Async-fork：从操作系统层缓解基于 fork 的快照机制导致的查询延迟峰值

*Pu Pang[1, 2], *Gang Deng[2]、Kaihao Bai[1, 2]、Quan Chen[1]、Shixuan Sun[3]、Bo Liu[1]、Yu Xu[2]、Hongbo Yao[2]、Zhengheng Wang[2]、Xiyu Wang[2]、Zheng Liu[2]、Zhuo Song[1, 2]、Yong Yang[2]、Tao Ma[2]、Minyi Guo[1]
[1]中国上海交通大学计算机科学与工程系
[2]中国阿里巴巴集团
[3]新加坡新加坡国立大学
{avengerispp,asbaikaihao,chen-quan,boliu98,myguo}@sjtu.edu.cn,sunsx@comp.nus.edu.sg
{denggang.dg,qiyu.xy,yuanzhi.yhb,zhengheng.wzh,xiyu.wxy,wenqing.lz,songzhuo.sz,boyu.mt}@alibaba-inc.com

arXiv:2301.05861v1 [cs.DB] 14 Jan 2023

摘要

内存键值存储（IMKVSes）凭借其高效性服务于众多在线应用。为了支持数据备份，业界主流的内存键值存储系统会定期通过系统调用 *fork* 对内存数据执行时间点快照。然而，由于 *fork* 会使引擎进入内核态，在此期间引擎无法为查询提供服务，因此查询若在快照期间到达，可能会出现延迟尖峰。不同于现有侧重于优化快照算法的研究，我们从操作系统（OS）级别优化 fork 操作，以解决延迟尖峰问题，同时保持内存键值存储系统中数据持久化机制不变。具体而言，我们首先开展深入研究，以揭示 fork 操作及其优化技术对查询延迟的影响。基于研究结果，我们提出了 Async-fork，将页表复制工作从引擎（父进程）卸载至子进程，因为页表复制占据了 *fork* 执行时间的主要部分。为保持父进程与子进程之间的数据一致性，我们设计了主动同步策略。Async-fork 已在 Linux 内核中实现，并部署到公有云上的在线 Redis 数据库中。实验结果表明，与操作系统中的默认 fork 方法相比，在 8GB 实例上，Async-fork 将快照期间到达的查询的尾延迟降低了 81.76%；在 64GB 实例上则降低了 99.84%。

PVLDB 引用格式：
*Pu Pang[1, 2], *Gang Deng[2], Kaihao Bai[1, 2], Quan Chen[1], Shixuan Sun[3], Bo Liu[1], Yu Xu[2], Hongbo Yao[2], Zhengheng Wang[2], Xiyu Wang[2], Zheng Liu[2], Zhuo Song[1, 2], Yong Yang[2], Tao Ma[2], Minyi Guo[1]。Async-fork：缓解操作系统层基于 fork 的快照机制引发的查询延迟峰值。PVLDB，16(5)：XXX-XXX，2023年。doi:XX.XX/XXX.XX

1 引言

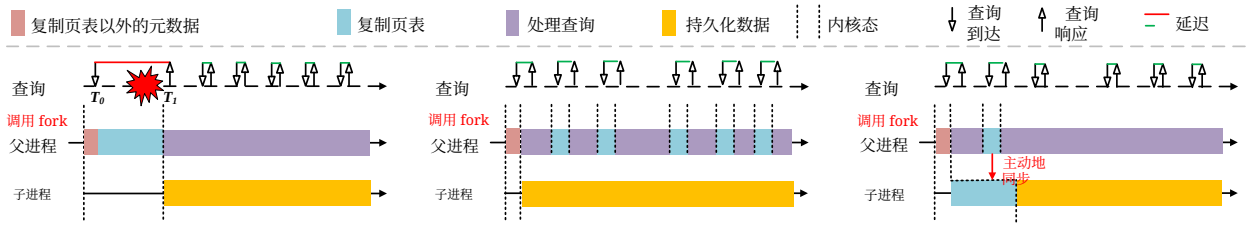
内存键值存储（IMKVS）凭借超快的查询处理速度，广泛应用于现实世界的应用程序，尤其是在线服务（例如电子商务和社交网络）。例如，Memcached [49], Redis [42], KeyDB [25] 及其变体 [7, 12, 19, 24] 已部署在 Facebook、Amazon 和 Twitter 等大型互联网公司的生产环境中。由于所有数据都驻留在内存中，数据持久化函数是 IMKVS 用于数据备份的一项关键功能。

一种常见的数据持久化方法是通过系统调用 *fork* 对内存数据进行时间点快照，并将快照写入文件系统。图1(a)展示了基于 fork 的快照方法示例。首先，存储引擎（父进程）调用 *fork* 创建一个子进程。由于*fork*通过复制父进程来创建新进程，子进程将持有与父进程相同的数据。因此，我们可以让子进程在后台将数据写入文件，同时让父进程持续处理查询。尽管存储引擎将繁重的输入输出任务委派给子进程，基于 fork 的快照方法仍可能导致延迟尖峰 [32, 52]。具体而言，在创建快照期间（从*fork* 开始到数据持久化完成），到达的查询可能会产生较长延迟，因为存储引擎会进入内核态，无法处理查询。例如，查询在图1(a)中的时间 T_0 到达。为简洁起见，我们将创建快照期间到达的查询称为快照查询，其他查询则称为普通查询。通常，内存键值存储系统会定期创建快照（例如，Redis 默认每 60 秒创建一次快照，前提是至少有 10000 条记录被修改过 [47, 59]）。因此，快照查询出现延迟尖峰并不少见。

为了解决这个问题，研究人员提出了多种快照算法，例如写时更新 [10, 33], Zigzag [11] 和 Ping-Pong [11]。这些算法致力于通过优化数据复制时机并减少复制的数据量，降低创建快照的成本。然而，近期一项研究 [32]发现，基于 fork 的方法整体性能与这些先进方法不相上下，甚至在写密集型工作负载下表现更优。此外，基于 fork 的方法实现起来非常简单，所需的工程投入也很少。因此，工业界的内存键值存储系统并未采用学术界提出的这些快照算法，而是继续将基于 fork 的方法作为内置的数据持久化方案。结果是，内存键值存储系统仍会因 fork 操作而出现延迟尖峰，从而损害在线应用程序的服务质量。

不同于现有专注于优化快照算法的研究，我们提出了一种直接的方法来解决这一问题。具体而言，我们优化了 fork 操作，以减少操作

*本文作者贡献均等。本文采用知识共享署名-非商业性使用-禁止演绎 4.0 国际许可协议。请访问 https://creativecommons.org/licenses/by-nc-nd/4.0/ 查看许可协议副本。若要进行超出许可协议范围的使用，请发送电子邮件至 info@vldb.org 获取许可。版权所有者为所有者/作者。出版权已授权给 VLDB Endowment。VLDB Endowment 会议录，第16卷，第5期，ISSN 2150-8097。
doi:XX.XX/XXX.XX



(a) 默认 fork　　(b) 基于 SPT 的分叉　　(c) 提出的异步分叉

图1：采用（a）默认 fork、（b）基于共享页表（SPT）的 fork，以及（c）所提出的 Async-fork 时，父进程和子进程在快照过程中的工作流程。

系统（OS）级别基于 fork 的快照所产生的长延迟，同时保持 IMKVSes 中的数据持久化机制不变。

我们首先进行了深入研究，以揭示 fork 操作对查询延迟的影响。对默认 fork 操作的性能分析结果表明，*fork* 的开销会导致快照查询出现较长的尾延迟（最高可达数百毫秒），而复制页表占据了*fork*执行时间的主要部分。这促使我们进一步研究能够加速 fork 操作的先进技术 [14, 15, 18, 35, 59] 的影响。我们考虑了两种优化策略：大页 [14, 15] 和共享页表 [18, 35, 59]。由于大页会降低 IMKVSes [28, 44], 的性能，我们的性能分析仅涉及最新的基于共享页表的*fork*（基于 SPT 的 *fork*）[59]。基于 SPT 的*fork* 提出以写时复制（CoW）的方式在父进程和子进程之间共享页表，从而降低 fork 操作的开销。然而，如图1(b)所示，CoW 会频繁中断父进程。我们的性能分析结果表明，尽管与默认 fork 操作相比，基于 SPT 的*fork* 显著降低了尾延迟，但频繁中断仍会给快照查询带来不可忽略的开销。此外，我们的分析发现，共享页表会引入数据泄露漏洞，可能导致快照不一致。因此，大页和共享页表技术均无法应用于这种场景。

受这些发现的启发，我们提出了 Async-fork，通过优化 fork 操作来缓解快照查询的延迟尖峰。图1(c)展示了总体思路。由于复制页表占据了 fork 操作的大部分开销，Async-fork 将这项工作从父进程转移到子进程，从而缩短父进程进入内核态的时间。这种设计还确保父进程和子进程各自拥有独立的页表，避免共享页表导致的数据泄露漏洞。

然而，从设计上实现这一点绝非易事，因为两个进程对页表执行的异步操作可能导致不一致快照：也就是说，父进程可能会在子进程的复制操作进行期间修改页表。为了解决这一问题，我们设计了主动同步技术。该技术使父进程能够检测对页表的所有修改（包括由用户或操作系统触发的修改）。如果父进程检测到某些页表项即将被修改，且这些条目尚未复制，就会主动将它们复制到子进程。否则，这些条目必定已复制到子进程，父进程将直接修改它们。通过这种方式，与基于 SPT 的 *fork* 相比，主动同步技术能够保持快照一致，并减少对父进程的中断次数。此外，我们并行执行子进程的复制操作，以进一步加快 Async-fork。

我们在 Linux 内核中实现了 Async-fork（支持 x86 和 ARM64）。Async-fork 集成于操作系统中，对内存键值存储系统透明。该技术也已部署在公有云中的在线 Redis 数据库中[1]。尽管如此，为了便于灵活测试，我们仍在本地机器上进行实验。在实验中，我们选取了两种流行的内存键值存储系统 Redis 和 KeyDB，并使用 Redis 基准测试工具 [41] 和 Memtier 基准测试 [46]。数据库实例大小从 1GB、2GB、4GB.....到 64GB 不等。尽管基于 SPT 的*fork* [59] 可能导致数据泄漏，但为了进行效率对比，我们的实验也采用了这种方法。对于 Redis，实验结果表明：1）与默认的*fork*相比，Async-fork 将快照查询的第99百分位延迟在 1GB 实例上降低了 17.57%（从 0.074ms 降至 0.061ms），在 8GB 实例上降低了 81.76%（从 0.435ms 降至 0.079ms），在 64GB 实例上降低了 99.84%（从 991.9ms 降至 1.5ms）；2）与最新的基于 SPT 的*fork*相比，Async-fork 将快照查询的第99百分位延迟在 1GB 实例上降低了 2.87%，在 8GB 实例上降低了 39.73%，在 64GB 实例上降低了 61.97%。我们在 KeyDB 上也取得了类似结果。这些结果证明了本文所提技术的有效性，尤其适用于可能导致较长延迟的大型实例。

总而言之，本文的贡献如下。

- 我们深入研究了 fork 操作对内存键值存储系统中快照查询延迟的影响。
- 我们提出了 Async-fork，可从操作系统层缓解快照查询的长延迟，这与针对该问题的现有研究相互正交。
- 该技术已在 Linux 内核（包括 x86 和 ARM64）中实现，并部署于公有云中的在线 Redis 数据库。
- 我们使用 Redis 和 KeyDB 开展了大量实验，以评估所提技术的有效性。

2 背景

本节首先介绍预备知识，然后讨论相关工作。

[1]https://www.alibabacloud.com/product/apsaradb-for-redis。最后访问于 2022/11/13。

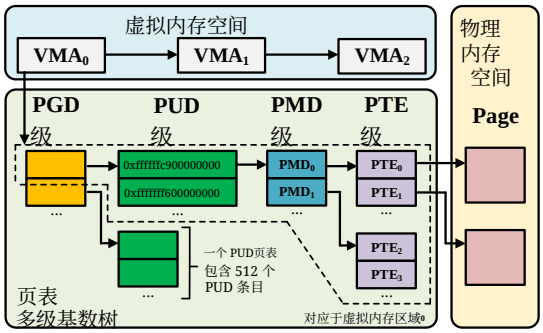


图 2：页表的组织结构。

2.1 初步介绍

我们首先简要回顾两个与本文密切相关的操作系统概念：虚拟内存 和*fork*。由于我们的技术是在 Linux 上实现并部署的，因此我们将结合 Linux 的背景介绍这些概念。然后，我们讨论*fork* 在数据库中的应用场景。

虚拟内存。 作为管理硬件内存资源的一种有效方法，虚拟内存广泛应用于现代操作系统中。每个进程都有自己的虚拟内存空间，该空间被组织为一组虚拟内存区域（VMA）。每个 VMA 描述虚拟内存空间中的一段连续区域。页表是一种用于将虚拟内存空间映射到物理内存的数据结构。页表由一组页表项（PTE）构成，每个页表项都保存虚拟地址到物理地址转换信息以及访问权限。一个 VMA 对应多个 PTE。

图 2 展示了页表的一个示例。为降低内存开销，页表以多级基数树的形式存储，PTE 位于叶节点中（即图 2 中的 PTE 级）。虚线标记区域中的部分对应于 VMA_0。该树最多有五级。从上到下依次为页全局目录（PGD）级、P4D 级、页上级目录（PUD）级、页中间目录（PMD）级和 PTE 级。由于 P4D 通常处于禁用状态，本文重点讨论其他四级。除 PTE 级外，每个条目都存储一个页的物理地址，该页用作下一级节点（页表）。当页大小设为 4KB 时，每一级的页表均包含 512 个条目。对于给定的 VMA，“VMA 的 PTE”指与该 VMA 对应的 PTE，而“VMA 的 PMD”指树中作为这些 PTE 父节点的 PMD 条目集合。例如，在图 2 中，PMD_0 和 PMD_1 属于 VMA_0的 PMD。

Fork 操作。 *Fork* 是一种系统调用，通过复制调用进程来创建新进程 [1]。两个进程拥有各自独立的内存空间。新进程（即子进程）和调用进程（即父进程）分别称为子进程和父进程。为加快该操作，Linux 使用写时复制（CoW）策略实现 *fork*。具体而言，调用 *fork* 时，父进程在内核态运行，并将元数据（例如虚拟内存区域、页表、文件描述符和信号）复制到子进程。父进程和子进程的页表项都被设置为写保护。此后，首先修改写保护页的进程会触发缺页异常，进而导致该页被复制。总之，得益于写时复制，*fork* 只复制元数据。

数据库中 *fork* 的用例。 *Fork* 有许多数据库用例，因为它能够轻松高效地创建内存数据的快照，而其一致性由操作系统保证。通常，根据快照的用途，这些用例可以分为两类。首先，使用*fork* 将输入输出或计算开销较大的专用任务委派给子进程，同时不阻塞父进程提供服务。MDC [40]使用*fork* 为内存数据库记录检查点。Redis 使用*fork* 执行日志重写 [45]，以优化仅追加文件（AOF）。FlurryDB [36] 提出在分布式环境中基于 *fork* 创建副本。其次，使用*fork* 创建快照，以支持并发事务处理，因为*fork* 能够在进程之间提供快照隔离。HyPer [30] 提出基于*fork*创建的快照来评估混合型联机事务处理和联机分析查询。AnKer [50] 设计了一种细粒度快照机制，以支持多版本并发控制。具体而言，AnKer 通过协同设计数据库引擎和 *fork*系统调用，对内存数据进行部分快照。与我们从操作系统层加速 fork 操作的研究不同，AnKer 关注的是优化快照应捕获哪些内存数据。

所有这些用例都有可能受益于 Async-fork，因为：1）它们可能会遇到 fork 操作导致的查询延迟峰值问题；2）Async-fork 可以加快快照创建。本文聚焦于使用*fork* 对内存数据创建时间点快照并持久化数据的场景 [27, 45]。具体而言，存储引擎调用*fork* 创建一个持有与自身相同数据的子进程。随后，子进程将数据写入硬盘，而存储引擎则可以继续为用户提供查询服务。尽管存储引擎将数据转储任务委托给子进程，但在调用*fork* 期间，由于进程会进入内核态，因此无法提供查询服务。我们尤其关注这一场景，因为 IMKVS 是公有云中最重要的服务之一，而 Redis 和 KeyDB 等流行的 IMKVS [51] 会使用这一机制持久化数据。因此，这些存储系统会遇到严重的查询延迟峰值问题（见第 3 节），而它们通常用于对延迟约束要求严格的关键任务应用。我们还评估了 Async-fork 在 Redis [45]日志重写中的效果。实验结果见技术报告 [2]。

备注。 Async-fork 的目标并非开发一种通用解决方案来替代操作系统中的默认 *fork*，而是为以下场景（尤其是内存键值存储系统）提供高效的 fork 操作：1）应用程序占用大量内存；2）父进程对延迟敏感。我们的设计允许 Async-fork 与操作系统中的默认 *fork* 并行运行。用户可以轻松选择应用程序所使用的 fork 方法（见第 5.2 节）。

2.2 相关工作

一致性快照对于内存数据库至关重要，可用于支持备份和灾难恢复 [5, 17, 32, 56]。已有一些一致性快照机制，通过权衡吞吐量、延迟和内存占用来实现这一目标 [9–11, 32, 33]。朴素快照 [9]会阻塞存储引擎，直到对所有内存中的

数据完成深拷贝，因此不适用于延迟至关重要的内存键值存储系统。此外，还有一些非阻塞快照机制。写时更新 [10, 33] 提出创建内存数据的影子副本；存储引擎可以自由访问任意数据，但首次更新数据时需要创建深拷贝。请注意，基于 fork 的快照是一种利用操作系统的写时更新变体。另一些机制采用多版本并发控制（MVCC） [8] 来保留内存数据的多个版本。Zigzag [11] 维护另一份未修改的内存数据副本，并引入元数据位来指示存储引擎应从哪个副本读取或向哪个副本写入。在 Zigzag 的基础上，Ping-Pong [11] 维护了三个数据版本，以降低管理元数据位的成本。Hourglass 和 Piggyback [32] 则通过结合 Zigzag 和 Ping-Pong 开发而成。

尽管基于 fork 的快照会导致快照过程延迟较高，主流工业内存键值存储系统（Redis 和 KeyDB）仍然采用基于 fork 的快照，原因有二：1）*fork* 为实现一致性快照提供了简单的工程方案，而将上述方法集成到内存键值存储系统中则需要付出巨大努力。2）上述方法都无法在写密集型工作负载下全面胜过基于 fork 的快照 [32]。例如，Ping-Pong 和 Hourglass 可以缓解延迟尖峰，但 Ping-Pong 会使内存占用增加 3 倍，而 Hourglass 在正常操作期间会导致更高的延迟。本文解决了基于 fork 的快照所引发的延迟尖峰，同时保留了其原有优势。

先前的工作 [33, 40] 指出，由于写时复制策略，快照过程中的内存占用会增加。MDC [40] 提出尽可能及时地释放已持久化的页。AnKer [50] 引入了一种细粒度的*fork*，以便在数据库无需持久化全部数据时进行部分快照。CCoW [23] 根据内存访问的空间局部性优化了写时复制机制。它优先复制局部性较高的内存区域，以提升写密集型工作负载下的性能。Async-fork 与这些方法正交且互为补充。

还有其他将数据持久化到内存键值存储系统中的方法。Redis 和 KeyDB 使用仅追加文件（AOF） [27, 45] 记录存储引擎收到的每个写操作，并在数据库重启后重新执行这些操作，以重建原始数据集。

快照和 AOF 互为补充，建议在内存键值存储系统中同时启用两者。

3 动机

本节展示性能分析结果，以说明 fork 操作对查询延迟的影响。我们首先评估*fork* 创建快照的性能，以确定影响性能的关键因素。随后，我们揭示 fork 操作对查询延迟的影响。除了 Linux 中默认的*fork* 外，我们的性能分析还涵盖了用于优化 fork 操作的最新方法 [59] 。最后，我们根据性能分析结果总结发现。

性能分析设置。 我们在实验中使用 Redis 基准测试工具 [41] 来研究性能。测试机器的详细配置见第 6.1 节。实验报告查询的延迟，即客户端发出查询到收到

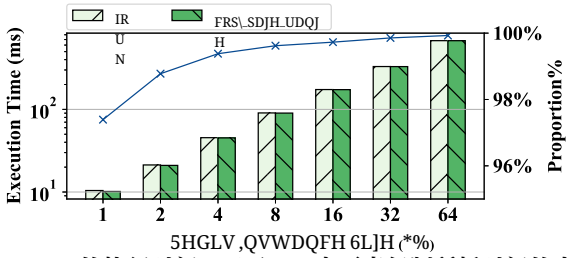


图 3：*fork* 的执行时间，以及*fork*.中页表复制所耗时间的占比。

响应之间经过的时间。在实验中，我们通过以开环模式 [48, 58] 生成查询来改进基准测试工具。这项改进会在不等待先前查询的响应时，继续向服务器发送命令，以模拟真实环境。数据库实例大小从 1GB 到 64GB 不等。默认情况下，如果至少有 10000 个键发生更改，Redis 每 60 秒创建一次快照。为了准确测量 fork 操作的影响，我们执行 BGSAVE 命令来触发快照操作。我们根据查询的到达时间将其分为两组：正常查询和快照查询。快照查询是指在创建快照期间到达的查询（即从调用*fork* 到内存数据持久化完成的这段时间内到达的查询），其余查询则为普通查询。我们分别测量正常查询和快照查询的 99 百分位（p99）延迟与最大延迟。这两种延迟会显著影响用户体验，常用于衡量面向用户的数据库的性能 [29, 32]。每项实验重复五次，并报告平均值。

3.1 用于创建快照的*fork* 性能

图 3 展示了*fork* 的执行时间，以及 fork 操作中复制页表所需的时间。可以看到，随着实例大小从 1GB 增加到 64GB，执行时间大致呈线性增长。1GB 实例耗时不到 10 ms，而 64GB 实例耗时超过 600 ms。我们还可以看到，页表复制耗时占据了执行时间的绝大部分。具体而言，在所有测试用例中，复制操作耗时占执行时间的比例均超过 97%，在 64GB 实例上更是高达 99.93%。为进一步研究复制操作，我们选取 8GB 实例，测量页表复制的详细指标。

在实验中，8GB 实例的页表包含一个 PGD 条目、八个 PUD、2^{12} 个 PMD 和 2^{21} 个页表项。总体而言，*fork* 沿着基数树从上到下逐级复制页表。复制一个 PGD/PUD/PMD 条目需要申请一页来存储其子节点并初始化该页。这项操作耗时约 500 ns。因此，复制 2^{12} 个 PMD 耗时约 2 ms，而复制 PGD 和 PUD 的开销微不足道，因为条目数量很少。其余时间（约 70 ms）用于复制页表项。根据这些结果，我们得出以下观察。

观察 ***1.*** 对于 *fork* 操作，随着实例大小增加，执行时间会显著增长，而页表复制占据了主要开销。在页表复制过程中，复制页全局目录（*PGD*）和 *PUD* 的开销微不足道，而复制 *PMD* 和页表项的开销则不可忽略。

表 1：父进程和子进程共享页表时，将页从“X”迁移到“Y”。

Step	操作	父进程(P)	子进程(C)
1	初始状态	转换后备缓冲器：V → X 页表项：V → X	转换后备缓冲器：V →X 页表项：V →X
2	P：设置页表项 →均不存在	转换后备缓冲器：V → X 页表项：V → N	转换后备缓冲器：V →X 页表项：V →N
3	P：刷新快表	转换后备缓冲器：不适用 页表项：V → N	转换后备缓冲器：V →X 页表项：V →N
4	C：已跳过因为 N！=X	转换后备缓冲器：不适用 页表项：V → N	转换后备缓冲器：V →X 页表项：V →N
5	P：更新页表项	转换后备缓冲器：不适用 页表项：V → Y	转换后备缓冲器：V →X 页表项：V →Y
6	权限检查：访问 V	转换后备缓冲器：V → Y 页表项：V → Y	转换后备缓冲器：V →X 页表项：V →Y

3.2 fork 操作对延迟的影响

本小节将揭示 fork 操作对查询延迟的影响。在介绍结果之前，我们先介绍现有的 fork 操作优化方法。

大页 [14, 15]。在操作系统（OS）中，我们可以增大页大小，以减少进程使用的页数，例如将页大小设置为 2MB，而不是 4KB。较大的页大小可以减少页表项的数量，并加快 fork 操作。然而，先前的研究发现，启用大页后，缺页延迟会从 3.6μs 增加到 378μs，因为缺页时的内存紧凑和内存清零在大页上的开销高于普通页 [31]。此外，启用大页后，Redis 消耗的内存空间会大幅增加，因为应用程序并不总是能充分利用大页（例如，在 [31], 的实验中，Redis 的内存消耗从 12.2GB 增加到 20.7GB）。此外，分配大页会导致物理内存中产生许多碎片，因为构建大页需要连续的物理内存区域。因此，内核需要执行开销很大的碎片整理操作，从而导致 CPU 利用率升高 [39]。 [39] 的实验表明，当内存高度碎片化时，SPEC CPU 2006 中的基准测试 *milc*会花费 343 秒（占总执行时间的 37%）在内核态执行碎片整理操作。

因此，许多数据库（例如 Couchbase、MongoDB、KeyDB 和 Redis 44] [18, 28, 37, ）都建议禁用大页。具体而言，Couchbase 和 MongoDB 这两个 NoSQL 数据库建议用户禁用该技术，因为在工作负载中，大页在随机内存访问场景下的表现不佳 [16, 37]。KeyDB 和 Redis 这两个内存键值存储系统建议用户禁用大页，因为该技术可能导致较大的延迟开销和较高的内存用量 [28, 44]。具体来说，如果启用大页，在调用 *fork* 将数据持久化到磁盘后，父进程和子进程会共享大页。在繁忙的实例中，任一进程中的少数几个事件循环都会访问数千个页，并因操作系统中的写时复制机制而触发大量进程内存的复制操作。因此，这会导致较高的延迟和较大的内存用量。因此，我们的性能分析不涉及这项技术。

共享页表 [18, 35, 59]。该技术提出在写时复制（CoW）方式下，让父进程与子进程共享页表。

具体而言，fork 操作在复制除页表之外的元数据后立即返回。页表则通过写时复制（CoW）方式进行复制。然而，我们发现，共享页表设计会引发数据泄漏问题、工作集大小估算问题和 NUMA 问题。首先，子进程的共享页表与转译后备缓冲器（TLB）之间的不一致可能导致数据泄漏问题。其次，我们无法准确估算工作集大小，而它反映了每个进程的内存用量，对云资源管理至关重要 [57]。这是因为内存用量是根据页表中的状态计算的 [21], , 但该页表由多个进程共享。第三，共享页表及其对应的进程可能位于不同的 NUMA 节点上，从而增加 TLB 未命中开销 [4, 38]。此外，由于共享页表，NUMA 均衡机制也无法按预期工作。受篇幅所限，我们将在技术报告中讨论工作集大小问题和 NUMA 问题 [2]。下文将通过表 1 中的示例说明数据泄漏问题，这是三种问题中最严重的一种。我们还编写了一个测试程序[2] ，以便在实际环境中复现该示例。

TLB 是一种硬件，通过缓存近期的地址转换结果，加速从虚拟地址到物理地址的转换。最初，虚拟地址“V”在页表项中映射到物理地址“X”，且该映射被缓存于 TLB 中。请注意，父进程和子进程共享页表项，而两个进程各自拥有独立的 TLB 条目。假设操作系统的内存管理机制（例如，内存压缩 [6, 53], 交换 [55]和 NUMA 平衡 [54]）在父进程中将页从“X”迁移到新的页框“Y”。随后，操作系统在步骤 2 中将映射从“V->X”设为“V->N”（不存在），以表示该映射无效，并在步骤 3 中刷新父进程的 TLB 条目。对于其他进程，操作系统会逐一检查其页表项是否包含“V->X”；如果包含，则将其值设为“V->N”并刷新 TLB；否则，跳过该进程。如果每个进程都有私有页表，这种机制就能正常工作。然而，ODF 使用共享页表设计。操作系统检查子进程时，由于父进程和子进程共享页表项，该页表项已在父进程中被设为“V->N”，因此操作系统无法在子进程中找到值为“V->X”的任何页表项。因此，操作系统在步骤 4 中跳过对子进程的更新。在步骤 5 中，父进程更新页表项，将“V”映射到“Y”。尽管页表项中的映射正确，但子进程的 TLB 条目与页表项不一致。因此，子进程后续访问“V”可能导致数据泄漏问题。尽管如此，为了进行比较，我们在实验中研究了基于共享页表的 *fork* 的性能。由于 ODF [59] 是最新的工作，也是采用共享页表设计的方法中唯一公开可用的[3] [18, 35, 59] ，我们的实验聚焦于 ODF。

实验结果。 图 4 和图 5 分别展示了普通查询和快照查询的第99百分位延迟和最大延迟。Snapshot-ODF 表示使用按需 fork [59] 创建快照的结果，这是最新的基于共享页表的 fork 方法。Snapshot-DEF 表示

[2]https://doi.org/10.5281/zenodo.7189585，最后访问于 2022/11/13。[3] https://github.com/rssys/on-demand-fork，最后访问于 2022/11/13。

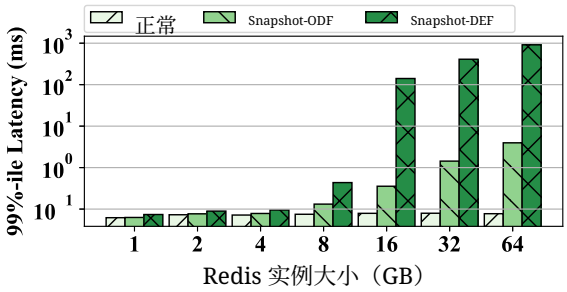


图 4：Redis 中普通查询和快照查询的 99 分位延迟。

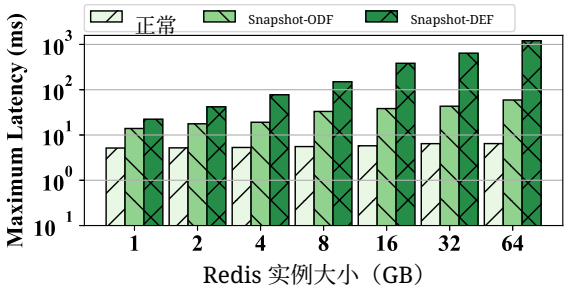


图 5：Redis 中普通查询和快照查询的最大延迟。

在 Linux 中使用默认 *fork* 的结果。如图所示，当实例大小从 1GB 增至 64GB 时，普通查询的延迟略有上升。相比之下，快照查询的延迟则急剧增长。共享页表技术显著降低了默认 *fork*的延迟，对于大实例尤为明显。例如，在 64GB 实例上，该优化将第99百分位延迟从 911.95ms 降至 3.96ms，并将最大延迟从 1204.78ms 降至 59.28ms。Snapshot-ODF 的延迟高于正常情况，因为页表的写时复制会频繁中断引擎进程。例如，在 16GB 实例上，引擎进程被中断 7000 多次，导致查询频繁出现服务中断。根据对现有优化方法的分析和实验结果，我们得出以下观察。

观察 *2*。*fork* 操作会显著影响快照查询的延迟，而默认函数的尾延迟高达数百毫秒。尽管共享页表技术降低了默认 *fork* 操作的延迟，但频繁中断带来的开销不可忽视，而且共享页表会引入潜在的数据泄漏问题。

3.3 总结

根据这些观察，我们发现，复制页表占据了 Linux 默认 *fork* 执行时间的绝大部分，尤其是在大型实例中。*fork* 的开销会导致在调用 *fork*期间发出的查询延迟很长（最高可达数百毫秒）。尽管已经提出了几种优化 fork 操作的方法 [14, 15, 18, 35, 59]，但它们要么会降低内存键值存储系统的性能，要么存在数据泄漏问题，可能生成不一致快照。因此，这些优化无法用于基于 fork 的快照机制

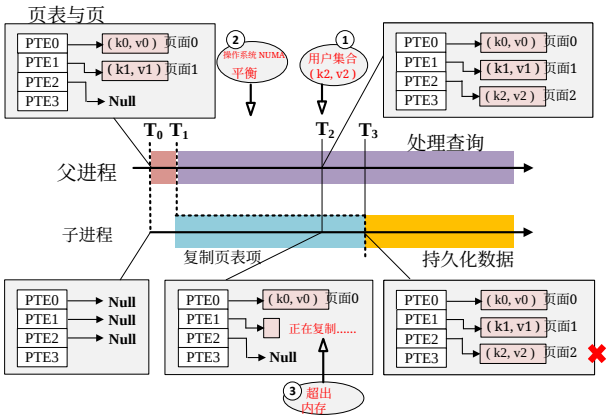


图 6：Async-fork 面临的挑战。

中。由于内存键值存储系统对在线场景下的服务延迟有严格要求，因此需要高性能的 *fork* 来降低快照查询的长延迟。

4 ASYNC-FORK 的设计

本节介绍 Async-fork 的设计。Async-fork 是一种基于操作系统的解决方案，能够有效降低快照查询的长延迟，且不会引入额外的漏洞。

4.1 总体思路

图1(c)展示了**Async-fork**的总体思路。在默认的*fork*中，父进程负责在创建快照时复制页表；如图所示，Async-fork 将复制页表的工作交给空闲的子进程，同时保持 fork 中的其他步骤不变。这样，父进程就能在子进程同时从父进程复制页表时处理查询。然而，要实现 Async-fork 的设计并非易事，因为对页表进行异步操作可能导致快照不一致。

当父进程在子进程复制某个页表项之前对其进行修改时，就会出现不一致。以图 6 为例。IMKVS 在时间 T_0创建快照，内存数据为 $\{(k_0, v_0), (k_1, v_1)\}$。假设在时间 T_2（图 6 中的 1）到达一条用户查询，要求设置新的键值对 (k_2, v_2)，而子进程在时间 T_1 至 T_3期间复制页表项。父进程处理这条查询时，会修改 PTE_2，使其指向包含 (k_2, v_2) 的新页（$Page_2$）。如果子进程尚未在修改前复制原始的 PTE_2，它就会复制修改后的 PTE_2，并在其内存空间中拥有新的键值对 (k_2, v_2)。这样一来，数据 (k_2, v_2) 得以持久化，从而产生不一致（在时间 T_0 创建快照时，键值对 (k_2, v_2) 并不存在）。

Async-fork 通过让父进程主动将已修改的页表项同步到子进程，解决了上述不一致问题。我们将在第 4.2 节详细介绍具体步骤。该解决方案需要应对两个主要挑战。

首先，需要检测所有页表项修改。然而，除了用户操作之外，操作系统中许多固有的内存管理操作也会导致页表项修改。例如，操作系统会定期在

算法 1： Async-fork 的框架

```
    // 父进程
1   对 每个 VMA_i （VMA_0,VMA_1,VMA_2...）执行
2       将 VMA_i 复制到子进程;
3       将 VMA_i 的页全局目录/页上级目录复制到子进程;
4       将所有 VMA_i 的 PMD 设置为写保护;
5   结束
6   将子进程放到一个 CPU 上运行;
7   当 true 时执行
8       如果检测到 PTE修改，且该页表项对应的页中间目录已设置为写保护则
9           将页中间目录复制到子进程;
10          将该页中间目录中的 512 个页表项复制到子进程;
11          将页中间目录设置为可写;
12          将 512 个页表项设置为写保护;
13      end
14  结束
    // 子进程
15  对于每个虚拟内存区域_i in (VMA_0, VMA_1, VMA_2...) 执行
16      对于 VMA_i 的每个页中间目录 j 执行
17          如果页中间目录 j 是写保护的则
18              从父进程复制页中间目录 j ;
19              从父进程复制页中间目录 j 的 512 个页表项;
20              将页中间目录设为可写;
21              将这 512 个页表项设为写保护;
22          end
23      end
24  结束
```

NUMA 节点之间迁移页 [54], , 导致相关页表项被修改为不可访问（图 6 中的 2）。我们将在第 4.3 节介绍检测页表项修改的方法。

其次，Async-fork 期间可能会发生错误。例如，子进程可能因内存不足而无法复制某个条目（图 6 中的 3）。在这种情况下，必须进行错误处理，因为我们应将进程恢复到调用 Async-fork 之前的状态。第 4.4 节介绍了如何处理错误。

通过解决上述挑战，父进程用于 *fork* 的时间大幅缩短，因此快照查询的延迟也得以降低。此外，子进程完成页表复制后，两个进程都拥有各自完整的私有页表。因此，Async-fork 不会引入数据泄露漏洞。

在接下来的章节中，我们使用 x86 Linux 中的术语来阐述我们的设计。我们也基于类似思路在 Arm Linux 上实现了 Async-fork，并取得了与 x86 相近的结果。算法 1 描述了 Async-fork 中父进程（第 1 至 14 行）和子进程（第 15 至 24 行）的工作方式。

4.2 主动同步

在详细介绍主动同步之前，我们先介绍如何在 Async-fork 中将复制页表的工作卸载到子进程。

异步复制页表。在默认的 *fork* 中，父进程会遍历其所有虚拟内存区域，并将页表中相应的部分复制到子进程。页表是从顶层向下复制的。在 Async-fork 中，父进程大致遵循上述流程，但只复制页全局目录、页四级目录（如果存在）和将 PUD 条目复制到子进程（算法 1 第 1 至 3 行）。之后，子进程开始运行，父进程返回用户态以处理查询（第 7 至 14 行）。子进程

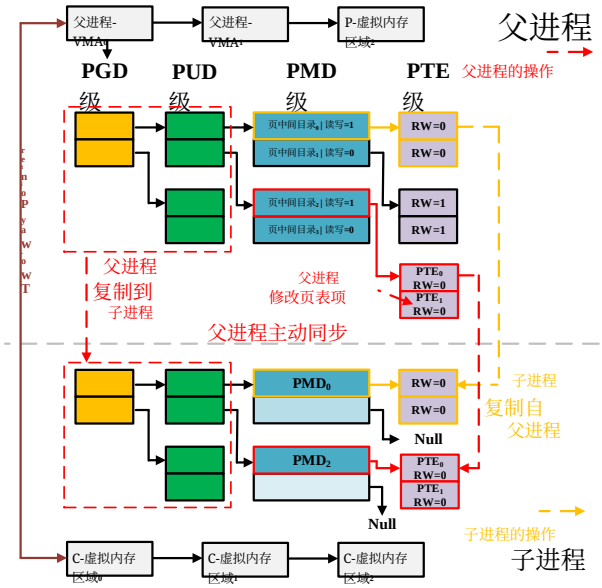


图7：Async-fork 中复制页表的示例。“RW=1”表示可写，“RW=0”表示写保护。

随后遍历虚拟内存区域，并从父进程复制页中间目录项和页表项。图7展示了异步页表复制的示例。在图中，父进程已经复制了 PGD/PUD条目，但页中间目录项和页表项尚未复制（部分页中间目录项指向“空”）。子进程从父进程复制页中间目录项和页表项（例如，页中间目录项$_0$ 及其页表项）。

我们将复制页中间目录项和页表项的工作交由子进程完成，因为如第 3.1 节所述，复制它们的开销不可忽略。同时，我们仍由父进程复制 PGD/PUD 条目。这是因为复制 PGD/PUD 条目的开销微不足道，而且尽量减少对 Linux 内核的改动会更稳妥。

主动同步已修改的PTE。当子进程负责复制页中间目录项和页表项时，父进程可能会在页表项实际复制之前对其进行修改。需要注意的是，只有父进程能够察觉这些修改（检测 PTE 修改的方法将在第 4.3 节介绍）。

通常，为保证一致性，有两种方式复制待修改的页表项：1）父进程主动将页表项复制到子进程；2）父进程通知子进程复制页表项，并等待复制完成。由于这两种方式都会导致父进程发生相同的中断，我们选择前一种方式（算法 1 的第 8 至 12 行）。

更具体地说，在快照期间修改页表项时，父进程会主动将该页表项以及同一个页表项表中的所有其他页表项（总计 512 个页表项），连同父进程的 PMD 条目，一并复制到子进程。例如，当图7中的 PTE$_1$ 被修改时，父进程会主动将 PMD$_2$、PTE$_0$ 和 PTE$_1$ 复制到子进程。我们选择复制整个页表项表，因为可以快速检测出将被修改的一组页表项，

但在实际应用中，准确识别其中具体哪个页表项将被修改，代价很高。

消除不必要的同步。始终让父进程复制已修改的页表项并无必要，因为待修改的页表项可能已由子进程复制。我们会检查页中间目录项和页表项是否已由子进程复制，以避免不必要的同步。为跟踪这一状态，需要一个标志。我们复用 PMD 条目的读写标志来记录该状态。由于在 x86 Linux 内核中，只有当 PMD 条目指向大页时才会使用读写标志，因此该标志能够正确跟踪状态。我们复用读写标志的原因有二：1）这种设计无需在内核数据结构中添加新字段；2）主流数据库（如 Redis、KeyDB、MongoDB 和 Couchbase）建议禁用大页以提升性能 [16, 28, 37, 44]。此外，如果应用程序使用大页，就没有必要使用 Async-fork，因为使用大页的应用程序不需要页表项，只需少量页全局目录项/页上级目录项/页中间目录项（即页表较小）。

另一种方法是使用*struct page* 中的一个未使用位作为标志。具体而言，每个 PMD 条目都指向一个页表项表，而操作系统会为每个页表项表维护一个数据结构（*struct page*）。*struct page* 中包含当前 Linux 内核未使用的位。以往的研究会将这些位用作标志，例如，ODF 使用*struct page* 中的一些位作为引用计数器。然而，这种方法需要进一步修改内核以初始化该位。因此，我们不采用使用*struct page* 的设计。

如果某个 PMD 条目及其 512 个页表项尚未复制到子进程，该 PMD 条目就会被设置为写保护（例如，图7中的 PMD_1 ）。请注意，这不会破坏*fork* 的写时复制策略，因为当 x86 [22] 上对应的页被写入时，仍会触发缺页异常。一旦 PMD/PTE 被复制到子进程（例如，PMD_0），该 PMD 条目就会改为可写（页表项则改为写保护，以维持写时复制策略）。由于父进程和子进程在复制 PMD 条目和页表项时，都会使用*trylock_page()* 锁定页表项表所在的页，因此它们不会同时复制由同一个 PMD 条目指向的页表项。

4.3 检测已修改的页表项

操作系统中修改页表项的操作可分为两类：1) *VMA* 范围修改。有些操作针对特定的虚拟内存区域，包括创建、合并、删除虚拟内存区域等。修改虚拟内存区域也可能导致其页表项被修改。例如，用户发送查询以删除大量键值对。随后，IMKVS（父进程）通过*munmap*回收相应的虚拟内存空间。因此，一些虚拟内存区域会被拆分或删除，其页表项也会随之删除。虚拟内存区域通常很大，因为操作系统总是倾向于合并相邻的虚拟内存区域。这意味着，VMA 范围修改通常会导致大量页表项修改，并涉及许多页中间目录项。2) *PMD* 范围修改。 其他操作会直接修改页表项。例如，父进程的页可能会被内存不足（OOM）杀手回收。在这种情况下，会涉及一个 PMD 条目。请注意，对 4KB 页进行换页或迁移会改变页表项，但不会改变数据，因此我们不会处理这类情况。由于篇幅有限，我们将操作系统中操作修改虚拟内存区域/页表项的位置总结为技术报告 [2] 中的检查点。

我们通过挂钩检查点来实现检测。到达检查点后，父进程会检查涉及的页中间目录项和页表项是否已被复制（通过检查 PMD 条目的读写标志）。对于 VMA 范围修改，会检查此虚拟内存区域中的所有 PMD 条目；而对于 PMD 范围修改，只检查一个 PMD 条目。未复制的 PMD 条目和页表项将在修改前复制到子进程。

如果虚拟内存区域很大，父进程通过逐一遍历所有 PMD/页表项进行检查可能需要相对较长的时间。因此，我们引入了一个双向指针，帮助父进程快速确定某个虚拟内存区域的所有条目是否已复制到子进程，从而降低开销。每个虚拟内存区域都有一个双向指针，该指针由父进程在调用 Async-fork 函数期间初始化。父进程的虚拟内存区域中的指针（相应地，子进程的虚拟内存区域中的指针）指向子进程（相应地，父进程）中对应的虚拟内存区域。这样，双向指针便在父进程和子进程的虚拟内存区域之间维持了连接。待该虚拟内存区域的所有 PMD/页表项复制到子进程后，连接将被关闭。具体而言，如果在复制该虚拟内存区域的 PMD/页表项期间没有发生 VMA 范围修改，则子进程会在复制操作完成后，将父进程和子进程虚拟内存区域中的指针都设置为 空 ，从而关闭连接。否则，父进程将同步该修改（即，将未复制的 PMD/页表项复制到子进程），并在复制操作完成后将指针设置为 空 ，从而关闭连接。由于父进程和子进程都可以访问双向指针，因此使用锁保护这些指针，以保持状态一致。当发生 VMA 范围修改时，父进程可以通过检查指针的值，快速确定某个虚拟内存区域的所有 PMD/页表项是否已复制到子进程，而无需遍历所有这些 PMD。此外，该指针还用于处理错误（见第 4.4 节）。

4.4 错误处理

由于复制页表涉及内存分配，默认 *fork* 和 Async-fork 过程中都可能发生一些错误。例如，进程可能因内存不足而无法初始化新的页表项表。在默认 *fork* 中，此类错误只可能发生在父进程中，并且有标准的错误处理方式。然而，在 Async-fork 中，页表复制工作被转交给子进程，因此此类错误可能发生在子进程中，需要有相应的方法来处理。具体而言，我们应将父进程恢复到调用 Async-fork 之前的状态，以确保父进程今后不会崩溃。

由于 Async-fork 可能会修改父进程 PMD 条目的读写标志，因此 Async-fork 发生错误时，我们会将这些条目回滚为可写状态。错误可能发生在以下情况：1）父进程复制 PGD/PUD 条目时；2）子进程复制 PMD/PTE 时；3）主动同步期间。对于第一种情况，父进程会回滚所有写保护的 PMD 条目。对于第二种情况，子进程会回滚所有剩余的未复制 PMD 条目。之后，我们会向子进程发送信号（*SIGKILL*）。子进程返回用户态并接收到该信号时，就会被终止。对于第三种情况，

父进程只回滚包含正在复制的 PMD 条目的虚拟内存区域（VMA）中的 PMD 条目。这样做是为了避免与子进程争用 PMD 条目锁。随后，错误码会存入 VMA 的双向指针中。在复制某个 VMA 的 PMD/PTE 之前（以及之后），子进程都会检查该指针以确认是否存在错误。如果存在错误，子进程就会立即停止复制 PMD/PTE 条目，并执行第二种情况中所述的回滚操作。

5 优化与实现

本节介绍进一步提升 Async-fork 性能的优化，以及在 Linux 中实现 Async-fork 的方法。

5.1 加速页表复制

触发主动 PTE 同步时，父进程仍会被中断。降低主动 PTE 同步开销的一种直接方法，是让子进程先复制可能被修改的页表项，再复制其他页表项。然而，这种方法并不实用，因为用户查询访问的数据相对随机，而且父进程调用 Async-fork 时，子进程无法确定哪些页表项会被修改。因此，我们提出通过缩短页表复制的持续时间，尽可能减少主动 PTE 同步的次数。

由于虚拟内存区域彼此独立，内核线程可以完全并行地执行复制，从而获得接近线性的加速效果。因此，子进程可以启动多个内核线程，并行复制多个虚拟内存区域中的 PMD/PTE，这样就能更快地完成复制。这可以有效减少主动 PTE 同步的次数，因为同步只会在子进程复制 PMD/PTE 期间发生。复制时间越短，触发主动 PTE 同步的概率就越低。第 6.3 节的实验展示了这项优化的效果。

多个内核线程会消耗 CPU 周期。这些线程会定期检查是否应被抢占，并通过调用 *cond_resched()* 来让出 CPU 资源，以减少对其他正常进程的干扰。

5.2 Async-fork 的实现

图 8 展示了采用模块化设计的 Async-fork 实现。我们将注入内核的代码封装在一个钩子函数中，并将其实例化为一个内核模块。根据需要，我们可以在内核中插入或移除该模块。该钩子

该函数被插入 Linux 内核（版本 4.19）中默认 *fork* 和内存子系统的调用路径。用户可以通过参数 F 确定使用默认 *fork* 还是 Async-fork。

钩子函数基于*copy_page_range()*进行了增强，可接受一个标志（快速或慢速）来控制页表的复制。使用“快速”标志时，函数会复制 PGD/PUD 条目，并将页中间目录项设为写保护。使用“慢速”标志时，函数会复制写保护的PMD条目和页表项；复制完成后，会将页中间目录项设为可写，并将页表项设为写保护。调用 Async-fork 时，父进程使用带有

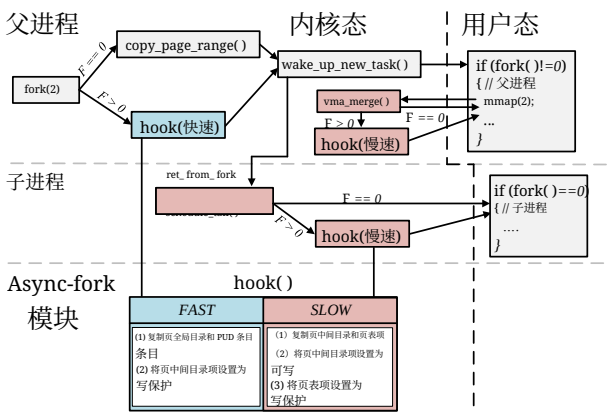


图 8：Async-fork 的实现。

“快速”标志的函数将页表复制到子进程。在 Async-fork 调用结束时，父进程将子进程放入某个 CPU 的运行队列 中，并返回用户态。在子进程返回用户态之前，它会使用带有“慢速”标志的函数复制页表。父进程会在修改尚未复制的 PMD/PTE 之前，主动使用带有“快速”标志的函数将它们复制到子进程。

灵活性。 对于内存占用较小的 IMKVS 工作负载，页表复制所需时间本来就很短。在这种情况下，Async- fork 带来的收益较小。针对这些工作负载，我们在内存控制组 中提供了一个接口，用于控制是否启用 Async-fork（以及用于加速子进程中 PMD/PTE 复制的内核线程数量）。具体而言，当用户将进程添加到一个内存控制组时，可以传递一个参数，以便在运行时启用或禁用 Async-fork（即图 8 中的参数F）。如图所示，如果参数值为 0，则进程将使用默认*fork*。否则，将启用 Async-fork。这样，用户便可以根据需要确定进程使用哪种 fork 操作，并且无需修改应用程序的源代码即可使用 Async-fork。如果未传递参数，进程将使用默认的 *fork* 。

内存开销。 Async-fork 唯一的内存开销来自进程中每个虚拟内存区域新增的指针（8B）。在这种情况下，进程的内存开销等于虚拟内存区域的数量乘以 $8B$。根据我们的统计，在一台拥有 512GB 主内存、同时运行 400 个进程的机器上，大约有 760,000 个虚拟内存区域。在这种情况下，内存开销为 $760000 \times 8\text{B} \approx 6\text{MB}$。通常，这项开销可以忽略不计。

支持 ARM64。 Async-fork 的设计也可以在 ARM64 上实现。具体而言，我们将 PMD 条目的页表描述符中的 *APTable[1: 0]* [34] 用作读写标志。Async-fork 也可以在其他支持页表分层属性的体系结构上实现。

连续快照。父进程可能会在上一个子进程完成 PMD/PTE 复制之前，使用 Async-fork 启动下一个快照，因为父进程会先于子进程返回用户态。在当前实现中，我们不会阻塞下一次 Async-fork 调用，但会确保每个虚拟内存区域的页表在任意时刻仅由一个子进程复制。当父进程将一个虚拟内存区域复制到子进程时，它会检查

双向指针，以确定是否存在正在复制此虚拟内存区域页表的前一个子进程。如果存在，它会主动将此虚拟内存区域的整个页表复制到前一个子进程。Async-fork 采用这种设计，是因为支持并发 fork 操作需要跟踪所有正在复制页表的子进程。这很困难，因为内核不提供相关信息，而且我们需要向内核中注入新的数据结构来记录状态并同步这些子进程。

由于 Async-fork 不支持在一个进程中并发执行 fork 操作（与操作系统中的默认 *fork* 相同），因此无法支持父进程需要以极高频次（例如每毫秒）执行 fork 操作的场景。尽管存在这一限制，Async-fork 仍适用于数据库用例，因为在实际应用中，存储引擎不会同时从父进程创建许多子进程。具体而言，频繁执行 fork 操作会导致父进程频繁切换到内核态，从而损害服务质量。此外，由于资源争用（例如 IO 带宽和 CPU 资源），许多进程并行执行会降低性能。因此，据我们所知，IMKVS（例如 Redis 和 KeyDB）不建议以极高频次创建数据快照（通常为 60 秒 [47]），也没有需要父进程以毫秒为间隔调用 Async-fork 的场景。对于使用 fork 操作支持并发事务处理的 HyPer，Async-fork 则能很好地发挥作用。这是因为父进程处理联机事务处理（单个更新器），对延迟有严格要求；而子进程执行联机分析处理，其执行时间通常较长，且对延迟的容忍度高于联机事务处理。此外，HyPer 注意到了 fork 操作的开销，并设计了一种新颖的机制，让多个联机分析查询共享一个快照，从而提升性能。

6 评估

在本节中，我们评估 Async-fork 的有效性。

6.1 实验设置

我们在一台配备两颗 Intel Xeon Platinum 8163 处理器的机器上评估 Async-fork，每颗处理器均有 24 个物理核心（48 个逻辑核心）。该机器配备 384GB 内存和一块 1TB NVMe 硬盘。在软件配置方面，实验平台运行 CentOS 7.9 和 Linux 4.19。除第 6.4 节中的实验外，所有其他实验均按照上述说明在单台机器上进行。

基准测试。我们使用 Redis（版本 5.0.10） [42] 和 KeyDB（版本 6.2.0） [25]，通过 gcc 6.5.1 编译，将其作为具有代表性的 IMKVS 服务器；并使用 Redis 基准测试工具 [41] 和 Memtier 基准测试 [46] 作为负载生成器。这些基准测试展示了多个客户端同时向 IMKVS 服务器发送请求的场景。与先前的工作 [13], 类似，我们改进了 Redis 基准测试工具，使其能够以开环模式生成查询，从而准确测量延迟 [48, 58]。

默认情况下，实验采用以下设置：1）Redis 基准测试工具使用 50 个客户端（默认设置），而为了保持一致性，Memtier 基准测试使用 50 个客户端。2）键范围设为 2×10^8，键大小为 8B，值大小为 1024B。3）每项实验重复进行五次，并报告平均结果。4）在 Async-fork 中，子进程会启动 7 个额外的内核线程（加上子进程本身，共有 8 个线程），以帮助其更快地复制 PMD/页表项。图14中的实验表明，即使只有一个子进程，Async-fork 的性能仍优于最先进的解决方案。5）KeyDB 服务器配置为使用 4 个线程。

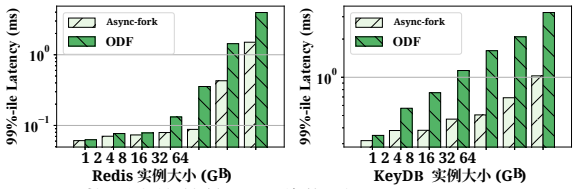


图9：快照查询的第99百分位延迟。

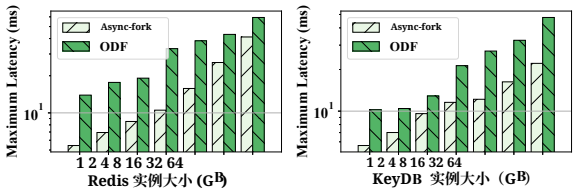


图10：快照查询的最大延迟。

指标。我们向 IMKVS 发起大量查询（5×10^6），并记录在快照过程中到达的查询的延迟（从父进程调用 *fork* 开始，直到子进程持久化所有内存数据）。具体而言，我们测量第99百分位延迟，因为延迟敏感型服务通常会针对某个百分位制定服务级别协议。此外，我们还报告最大延迟，因为 IMKVS 通常用于对最坏情况延迟有严格要求的高要求用例 [43]。例如，Redis 最大延迟的增加可能导致连接出现读取错误 [52]。因此，在生产环境中 [20], ，最大延迟是系统稳定性的一项重要指标。由于篇幅有限，查询处理吞吐量和父进程总停服时间的实验结果见技术报告 [2]。报告中还讨论了如何调整内存键值存储系统，以进一步提升性能。

基线方案。 我们将 Async-fork 与 On-Demand-Fork（简称 *ODF*）进行比较，后者是当前最先进的基于共享页表的 fork 方案。在 ODF 中，每当一个进程修改共享页表项时，不仅会复制一个页表项，还会同时复制位于同一页表中的 512 个页表项。由于默认的 *fork* 在大多数情况下产生的延迟比 Async-fork 和 ODF 高出 10 倍（已在第 3.2 节中介绍），因此本节不报告其结果。为准确测量实验中内存键值存储系统的性能，我们使用 BGSAVE 命令手动触发快照操作。

6.2 总体评估

我们首先使用需要频繁进行快照以实现数据持久化的写密集型工作负载来评估 Async-fork。

延迟结果。在此实验中，我们使用 Redis 基准测试 [41]，向内存键值存储系统发送 *SET* 查询，以生成写密集型负载，并将客户端配置为每秒发送 50,000

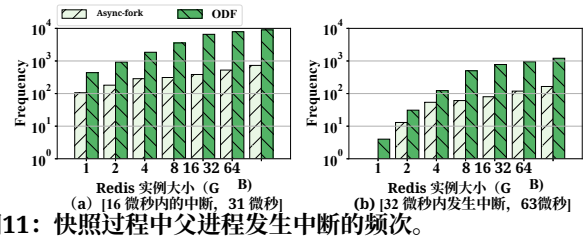


图11：快照过程中父进程发生中断的频次。

次此类查询。由于 IMKVS 常用于高要求用例，因此通常对第99百分位延迟和最坏情况延迟 [26, 43] 都有严格要求。

图9展示了采用 ODF 和 Async-fork 时，Redis 与 KeyDB 中快照查询的第99百分位延迟。可以看出，所有情况下 Async-fork 的性能都优于 ODF，且实例越大，性能差距越明显。例如，在 64GB 的 IMKVS 实例上，采用 ODF 时，快照查询的第99百分位延迟在 Redis 中为 3.96ms，在 KeyDB 中为 3.24ms；采用 Async-fork 后，延迟分别降至 1.5ms（Redis，降低 61.9%）和 1.03ms（KeyDB，降低 68.3%）。

图10展示了快照查询的最大延迟。可以看出，与 ODF 相比，即使实例较小，Async-fork 也能大幅降低基准测试的最大延迟。对于 1GB 的 IMKVS 实例，采用 ODF 时，快照查询的最大延迟分别为 13.93ms（Redis）和 10.24ms（KeyDB）；采用 Async-fork 后，最大延迟降至 5.43ms（Redis，降低 60.97%）和 5.64ms（KeyDB，降低 44.95%）。

深入探究。 Async-fork 的中断由主动同步引起，而 ODF 的中断则由页表的写时复制引起。我们测量快照期间父进程受到的中断，以了解 Async-fork 优于 ODF 的原因。具体而言，父进程在执行 *copy_页中间目录_range()* 函数时会进入内核态，并停止为查询提供服务。执行 *copy_页中间目录_range()* 的开销占内核态执行成本的主要部分。为了检查中断次数和停服时间，我们使用*bcc* 工具 [3] 统计 *copy_页中间目录_rage()* 的调用次数，并测量每次调用的执行时间。*bcc* 的结果是一个直方图，其中区间表示时间长度，频次表示执行时间落在该区间内的调用次数。类别 [16*us*, 31*us*] 和 [32*us*, 63*us*]是*bcc*的两个默认区间。在我们的实验中，所有调用都落在这两个区间内。

图11显示了在 [16us、31us] 和 [32us、63us] 内中断的频次。可以看出，Async-fork 显著降低了中断频次。例如，Async-fork 在 16GB 实例上，中断频次从 7348 次降至 446 次。Async-fork 大幅减少了中断次数，因为只有在子进程复制页中间目录/页表项时才会发生中断（如图15所示，所需时间在 600ms 以内）。然而，在子进程通过 ODF 持久化所有数据之前，中断都可能发生，而数据持久化操作需要数十秒（例如，持久化 8GB 内存数据约需 40 秒）。在相同工作负载下，使用 ODF 时，父进程更容易受到中断影响。

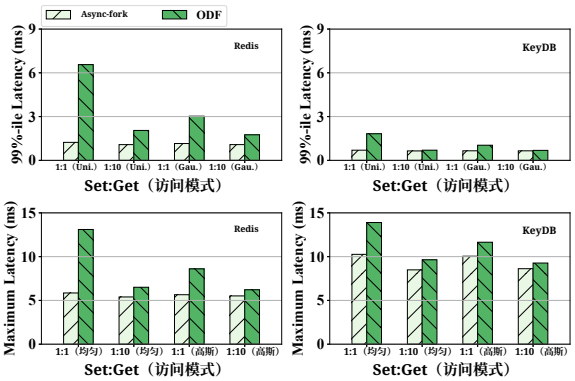


图12：8GB IMKVS 在不同工作负载下快照查询的第99百分位延迟和最大延迟。

6.3 详细评估

对读写模式的敏感性。图12展示了使用 Memtier [46] 生成的四种不同读写模式工作负载所得到的结果。图中，“1:1 (Uni.)”表示 Set:Get 比为 1:1 且采用均匀随机访问模式的工作负载，而“1:10 (Gau.)”表示 Set:Get 比为 1:10 且采用高斯分布访问模式的工作负载。

从图12可以看出，Async-fork 仍优于 ODF。对于 GET 查询更多的工作负载，其收益较小。这是因为在 GET 密集型工作负载下，服务（父进程）仅复制少量页表项。此外，与均匀随机访问模式的实验相比，高斯分布实验中被修改的内存更少。在随机访问模式下，IMKVS 中的键值对被访问的概率相同；而在高斯分布访问模式下，部分键值对可能会被重复访问。由于在 ODF 中，共享页表项仅在首次修改时才会被复制，因此在高斯分布访问模式下，父进程被中断的次数更少。

总体而言，Async-fork 在写密集型工作负载下表现更好。被修改的内存越多，Async-fork 的性能就越好。将 Async-fork 与 CCoW [23]结合以提升写密集型工作负载的性能，是一个值得探索的研究方向，因为 Async-fork 可以利用 CCoW 预先复制高局部性内存页的页表项，从而减少主动同步的次数。

客户端数量的影响。 在此实验中，我们调整 Redis-benchmark 中的客户端数量，同时持续以每秒 50,000 次的速率向一台 8GB 的 IMKVS 服务器发送 *SET* 查询。图13展示了客户端数量为 10、50、100 和 500 时的 99% 分位延迟和最大延迟。结果表明，Async-fork 的性能优于 ODF，且客户端数量越多，性能差距越大。这是因为客户端数量增加时，IMKVS 会同时收到更多请求。因此，可能会有更多页表项同时被修改，而且父进程一次中断的持续时间可能会更长。

子进程中线程数量的影响。在 Async-fork 中，可以使用多个内核线程在子进程中并行复制 PMD/PTE。图14展示了

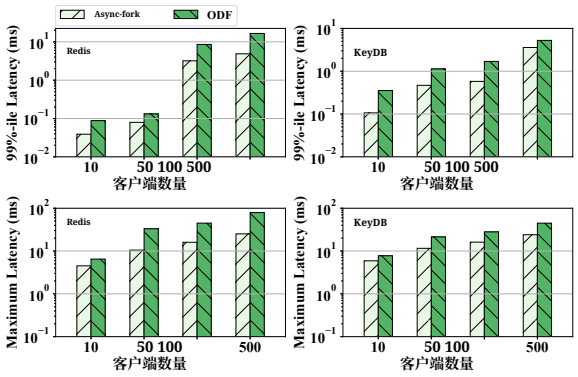


图13：在 8GB IMKVS 中，不同客户端数量下快照查询的第99百分位延迟和最大延迟。

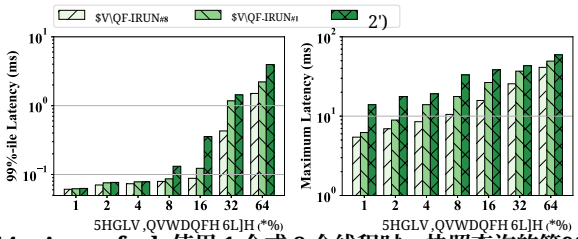


图14：Async-fork 使用 1 个或 8 个线程时，快照查询的第99百分位延迟和最大延迟。

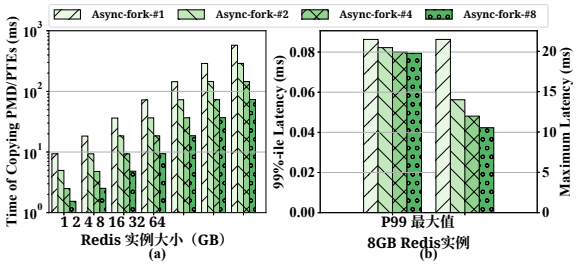


图15：（a）Async-fork 中子进程复制 PMD/PTE 所需的时间。（b）8GB Redis 实例中的第99百分位延迟和最大延迟。

不同 Redis 实例大小下，快照查询的第99百分位延迟和最大延迟。Async-fork#*i* 表示使用总计 *i* 个线程复制 PMD/PTE 的结果。

从图14可以看出，Async-fork#1（子进程本身）的延迟仍低于 ODF。与 ODF 相比，快照查询的最大延迟平均降低了34.3%。我们还发现，使用更多线程（ Async-fork#8 ）可以进一步降低最大延迟。这是因为子进程越早完成 PMD/PTE 的复制，父进程被中断以主动同步 PTE 的概率就越低。

具体而言，图15(a)展示了子进程在不同数量的内核线程下复制 PMD/PTE 所花费的时间，图15(b)则展示了一个 8 GB Redis 实例中相应的 99% 分位延迟和最大延迟。可以看到，启动更多内核线程可以有效缩短子进程复制 PMD/PTE 的时间。耗时越短，延迟就越低。

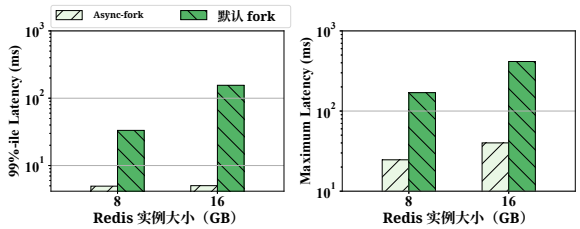


图16：在生产环境中，使用 Async-fork 执行快照查询时的第99百分位延迟和最大延迟。

6.4 生产环境中的评估

Async-fork 已部署在我们的生产环境 Redis 云中。我们从公有云租用一个 Redis 数据库，以评估 Async-fork 在生产环境中的表现。租用的 Redis 服务器配备 16GB 内存和 80GB SSD。我们还从同一云服务商租用另一台配备 4 个 vCPU 核心和 16GB 内存的虚拟机，用作运行 Redis 基准测试工具的客户端。客户端设置与先前实验相同。IMKVS 服务器与客户端之间的网络带宽为 3Gbps。

由于我们的云中无法使用 ODF，本小节将 Async-fork 与默认的*fork* 进行比较。图16展示了使用默认*fork* 和 Async-fork 执行快照查询时的第99百分位延迟和最大延迟。可以看到，在 8GB 实例上，快照查询的第99百分位延迟从 33.29ms 降至 4.92ms，最大延迟从 169.57ms 降至 24.63ms。在 16GB 实例上，前者从 155.69ms 降至 5.02ms，后者则从 415.19ms 降至 40.04ms。

在将 Async-fork 部署到我们的 Redis 云之前，我们收到大量租户投诉，称执行快照时延迟过长。一些租户甚至不得不禁用快照功能，而且他们无法重启 Redis 服务器，否则会导致数据丢失。部署 Async-fork 后，我们再也没有收到延迟方面的投诉。

7 结论

本文研究了内存键值存储系统中基于 fork 的快照机制所引发的延迟尖峰，并从操作系统层面解决这一问题。具体而言，我们深入研究了 fork 操作对延迟尖峰的影响。基于这项研究，我们提出了 Async-fork。它通过将页表复制的工作负载从父进程转移到子进程，优化了 fork 操作。为保证父进程与子进程之间的数据一致性，我们设计了主动同步策略。
Async-fork 已在 Linux 内核中实现，并部署于生产环境。大量实验结果表明，本文提出的技术能够显著降低快照期间到达的查询的尾延迟。

致谢

本工作部分由中国国家自然科学基金（62232011、62022057）和上海国际科技合作项目（21510713600）资助。阿里巴巴集团还通过阿里巴巴创新研究计划支持了本工作。Quan Chen 和 Minyi Guo 为通讯作者。

参考文献

[1] [无日期]。fork(2)。https://linux.die.net/man/2/fork。[2] 2022。Async-fork：从操作系统层缓解基于 fork 的快照机制引发的查询延迟峰值（完整版本）。https://drive.google.com/drive/folders/1xkwhyYmyDV5dD-X03ge1CmwUBNnGgN4X?usp=sharing。[3] 2022。BPF 编译器集合 (BCC)。https://github.com/iovisor/bcc。[4] Reto Achermann、Ashish Panwar、Abhishek Bhattacharjee、Timothy Roscoe 和 Jayneel Gandhi。2020年。Mitosis：为大内存机器提供透明的自复制页表。载于第二十五届编程语言与操作系统架构支持国际会议论文集。283–300。[5] Antirez。 [无日期]。Redis 持久化揭秘。http://antirez.com/post/redis-persistence-demystified.html。[6] Vlastimil Babka。2016年。mm，compaction：引入 kcompactd。https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=698b1b30642f1ff0ea10ef1de9745ab633031377。[7] Katelin A Bailey、Peter Hornyack、Luis Ceze、Steven D Gribble 和 Henry M Levy。2013年。探索键值存储中的存储级内存。载于第一届非易失性内存/闪存与操作系统及工作负载交互研讨会论文集。1–8。[8] Philip A Bernstein、Vassos Hadzilacos 和 Nathan Goodman。1987年。数据库系统中的并发控制与恢复。第370卷。Addison-Wesley Reading。[9] Greg Bronevetsky、Rohit Fernandes、Daniel Marques、Keshav Pingali 和 Paul Stodghill。2006年。检查点/恢复系统的最新进展。载于第二十届 IEEE 并行与分布式处理国际研讨会论文集。IEEE，8–页。[10] Tuan Cao。2013年。云中主存应用程序的容错。康奈尔大学。[11] Tuan Cao、Marcos Vaz Salles、Benjamin Sowell、Yao Yue、Alan Demers、Johannes Gehrke 和 Walker White。2011年。面向频繁保持一致的应用程序的快速检查点恢复算法。载于2011年 ACM SIGMOD 数据管理国际会议论文集。265–276。[12] Badrish Chandramouli、Guna Prasaad、Donald Kossmann、Justin Levandoski、James Hunter 和 Mike Barnett。2018年。Faster：支持原地更新的并发键值存储。载于2018年数据管理国际会议论文集。275–290。[13] Shuang Chen、Christina Delimitrou 和 José F Martínez。2019年。Parties：面向 QoS 的多交互式服务资源划分。载于第二十四届编程语言与操作系统架构支持国际会议论文集。107–120。[14] 内核开发社区。2021年。HugeTLB 页。https://www.kernel.org/doc/html/latest/admin-guide/mm/hugetlbpage.html。[15] 内核开发社区。2021年。透明大页支持。https://www.kernel.org/doc/html/latest/admin-guide/mm/transhuge.html。[16] Couchbase。2021年。禁用透明大页 (THP)。https://docs.couchbase.com/server/current/install/thp-disable.html。[17] Cristian Diaconu、Craig Freedman、Erik Ismert、Per-Ake Larson、Pravin Mittal、Ryan Stonecipher、Nitin Verma 和 Mike Zwilling。2013年。Hekaton：SQL Server 的内存优化 OLTP 引擎。载于2013年 ACM SIGMOD 数据管理国际会议论文集。1243–1254。[18] Xiaowan Dong、Sandhya Dwarkadas 和 Alan L Cox。2016年。重新审视共享地址转换。载于第十一届欧洲计算机系统会议论文集。1–15。[19] Bin Fan、David G Andersen 和 Michael Kaminsky。2013年。Memc3：采用更简单的缓存和更智能的哈希算法的紧凑型并发 memcache。载于第十届 {USENIX} 网络系统设计与实现研讨会（{NSDI} 13）。371–384。[20] Caixin Gong、Chengjin Tian、Zhengheng Wang、Sheng Wang、Xiyu Wang、Qiulei Fu、Wu Qin、Long Qian、Rui Chen、Jiang Qi、Ruo Wang、Guoyun Zhu、Chenghu Yang、Wei Zhang 和 Feifei Li。2022年。Tair-PMem：完全持久化的非易失性内存数据库。《VLDB 学报》15，12（2022年8月），3346–3358。[21] Brendan Gregg。2018年。工作集大小估算。https://www.brendangregg.com/wss.html。[22] Part Guide。2011年。Intel ® 64 和 IA-32 架构软件开发人员手册。第3B卷：系统编程指南，第2部分，11（2011年）。[23] Minjong Ha 和 Sang-Hoon Kim。2022年。CCoW：根据工作负载的空间局部性优化写时复制。《电子学》11，3（2022年）。https://doi.org/10 3390 11030461 。/electronics Alan Harris。无日期。通过 memcached 实现分布式缓存。载于 Pro ASP.NET 4 CMS。Springer，165–196。[25] Snap Inc. 2022年。KeyDB。https://keydb.dev/。[26] Snap Inc. 2022年。延迟监控工具。https://docs.keydb.dev/docs/latency-monitor/。[27] Snap Inc. 2022年。持久化。https://docs.keydb.dev/docs/persistence。[28] Snap Inc. 2022年。延迟问题排查。https://docs.keydb.dev/docs/latency/。[29] Harshad Kasture and Daniel Sanchez。2016年。Tailbench: a benchmark suite and 面向延迟关键型应用的评估方法。在2016年IEEE工作负载特征研讨会（IISWC）上。1–10。[30] Alfons Kemper和Thomas Neumann。2011年。HyPer：一种基于虚拟内存快照的混合型联机事务处理与联机分析处理主存数据库系统。载于2011年IEEE第27届数据工程国际会议。IEEE，195–206。[31] Youngjin Kwon、Hangchen Yu、Simon Peter、Christopher J Rossbach和Emmett Witchel。2016年。使用ingens实现协调且高效的大页管理。载于第12届{USENIX}操作系统设计与实现研讨会（{OSDI} 16）。705–721。[32] Liang Li、Guoren Wang、Gang Wu和Ye Yuan。2018年。内存数据库系统的一致性快照算法：实验与分析。载于2018年IEEE第34届数据工程国际会议（ICDE）。IEEE，1284–1287。[33] A-P Liedes和Antoni Wolski。2006年。Siren：一种节省内存且快照一致的内存数据库检查点算法。载于第22届数据工程国际会议（ICDE'06）。IEEE，99–99。[34] Arm有限公司。 [无日期。]。适用于A-profile架构的Armv8架构参考手册。https://developer.arm.com/documentation/ddi0487/gb/。[35] Dave McCracken。2003年。在Linux 内核中共享页表。载于Linux研讨会。315。[36] Michael J Mior和Eyal de Lara。2011年。FlurryDB：一种通过虚拟机克隆实现动态扩展的关系型数据库。载于第4届系统与存储年度国际会议论文集。1–9。[37] MongoDB。2021年。禁用透明大页 (THP)——MongoDB手册。https://docs.mongodb.com/manual/tutorial/transparent-huge-pages。[38] Ashish Panwar、Reto Achermann、Arkaprava Basu、Abhishek Bhattacharjee、K Gopinath和Jayneel Gandhi。2021年。借助vMitosis为虚拟化NUMA服务器提供快速本地页表。载于第26届ACM编程语言与操作系统架构支持国际会议论文集。194–210。[39] Ashish Panwar、Aravinda Prasad和K Gopinath。2018年。让大页真正发挥作用。载于第二十三届编程语言与操作系统架构支持国际会议论文集。679–692。[40] Jiwoong Park、Yunjae Lee、Heon Young Yeom和Yongseok Son。2020年。面向内存数据库系统的节省内存的基于fork的检查点机制。载于第35届ACM应用计算年度研讨会论文集。420–427。[41] RedisLab。2021年。Redis有多快？https://redis.io/topics/benchmarks。[42] RedisLab。2021年。Redis。https://redis.io/。[43] RedisLab。2021年。Redis延迟监控框架。https://redis.io/topics/latency-monitor。[44] RedisLab。2021年。Redis延迟问题故障排查。https://redis.io/topics/latency。[45] RedisLab。2021年。Redis持久化。https://redis.io/topics/persistence。[46] RedisLab。2022年。memtier。https://github.com/RedisLabs/memtier_基准测试。[47] RedisLab。2022年。redis.conf。https://github.com/redis/redis/blob/7.0/redis.conf。[48] Bianca Schroeder、Adam Wierman和Mor Harchol-Balter。2006年。开放与封闭：一个警示故事。USENIX。[49] Matthew Shafer。2012年。Memcached。https://github.com/memcached/memcached。[50] Ankur Sharma、Felix Martin Schuhknecht和Jens Dittrich。2018年。利用细粒度高频虚拟快照加速多版本并发控制中的分析处理。载于2018年数据管理国际会议论文集（美国得克萨斯州休斯敦）（SIGMOD '18）。计算机协会，美国纽约州纽约，245–258。[51] solid IT。2022年。DB-Engines键值存储排名。https://db-engines.com/en/ranking/key-value+存储。[52] Trivago Technology。2017年。用困难的方式学习Redis（在生产环境中）。https://tech.trivago.com/2017/01/25/learn-redis-the-hard-way-in-production/。[53] Linus Torvalds。2018年。linux/mm/compaction.c。https://github.com/torvalds/linux/blob/v4.19/mm/compaction.c。[54] Linus Torvalds。2018年。linux/mm/memory.c。https://github.com/torvalds/linux/blob/v4.19/mm/memory.c。[55] Linus Torvalds。2018年。linux/mm/vmscan.c。https://github.com/torvalds/linux/blob/v4.19/mm/vmscan.c。[56] Hao Zhang、Gang Chen、Beng Chin Ooi、Kian-Lee Tan和Meihui Zhang。2015年。内存大数据管理与处理：综述。IEEE知识与数据工程汇刊27，7（2015年），1920–1948。[57] Wei Zhang、Ningxin Zheng、Quan Chen、Yong Yang、Zhuo Song、Tao Ma、Jingwen Leng和Minyi Guo。2020年。URSA：基于底层统计数据的公有云精准容量规划与公平调度。载于第49届并行处理国际会议（ICPP）（加拿大阿尔伯塔省埃德蒙顿）（ICPP '20）。计算机协会，美国纽约州纽约，文章73，共11页。[58] Yunqi Zhang、David Meisner、Jason Mars和Lingjia Tang。2016年。Treadmill：通过精准负载测试和统计推断确定尾延迟的来源。载于2016年ACM/IEEE第43届年度国际计算机体系结构研讨会（ISCA）。IEEE，456–468。

[59] Kaiyang Zhao、Sishuai Gong和Pedro Fonseca。2021年。按需 fork：面向内存密集型和延迟敏感型应用的微秒级fork。载于第十六届欧洲计算机系统会议论文集。540–555。

A 共享页表设计引发的问题

我们最初考虑在云中使用 ODF 来解决查询延迟峰值问题。然而，我们发现，ODF [59]（以及之前的研究 [18, 35]）采用的共享页表设计会引发若干问题，阻碍其在生产环境中的使用。

数据泄漏问题。如第3.2节所述，由于页表与转换后备缓冲器之间不一致，ODF 存在数据泄漏问题。修复数据泄露漏洞的一种潜在方法如下：1）记录每个页表项被哪些进程共享；2）如果父进程将某个页的页表项设为无效，操作系统便通知共享该页表项的其他进程刷新转换后备缓冲器条目。然而，要记录所有共享某个页表项的进程并不容易；这个问题类似于反向映射 问题，即将物理页反向映射到与之对应的页表项。虽然反向映射的概念看起来简单易实现，但实际操作中却很困难，因为它会带来空间和访问效率方面的问题。正如《Linux设备驱动程序》的作者 Jonathan Corbet 在博客[4,5]中所述，每种反向映射方法都需要复杂的数据结构和更新机制。遗憾的是，ODF 只记录每个页表项被多少个进程共享，却没有反向映射信息。因此，ODF 无法支持这种修复方法。

工作集大小（WSS）估算问题。Linux 内核无法使用 ODF 准确估算进程的工作集大小（WSS），而 WSS 是一项重要指标，云服务提供商利用它来提高资源利用率。具体而言，WSS 是进程在运行时实际需要的内存大小。进程可以占用远大于实际使用量的内存空间（例如，启动时申请固定大小的内存池，但运行时不缩减）。Linux 内核通过统计进程访问过的页数来估算其 WSS（依据页表项的访问位 [21]）。然而，ODF 的共享页表设计使内核无法区分访问某个页的是哪个进程。因此，Linux 内核无法准确估算 WSS。在实践中，提高资源利用率对公有云非常重要，因为租户通常希望申请超过实际需求的资源（例如，我们的云中有 68.4% 的内存空间被浪费 [57]）。

NUMA问题。我们发现，NUMA均衡机制在 ODF 中无法按预期工作。具体来说，当 NUMA均衡机制决定更改子进程的页面位置时，会将页表项设置为 PROT_NONE（即后续访问这些页时会触发缺页异常），以便让子进程处理这些异常。然而，由于采用了共享页表设计，父进程和子进程都可能捕获这些异常。此外，当共享页表的进程在不同的 NUMA 节点上运行时，TLB 未命中的开销会增加，因为页表项只有一份副本，而且部分进程和页表项可能位于

[4]https://lwn.net/Articles/23732. 最后访问于 2022/11/13。[5] https://lwn.net/Articles/383162. 最后访问于 2022/11/13。

表 2：在子进程于 Async-fork 中复制页表项（“V->X”）之前，将页从“X”迁移到“Y”。

Step	操作	父进程（P）	子进程（C）
1	初始状态	转换后备缓冲器：V->X 页表项：V->X	转换后备缓冲器：不适用 页表项：不适用
2	P：设置页表项 → 均不存在	转换后备缓冲器:V->X 页表项:V->N	转换后备缓冲器:N/A 页表项:N/A
3	P: 刷新快表	转换后备缓冲器:N/A 页表项:V->N	转换后备缓冲器:N/A 页表项:N/A
4	P: 更新页表项	转换后备缓冲器:N/A 页表项:V->Y	转换后备缓冲器：不适用 页表项：不适用
5	C：复制页表项	转换后备缓冲器：不适用 页表项：V->Y	转换后备缓冲器：不适用 页表项：V->Y
6	权限检查：访问 V	转换后备缓冲器：V->Y 页表项：V->Y	转换后备缓冲器：V->Y 页表项：V->Y

不同节点上。由于需要在远程节点上遍历页表，TLB 未命中的开销可能会增加 [4, 38]。尽管我们的实验未考虑 NUMA 环境，但我们提及这一问题，是为了说明 ODF 与实际情况之间的差距。

要解决这些问题绝非易事，需要付出大量努力（例如，添加新的数据结构并进一步修改内核）。这促使我们开发一种可在生产环境中运行的新 fork 机制。与共享页表设计不同，Async-fork 的设计与默认 fork 操作相同，即父进程和子进程各自拥有私有页表。如果父进程中要更新某个页表项（属于父进程的页表项表），则会锁定父进程的页表项表。因此，在父进程更新页表项时，该页表项要么已在更新前被子进程复制，要么尚未被复制。对于第一种情况，子进程和父进程各自拥有自己的页表项，因此 Async-fork 不存在数据泄漏问题。对于第二种情况，由于父进程的页表项表处于锁定状态，子进程会在更新后复制该页表项。我们使用表 2 中的示例来说明这种情况。在页面迁移期间（步骤 2 至步骤 4），父进程的页表项表处于锁定状态。步骤 4 之后，子进程即可访问该页表项。随后，子进程会复制更新后的页表项。因此，Async-fork 不会出现数据泄漏问题。

尽管如此，在多个进程之间共享页表并使用写时复制来保持一致性的思路很有前景，因为它可以减少复制页表的工作量。解决上述所有问题并开发一种基于共享页表设计的 fork 方法，是一项有趣且具有挑战性的未来研究。

B 实现细节

我们将操作系统中会修改虚拟内存区域/页表项的操作归纳为检查点。通过对 Linux 内核进行全面分析，表 3 列出了能够修改虚拟内存区域或页表项的函数（即检查点）。如表所示，有些操作由用户查询触发，另一些则由操作系统固有的内存管理操作触发。不同内核版本的调用路径相当稳定。我们的分析表明，从 Linux 内核 2.6 到 6.0，只有一个与 Async-fork 相关的函数曾被修改（见表 4）。因此，与我们的工作相关的内核非常稳定。

表 3：可能修改虚拟内存区域/页表项的操作。

虚拟内存区域范围	描述	检查点		页中间目录范围	描述	检查点
mmap(2),munmap(2), madvise(2),mprotect(2), mlock(2),munlock(2), mremap(2)	用户使用系统调用主动地，导致虚拟内存区域的属性被修改，或虚拟内存区域被删除、拆分、扩展或合并。	虚拟内存区域_合并( )，__拆分_虚拟内存区域( )，分离_待_取消映射_的_虚拟内存区域( )，建议_虚拟内存区域( )，执行_内存保护_密钥( )，内存锁定_修正( )，虚拟内存区域_调整_大小( )		缺页异常	一个虚拟地址被首次访问时，操作系统会分配一个与之映射的页。	__处理_内存管理_故障( )
扩展_栈( )	用户/操作系统扩展栈的地址空间。	向上扩展_( )，向下扩展_( )，		内存不足	OOM 杀手回收页。	*zap_pmd_范围( )*
NUMA 平衡	操作系统会定期迁移非一致性内存访问节点之间的页。页表项会被修改，以无法访问。	更改_*prot_numa( )*		获取用户页	直接 I/O、VFIO 固定获取页并修改它们。	后续_页_页表项( )

表 4：相关函数的生命周期和位置。

函数	生命周期	位置
虚拟内存区域_合并()	v2.6.12 - v6.0	mm/mmap.c
__拆分_虚拟内存区域( )	v2.6.33 - v6.0	mm/mmap.c
分离_虚拟内存区域_以_供_取消映射( )	v2.6.12 - v6.0	mm/mmap.c
madvise_虚拟内存区域( )	v2.6.12 - v5.16.20	mm/madvise.c
do_mprotect_pkey( )	v4.9 - v6.0	mm/mprotect.c
mlock_fixup( )	v2.6.12 - v6.0	mm/mblock.c
虚拟内存区域_to_resize( )	v2.6.33 - v6.0	mm/mremap.c
向上扩展_( )	v2.6.15 - v6.0	mm/mmap.c
向下扩展_( )	v2.6.23 - v6.0	mm/mmap.c
change_prot_非一致性内存访问( )	v3.8 - v6.0	mm/mempolicy.c
__handle_mm_故障( )	v3.12 - v6.0	mm/memory.c
zap_页中间目录_range( )	v2.6.12 - v6.0	mm/memory.c
跟随_页_页表项( )	v3.16 - v6.0	mm/gup.c

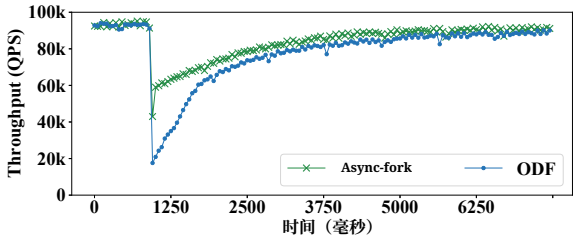


图17：16GB Redis 服务器在快照过程中吞吐量的变化。

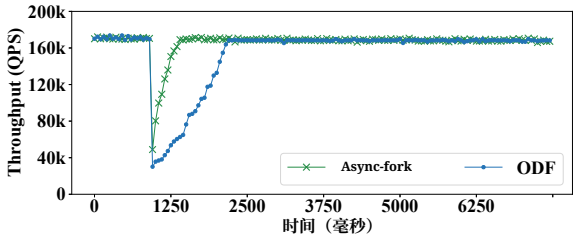


图18：16GB KeyDB 服务器在快照过程中吞吐量的变化。

实际上，无论对云服务提供商还是用户而言，更新内核都是一项关键任务。为了保持服务稳定性，他们往往会长期使用同一内核版本。因此，即使对内核的修改无法合并到上游，在生产环境中，云服务提供商为重要应用程序或特定内部需求定制操作系统内核也很常见。例如，Facebook 在内核中使用定制的磁盘缓存组件[6] 来加速 MySQL，而 Google 则在其内核中为内存不足终止程序添加了特定规则[7] 。Async-fork 的实现采用模块化设计，可以轻松地插入内核或从内核中移除。

C 补充实验

本节首先报告快照过程中父进程的查询处理吞吐量和总停服时间的实验结果。随后报告在日志重写中使用 Async-fork 的实验结果。此外，我们还报告 Async-fork 和 ODF 的执行时间。最后，讨论调整内存键值存储系统的方法，以进一步提升性能。

吞吐量结果。本实验测量内存键值存储系统对内存数据进行快照时的处理吞吐量。为此，我们使用 Redis 基准测试工具 [41] 发起大量查询，并将服务器端完成的查询数量作为处理吞吐量进行报告。

图 17 和图 18 分别展示了 16GB Redis 实例和 16GB KeyDB 实例的处理吞吐量。吞吐量每 50ms 报告一次。由于 KeyDB 使用多个线程处理查询，我们按以下方式测量 KeyDB 的吞吐量：1）每个线程独立测量每 50ms 的吞吐量，并记录带有时间戳的数值；2）对于每个时间间隔（50ms），将时间戳落在该间隔内的所有线程的数值相加，以估算引擎的吞吐量。

如图所示，执行快照时吞吐量会迅速下降，之后则逐渐回升。例如，在最坏情况下，使用 ODF 和 Async-fork 时，Redis 服务器的吞吐量分别降至每秒 17,592 次查询（QPS）和 42,980 QPS。此外，与 ODF 相比，使用 Async-fork 时吞吐量能更快恢复至正常水平。在我们的实验中，KeyDB 使用四个线程处理查询，因此其吞吐量高于 Redis。

在其他 Redis 和 KeyDB 实例规模下进行的实验也显示出类似结果。图19进一步展示了不同实例规模下 Redis 和 KeyDB 的最低处理吞吐量。与 ODF 相比，Async-fork 使 Redis 的最低吞吐量平均提高了 2.44 倍（最高可达 2.9 倍）。KeyDB 的最低吞吐量平均提高了 1.6 倍（最高可达 2.69 倍）。当负载超过服务器的处理能力时，一些查询会排队等待，并遭受较长延迟。处理吞吐量越低，延迟就越长，这与第 6.2 节的结果一致。

总停服时间。 如第 6.2 节所述，在调用 *copy_pmd_range()*期间，父进程无法处理查询。我们报告总停服时间（每次调用 *copy_pmd_range()* 的持续时间之和），如图

[6]https://engineering.fb.com/2013/10/09/core-data/flashcache-at-facebook-from-2010-to-2013-and-beyond。最后访问日期：2022/11/13。 [7] https://lwn.net/Articles/871195。最后访问日期：2022/11/13。

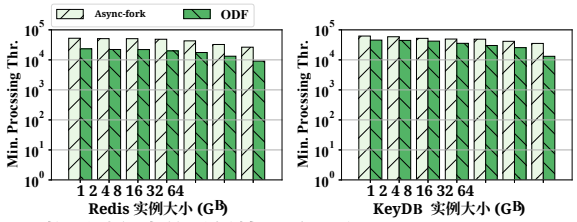


图19：快照过程中的最低处理吞吐量。

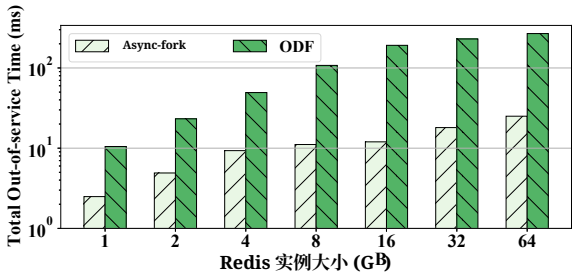


图20：Redis 服务器的总停服时间。

20 所示。与 Async-fork 相比，采用 ODF 时父进程停服时间更长。

Async-fork 对日志重写查询的影响。除了创建快照之外，Redis 还可以通过仅追加文件（AOF）持久化数据。Redis 会将服务器收到的每个写操作记录到文件中；在服务器启动时，可以重新执行这些操作，以重建原始数据集。然而，随着写操作不断执行，AOF 日志文件会越来越大。因此，Redis 会执行日志重写来优化 AOF 日志文件 [45]。具体来说，存储引擎调用*fork* 来创建子进程。子进程会将 AOF 文件重建为一组最短的命令序列，用于重建内存中的当前数据集；与此同时，父进程继续处理用户的查询。由于使用了*fork*，日志重写过程中也会出现延迟尖峰。

在实验中，我们使用 Redis 基准测试工具 [41] 生成负载，并执行 BGREWRITEAOF 命令，以触发 Redis 中的日志重写操作。我们记录日志重写期间（从 fork 操作开始到日志重写结束）到达的查询的延迟。为简洁起见，我们将这些查询称为日志重写查询 。

图21展示了采用不同 fork 方法时日志重写查询的第99百分位延迟和最大延迟。与默认的 *fork* 相比，在1GB 实例上，Async-fork 将日志重写查询的第99百分位延迟降低了71.81%（从11.53ms降至3.25ms）；在8GB 实例上降低了90.29%（从84.03ms降至8.16ms）；在64GB 实例上降低了97.66%（从1093.35ms降至25.59ms）。与 ODF 相比，在1GB 实例上，Async-fork 将日志重写查询的第99百分位延迟降低了39.7%（从5.39ms降至3.25ms）；在8GB 实例上降低了43.92%（从14.55msto 8.16ms）；在64GB 实例上降低了71.09%（从88.51ms降至25.59ms）。日志重写查询的延迟高于快照查询，这是因为启用 AOF 后 Redis 的整体性能会下降。具体而言，启用 AOF 的存储引擎除了处理查询外，还需要将日志同步到磁盘 [45] 。我们同样观察到了普通查询的性能下降。当 AOF

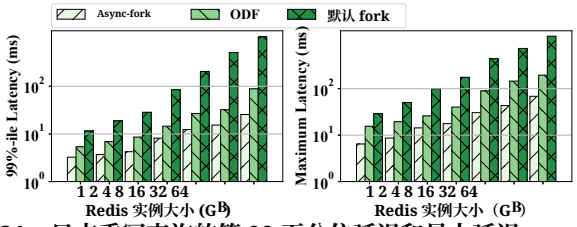


图 21：日志重写查询的第 99 百分位延迟和最大延迟在 Redis 中。

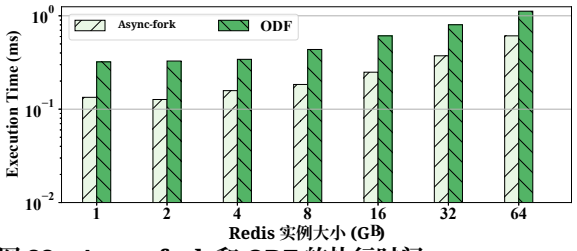


图 22：Async-fork 和 ODF 的执行时间。

被禁用时，16GB 实例上普通查询的第99百分位延迟和最大延迟分别为0.079ms和5.78ms。相比之下，启用 AOF 后，这两个值分别增至1.56ms和11.3ms。然而，如图21所示，Async-fork 显著降低了日志重写查询的延迟。结果证明了 Async-fork 的有效性。

fork 的执行时间。 fork 操作的持续时间会极大地影响快照查询的延迟。图 22 展示了在不同实例规模下，父进程调用 Async-fork 和 ODF 后返回所用的时间。与默认的*fork*（图 3）相比，Async-fork 和 ODF 都能有效缩短调用 fork 操作的执行时间。在大型的 64GB 实例中，父进程完成 Async-fork 需要 0.61ms，完成 ODF 则需要 1.1ms。时间之所以缩短，是因为它们移除了默认*fork*中最耗时的阶段。我们还可以观察到，Async-fork 略快于 ODF。一个可能的原因是，ODF 引入了逐页计数器以支持写时复制共享，因此需要额外的初始化时间。

调优内存键值存储系统。 在我们的实验中，Redis 和 KeyDB 均采用默认设置。调优内存键值存储系统的超参数或许能进一步降低快照查询的延迟。下文将讨论这一问题，但将其留待未来研究。

通过减少页表项修改，可以进一步减少 Async-fork 中的主动页表项复制。通过适当配置内存分配器，可以减少页表项修改。一般而言，为存储用户数据和元信息分配足够大的虚拟内存空间，可以大幅减少运行时所需的*mmap* 操作。此外，最好保留未使用的虚拟内存以供后续使用，而不是调用*munmap*将其释放。*mmap* 和*munmap* 操作都会导致页表项修改。Redis 和 KeyDB 的默认内存分配器*jemalloc*[8] 支持上述配置：1）通过*zmalloc/zfree* 预分配虚拟内存空间，但不填充其中的数据（因此不会额外消耗物理内存）。2）将内置属性‘retain’设为 true。

[8]https://github.com/jemalloc/jemalloc。最后访问于 2022/11/ 13.