器。 跟踪完成后，离线跟踪后处理器 会对原始跟踪进行清理和压缩。它会移除不完整的映射跟踪记录（例如缺少创建事件的记录）、合并迭代器操作、插入对象释放事件，并将操作编码为适合高效回放的紧凑格式。此步骤的输出是处理后的跟踪。

重放基础设施与重放工作负载。最后一个组件是重放基础设施，这是一个独立的执行框架，用于解析处理后的跟踪记录并执行相应的HashMap 操作。重放工作负载是将重放基础设施与处理后的跟踪记录配对后得到的一个特定基准测试实例。一个重放工作负载包含两个阶段：用于初始化模拟对象并分配数据结构的设置阶段，以及执行所记录映射操作序列的重放阶段。重放阶段构成由性能测量框架执行的基准测试方法（在 MapReplay 中即为 JMH [13]）。其单线程设计和极低的运行时开销，可确保对HashMap 实现进行忠实的性能评估。

3.2 目标

MapReplay 的核心思想是将应用建模为一系列映射操作，这些操作与其他非映射计算交错进行。我们只跟踪与映射相关的操作，例如插入操作、查找操作、删除操作和迭代操作，而忽略应用的所有其他逻辑。随后，重放工作负载会在等效的映射状态下重新执行相同的映射操作序列，从而重建HashMap 实现在原始应用中所经历的内部控制流。这种简化模型将映射本身的行为独立出来，使我们能够精确测量性能，并将关注点仅放在库代码上。自然，这种抽象忽略了若干可能影响实际执行性能的因素，包括并发和线程间竞争、垃圾回收活动、动态编译效应，以及回放产生的跟踪解释开销。尽管如此，它仍为现实且高效地评估HashMap 性能提供了实用而有效的基础。

忠实执行路径。MapReplay 的首要目标是确保重放工作负载中的映射操作——尤其是性能关键的操作——在库实现中执行与被跟踪应用相同的代码路径。这对于在真实条件下准确评估HashMap 实现中提出的优化至关重要。库代码中的控制流一旦出现偏差，就可能导致性能评估产生误导。

等效的映射状态。为了忠实地复现相关HashMap 执行路径，回放时每个映射的内部状态必须与跟踪执行期间的状态等效。由于许多HashMap 操作取决于桶占用情况、冲突链和调整大小状态等结构属性，因此确保回放后的映射状态等效至关重要。因此，我们的追踪与回放机制只捕获复现映射状态相同演变过程所需的信息。

3.3 设计决策

在上述设计目标的基础上，MapReplay 采用了若干关键决策，以兼顾保真度、效率以及轨迹收集和回放的实际可用性。

操作分类。 我们从两个维度对映射操作进行分类：(a) 是否属于性能关键 操作（需要忠实回放库代码路径）；(b) 是否会改变映射状态 （从而影响后续操作）。这种分类决定了追踪和回放的要求：

- (a)：性能关键的操作（可能改变状态，也可能不改变状态）会被追踪并忠实回放。例如：`get()`、`put()`、`remove()`。
- ((非 a) 且 b)：非性能关键但可能改变状态的操作会被追踪，但在回放期间可以用等效的状态转换来替代。例如：`compute()`。
- ((非 a) 且 (非 b))：既非性能关键也不改变状态的操作会被忽略。例如：`containsValue()`。这种二维分类为兼顾保真度和跟踪记录紧凑性提供了概念基础。

应用程序键的替换，最大限度减少回放中的回调。 库代码经常通过键对象上的 `hashCode()` 和 `equals()` 等方法调用应用代码中的回调（这些原语本身也是现代 JVM 上采用 SIMD 风格加速的目标 [3, 30]）。此外，HashMap API 的某些方法（例如 `compute()`）接受用户定义的函数或 lambda。由于 MapReplay 专注于对库代码而非应用逻辑进行基准测试，我们将用户定义的行为抽象掉。我们只记录键的必要属性（尤其是其哈希码），并在回放环境中将所有键替换为统一的模拟键类型。这种设计大幅减小了跟踪记录的大小，并避免了对应用类的复杂依赖。

保留哈希冲突。 为保持映射结构等价，我们准确保留所有键的哈希码。这确保了在回放期间，相同的哈希桶分配和冲突链能够再次出现，从而产生相同的库级控制流。

简化的相等语义。 保留键的相等关系需要耗费大量资源来跟踪等价类。

相反，MapReplay 强制采用基于标识的相等性：当某个操作引用了一个与现有键相等（但并非同一对象）的键时，跟踪记录会改为记录现有键。这样一来，回放时模拟键可以使用简单的 `equals()` 实现，同时仍能在映射中复现正确的行为和控制流。

忽略值。 我们不会保留映射中所存值的任何信息。在实际应用中，`containsValue()` 等操作很少见（且效率低下）；此外，应用可能直接修改映射中作为值存储的对象状态，而这将需要追踪任意字段写入。[3] 因此，这类操作不纳入追踪和回放。

追踪迭代器和分割迭代器。 涉及映射视图的操作——具体来说，是迭代器和分割迭代器——会被追踪，因为它们在实际应用中广泛使用，且往往与性能密切相关。为准确反映迭代和遍历模式，这些操作也会纳入回放。

异常处理。 应用中会抛出异常的操作（例如 `ConcurrentModificationException`）通常很少见，也不会修改映射状态（它们往往表明应用存在错误）。因此，我们不追踪或回放会引发异常的操作，使重放基础设施能够假设执行过程中不会发生异常。这样可以简化重放工作负载，并避免不必要的控制流开销。

无哈希映射多态性。为确保结果一致且可比较，MapReplay 会确保一次基准测试运行中实例化的所有哈希映射都属于同一类型。这一设计选择保证重放工作负载中的每个映射操作都从单态调用点调用，使即时编译器能够应用内联等激进优化。生成的编译后代码降低了回放的解释开销，并确保基准测试的大部分时间都用于执行待评估的库代码。为求简便，MapReplay 目前在追踪期间不会区分 `HashMap` 子类型；例如，原始应用中的`LinkedHashMap`实例不会单独回放。不过，扩展 MapReplay 以追踪 `LinkedHashMap` 等特定子类型并不困难。

离线跟踪清理。为尽量降低追踪的运行时开销，插桩保持简单。耗时的清理和一致性检查则在离线后处理阶段进行，其中包括移除无法捕获创建事件的映射操作，例如在 JVM 引导期间创建的映射，因为此时尚无法进行追踪。[4] 排除这些操作可确保重放一致性。

单线程重放。 即使原始应用使用了多个线程，重放工作负载仍是单线程的。如果线程操作的是不同的 `HashMap` 实例，则可以安全地重新排序这些操作。对于不可变共享映射，只读操作也可以重新排序。对于可变的共享映射，我们假设原始应用中采用了恰当的同步，从而保留临界区内映射操作的相对顺序。这种推理适用于 `HashMap` 这类非线程安全映射，但不适用于 `ConcurrentHashMap`等并发集合，我们将在第 6.3 节讨论这一点。

[3]请注意，映射中的键不会出现这一问题。Java 文档指出，“如果对象作为映射中的键时，其值发生了会影响相等比较的变化，则映射的行为未作规定” [15]。因此，我们假设所有键的相等关系保持稳定。[4]虽然 `HashMap` 实例早在 JVM 引导阶段就已创建，但即使是用本地代码编写的跟踪器（如我们的实现），也无法被调用，因为使用 JNI 会导致引导过程崩溃 [29]。

3.4 性能注意事项

为确保重放工作负载准确反映库级性能，MapReplay 的设计既保证跟踪记录紧凑，又最大限度地减少回放开销，同时保持对 `HashMap` 操作的忠实执行。

高效回放。 回放机制相对于库代码的执行必须产生可忽略不计的开销。因此，基准测试方法本身经过设计，仅执行最少的解码和分派工作，从而将重点放在待测映射操作的性能上。

合并迭代器操作。在许多跟踪记录中，`hasNext()` 和 `next()` 等迭代器方法出现得极为频繁，而且单次开销很低。逐一回放这些操作会使回放开销占据主导地位。因此，在跟踪记录后处理中，我们会将连续的迭代器或分割迭代器操作（例如 `tryAdvance()`）合并为聚合的回放事件，除非遇到`remove()`等修改操作而中断。基准测试通过短循环回放这些事件，执行相应数量的步骤。

紧凑的运行时编码。为降低解释开销，我们使用基本类型数组表示回放跟踪记录，将操作编码为包含操作数（例如映射标识符）的操作码。数组存储活动的`HashMap` 实例、迭代器和分割迭代器。这种底层编码可在回放期间实现快速解码和分派。

尽量减少对象分配。所有基准测试相关对象——包括模拟键和数据结构——都在基准测试设置阶段预先分配。回放期间，只创建必要的运行时对象，例如映射和迭代器。这可以减少回放期间无关的内存分配和垃圾回收活动；在分离<HashMap>本地影响时，也有助于降低测量噪声。

显式对象释放事件。为进一步减少内存占用和垃圾回收开销，跟踪记录包含显式对象释放事件，这些事件会在对象最后一次使用后立即发出。这些事件会将引用置为空，从而允许垃圾收集器提前回收对象。由于这些事件是在离线后处理阶段插入的，因此几乎不会影响回放开销，但有助于在重复进行跟踪记录回放期间保持自动内存管理的高效性。

3.5 跟踪器设计与实现

这里，我们讨论有关 MapReplay 跟踪器的主要设计选择，并介绍其实现细节。

源代码级插桩与部署。为了以最小干扰收集跟踪记录，MapReplay 通过在精心选择的内部位置直接修改 Java 源代码，对`HashMap`进行插桩。此插桩会记录与`HashMap`相关的事件

（例如，公开操作和选定的内部结果），同时保留原有的控制流和数据结构语义。无需重新编译 JVM 来使用经过插桩的 HashMap：正如相关工作 [48]，所示，我们在启动时通过修补 java.base 模块来部署经过插桩的HashMap （使用自 Java 9 起提供的模块修补机制 [35] ），使应用能够执行我们插桩后的类，以取代标准库版本。

基于 ***JNI*** 的跟踪记录。插入的插桩探针有意保持精简，并立即委托给以本地方法实现的跟踪记录例程。具体而言，每个探针只执行轻量的参数准备工作（例如计算键标识符），然后调用（通过 JNI [17]）一个本地方法，将相应事件追加到原始跟踪缓冲区中。在本地代码中实现记录逻辑，可以让 Java 侧探针保持简单，并避免内存分配和库调用，同时仍能实现高效 I/O 和紧凑的事件二进制编码。

避免插桩重入。一个关键挑战是防止插桩重入：如果追踪或分析逻辑本身执行了经过插桩的库代码，那么探针就可能在插桩执行期间再次触发，导致无限递归并最终失败（即栈溢出错误）。在对HashMap等核心库类进行插桩时，这个问题尤为突出，因为 Java 类库中处处都会用到这些类；因此，轨迹收集所需的看似无关的功能（例如初始化支持 I/O 写入的常见 Java 类）也可能间接分配或访问HashMap 实例。关于 JVM 可观测性的先前研究指出，这种自干扰是实现可靠库插桩的一项实际障碍，通常需要仔细隔离应用执行与分析逻辑 [29]。为了在 MapReplay 中可靠地避免重入，我们将 Java 侧探针保持精简，并在本地代码中实现跟踪记录逻辑，依赖 C 库而非 Java 类库。

源代码级插桩的替代方案。源代码级插桩的一种替代方案是，使用加载时字节码转换，或借鉴面向切面编程（AOP）原则的插桩框架来对 HashMap 进行插桩。实际上，对 Java 类库进行插桩往往很脆弱：许多字节码插桩框架主要针对应用类，而非库类；对 JVM 生命周期早期加载的类（如 HashMap）进行插桩，可能会干扰 JVM 初始化或引发自干扰问题，除非采取特别措施 [29]。不过，也有一些框架明确支持对核心类进行稳健的字节码插桩。例如，DiSL [31, 42, 43] 提供全面的字节码覆盖，并包含一种基于多态字节码插桩 [6, 33] 的动态绕过机制。该机制可确保分析代码针对选定类的未插桩版本执行，同时应用运行经插桩的版本，从而防止插桩重入。

我们没有为 MapReplay 采用这类框架，原因有三。首先，我们必须在 HashMap 内部多个细粒度位置布置探针（包括特定于控制流的方法中间位置），而且有时还必须根据映射状态采取不同操作；使用插桩框架来编码这些语义——这通常意味着要用受限的领域特定语言表达语义——其复杂度和出错风险都远高于直接修改源代码。其次，多个公开的 HashMap 入口点会相互委托（例如便捷重载和构造函数链式调用），我们必须避免重复记录同一逻辑操作；通过手动修改源代码来实施恰当的插桩很简单，但要通过 AOP 框架所需的通用、基于连接点的插桩可靠地实现这一点则很困难。第三，基于框架的方法会带来额外的运行时复杂度和开销，例如启动阶段的类转换，以及某些设计（如 DiSL）中的辅助进程和更大的内存占用 [44, 45]。而这些都不利于收集旨在如实反映库基准行为的跟踪记录。总的来说，我们发现，直接进行源代码级插桩并结合基于 JNI 的记录，是收集 HashMap 跟踪记录的最简单、最可靠的方法，同时还能将测量扰动和资源消耗控制在较低水平。

4 MapReplayBench

本节介绍 MapReplayBench，这是一个针对基于哈希的 Java 映射的回放工作负载基准测试套件。它通过将 MapReplay 应用于两个广泛使用的 Java 应用基准测试套件——DaCapo-Chopin [7, 8] 和 Renaissance [41]——生成。MapReplayBench 使研究人员和从业者能够在具有代表性的、源自应用的工作负载下，分析 HashMap 类以及 JVM 上其他基于哈希的映射实现的性能。

这两个套件中的基准测试并非都表现出显著的映射活动。表 1 描述了各基准测试对映射的使用情况。第一列“% CPU in HM”报告了原本在 HashMap 类中耗费的 CPU 时间百分比；我们用这一指标来定义工作负载的映射强度，并据此对表格排序。其余各列描述由 MapReplay 跟踪器收集并经离线后处理器优化的事件。

“# Event”列报告后处理后的回放事件总数。“# Create”列统计 HashMap 构造函数的所有调用。“# Read”包括所有检查映射状态的已跟踪公共方法（例如 get、containsKey）；“# Write”则包括所有修改映射的方法（例如 put、remove）。“# Iterate”概述遍历映射或其视图的操作，包括针对条目、键或值的迭代器和分割迭代器。最后，“Size”列报告处理后的跟踪记录的字节数。[5]

5 评估

在此，我们将 MapReplay 与现有的微基准测试和应用基准测试方法进行比较。我们通过一个简单但具有代表性的案例研究，展示这些方法的局限性：当前的基准测试方法在帮助开发人员确定 HashMap 的最佳默认初始容量（*DIC*）方面有多有效？

我们设计了一项实验，以评估将 DIC 从默认值 16 提高到 32、64 或 128 是否能带来可测量的

[5]每条跟踪记录都是一个压缩归档文件。跟踪记录的事件数量可能相近，但由于压缩的缘故，其大小可能存在显著差异（例如，助记符和参数助记符）。

表 1：DaCapo-Chopin（C-）和 Renaissance（R-）基准测试中的 HashMap 使用情况。原始基准测试执行过程中，HashMap 所占用的 CPU 时间百分比源自原始基准测试的运行。其他指标则源自 MapReplay 处理后的跟踪数据。

基准测试	CPU 百分比 在 HM 中	# 事件	# 创建	# 读取	# 写入	# 迭代	大小
R-par-mnemonics	27.43	66236024	1415980	43872832	18678894	1252797	102M
R-mnemonics	26.00	66203404	1415852	43872832	18646430	1252783	60M
C-biojava	23.49	394224921	150337	364892378	29031647	43352	258M
R-scrabble	16.44	9600913	1244173	672755	5197588	1243212	33M
R-scala-stm-bench7	13.67	49156302	35940	42071490	7013480	185	222M
R-fj-kmeans	6.76	102274181	411953	6565	101033413	410939	200M
C-h2	5.38	139423811	7242709	67714731	45978767	12938167	435M
R-rx-scrabble	3.87	1746158	50357	63296	1533327	49593	6.8M
C-tomcat	3.34	13220831	70100	6762075	5924407	317981	59M
C-kafka	3.16	12116302	67034	8166290	3503230	328965	41M
C-pmd	2.93	15514296	1921133	6116013	4252211	2908055	68M
C-xalan	0.98	3086931	74250	2419000	534399	17597	9.0M
C-spring	0.91	7762498	431686	5040616	1524973	462283	28M
C-fop	0.69	937499	40275	759269	94830	19579	976K
C-avrora	0.61	2513654	319	2509962	3019	161	1.6M
C-eclipse	0.61	4089948	116315	2762388	1030205	93647	8.5M
R-movie-lens	0.49	2909774	401752	1303789	788540	36264	8.7M
R-log-regression	0.46	619471	78414	320235	161589	9802	1.6M
R-finagle-http	0.42	3141156	18302	3102359	18844	539	8.5M
C-h2o	0.37	615037	26745	440792	114519	9474	1.1M
C-lusearch	0.36	7956080	48572	7732258	99144	34982	18M
R-finagle-chirper	0.28	2598361	5806	2576093	15241	357	6.4M

基准测试	CPU 百分比 在 HM 中	# 事件	# 创建	#读取	#写入	# 迭代	Size
R-naive-bayes	0.28	501175	57466	264564	140864	15823	1.1M
C-batik	0.27	396511	2031	338706	52196	1769	400K
C-jython	0.21	2563693	540457	1939448	72575	9763	2.0M
R-gauss-mix	0.11	497218	75100	214081	129394	11954	1.4M
R-决策树	0.10	495801	47258	311509	104307	9659	1.1M
R-交替最小二乘	0.10	503414	65069	255852	131942	7855	1.4M
C-luindex	0.08	2393138	1001	2388438	2655	559	232K
R-neo4j-analytics	0.08	8078074	593295	2930709	3411231	570423	3.8M
R-卡方	0.02	111593	12953	53919	37082	2809	256K
C-zxing	0.02	578096	23171	235569	254232	42651	1.1M
R-page-rank	0.02	127214	16085	57973	41032	5498	304K
R-dotty	0.02	27815	9816	9054	8576	171	80K
C-jme	0.02	11189	911	4627	4946	366	36K
R-Akka-UCT	0.01	157637	5932	80118	60358	6895	460K
C-graphchi	0.00	14595	2180	4016	5987	508	44K
R-scala-doku	0.00	16087	1087	6736	8013	139	52K
R-reactors	0.00	16651	1487	6898	7869	216	52K
C-sunflow	0.00	3293	376	474	2138	167	16K
R-future-genetic	0.00	10285	1117	3594	5323	139	32K
R-philosophers	0.00	15938	954	6713	8017	140	52K
R-scala-kmeans	0.00	15554	722	6574	8009	138	52K

性能提升。提高初始容量可以减少调整大小的频率、缩短冲突链，并通过提高探测到空桶的可能性来加快未命中查找。不过，更大的表会带来更高的内存分配成本，在稀疏填充的桶上迭代速度也更慢，且局部性更差。这些权衡的总体影响高度取决于具体应用。我们的目标是比较 MapReplay 与微基准测试及应用基准测试，以确定较高的 DIC 值是否能提升性能。

5.1 实验设置

我们在一台配备 16 核 Intel Xeon Gold 6326 处理器（主频 2.9 GHz）和 256 GB 内存（频率 3.2 GHz）的机器上进行评估，运行 Ubuntu 22.04 LTS（内核 5.15.0-25-generic）。为确保性能测量稳定且可复现，我们禁用了超线程和睿频。所有实验均使用 JDK 25（Oracle JDK25，构建版本 25.0.1+8-LTS-27）。

对于每个案例研究，我们采用撰写本文时可用的最新版本 JMH （ 1.37 版），并严格遵循最佳实践，以避免常见的基准测试陷阱 [19]。每项 JMH 配置均包含 5 次 JVM 运行，每次运行先进行 5 次预热迭代，再进行 5 次测量迭代。对于微基准测试，我们将每次迭代的时长设为 500 毫秒，因为单次基准测试调用执行很快（通常为几纳秒到 1 毫秒）。对于回放工作负载，我们将每次迭代的时长设为 10 秒，因为单次调用可能耗时更长（最长约 3.5 秒）。对于应用基准测试，我们执行 30–100 次迭代（视工作负载而定），以确保充分预热，并收集 30 次 JVM 运行的结果，以减轻测量噪声。

5.2 微基准测试

在此，我们展示微基准测试的局限性：它们往往无法提供足够的信息，来判断是否应将某项优化纳入标准库。

为此，我们使用了一个现有的微基准测试套件，旨在研究 Java 集合 [18] 的性能。我们围绕案例研究开展性能评估，以确定哪个 DIC 值能为`HashMap`带来最佳性能。具体而言，我们测量了不同输入规模下的性能，并特别关注接近触发扩容操作阈值的规模。

我们仅关注执行时间，而不考虑内存消耗等次要因素。观察发现，性能很大程度上取决于操作的组成、每种映射大小下执行的操作数量，以及键分布的选择。值得注意的是，由于微基准测试具有合成性质，它们采用三种固定的键分布，在每种映射大小下执行定时限定的工作量，并保持由四种权重相同的操作（即 `contains`、复制、迭代 和 填充）构成的均匀操作组合。

图 2 报告了在不同 DIC 设置下，这四种操作的观测吞吐量。`contains` 和`populate` 操作通常受益于更大的 DIC，因为更大的表有助于减少哈希冲突。对于`contains`而言，查找延迟与哈希冲突链长度直接相关：在所有映射达到相当的内部大小（即 ≥ 49 个元素）之前，采用 DIC-128 的映射性能优于较小的配置。每次扩容事件后，性能通常都会提升（例如，在 13、25 和 49 个元素时）。由于每次执行中键的随机选择，性能仍存在一定波动 [18], 不过，该基准测试确保了不同运行和 DIC 配置之间的确定性。

对于`populate`，每次迭代都会分配一个新的映射。只有当插入元素的数量足够多时，较大的 DIC 才能提升性能；否则，中等大小的表能在分配成本和冲突率之间实现更好的平衡。

相比之下，复制 和迭代 操作的性能往往会随着 DIC 增大而下降。两者都会遍历所有桶，因此在底层表较稀疏时会产生额外开销。调整大小的影响在迭代操作中尤为明显：每次调整大小都会使桶数组翻倍，导致较小 DIC 配置的性能

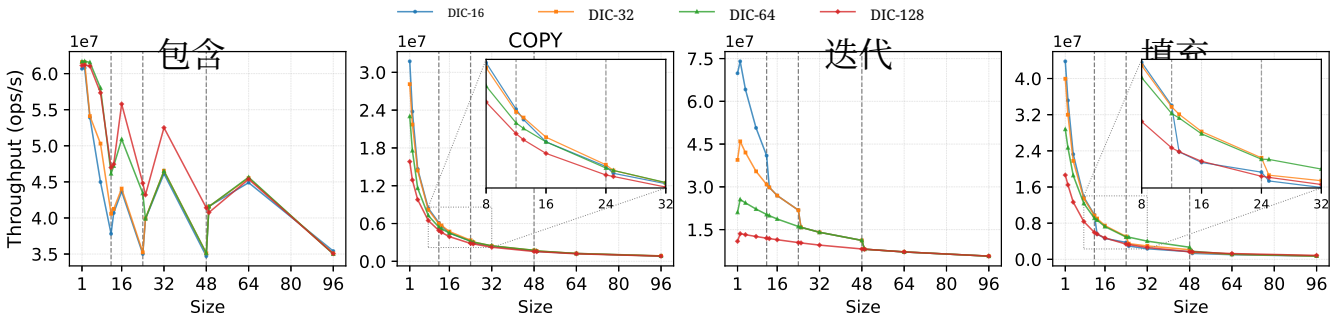


图 2：不同操作、映射大小和 DIC 配置下的微基准测试性能。垂直灰线表示发生调整大小时的阈值。

与下一个更大的配置趋于一致。当所有变体的内部容量都达到 128（含 ≥ 49 个元素）时，它们的性能便趋于一致。复制操作也呈现出类似的趋势，不过由于其内存分配开销较大，趋势较难解读。

基于微基准测试，很难确定库中应使用哪个 DIC 值。最优选择取决于映射大小的分布、操作的相对频率以及键分布等因素——这些因素在微基准测试中是合成的，可能无法反映真实的应用行为。MapReplay 通过对源自真实工作负载的实际映射使用情况进行追踪和回放，克服了这些局限性。

5.3 应用程序基准测试分类

为了评估不同工作负载对 DIC 变化的反应，我们根据各基准测试在HashMap 操作上花费的执行时间百分比[6]（表 1 中的“% CPU in HM”列）对所有基准测试进行分类。该指标反映了各工作负载使用HashMap 的强度，因此也反映了它们对 DIC 变化的敏感程度。我们区分出三类：

映射密集型（$\%CPU in HM \geq 5\%$）这些基准测试在 HashMap 上花费了大量执行时间，预计会对 DIC 变化较为敏感。

映射中等型（$\%\ CPU\ in HM < 5$ 且事件数 $\geq 100,000$）。这些工作负载会较为频繁地使用HashMap，但映射操作并非主要开销。由于HashMap 操作仅占整体执行的一小部分，因此其性能通常较少受到 DIC 变化的影响。

映射极少型（$\%\ CPU\ in\ HM < 5$ 且事件数 $< 100,000$）。这些工作负载中的HashMap 活动微乎其微。DIC 修改通常不会产生可测量的影响，而且收集的跟踪数据往往太小，缺乏代表性。因此，我们将这些基准测试排除在评估之外。如表 1 所示，它们的跟踪数据大小通常只有几千字节。

5.4 应用基准测试

表 2（左）报告了 DaCapo-Chopin 和 Renaissance 基准测试套件中不同 DIC 配置的执行时间和加速比

[6]我们使用 profhot [36]收集了 HashMap中的 CPU 时间。

大多数应用基准测试（不包括地图密集型类别中的少数几个）在改变 DIC 时都没有出现可测量的性能变化。这是意料之中的——许多工作负载并非映射密集型（如表 1 所示）。

相比之下，地图密集型类别中的几乎所有基准测试在改变 DIC 时都会表现出性能差异（ R-fj-kmeans 除外）。少数中等映射密集度基准测试也表现出敏感性，其中最明显的是 R-rx-scrabble。然而，表现出显著差异的基准测试数量较少，因此难以确定理想的 DIC 配置。

此外，要检测应用基准测试中幅度较小但具有统计显著性的性能变化，需要收集多次运行的测量数据；每次运行都必须持续足够长的时间，以度过大量代码编译所需的初始预热阶段。这意味着开发者需要等待很长时间，才能获得有关配置变更的反馈。在这个具体案例中，我们收集了 30 次运行的数据，每个基准测试采用特定的重复次数，每种 DIC 配置耗时约 72 小时。

5.5 MapReplayBench

表2（右侧）报告了使用 MapReplayBench 得到的结果，即通过追踪并回放 DaCapo-Chopin 和 Renaissance 套件中的工作负载所生成的回放工作负载。

MapReplayBench 比应用基准测试更能凸显与 ***HashMap***相关的性能变化。尽管回放工作负载可能带来诸多好处，但对于我们研究的场景，尤其是在性能表现方面，必须确认它们能够保留相应应用基准测试中观察到的 HashMap 相关性能变化趋势。鉴于回放工作负载提取的是HashMap特定事件，我们预计它们对HashMap相关代码变更会更加敏感，从而在推动代码优化时助力做出更好的决策。

一般而言，尽管通过回放工作负载发现的性能变化可能无法通过应用基准测试检测到（或者需要运行多次才能检测到），但我们预期回放工作负载能够确认应用基准测试发现的变化，而且这两类基准测试不会

表2：DaCapo-Chopin（C-）和 Renaissance（R-）在不同 DIC 值下的应用基准测试与回放工作负载的性能比较（执行时间，单位为 ms/op）。水平线以上的基准测试为映射密集型 （% CPU 占 HM ≥ 5），水平线以下的基准测试为映射中等型。由于无关，省略了映射极少型类别中的基准测试。

	应用基准测试				回放工作负载			
基准测试	DIC-16	DIC-32	DIC-64	DIC-128	DIC-16	DIC-32	DIC-64	DIC-128
R-par-mnemonics	2051±25.7	2050±29.0（1.00x）	2007±26.9（1.02 倍）	2069±31.9（0.99 倍）	958±38.2	928±37.7（1.03 倍）	878±40.5（1.09 倍）	885±22.6（1.08 倍）
R-mnemonics	2499±28.6	2473±26.3 (1.01 倍)	2420±19.3 (1.03x)	2532±35.9（0.99 倍）	1022±101.8	979±56.4 (1.04 倍)	932±10.0（1.10 倍）	963±53.2（1.06 倍）
C-biojava	8099±22.0	8062±24.2 (1.00x)	7944±32.8（1.02 倍）	7924±31.1（1.02 倍）	3781±861.3	3592±137.4（1.05 倍）	3344±59.4 (1.13 倍)	3349±137.7（1.13 倍）
R-scrabble	59±0.5	61±0.6（0.97 倍）	67±1.4（0.88 倍）	80±3.2（0.73 倍）	331±2.8	358±5.5（0.92 倍）	378±8.4（0.88x）	463±8.2（0.72x）
R-scala-stm-bench7	758±15.9	756±14.1（1.00 倍）	744±10.3（1.02 倍）	748±10.9 (1.01 倍)	1006±8.3	982±41.5（1.02 倍）	986±4.6（1.02 倍）	953±32.8（1.06 倍）
R-fj-kmeans	988±2.3	990±3.5（1.00x）	985±3.0（1.00x）	987±3.0（1.00x）	1730±68.8	1600±142.1（1.08 倍）	1460±188.1（1.18 倍）	1559±57.4（1.11 倍）
C-h2	2437±8.9	2435±11.9（1.00倍）	2452±12.7 (0.99 倍)	2480±11.2（0.98x）	3065±139.6	3034±43.3 (1.01 倍)	3086±316.6（0.99 倍）	3170±56.4（0.97 倍）
R-rx-scrabble	102±0.7	102±0.7 (1.00x)	103±0.7 (0.99 倍)	107±0.6（0.95x）	67±1.0	67±1.5（1.01 倍）	69±0.8 (0.98 倍)	69±1.9（0.98 倍）
C-tomcat	6809±1.7	6809±2.1（1.00 倍）	6810±1.6 (1.00倍)	6811±1.6 (1.00倍)	398±8.5	377±21.2 (1.05倍)	364±5.9 (1.09倍)	368±22.8 (1.08倍)
C-kafka	5014±0.9	5015±1.3（1.00x）	5015±1.2（1.00x）	5014±1.0（1.00x）	203±10.9	198±9.3（1.02x）	184±6.8（1.10x）	188±7.6（1.08倍）
C-pmd	1395±7.9	1399±10.1（1.00倍）	1393±8.9（1.00倍）	1394±12.2（1.00倍）	489±26.7	495±22.2（0.99x）	483±25.7（1.01x）	473±10.0（1.04x）
C-xalan	517±2.6	514±3.1（1.00 倍）	517±2.7（1.00 倍）	516±3.1（1.00 倍）	100±1.8	106±3.3（0.94 倍）	103±5.2（0.97 倍）	96±1.9（1.04 倍）
C-spring	1940±35.9	1941±38.3（1.00 倍）	1931±48.6（1.00 倍）	1938±40.1（1.00 倍）	201±5.2	178±5.0 (1.13 倍)	182±12.2（1.11 倍）	184±1.7（1.09倍）
C-fop	487±1.0	487±0.7（1.00 倍）	486±0.9（1.00 倍）	488±1.5（1.00 倍）	7±0.4	6±0.4（1.15 倍）	6±0.1（1.11 倍）	6±0.3（1.12x）
C-avrora	3673±6.0	3673±4.3（1.00x）	3673±3.7（1.00x）	3671±5.6（1.00x）	13±0.4	12±0.1（1.10 倍）	10±0.0（1.24 倍）	10±0.1 (1.23倍)
C-eclipse	12337±18.1	12333±21.6 (1.00倍)	12339±23.5（1.00倍）	12336±21.2 (1.00 倍)	54±10.5	61±7.5（0.89 倍）	62±13.3 (0.88x)	61±10.2（0.89 倍）
R-movie-lens	6793±15.0	6806±14.5（1.00x）	6809±19.9（1.00x）	6809±13.8 (1.00x)	56±0.3	52±2.3（1.07 倍）	51±3.5（1.10 倍）	51±3.6（1.10 倍）
R-log-regression	561±0.9	566±1.1（0.99倍）	561±0.7（1.00倍）	561±1.1（1.00倍）	12±0.6	12±0.6（1.02倍）	12±0.5（1.03倍）	12±0.2（1.04 倍）
R-finagle-http	2009±8.9	2007±7.5（1.00x）	2002±8.0（1.00x）	2005±7.6 (1.00倍)	16±3.3	15±0.2 (1.05倍)	16±3.2 (0.98倍)	17±2.6 (0.94倍)
C-h2o	2942±11.9	2929±14.9（1.00 倍）	2937±20.2（1.00 倍）	2934±18.9（1.00 倍）	7±2.3	6±1.6（1.07 倍）	7±1.3（1.03 倍）	6±1.4（1.16 倍）
C-lusearch	1830±19.7	1838±12.2 (1.00x)	1829±17.4（1.00 倍）	1829±17.7 (1.00x)	63±1.8	63±1.0（1.00x）	58±20.6（1.08 倍）	65±2.7（0.97x）
R-finagle-chirper	1597±9.2	1598±5.9 (1.00x)	1592±5.7 (1.00x)	1594±7.2 (1.00x)	14±0.3	14±0.3 (0.99 倍)	14±0.2（0.98 倍）	14±0.1 (0.97x)
R-naive-bayes	256±1.9	255±2.2 (1.00x)	257±3.1 (0.99x)	257±6.4（1.00 倍）	10±0.8	10±1.1（1.01 倍）	10±1.0 (1.01 倍)	10±0.7 (1.02倍)
C-batik	1670±1.4	1671±2.2 (1.00 倍)	1670±0.9（1.00x）	1670±1.3 (1.00x)	4±0.2	4±0.1（1.01 倍）	4±0.1（1.05 倍）	4±0.3 (1.03x)
C-jython	3894±83.0	3866±6.5（1.01 倍）	3868±6.4 (1.01 倍)	3866±7.5（1.01 倍）	18±0.3	18±0.3（1.00 倍）	18±0.8（0.99 倍）	18±0.9（0.98 倍）
R-gauss-mix	4928±18.3	4933±13.5（1.00x）	4924±16.1（1.00 倍）	4922±18.3（1.00 倍）	11±0.2	11±0.4（1.00x）	11±0.2（0.98 倍）	11±0.4（0.96 倍）
R-决策树	631±4.7	632±4.8（1.00x）	631±4.8（1.00x）	632±4.6 (1.00倍)	9±0.5	9±0.5（1.00x）	9±0.1（1.02 倍）	9±0.8 (1.03 倍)
R-交替最小二乘	1337±1.5	1337±2.5（1.00 倍）	1336±1.2（1.00 倍）	1338±2.0（1.00 倍）	11±0.5	10±0.7（1.06 倍）	10±0.1（1.07 倍）	10±0.6（1.10 倍）
C-luindex	4643±20.7	4654±21.6 (1.00倍)	4646±21.7 (1.00x)	4643±15.2 (1.00x)	10±0.0	10±0.2（1.02 倍）	10±0.3 (0.99 倍)	10±0.0（1.00x）
R-neo4j-analytics	1229±8.2	1227±7.3（1.00x）	1226±8.3 (1.00x)	1226±6.3（1.00x）	107±20.6	108±8.5（1.00倍）	117±12.0（0.92倍）	141±12.2（0.76 倍）
R-卡方	401±1.7	402±2.0（1.00x）	401±1.8（1.00x）	401±3.3（1.00 倍）	3±0.1	3±0.1（0.99 倍）	3±0.1（0.99x）	3±0.0（1.00x）
C-zxing	1224±3.0	1223±3.2 (1.00 倍)	1223±3.2 (1.00 倍)	1222±3.8 (1.00 倍)	8±1.2	8±1.5 (1.04 倍)	7±0.4 (1.11 倍)	9±0.9（0.93 倍）
R-page-rank	2702±19.7	2701±22.2 (1.00x)	2703±24.2 (1.00x)	2694±13.8（1.00x）	4±0.1	4±0.1（1.00 倍）	4±0.1（1.01 倍）	4±0.1（1.00 倍）
R-Akka-UCT	6209±33.2	6207±48.2（1.00 倍）	6193±45.9（1.00 倍）	6180±37.5（1.00 倍）	3±0.3	2±0.3（1.04 倍）	2±0.4（1.06 倍）	3±0.3（1.00x）
几何平均值		（1.00x）	（1.00x）	（0.99x）		（1.02x）	(1.04x)	(1.01x)

报告在统计上显著且相互矛盾的变化。请注意，由于我们缺乏真实值，评估基于统计假设检验，因此可能会产生统计误差。

假设有一位开发者希望使用回放工作负载来确定 DIC 参数的最佳值，我们将重点关注与本案例研究最相关的 7 个映射密集型类别基准测试。

我们首先考察回放工作负载的变化与应用基准测试之间的相关性。图 3 展示了观测到的加速比散点图，并附有线性趋势线。符号分别表示统计显著变化、方向一致的变化（统计上不显著，方向相同）以及弱不一致变化（至多只有一项具有统计显著性，且方向相反）。[7]矩形表示不一致结果的置信区间。

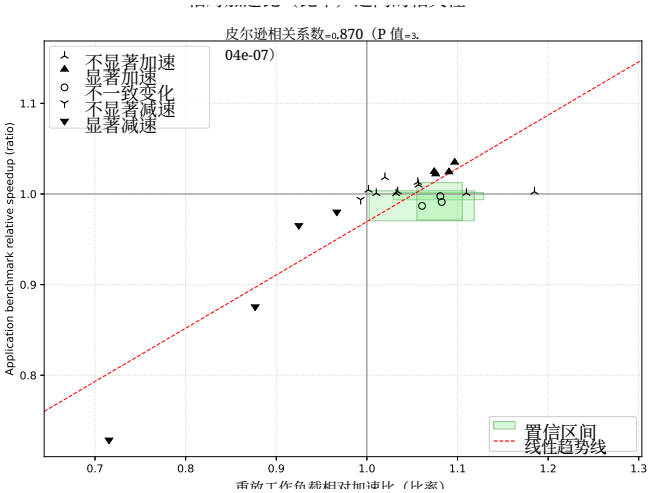
这些比率的皮尔逊相关系数为 $r = 0.870$，这表明二者之间存在很强的正线性关系。极低的 P 值使我们有 99% 的把握拒绝「不存在相关线性关系」这一零假设。虽然这与我们的预期一致，即两类基准测试之间性能变化的方向和幅度彼此相关，但相关性并未提供有关具体变化的信息。

为此，我们按照表 3 所示对观测到的性能变化进行分类。应用基准测试报告了 10 项显著性能变化，而回放工作负载报告了 18 项显著变化。表 3 中的重叠组显示，显著变化中有 8 项也由回放工作负载报告；其余两种情况（DIC=32 对应的 C-biojava，以及 DIC=64 对应的 C-h2）中，回放工作负载显示执行时间平均值发生了方向一致（但统计上不显著）的变化。这一结果符合预期。

重叠组显示了 3 个弱不一致结果的案例，即统计显著与统计不显著且方向相反的结果组合。图 3 中的图表表明，这些结果接近边界；鉴于回放工作负载的结果仅来自 5 次运行（我们将在后文讨论），而应用基准测试的结果来自 30 次运行，因此这可能是实验设置造成的伪象。

[7]如果执行时间平均值之差的 99% 自助法置信区间不包含零，则认为性能变化（加速或减速）具有统计显著性。每项比较的 BCI 均使用 50000 个自助抽样样本计算得出。

图 3：应用基准测试与回放工作负载中相对加速比（比值）之间的相关性。不同符号表示加速比组合在统计学上的显著性。矩形表示使用 ∘ 符号绘制的不一致程度较弱的变化所对应加速比的 99% 自助法置信区间。



虽然假设检验无法对单个在统计学上不显著且不一致程度较弱的变化作出进一步判断，但我们可以检验：在 21 次比较中观察到 18 次一致变化是否可能纯属偶然。如果两个基准测试系列中的变化方向相互独立，那么观察到一致变化和不一致变化的可能性应当相同。

因此，我们进行单侧二项检验，以 $p = 0.5$ 作为零假设概率，检验 21 次试验中出现 18 次正向结果（一致变化）的情况。正向结果的比例为 0.857，检验得出的 P 值为 0.0007，因此可以拒绝零假设。此外，该比例与 0.5 相距甚远（Cohen's $h = 0.796$），因此我们得出结论：在回放基准测试和应用基准测试中观察到的变化方向并不相互独立。

因此，对于所考察的用例，我们得出结论：回放工作负载可以确认在应用基准测试中同样可观测到的、与 `HashMap` 相关的性能变化，但由于其中包含大量非 `HashMap` 代码，这些变化在应用基准测试中往往不太明显。

MapReplayBench 比应用基准测试更敏感。借助 MapReplayBench，即使在映射中等型基准测试之间，我们也能观察到明显的性能差异——这与应用基准测试形成鲜明对比。虽然只有 7 个应用基准测试在不同 DIC 配置之间表现出可测量的差异，但有 27 个回放工作负载显示出统计显著变化。

MapReplayBench 还使我们能够找出理想的 DIC 值——这是微基准测试和应用基准测试都无法可靠揭示的。微基准测试只能针对孤立操作提供见解，而应用基准测试对 DIC 变化的敏感度很低。相比之下，MapReplayBench 表明，DIC 为 64 时几何平均加速比最高（1.04×）。我们

表 3：应用基准测试和回放工作负载所显示的、相对于 DIC =16 对应基线的性能变化。 ⊕ 和 ⊖ 符号分别表示具有统计显著性的加速或减速。 + 和 – 符号表示执行时间平均值的变化趋势，但不能视为具有统计显著性。重叠项综合了应用基准测试和回放工作负载的指标。两个方向一致且均具有统计显著性的变化组合予以保留；具有统计显著性和不具有统计显著性的变化方向一致时，报告为不显著；不一致变化则用 | 符号连接后报告。

	回放			应用			重叠		
基准测试/DIC	32	64	128	32	64	128	32	64	128
R-par-mnemonics	⊕	⊕	⊕	+	⊕	-	+	⊕	⊕|-
R-mnemonics	⊕	⊕	⊕	+	⊕	-	+	⊕	⊕|-
C-biojava	+	⊕	⊕	⊕	⊕	⊕	+	⊕	⊕
R-scrabble	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖	⊖
R-scala-stm-bench7	⊕	⊕	⊕	+	+	+	+	+	+
R-fj-kmeans	⊕	⊕	⊕	-	+	+	⊕|-	+	+
C-h2	+	-	⊖	+	⊖	⊖	+	-	⊖

不过请注意，我们的评估未考虑较大初始容量所带来的内存占用；要得出最终结论，还需要进一步调查。

MapReplayBench 比应用基准测试能够更快地提供反馈。基于我们的案例研究，在为回放工作负载选择 JMH 配置时，我们设想自己是一位熟悉 JMH、但不想花太多时间针对特定基准测试微调 JMH 配置的开发者。在这种情况下，选择从 5 次运行（JMH 分叉）中收集数据，每次运行包含 5 次预热迭代和 5 次测量迭代，每次迭代持续 10 秒，似乎是合理的选择，也能兼顾运行之间和运行内部的一些差异。采用此配置，所有回放工作负载的实验执行时间约为每种 DIC 配置 8 小时。[8] 尽管运行次数非常少，回放工作负载仍报告了许多统计显著变化。

6 讨论

本节讨论 MapReplay 方法更广泛的影响、优势和局限性。

6.1 MapReplay 作为互补的基准测试方法

MapReplay 是一种补充性基准测试方法，介于微基准测试和完整的应用基准测试之间。它的主要优势在于，能够从真实程序执行中提取与 `HashMap` 相关的活动，生成回放工作负载，从而在保留真实使用模式的同时减少应用程序级噪声。这使得人们能够在受控环境中，快速且可重复地试验配置选项和实现变体。

[8]我们注意到，对于某些多线程基准测试（例如 C-h2、R-scrabble），原始基准测试运行速度快于相应的重放工作负载，因为后者是单线程的。

在实践中，当应用基准测试运行成本高昂，或对 HashMap 级别的变化不够敏感时，这一点尤其有价值，因为在流行的 Java 基准测试套件中，基于哈希的映射操作通常只占端到端执行时间的一小部分（如第 4 节所示）。

回放工作负载会有意省略与 HashMap无关的事件，因此并不旨在重现跨组件效应，例如由非映射分配引发的垃圾回收压力、横跨整个应用的复杂即时编译交互、受周边代码影响的底层架构效应，或多线程竞争与同步模式。因此，MapReplay 最适合用于独立研究库级设计选择和性能权衡，以及快速筛选假设；应用基准测试仍适用于验证端到端影响和系统级交互。

总体而言，我们将 MapReplay 定位为一种实用的折中方案：它保留了源自应用的真实 HashMap 用法——通常比高度合成的微基准测试更具代表性——同时显著降低了实验成本，并在目标是理解 HashMap局部行为时提升了可操作性。

6.2 MapReplay 的优势

MapReplay 具有多项显著优势，能够提升基于哈希的映射基准测试的实用性和分析能力。下面概述其最重要的优势，重点介绍它的灵活性、可复现性，以及它在广泛场景下进行详细性能探索的适用性。

跨应用程序自动生成基准测试。 MapReplay 的一项实用优势是，它能够从各种 Java 应用程序中生成真实的 HashMap 回放工作负载，即使由于运行时依赖或环境限制，无法实际运行完整的端到端基准测试也是如此。MapReplay 会在目标应用程序正常执行期间直接捕获相关的 HashMap 活动，回放时无需重新实例化任何外部资源。由此生成的回放工作负载完全自包含，可在任何 JVM 上执行，不受原始应用程序环境的限制。这一能力使得以极少的工作生成新的、有代表性的基准测试成为可能，极大拓宽了可在受控条件下研究的真实场景范围。

设计变体的灵活性与受控探索。MapReplay 的一大优势在于，它能够提供灵活且可控的环境，用于评估兼容HashMap的替代实现和设计选择。由于在 Java 类库中直接修改HashMap 并不安全——它与 JVM 深度集成——MapReplay 将基准测试与 JVM 对该类的内部使用解耦，使记录的轨迹能够通过任何遵循相同 API 的基于哈希的映射实现进行回放。这包括标准 HashMap、其实现的隔离副本（以避免现有即时编译配置文件造成的干扰），以及包含复杂依赖项或优化的实验性变体。同时，回放的基准测试会复现完全相同的映射操作序列，使研究人员能够探索不同负载因子、哈希函数、调整大小策略等设计变体，或在恒定工作负载下采用不同的哈希冲突解决机制。对输入和执行环境的严格控制，有助于进行公平、直接的比较；使用传统应用基准测试很难做到这一点，同时还能避免 JVM 不稳定，并确保运行时编译行为不受偏差影响。因此，MapReplay 不仅是一个安全且可扩展的平台，可用于隔离和测试新的映射实现，也是一个系统化的实验框架，可用于分析类似HashMap的数据结构对特定设计决策的敏感性。

隔离应用程序级噪声。 我们的方法只复现应用程序与目标库交互时的行为，舍弃观测到的库之外的无关计算。通过仅捕获并回放映射操作序列及其运行时状态，MapReplay 消除了应用程序其余部分带来的测量开销。这种有针对性的回放能够隔离实现变更在库级别产生的影响，同时保留真实的操作组合和状态，从而确保测量结果具有代表性。

6.3 MapReplay 的局限性与未来工作

尽管 MapReplay 提供了一个强大而高效的框架来研究映射性能，但其设计也伴随着固有的权衡和范围局限。下文将探讨这些约束，阐明其根本原因，以及它们为未来扩展和方法改进带来的机会。

所采集跟踪记录的代表性。 一个核心的方法论问题在于记录的轨迹是否具有代表性。MapReplay 会重现跟踪记录所捕获的 HashMap 操作序列（例如，与控制流相关的键属性），同时抽象掉其他应用影响，但它无法推广到该特定工作负载之外。HashMap 的性能表现可能因输入特征、应用程序阶段和工作负载强度而有很大差异。因此，如果跟踪记录的执行仅反映了一个狭窄的运行阶段或非典型的输入分布，重放工作负载就可能无法涵盖现实世界使用模式的多样性。这是基于轨迹的基准测试固有的局限：回放的保真度受所收集跟踪记录的覆盖范围和真实性所限。为缓解这一问题，可以汇总多次具有代表性的执行所产生的跟踪记录。

即时编译的影响。 我们的方法存在一项根本局限，源于多层次 JVM 中动态（即时）优化编译器的行为，例如 HotSpot VM 中的 C2 编译器或 GraalVM 中的 Graal。跟踪记录所对应的应用与重放工作负载之间，内联和优化决策可能存在显著差异。这种差异在回调中最为明显，例如调用 hashCode() 或 equals() 方法；在实际应用中，这些方法可能呈现多态性，并执行复杂的用户代码。相比之下，在我们的基准测试中，回调是单态且简单的，只包含一个基本块。尽管这些差异可能改变内联行为，但通常会降低重放工作负载中的回调开销，从而符合我们的目标：隔离并衡量库级性能，而非特定于应用的逻辑。

不涵盖低层次架构效应。我们的方法并不试图重现缓存效应、内存局部性或瞬时内存压力等低层次架构现象。这些特性本质上取决于应用的计算模式和数据访问行为，而这些内容在我们的回放中被刻意排除。MapReplay 将执行集中在 HashMap 库代码上，从而抽象掉无关的应用逻辑。因此，尽管我们的基准测试能够高度忠实地呈现映射操作的控制流和结构演变，但并不能重现硬件级性能特征。对于我们评估算法和库级优化的目标而言，这一局限可以接受；但研究微体系结构效应的研究人员则需要采用互补的方法。

关于并发映射的局限性。目前，即使被跟踪的应用是多线程的，MapReplay 也只提供单线程基准测试。我们对正确性的推理依赖于这样一个假设：应用中的任何共享可变 HashMap 实例都以线程安全的方式使用——通常通过同步，确保每个映射操作都在执行时所在的同一临界区内被原子地跟踪。此假设可确保跟踪记录中的映射状态演变与回放一致。然而，这一推理并不适用于依赖细粒度同步或无锁机制的并发映射实现，例如 ConcurrentHashMap [14]。将 MapReplay 扩展到这类场景留待未来工作。

扩展到基于树的映射。目前，MapReplay 专注于基于哈希的映射，不支持 TreeMap [16] 等基于树的数据结构。要将我们的方法扩展到树映射，需要保留键顺序信息，以确保模拟键与原应用程序键的比较结果一致。HashMap本身偶尔会对较长的桶进行树化，将其转换为红黑树；不过，这种情况在实践中很少见。MapReplay 在发生树化时能够正确运行，但无法保证跟踪执行与回放执行中的树化桶在结构上等价。要在这些情况下实现忠实回放，需要捕获键之间的比较关系。如何处理有序映射、偏序映射以及树化的HashMap 桶，仍是持续研究的课题。

推广到其他集合类型。尽管支持 HashMap及其衍生类型 HashSet，MapReplay 尚未涵盖其他集合类型，例如列表或内部结构各异的集合。我们预计这一方向的主要挑战在于，集合操作可能依赖存储对象的状态，而应用可能会修改这些状态。捕获并回放影响此类对象的所有状态变化，代价将高得难以承受。一种可行的替代方案是记录每项操作的内部执行步骤序列，并合成人工构造的容器状态，使基准测试能够触发等效的内部行为，方式类似于我们对模拟键的处理。这一策略也适用于方法 HashMap.containsValue()，该方法目前尚未纳入支持范围。将 MapReplay 推广到类映射数据结构之外，在概念上很直接，但在实践中颇为复杂，因此是未来工作的重要方向。

7 相关工作

关于基准测试、工作负载缩减和程序分析的研究，为 MapReplay 奠定了方法论基础。本节回顾六个相关领域的关键成果：（1）工作负载提炼与基准测试合成；（2）记录/回放基础设施；（3）程序切片；（4）集合框架的基准测试；（5）哈希映射算法与 JVM 实现；以及（6）研究与实践中的高性能哈希表。

工作负载提炼与基准测试合成。在体系结构和系统研究中，人们已经探索了将复杂工作负载缩减为更小但具有代表性的基准测试，例如 SimPoint [40, 51]、MiBench [23],，以及编译器和运行时基准测试领域的相关研究 [21]。另一类互补工作则通过匹配目标工作负载特征来合成新的紧凑型基准测试：Bell 和 John [4]提出自动合成微型基准测试，以复现参考工作负载的动态指标；Van Ertvelde 和 Eeckhout [52] 通过针对 CPU 级特征，为体系结构/编译器探索合成工作负载；Joshi 等人 [26] 论证了合成基准测试的实用性，并讨论了如何构建这类基准测试以开展性能研究；Panda 等人 [38] 则提出代理基准测试，通过硬件性能计数器跟踪性能指标，以保留真实世界应用的性能特征。MapReplay 遵循类似原则，但专注于复现 HashMap 操作的控制流和状态转换，在不保留完整应用逻辑的情况下，实现库级别的真实性。在数据库领域，也有多种方法用于生成合成工作负载。Stitcher [55] 通过组合标准基准测试的片段来合成工作负载（还可以选择性地通过优化来匹配期望的性能模式）。类似地，PBench [59]生成能够近似观测到的执行统计数据的合成工作负载。MapReplay 的设计目标不同：它并非合成端到端工作负载来匹配总体性能特征，而是从真实应用中提取特定关注功能（HashMap 操作）的真实使用模式，并在基准测试环境中对其进行回放。这样便能够针对 HashMap 优化和配置选择开展有针对性的评估。

记录/回放基础设施。PinPlay [39] 和 DejaVu [12] 等通用记录/回放系统能够为调试、性能分析或容错提供强确定性。这些工具通常在进程级或指令级运行，由于记录的事件数量庞大，且需要处理非确定性交互，因此会产生显著开销。在这条研究路线中，Dolos [11] 专注于 Web 应用场景下的记录/回放，最终目标是调试。JaRec [22] 提出了一种面向多线程应用的轻量级 Java 记录/回放基础设施，其最终目标同样是调试。类似地，rdb [24] 是一种用于调试的硬件辅助记录/回放基础设施，尤其注重精确、确定性的回放。MapReplay 的研究范围则截然不同：它只记录应用行为的严格子集，即 API 层级上与 HashMap 相关的事件，从而生成轻量级跟踪记录，既能快速回放，又能忠实保留被跟踪库的语义行为。

动态程序切片与保持行为的提取。程序切片 [56] 旨在提取与特定执行行为相关的代码子集及

数据。动态切片 [2,57] 利用运行时依赖项进一步细化这一过程，使人们能够仅重建影响特定输出或副作用的程序部分。从概念上讲，MapReplay 采用了类似的精简方式：它仅捕获与 HashMap 操作相关的执行子集，同时在回放期间保留HashMap 对象的内部状态转换。具体而言，MapReplay 借鉴通用动态切片技术的思想，生成能够快速、可靠地评估HashMap 变体性能的工作负载。

集合的基准测试与实证研究。Java 集合的实证研究表明，性能对使用模式和工作负载多样性高度敏感 [18]。DaCapo Chopin [7, 8],Renaissance [41]和 JEDI [47]（使用 S2S [49] 生成的 Java 流基准测试套件）等基准测试套件包含大量集合操作的工作负载，但其运行时间长且计算内容混杂，因此在隔离集合层面的影响时成本高昂。以往研究指出了集合微基准测试中的一些陷阱，包括缺乏代表性的访问分布和测量干扰 [28]。代码仓库挖掘方法通过从公共代码仓库自动提取并执行单元测试工作负载，构建特定领域的基准测试套件 [53, 58]）。MapReplay 通过支持对源自真实应用程序跟踪数据的哈希映射活动进行快速回放，补充了这类研究。

JBrainy [20], 是 Brainy 的 Java 实现，它通过生成随机化调用序列，自动生成大量合成的集合微基准测试。与 MapReplay 一样，它支持对 Java HashMaps进行快速实验；不过，它依赖合成的使用模式，而 MapReplay 则从应用程序执行中提取真实的使用模式。我们预计，MapReplay 从应用程序中提取的跟踪记录，比随机化调用序列更能反映真实世界的行为。

哈希映射算法和 *JVM* 实现。 大量研究探索了哈希表的设计与优化，涵盖开放寻址、布谷鸟哈希，以及能够改善局部性并降低哈希冲突开销的缓存高效变体 [37, 46]。另一些研究则关注并发和无锁哈希表，例如 Java 的 ConcurrentHashMap [14]，以及后续关于可扩展非阻塞映射的研究 [50]。这些研究凸显了哈希表对性能的敏感性，也说明了采用 MapReplay 等真实评估方法的必要性。

研究与实践中的高性能哈希表。哈希表性能一直是活跃的研究课题，近年来人们重新将重点放在现代硬件和内存技术上。近期研究提出了能够协同优化实用指标（如吞吐量、内存占用和缓存行为）的设计 [5], 改进了跨架构的向量化探测策略 [10], 并开发了可优化写入的持久内存感知型哈希表 [54, 60]。在工业界，主要平台和库出于稳健性与性能方面的考虑，对哈希表实现进行了大量改进，从 OpenJDK 对HashMap 哈希冲突处理的更新 [34] 到高度优化、可用于生产环境的哈希表，例如 Google 的 Abseil SwissTables [1] 和 Meta 的 F14 [32]。这些进展表明，哈希表优化仍在持续推进，也促使人们开发能够在真实使用模式下准确评估性能影响的方法。

8 结论

本文介绍了 MapReplay，这是一种轨迹驱动基准测试方法，可在不重新运行完整应用的情况下，使用源自应用的工作负载评估 HashMap 的性能。MapReplay 会在执行期间记录与HashMap相关的 API 事件，并通过轻量级回放基础设施重放这些事件，同时保留跟踪记录中操作序列对应的内部映射状态。其目标是在实现可复现、低噪声性能测量的同时，保留与性能相关的HashMap 行为。

我们将 MapReplay 应用于 DaCapo-Chopin 和 Renaissance，构建了 MapReplayBench，这是一套源自广泛使用的 Java 基准测试的回放工作负载。我们通过一项案例研究展示了该方法：在四个候选值中确定 HashMap 最快的默认初始容量（DIC）。评估结果表明，在所研究的映射密集型工作负载中，回放工作负载的结果与相应应用基准测试的结果一致：在 7 个工作负载上的 21 项 DIC 比较中，相对加速比呈现强正相关（$r = 0.870$），且大多数比较（18/21）的变化方向一致，这表明回放工作负载与应用基准测试得出了相似的性能结论。此外，采用实用的 JMH 配置（运行 5 次；预热和测量各迭代 5 次，每次 10 秒），回放工作负载以更低的成本提供反馈：评估某个 DIC 的所有回放工作负载约需 8 小时，而相应的应用基准测试测量约需 72 小时。回放工作负载对HashMap局部变更的敏感度更高：在同一案例研究中，它们报告了 18 项具有统计显著性的 DIC 效应，而应用基准测试仅报告 10 项。在整个基准测试集合中，只有 7 个应用工作负载呈现可测量且与性能相关的 DIC 差异，而 27 个回放工作负载则显示出统计显著的变化。最后，与应用基准测试相比，回放工作负载能更清晰地呈现与 HashMap相关的性能变化：它们帮助我们找出了加速比最高的 DIC（几何平均执行时间改善了 1.04×），而应用基准测试无法得出这样的结论。

鉴于上述结果，我们将 MapReplayBench 定位为介于微基准测试和完整应用基准测试之间的实用折中方案。当目标是在受控环境中快速、可复现地评估 HashMap 配置选择或实现变体，同时保留从应用中提取的真实使用模式，并减少无关计算带来的噪声时，MapReplayBench 尤其有用。回放工作负载有意抽象掉非HashMap 代码，也不旨在捕捉跨组件影响（例如，由非映射分配引发的垃圾回收、贯穿整个应用的复杂即时编译交互、受周边代码影响的底层架构效应，或多线程争用与同步）。因此，只要预期这些端到端交互会产生影响，就应通过应用基准测试验证回放工作负载得出的结论。

致谢

本工作得到 Hasler Fundation（2025-03-13- 355）、瑞士国家科学基金会（IZSEZ0_229176）以及 USI FIR 项目“理解并缓解托管运行时中的性能波动”的支持。

引用

[1] Abseil。2018。Abseil C++ 库中瑞士表的引入。https://abseil.io/blog/20180927-swisstables。[2] Hiralal Agrawal 和 Joseph R. Horgan。1990。动态程序切片。载于《ACM SIGPLAN 1990 编程语言设计与实现大会论文集》（美国纽约州怀特普莱恩斯）（PLDI '90）。计算机协会，纽约州纽约，美国，246–256。10 1145 93542 93576 – 。doi: . / .

[3] Matteo Basso、Andrea Rosà、Luca Omini 和 Walter Binder。2023。。Java Vector API：基准测试与性能分析。载于《第 32 届 ACM SIGPLAN 编译器构造国际会议论文集》（加拿大魁北克省蒙特利尔）（CC 2023）。计算机协会，纽约州纽约，美国，1–12。doi:10.1145/3578360.3580265[4] Robert Bell Jr. 和 Lizy K. John。2005。微型基准测试自动合成的理由。载于建模、基准测试与仿真研讨会（MoBS）。4–8。[5] Michael A. Bender、Martin Farach-Colton、William Kuszmaul 和 Giuliano L. Tagliavini。2023。冰山哈希：同时优化多项哈希表标准。《ACM学报》（2023）。10 1145 3625817 ）。doi: . / Walter Binder、Philippe

[6] Moret、Éric Tanter 和 Danilo Ansaloni。2016。多态字节码插桩。《软件实践与经验》46，10（2016），1351–1380。doi:10.1002/SPE.2385[7] Stephen M Blackburn、Zixian Cai、Rui Chen、Xi Yang、John Zhang 和 John Zigman。2025。重新思考 Java 性能分析。载于《第 30 届 ACM 编程语言与操作系统体系结构支持国际会议论文集》，第 1 卷，ASPLOS 2025，荷兰鹿特丹，*30* *2025* *3* *2025* 10 1145 3669940 3707217 3 月 - 4 月。ACM。doi: . / .

[8] Stephen M. Blackburn、Robin Garner、Chris Hoffmann、Asjad M. Khang、Kathryn S. McKinley、Rotem Bentzur、Amer Diwan、Daniel Feinberg、Daniel Frampton、Samuel Z. Guyer、Martin Hirzel、Antony Hosking、Maria Jump、Han Lee、J. Eliot B. Moss、Aashish Phansalkar、Darko Stefanović、Thomas VanDrunen、Daniel von Dincklage 和 Ben Wiedermann。2006。DaCapo 基准测试：Java 基准测试的开发与分析。载于《第 21 届 ACM SIGPLAN 面向对象编程系统、语言与应用年会论文集》（美国俄勒冈州波特兰）（OOPSLA '06）。计算机协会，纽约州纽约，美国，169 190 10 1145 1167473 1167488 – 。doi: . / . Peter

[9] Boncz、Thomas Neumann 和 Viktor Leis。。FSST：快速随机访问字符串压缩。《VLDB基金会会刊》13，12（2020），2649–2661。[10] Maximilian Böther、Lawrence Benson、Ana Klimovic 和 Tilmann Rabl。2023。跨 CPU 架构分析向量化哈希表。《VLDB基金会会刊》16, 11 2023 2755 2768 10 14778 3611479 3611485 ( ), – 。doi: . / .

[11] Brian Burg、Richard Bailey、Amy J Ko 和 Michael D Ernst。。用于 Web 应用程序调试的交互式记录/回放。载于《第 26 届 ACM 用户界面软件与技术年会论文集》。473–484。[12] Jong-Deok Choi 和 Harini Srinivasan。1998。Java 多线程应用程序的确定性回放。载于 SIGMETRICS 并行与分布式工具研讨会论文集。48–59。[13] OpenJDK 社区。2025。Java 微基准测试框架（JMH）。OpenJDK。https://github.com/openjdk/jmh [14] 甲骨文公司。2025。ConcurrentHashMap 类。https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/concurrent/ConcurrentHashMap. html[15] 甲骨文公司。2025。HashMap 类。甲骨文。https://docs.oracle.com/en/java/javase/25/docs/api//java.base/java/util/HashMap.html [16] 甲骨文公司。2025。TreeMap 类。https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/TreeMap.html [17] 甲骨文公司。2025。Java 本地接口规范。https://docs.oracle. com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html [18] Diego Costa、Artur Andrzejak、Janos Seboek 和 David Lo。2017。Java 集合的使用与性能实证研究。载于《第 8 届 ACM/SPEC 国际性能工程大会论文集》（意大利拉奎拉）（ICPE '17）。389 400 10 1145 3030207 3030221 ）。ACM，纽约州纽约，美国，– 。doi: . / . Diego Costa、Cor-Paul Bezemer、Philipp Leitner 和 Artur Andrzejak。2021。我的基准测试结果有什么问题？研究 JMH 基准测试中的不良实践。《IEEE软件工程汇刊》47，7（2021），1452–1467。doi:10.1109/TSE.2019.2925345[20] Noric Couderc、Emma Söderberg 和 Christoph Reichenbach。2020。JBrainy：通过干扰对 Java 集合进行微基准测试。载于 ACM/SPEC 国际性能工程大会伴随会议（加拿大阿尔伯塔省埃德蒙顿）（ICPE '20）。计算机协会，纽约州纽约，美国，42–45。doi:10.1145/3375555.3383760[21] Lieven Eeckhout、Hans Vandierendonck 和 Koen De Bosschere。2003。量化输入数据集对程序行为的影响及其应用。《指令级并行》5，1（2003），1–33。[22] Andy Georges、Mark Christiaens、Michiel Ronsse 和 Koenraad De Bosschere。2004。JaRec：用于多线程 Java 应用程序的可移植记录/回放环境。《软件：实践与经验》34，6（2004），523–547。[23] M.R. Guthaus、J.S. Ringenberg、D. Ernst、T.M. Austin、T. Mudge 和 R.B. Brown。2001。MiBench：一个免费且具有商业代表性的嵌入式基准测试套件。载于第四届 IEEE 工作负载特征分析年度国际研讨会论文集。WWC-*4* *01* *538* 3 14 10 1109 2001 990739（目录号EX）。– 。doi: 。/WWC. 。Nima Honarmand 和 Josep Torrellas。。重放调试：利用记录与重放进行程序调试。《ACM SIGARCH计算机体系结构新闻》42, 3（2014），445–456。[25] Hao Jiang、Chunwei Liu、John Paparrizos、Andrew A Chien、Jihong Ma 和 Aaron J Elmore。2021。好到最后一位：使用 codecdb 实现数据驱动编码。载于 2021 年国际数据管理会议论文集。843–856。[26] Ajay M. Joshi、Lieven Eeckhout 和 Lizy K. John。2008。合成基准测试的回归。载于 2008 年 SPEC基准测试研讨会论文集。1–11。[27] Changhee Jung、Silvius Rus、Brian P. Railing、Nathan Clark 和 Santosh Pande。2011。Brainy：有效选择数据结构。SIGPLAN Not. 46, 6（2011 年 6 月 2011 86 97 10 1145 1993316 1993509 ），–。doi: 。/。Tomas Kalibera 和 Richard Jones。。在合理时间内进行严格的基准测试。载于 2013 年国际内存管理研讨会论文集（美国华盛顿州西雅图）（ISMM '13）。计算机协会，纽约州纽约市，美国，63 74 10 1145 2491894 2464160 –。doi: 。/。Stephen Kell、Danilo Ansaloni、Walter Binder 和 Lukáš Marek。2012。JVM 的可观测性不足（以及应对之策）。载于第六届 ACM 虚拟机与中间语言研讨会论文集（美国亚利桑那州图森）（VMIL '12）。计算机协会，纽约州纽约市，美国，33–38。doi:10.1145/2414740.2414747[30] Júnior Löff、Filippo Schiavio、Andrea Rosà、Matteo Basso 和 Walter Binder。2024。向量化内建函数可替换为纯 Java 代码，而不会损害稳态性能。载于第 15 届 ACM/SPEC 国际性能工程大会论文集（ICPE）。doi:10.1145/3629526.3645051[31] Lukáš Marek、Alex Villazón、Yudi Zheng、Danilo Ansaloni、Walter Binder 和 Zhengwei Qi。2012。DiSL：一种用于字节码插桩的领域特定语言。载于第 11 届面向方面软件开发国际年会论文集（德国波茨坦）（AOSD '12）。计算机协会，纽约州纽约市，美国，239 250 10 1145 2162049 2162077 –。doi: 。/。Meta Engineering。。开源 F14，打造更快、内存效率更高的哈希表。https://engineering.fb.com/2019/04/25/developer-tools/f14/。[33] Philippe Moret、Walter Binder 和 Éric Tanter。2011。多态字节码插桩。载于第十届面向方面软件开发国际会议论文集（巴西加利尼亚斯港）（AOSD '11）。计算机协会，纽约州纽约市，美国，129–140。doi:10.1145/1960275. 1960292[34] OpenJDK。2014。JEP 180：使用平衡树处理频繁的 HashMap冲突。https://openjdk.org/jeps/180。[35] OpenJDK。2025。Project Jigsaw：模块系统快速入门指南。https://openjdk.org/projects/jigsaw/quick-start。[36] 甲骨文。2025。Proftool。https://github.com/graalvm/mx/blob/master/README- proftool.md。[37] Rasmus Pagh 和 Flemming Friche Rodler。2004。布谷鸟哈希。《算法杂志》51, 2 2004 122 144 10 1016 2003 12 002 （5 月），–。doi: 。/j.jalgor. . 。Reena Panda 和 Lizy Kurian John。。面向新兴大数据工作负载的代理基准测试。载于 2017 年第 26 届并行体系结构与编译技术国际会议（PACT）。IEEE，105–116。[39] Harish Patil、Cristiano Pereira、Mack Stallcup、Gregory Lueck 和 James Cownie。2010。PinPlay：用于并行程序确定性回放与可复现分析的框架。载于第八届 IEEE/ACM代码生成与优化国际年会论文集（加拿大安大略省多伦多）（CGO '10）。计算机协会，纽约州纽约市，美国，2–11。doi:10.1145/1772954.1772958[40] E. Perelman、G. Hamerly 和 B. Calder。2003。选择统计有效且较早的仿真点。载于 2003 年第 12 届并行体系结构与编译技术国际会议。244–255。doi:10.1109/PACT.2003.1238020[41] 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。2019。Renaissance：面向 JVM 并行应用程序的基准测试套件。载于第 40 届 ACM SIGPLAN 编程语言设计与实现会议论文集（PLDI '19）。31–47。[42] Andrea Rosà 和 Walter Binder。2018。利用反射式超类型信息优化 JVM 上的类型特定插桩。《可视化语言与计算》49 2018 29 45 10 1016 2018 10 007（），–。doi: 。/j.jvlc. . 。[43] Andrea Rosà、Eduardo Rosales 和 Walter Binder。。在 JVM 上为动态分析准确具体化完整超类型信息。载于第 16 届 ACM SIGPLAN 生成式编程：概念与实践国际会议（GPCE）论文集。216 224 10 1145 3136040 3136061 –。doi: 。/。Andrea Rosà、Eduardo Rosales 和 Walter Binder。。在 JVM 上分析并优化任务粒度。载于 2018 年代码生成与优化国际研讨会（CGO）论文集。doi: 10 1145 3168828 . /

[45] Andrea Rosà、Eduardo Rosales 和 Walter Binder。。在 Java 虚拟机上分析与优化任务粒度。《ACM编程语言与系统汇刊》41, 3，文章 19（2019）。doi:10.1145/3338497

[46] Kenneth A. Ross。2007。现代处理器上的高效哈希探测。载于*2007 IEEE第23届国际数据工程会议*。1297–1301。doi:10.1109/ICDE. 2007.368997[47] Filippo Schiavio 和 Walter Binder。2026。JEDI：Java声明式与命令式查询评估——Java Stream API 基准测试。载于第 *48* 届国际软件工程会议论文集。doi:10.1145/3744916. 3773165[48] Filippo Schiavio、Lubomír Bulej 和 Walter Binder。2026。Java虚拟机上的误导性微基准测试。载于第 *41* 届 *ACM/SIGAPP*应用计算研讨会论文集。doi: 10 1145 3748522 3779882 . / 。Filippo Schiavio、[49] Andrea Rosà 和 Walter Binder。 。借助 S2S 将 SQL 转换为 Stream：Java Stream API 的自动基准测试生成器。载于第 *21* 届 *ACM/SIGPLAN*生成式编程：概念与实践国际会议（*GPCE*）论文集。ACM, 179 186 10 1145 3564719 3568699 –。 2022 doi: 。/。Ori Shalev 和 Nir Shavit[50]。 。分割有序列表：无锁可扩展哈希表。《*ACM*期刊》 53, 3 2006 379 405 10 1145 1147954 1147958 （5 月），–。 2006 doi: 。/。Timothy Sherwood、Erez Perelman、Greg Hamerly 和 Brad Calder[51]。 。自动刻画大规模程序行为。载于第 *10* 届编程语言与操作系统的体系结构支持国际会议论文集（美国加利福尼亚州圣何塞）（*ASPLOS X*）。计算机协会，纽约州纽约市，美国, 45 57 10 1145 605397 605403 –。doi: 。/。 2002 Luk Van Ertvelde 和 Lieven Eeckhout[52]。 。面向体系结构与编译器探索的基准测试合成。载于*2010 IEEE*国际工作负载特征分析研讨会（*IISWC*）。IEEE，1–11。[53] Alex Villazón、Haiyang Sun、Andrea Rosà、Eduardo Rosales、Daniele Bonetta、Isabella Defilippis、Sergio Oporto 和 Walter Binder。2019。在真实环境中自动开展大规模多语言动态程序分析。载于第 33 届欧洲面向对象编程会议（ECOOP 2019）（莱布尼茨国际信息学会议录 *(LIPIcs))*. doi:10.4230/LIPIcs.ECOOP.2019.20[54] Lukas Vogel、Alexander van Renen、Satoshi Imamura、Jana Giceva、Thomas Neumann 和 Alfons Kemper。2022。Plush：一种写入优化型持久性日志结构哈希表。*VLDB*基金会会刊 15, 11 (2022), 2895–2907. doi:10.14778/3551793.3551839[55] Chengcheng Wan、Yiwen Zhu、Joyce Cahoon、Wenjing Wang、Katherine Lin、Sean Liu、Raymond Truong、Neetu Singh、Alexandra M Ciortea、Konstantinos Karanasos 等。2023。Stitcher：基于历史性能特征的工作负载合成。在*EDBT*。417–423. [56] Mark Weiser。1984。程序切片。*IEEE*软件工程汇刊SE- 10, 4 1984 352 357 10 1109 1984 5010248 ( ), – . doi: . /TSE. . Xiangyu Z [57] hang 和 Rajiv Gupta。高性价比动态程序切片。(2004), 94–106. 2004 doi:10.1145/996841.996855[58] Yudi Zheng、Andrea Rosà、Luca Salucci、Haiyang Sun、Omar Javed、Lubomír Bulej、Walter Binder、Yao Li、Zhengwei Qi 和 Lydia Y. Chen。2016。AutoBench：借助可插拔混合分析，找到您需要的工作负载。在*2016 IEEE第23届*软件分析、演化与再工程国际会议（*SANER*）。IEEE，639–643. doi:10.1109/SANER.2016.70[59] Yan Zhou、Chunwei Liu、Bhuvan Urgaonkar、Zhengle Wang、Magnus Mueller、Chao Zhang、Songyue Zhang、Pascal Pfeil、Dominik Horn、Zhengchun Liu、Davide Pagano、Tim Kraska、Samuel Madden 和 Ju Fan。2025。PBench：用于云分析基准测试、基于真实统计数据的工作负载合成器。*VLDB*基金会会刊 18, 11 2025 3883 3895 10 14778 3749646 3749661 (July), – . doi: . / . Pengfei Zuo [60]、 Yu Hua 和 Jie Wu。面向持久性内存的写入优化型高性能哈希索引方案。在第*13*届*USENIX*操作系统设计与实现会议论文集(卡尔斯巴德，CA，美国) *(OSDI'18)*。USENIX协会，美国，461–476。 2018