arXiv:2603.05162v1 [cs.DB] 5 Mar 2026

R E SY S TA N C E：借助 eBPF 释放 LSM 树压实的隐藏性能

Hongsu Byun*, Seungjae Lee*, Honghyeon Yoo, Myoungjoon Kim, Sungyong Park†西江大学，首尔，大韩民国{byhs, ctaaone, yhh, myoungjoon,parksy}@sogang.ac.kr

摘要—**随着 NVMe 固态硬盘等高速存储设备的发展，主要的输入/输出瓶颈已从硬件转移到软件。现代数据库系统也依赖基于内核的输入/输出路径，频繁调用系统调用以及内核与用户空间切换会带来相当大的开销并导致性能下降。这个问题在基于日志结构合并树（LSM树）的 NoSQL 数据库中尤为突出。我们发现，后台压缩过程尤其会产生大量读取系统调用，从而造成显著开销。为解决这一问题，我们提出了 RESYSTANCE，它利用 eBPF 和 io_uring 使压实摆脱系统调用，从而释放潜在性能。RESYSTANCE 借助 io_uring 提高读操作期间的磁盘 I/O 效率，并通过 eBPF 在内核中直接处理压实，大幅降低软件栈开销。此外，RESYSTANCE 将关键 I/O 例程卸载到内核中，在不修改 LSM 树结构或压缩算法的情况下，最大限度地减少用户与内核切换。我们使用 db bench、YCSB 和 OLTP 工作负载对 RESYSTANCE 进行了全面评估。与基线 RocksDB 相比，它将压缩期间的系统调用平均次数减少了 99%，并将压缩时间缩短了 50%。因此，在写密集型工作负载中，RESYSTANCE 将吞吐量最多提升了 75%，并将 p99 延迟降低了 40%。**

索引词—**eBPF、键值存储、日志结构合并树**

I. 引言

I/O 瓶颈的转变。 NVMe 固态硬盘等超高速存储设备的出现 [1]–[3] 从根本上改变了系统性能瓶颈的性质。过去，有限的磁盘带宽和较高的延迟是主要制约因素；而如今，现代存储设备可提供数 GB/s 的带宽和微秒级延迟 [4]–[6]。因此，I/O 操作的性能上限不再取决于硬件层。I/O 瓶颈由此从硬件转向软件 [7]。每个 I/O 请求都需要多次系统调用，并频繁在内核态与用户空间之间进行上下文切换，浪费 CPU 周期，也妨碍高性能设备得到充分利用。这一转变不仅影响内核级操作，也影响依赖这些操作的数据密集型应用。数据库系统也不例外：由于缓冲区管理、日志记录和文件访问涉及多层 I/O 路径，它们尤其容易受到软件引入的开销影响。因此，数据库性能的主要决定因素已从硬件限制转变为操作系统与数据库管理系统边界产生的系统调用开销。

*这些作者为共同第一作者，贡献相同。†S. Park 为通讯作者。

LSM 树中的系统调用瓶颈。 这种由软件引起的开销在各种数据库中普遍存在，但在 NoSQL 系统中尤为明显。RocksDB [8], 、LevelDB [9],和 Cassandra [10]等代表性示例均采用日志结构合并树（LSM树） [11] 作为其存储引擎。LSM 树架构通过将写入缓冲在内存中，并定期批量刷新到磁盘，同时在后台持续合并（压实）数据，从而最大限度地提高写入效率 [12], [13]。尽管这种设计能够提供较高的写入吞吐量，但刷新和压实操作会触发大量文件访问，进而导致频繁的系统调用。

我们的性能分析表明，压实在 LSM 树引擎的系统调用总数中占据主导，其中`pread()` 占大多数。这是因为压实需要对反复读取并合并的数千个 SSTable 数据块逐一执行经由内核的输入/输出。因此，压实成为基于 LSM 树的数据库 I/O 路径中最严重的系统调用瓶颈。详细分析见第II节。

内核旁路的约束。 已有多项工作致力于缓解此类内核开销 [14]–[19]。具体而言，SPDK（存储性能开发套件）等内核旁路框架 [17]通过允许用户空间应用程序直接访问 NVMe 设备来消除系统调用开销，从而实现极高的 I/O并行度。然而，这种方法对数据库系统存在显著的实际限制。由于 SPDK 旁路了操作系统文件系统，采用它需要放弃传统的基于文件的数据管理，并重新设计整个 I/O 子系统。如此彻底的架构变更会带来巨大的实现复杂度、维护成本，以及与标准操作系统接口的不兼容。因此，尽管内核旁路在理论上高效，但对于必须保留现有架构的成熟数据库系统而言，它仍然不切实际。

eBPF在LSM树压实中的论证。 为了在避免内核旁路弊端的同时解决这些瓶颈，我们将目光投向扩展伯克利数据包过滤器（eBPF） [20]。作为一种内核内可编程框架，eBPF适合用于优化LSM树压实，原因有三。第一，安全性：eBPF的验证器可确保注入的逻辑不会使内核崩溃，从而保持数据库的可靠性。第二，效率：它允许在内核上下文中直接执行用户定义逻辑，消除上下文切换和数据复制。第三，实用性：它能够实现动态加速，而无需修改-

内核源代码或底层文件系统，从而为性能优化提供一条非侵入式路径。

挑战。 然而，使用 eBPF 设计压实面临以下挑战：

• eBPF 对指令数、循环结构和栈使用量施加了严格限制。在这些限制下设计压缩逻辑极其复杂。

• eBPF 中的压实会与外部数据和内核组件交互。由于验证器只检查 eBPF 内部逻辑，因此无法完全防止此类交互可能引发的无限循环或内核崩溃。

•该解决方案必须确保内核可维护性，并保证在各种基于 LSM 树的系统间具备可移植性，同时避免对内核进行侵入式修改，或针对特定系统对 LSM 树引擎进行深度重构。

RESYSTANCE：基于 eBPF 的压缩系统调用解放。 为应对这些挑战，我们提出了 RESYSTANCE，这是一个基于 eBPF 的框架，可使 LSM 树压实摆脱系统调用开销。其名称“Resystance”灵感源自“Resistance”，体现了其抵御系统调用带来的开销、使压实摆脱对系统调用的依赖这一目标。RESYSTANCE 结合 eBPF [20]和 io uring [21]，在不修改 RocksDB 等现有 LSM 树引擎的结构或压实算法，也不需要额外的硬件加速器或非易失性内存的情况下，将压实的核心 I/O 例程迁移至内核。通过最大限度地减少用户态与内核态边界切换，它消除了数据路径上不必要的系统调用，并纯粹通过软件栈优化提升性能。

据我们所知，RESYSTANCE 是首个仅通过消除软件栈开销来实现这一目标的系统。RESYSTANCE 摆脱了 LSM树压实对系统调用的依赖，并提升了数据库 I/O 效率。

贡献。 本文的贡献总结如下：

• 我们发现，压实是基于 LSM树的 NoSQL 数据库中系统调用开销的主要来源。

•我们提出了 RESYSTANCE，它通过集成 eBPF 和 io_uring，将压实的核心 I/O 逻辑卸载到内核中。

•RESYSTANCE 显著降低了系统调用开销并提高了磁盘 I/O 效率，且无需进行任何硬件修改或重新设计 LSM树。

RESYSTANCE 在一个具有代表性的基于 LSM树的键值存储 RocksDB v10.1.3 上实现，并进行了广泛的使用 db bench、YCSB 基准测试和 OLTP 工作负载进行了评估。与基线 RocksDB 相比，该方案将压实期间的平均系统调用次数减少了 99%，并将压实延迟缩短了 50%。具体而言，在 db bench 的 FillRandom 工作负载中，RESYSTANCE 将吞吐量提升了 75%，并使 p99 延迟降低了 40%。此外，在混合读写 OLTP 工作负载中，RESYSTANCE 的吞吐量比基线 RocksDB 高出 22%。

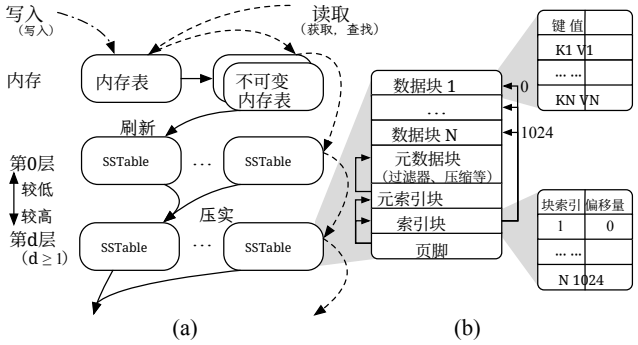


图1：LSM树的架构。

II. 背景与问题定义

本节介绍日志结构合并树（LSM 树）的背景，并分析当今基于 LSM 树的 NoSQL 数据库中观察到的性能瓶颈和局限性。

A. 日志结构合并树

图1（a）展示了LSM树的结构。LSM树将键值写入缓冲在称为内存表的内存结构中，并定期执行刷新，以将数据持久化为磁盘结构SSTable。这种批处理效果将频繁的随机写入转换为大块顺序写入，从而提供较高的写入吞吐量。新刷新的SSTable首先写入第0层；当第0层中的SSTable数量超过预定义阈值时，压实过程会合并并排序它们，并在下一层生成新的SSTable。每一层都维护各自的压缩阈值，且各层容量随层号呈指数增长。

一次读取操作首先搜索内存表，然后从较低层到较高层顺序扫描SSTable。当冗余或未排序的键不断累积时，查找时间会增加；因此，压实——它会去除重复项并重新组织键范围——对于维持快速读取至关重要。当压实被延迟时，读取必须遍历大量SSTable，导致性能急剧下降。因此，LSM树无法在压实积压期间维持读取性能，前台写入常常被暂停以防止进一步恶化，从而导致写入停顿 [22]–[26]。

压实性能的重要性。压实会影响 LSM 树的写入和读取性能，因此是决定系统整体效率的关键因素。因此，提升压缩性能一直是研究的核心课题，研究方向包括使用 FPGA、GPU 或 DPU 设备进行硬件加速、算法优化以及制定新的压实策略。第七节将详细回顾这些研究。

B. 软件栈的瓶颈

近期研究 [7] 表明，即使是简单的文件I/O调用，例如 `read()` 或`write()` 在现代 NVMe 固态硬盘上，其延迟也更多来自软件栈，而非硬件本身。随着固态硬盘的延迟显著降低、内部并行度大幅提高，内核级操作——数据复制、

表 I：LSM 树中 pread() 的延迟分解。

软件	文件系统	110.4 µs	61.2%
	块层	5.9 µs	3.8%
	NVMe 驱动程序	6.3 µs	3.5%
硬件	存储设备	56.8 µs	31.5%
	总计	180.4 µs	100%

表 II：RocksDB 中每次操作的平均系统调用次数。

	Put	Get	Seek	Next	刷新	压实
系统调用/操作	0	0.5	0.5	0.01	107	66,983

表 III：压实期间的系统调用分布。

	写入	fsync	pread	unlink	其他
比例（%）	0.38	0.005	96.09	0.01	3.52

缓冲区管理和上下文切换——在总 I/O 延迟中占据主导地位。换言之，瓶颈已从速度缓慢的存储硬件转移到操作系统的软件层及其系统调用接口。

这一观察结果可直接推广到数据库系统。表I 展示了 RocksDB（一种具有代表性的、基于 LSM树 的键值存储）中一次`pread()` 在一次获取操作期间的延迟分解。软件栈占总延迟的 68.5%。用户查询和刷新、压实等内部操作都严重依赖内核介导的系统调用。因此，现代数据库的瓶颈不再是存储硬件，而是内核 I/O 路径中系统调用带来的软件开销。

C. LSM 树系统调用分析

我们使用 db bench，在 FillRandom 和 MixGraph 工作负载下，定量分析 LSM树 中哪些操作会产生最多的系统调用。MixGraph 工作负载 [27]，基于 Facebook 的 RocksDB 使用模式建模，采用默认比例：83%获取、14%写入和 13%查找。实验详情见第VI节。表II 比较了每次操作的平均系统调用次数，结果显示压实占据绝对主导地位。这是因为压实会反复读取和写入大量 SSTable 文件。因此，压实不仅是数据重组任务，也是 LSM 树引擎中系统调用最密集的操作。

为进一步理解这一行为，我们考察压实期间的系统调用构成。如表III所示，`pread()` 占据了绝大多数调用。这是由于其采用基于块的设计，SSTable 按块读取，键值对则经过合并并重新写入。图2 展示了压实的详细操作：压实迭代器以归并排序的方式，反复读取（`ReadNextKV`）和写入（`WriteKV`）键值对。对于每个所需的数据块，一个

发出 pread() 调用。假设一个

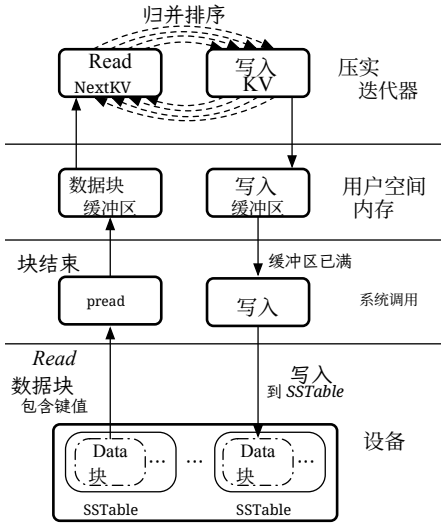


图 2：LSM 树中基于块的压实操作。

64MB 的 SSTable 包含 4KB 数据块，则每个 SSTable 包含2^{14} 个数据块，从而导致大量 pread() 调用。因此，这种基于数据块的结构是软件栈开销的主要来源。

D. LSM 树为何采用基于块的 SSTable

LSM 树为何采用基于块的 SSTable 格式？原因在于兼顾写入和读取效率。

在写入方面，LSM 树会将随机写入转化为顺序写入，以最大限度地提高磁盘效率。如图 1(b) 所示，SSTable 被组织成多个数据块。在刷新或压实时，数据会沿连续区域顺序写入，从而有效利用文件系统和固态硬盘的内部缓冲区及并行通道，提高吞吐量并降低写放大。因此，基于块的存储实现了 LSM 树旨在利用的“大块顺序写入”特性。

在读取端，基于数据块的结构同样至关重要。SSTable 由元数据块和数据块组成；索引块 存储每个数据块的偏移量、大小和键范围。查找操作首先查询索引块，然后仅访问相应的数据块，从而减少不必要的输入/输出并提高效率。这种多级查找减少了元数据大小，并提升了缓存效率。因此，基于数据块的 SSTable 既为写入提供了顺序化，也为读取提供了局部性，构成了 LSM树性能的结构基础。

III. 减少系统调用的机会

本节介绍能够减少压实过程中系统调用的框架，并解释现有内核旁路方法的限制。

A. 扩展伯克利数据包过滤器

eBPF（扩展伯克利数据包过滤器）通过将经过验证的字节码程序动态附加到各种挂钩点（例如内核探针/跟踪点、用户空间探针、XDP 等），使其能够在 Linux 内核中执行。每个程序

均由内核验证器静态验证其内存安全、保证终止以及辅助函数调用限制，随后经即时编译，以事件驱动的方式在所附加的挂钩点执行。借助 eBPF 映射（共享的键值状态）和辅助函数，eBPF 已被广泛应用于网络（数据包过滤和负载均衡）、可观测性（性能分析和跟踪）以及安全（策略执行和系统调用监控）。BTF 和 CO-RE（一次编译，到处运行）等特性使单一 eBPF 字节码能够在不同内核版本上无缝运行。

eBPF 允许用户定义的逻辑在内核级执行。具体而言，通过将选定的应用功能从 I/O 路径卸载到 eBPF 程序中，可在单次内核调用内实现多种系统调用效果。此外，数据可在传递到用户空间之前于内核中进行预处理，从而降低用户内存需求并提高工作集效率。该方法消除或摊销了上下文切换以及用户态–内核态边界的跨越，在改善缓存局部性的同时降低 TLB 压力和内存带宽浪费。尽管这些收益必须在验证器约束和内核稳定性考量下运作，eBPF 仍为优化 I/O 路径提供了切实的性能机会。

B. io_uring

io_uring 是一种现代异步 I/O 接口，它通过 mmap 在用户空间和内核空间之间共享提交环和完成环（提交队列/完成队列），从而消除每次 I/O 的系统调用，并通过批量操作实现系统调用摊销。应用程序将读取、写入、轮询和 fsync 等操作填入提交队列条目（SQE），而内核则通过完成队列条目（CQE）异步返回结果。凭借共享内存以及对丰富异步原语的支持，io_uring 将上下文切换开销和数据复制开销降至最低，提供一种高效的执行模型，使多次 I/O 操作可通过单次内核入口提交并完成。

C. 内核旁路的约束与局限

同样，XRP 等方法在存储驱动程序层提供可编程的快速路径，以便在驱动程序内部处理或转换请求。这类方法虽能缩短系统调用路径，却绕过了虚拟文件系统和块层。因此，它们无法充分利用内核的功能，例如 I/O 调度、回写和预读。再加上文件系统特性的限制，实现复杂度大幅增加。在涉及大量文件和目录的工作负载中（例如，分布在多个文件中的大量小型 I/O 操作，或逐文件策略），确保一致性并控制共享资源尤其困难。归根结底，依赖内核旁路或省略块层/文件系统层的现有方法，难以兼顾通用性、安全性和运行可行性。因此，我们需要一种新方法：既保留内核层的优势，又降低用户态–内核态边界的开销。

IV. 设计目标与挑战

本节介绍使用 eBPF 和 io_uring 加速压实的设计目标与挑战。

A. 目标

目标 #1：通过削减系统调用来降低开销。传统的基于迭代器的压实在每次读取数据块时都会发起一次 pread() 系统调用，导致系统调用次数激增，并产生巨大的上下文切换开销。RESYSTANCE 利用 eBPF 将压实的核心输入/输出和归并排序阶段卸载到内核中，从而减少用户态与内核态切换，并缓解系统调用引起的开销。

目标 #2：借助内核内异步 I/O 提高磁盘 I/O 效率。 在传统路径中，每次访问数据块都要通过 pread() 调用穿过内核才能到达存储设备，这实际上使每次调用的 I/O 串行化，迫使每个数据块读取都采用同步输入/输出。相比之下，由于 RESYSTANCE 在内核中执行压实，因此可以使用 io_uring 异步并行地请求多个数据块。这样既减少了进入内核的次数，也最大限度地发挥了高性能 NVMe 固态硬盘固有的并行能力和带宽。

目标 #3：实现动态压实算法。传统压实通常采用固定算法（例如基于最小堆的归并），难以适应工作负载特征或设备状况。RESYSTANCE 通过允许在用户空间定义压实算法本身，并使用 eBPF 将其注入内核，消除了这一限制。因此，系统可以在运行时替换压缩逻辑——无需重启数据库或重构内核——并动态采用最合适的归并策略，以提升效率和适应性。

目标 #4：尽量减少内核修改。过多的内核修改可能损害稳定性，并阻碍实际部署。因此，RESYSTANCE 尽量减少内核变更，以提升在不同环境中的可移植性。为尽量减少应用程序与操作系统接口之间的干扰，RESYSTANCE 设计为通过现有系统调用运行，而非引入新的系统调用。此外，为降低对原有内核控制流的影响，本工作引入的附加功能由调用频率较低的顶层函数调用，从而确保修改保持最少且局部化。

B. 挑战

挑战 #1：eBPF 性能限制与验证器约束。尽管 RESYSTANCE 通过 eBPF 将部分压实逻辑移入内核，但出于安全性考虑，eBPF 对指令数、循环结构、栈使用量和调用深度都施加了严格限制。这些约束使得仅用 eBPF 实现归并排序或迭代器等操作变得困难，而且复杂度越高，验证器拒绝的可能性就越大。此外，由于映射访问和辅助函数调用，eBPF 的性能往往不如原生内核代码 [28], [29]，这可能成为输入/输出与排序密集型压实的瓶颈。

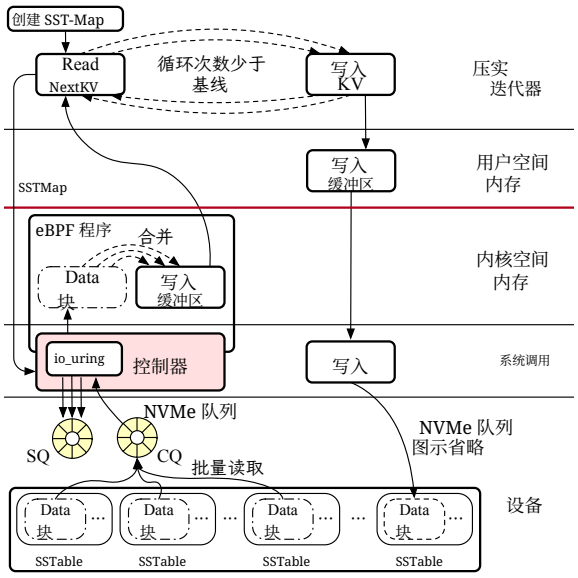


图 3：RESYSTANCE 架构概览。

挑战 #2：内核与 eBPF 逻辑之间的功能划分。鉴于这些约束，RESYSTANCE 必须谨慎地划分内核代码与 eBPF 程序之间的职责。给 eBPF 分配过多任务会增加验证器失败和系统不稳定的风险；而对 eBPF 规定不足，则会使其无法充分表达所需的压缩行为。因此，RESYSTANCE 需要明确划分边界并制定划分策略，在遵守验证器限制的同时保留足够的灵活性。

挑战 #3：内核稳定性。虽然 eBPF 验证器可以防止 eBPF 程序中的无限循环和无效内存访问等问题，但它不会验证与外部内核代码的交互。如果用户提供的逻辑与内核资源交互不当，可能导致无限循环或内核崩溃。为保障整个系统的稳定性，RESYS- TANCE 必须限制资源访问，并纳入故障隔离和恢复机制。

V. RESYSTANCE 设计

本节介绍 RESYSTANCE 的架构。我们首先概述其整体设计，然后介绍执行工作流程。

A. 概述

核心思想。图3 展示了 RESYS- TANCE 的分层架构，它由三个组件组成：控制器、io_uring 和 eBPF 程序。其核心思想是通过以下方式减少压缩迭代器的调用次数：(i) 在一次 ReadNextKV 调用中通过 io_uring 发起多个读 I/O；(ii) 在附加的 eBPF 程序中执行归并排序。由于使用了 io_uring，RESYSTANCE 可以通过传统的内核 VFS 和块层运行，从而保持通用性。

面向 eBPF 程序的 SST-Map。执行压实时，内核需要 I/O 描述符，例如目标 SSTable、块偏移量和块大小，但 eBPF 本身无法获取这些信息。因此，RESYSTANCE 构建了一个 *SST-Map*来传递此类元数据。SST-Map 构建于用户-内核共享内存中（消除了拷贝开销），且仅根据已加载到主存中的 SSTable 元数据生成，因此构建成本可以忽略不计。下一小节将进一步介绍 SST-Map。

使用 eBPF 和 io_uring 进行读取与合并。当压实迭代器调用 ReadNextKV 时，RESYS-TANCE 会通过 io_uring 对所需数据块发起异步读取。随后，挂接到相关事件上的 eBPF 程序会对读取的数据执行归并排序，并将结果追加到输出缓冲区。此时，eBPF 程序会读取 SSTable 中数据块的键值结构，因此无论键值对大小如何，都能执行处理。当 eBPF 程序耗尽一个数据块后，控制器会从 SST 映射中获取下一个数据块的元数据。为避免用户承担复杂的 eBPF 逻辑（这可能导致验证器拒绝和稳定性问题），控制器提供了针对“顺序读取后合并”模式优化的控制骨架。在 eBPF 中，程序只需检查逐块有效性以及继续压实所需的最少进度状态，从而减轻用户的实现负担，并最大限度地降低内核侵入风险。循环会持续进行，直到 (i) 输出缓冲区达到预设大小，或 (ii) 没有更多数据块可供读取。结果通过用户-内核共享内存返回，从而消除拷贝开销。

通过传统路径执行写入过程。此后，我们保留原有压实迭代器的 `WriteKV()` 来执行写入操作，并通过传统的用户空间路径写入数据块。如表 III 所示，`WriteKV()`通过 `write()`发起的系统调用远少于 `pread()`，且它还负责构建 SSTable 元数据。因此，在 eBPF 中重新实现它只能带来微乎其微的收益，却会增加验证器风险。

基于 eBPF 的压缩逻辑的设计选择。从以合并为中心的视角来看，将 eBPF 用于压实时有几种设计选择：(i) 使用 eBPF 执行读取、合并和写入；(ii) 执行读取和合并；(iii) 执行合并和写入；以及 (iv) 仅使用 eBPF 执行合并。在 RESYSTANCE 中，我们选择只将合并功能卸载到 eBPF 程序中，以确保内核安全性、尽量减少代码修改，并最大限度地兼容现有压缩逻辑，同时仍能受益于 eBPF。第 VIII 节将详细讨论这一设计选择。

B. 内核稳定性维护

为保障内核稳定性而修改 eBPF 验证器。为实现 RESYSTANCE 并保障内核稳定性，我们修改了 eBPF 验证器。首先，我们放宽了验证指令上限；默认验证器将该上限设为一百万。在 eBPF 中实现压实时，验证器会探索压缩逻辑中的所有可能合并状态，从而大幅增加验证指令数。不过，该上限旨在限制验证过程本身的复杂度和终止时间，而非限制 eBPF 程序的运行时或执行指令数。放宽指令数上限并

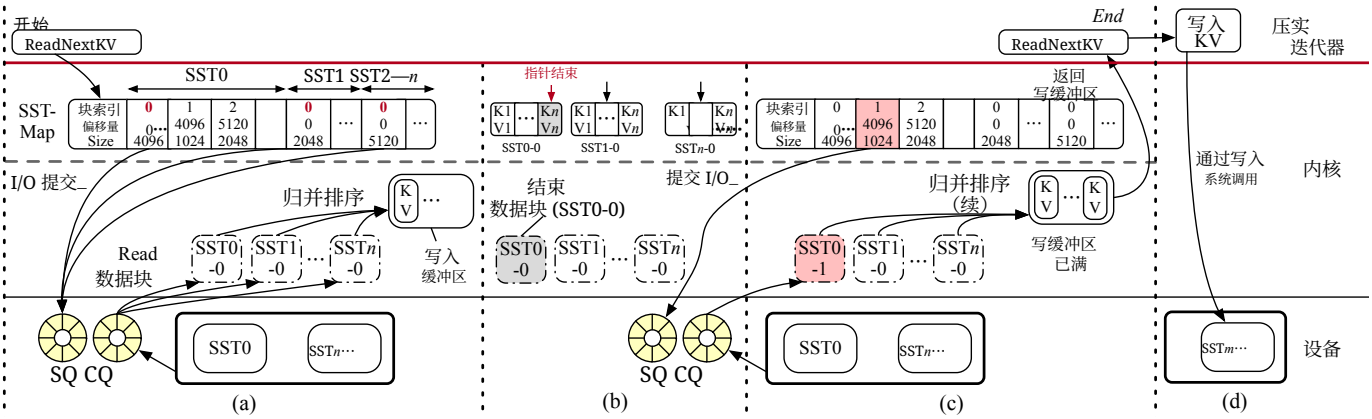


图 4：RESYSTANCE 控制器的详细操作。

不会削弱验证器的核心安全检查。验证器仍会分析循环结构是否能够终止，并拒绝存在潜在无限循环的程序。因此，提高这一上限可以扩大验证范围，同时不影响终止性保证。

其次，我们允许 eBPF 程序访问一个用于存储压实所需数据的内核内存区域，同时严格限制其访问预定义的地址范围。由于 eBPF 栈大小有限，无法提供足够的空间来以千字节为粒度执行归并操作。为防止任意访问内核内存，我们自定义了验证器的 is_valid_access 逻辑，仅允许访问 RESYSTANCE 明确指定的地址。由于验证器会检查所有可能的程序状态，因此任何控制流可能访问允许的内核内存区域之外的地址的 eBPF 程序都会被拒绝。这一设计消除了意外访问内核内存的可能性。

通过 SST-Map 实现确定性 IO。SST-Map 会按顺序且仅执行一次用户注册的 IO 操作。相比之下，数据追踪等非确定性执行模式中，IO 操作的顺序取决于运行时数据，因此无法保证在有限时间内完成。RESYSTANCE 采用简单模型，以固定顺序各执行一次 IO，因此只需跟踪 SST-Map 中哪些 IO 已完成、哪些仍待处理。文件描述符或数据偏移量上的 IO 错误由底层 io_uring 子系统安全处理。因此，RESYSTANCE 能够在内核中执行用户逻辑，同时保持内核稳定性，并精确控制 IO 的执行。

C. RESYSTANCE 的详细操作

ReadNextKV() 的开始`ReadNextKV()`。下面详细介绍一次 `ReadNextKV()` 调用。如图4(a) 所示，首先将 SST-Map 传递给 io_uring。对于每个目标 SSTable（SST0、SST1、.....），SST-Map 都包含待压实的各个数据块的索引、偏移量和大小。由于 eBPF 和 io_uring 均通过文件描述符进行管理，RESYSTANCE 可以为每个压缩线程创建独立实例，并将各线程的 SST-Map 存放在不同的空间中。

使用 SST-Map，控制器最初会读取每个目标 SSTable 的第一个数据块（SST0-0、SST1-0、...、SSTn-0）并通过 io_uring 将其加载到内核内存中。所有必需的数据块都可以一次性请求，以减少开销并提高并行度。随后，eBPF 程序会对内核中的数据块执行传统的迭代器式归并排序，并将结果写入内核侧写入缓冲区。在此阶段，RESYSTANCE 可以直接访问指定的内核缓冲区区域，而无需依赖辅助函数，从而消除执行期间辅助函数调用的开销。

数据块指针与块结束处理。由于 SST-Map 仅提供元数据，且控制器本身不执行压实，因此必须跟踪进度。eBPF 程序会为每个数据块维护一个读取指针，并与控制器共享该指针（图4(b)）。当数据块 SST0-0 的读取指针到达末尾时，控制器会查阅 SST-Map 以定位下一个数据块，并发起 io_uring 读取。通过批处理这些请求，可以并发处理多个已读完的数据块。

结束 `ReadNextKV()` 和 `WriteKV()`。RESYSTANCE通过以下方式迭代：(i) 经由控制器和 io_uring 发起输入/输出；(ii) 通过 eBPF 程序执行归并排序和追加操作。当 (a) 内核写缓冲区达到预定义阈值，或 (b) 所有 SSTable 均已完全归并且没有剩余数据块时，循环终止（图4(c)）。在这两种情况下，位于用户-内核共享内存中的写缓冲区都会返回用户空间，而 `ReadNextKV()` 会返回其大小。最后，`WriteKV()` 通过传统路径持久化数据（图 4(d)）。用户代码根据返回的大小，决定是否再次调用`ReadNextKV()` 。

D. 归并排序算法

RESYSTANCE 通过 eBPF 程序将压实归并排序移植到内核空间。该算法比较多个数据块中的键，以选出下一个键值对。用户可以提供自定义比较器或过滤逻辑，并通过 eBPF 在内核中执行。RocksDB 通常因其归并迭代器结构而采用最小堆，但 RESYSTANCE 不受这一选择的限制。由于文件数量较少时堆管理开销可能不可忽视，我们考虑了两种用于选取最小值的实现。**线性搜索**。 线性搜索在每次迭代时扫描所有有效数据块并比较其中的键。由于它涉及

较少的分支，因此在文件数量较少时效率较高；但其开销会随着文件数量和键长度的增加而成比例上升。伪代码见算法 1。

最小堆搜索。 最小堆搜索会在压实开始时，使用每个数据块的当前键初始化一个最小堆。每次迭代都会弹出最小的键，并将同一数据块中的下一个键压入堆中，同时记录后续调用中需要额外读取以便重新插入的数据块。专用的 BPF 映射会在 eBPF 调用之间保留堆状态。使用堆时，每次迭代的开销为 $O(K \log N)$，适用于 N 个文件和键大小为 K 的情况，能够在文件数量较大时快速完成选择；不过，当 N 较小时，维护堆的开销可能使其速度低于线性搜索。伪代码见算法2。

VI. 评估

A. 实现

RESYSTANCE 基于 Linux 内核 6.14 中的 io_uring 系统调用实现，仅对内核进行了少量修改，以添加 eBPF 调用的附着点。因此，RESYSTANCE 通过专用标志作为 io_uring 实例运行来执行压实，无需引入任何新的系统调用。eBPF 程序使用 `bpf_loop` 循环辅助函数，使验证器能够安全地验证复杂度适中的压实逻辑。在本研究中，选择搜索算法的输入文件数量阈值经实测设为六：当压实文件数小于或等于六时，使用线性搜索算法；否则，使用最小堆搜索。该算法的详细评估见评估章节 VI-D。

RocksDB [8]版本 10.1.3 被用作基线。我们修改了 compaction_job.cc 源文件（其中实现了压实作业），通过调用带有特殊标志的 io_uring_enter 系统调用来调用 RESYSTANCE，而不再使用 c_iter()。在同一函数中，io_uring_setup 系统调用用于初始化 io_uring 实例，而 mmap 系统调用则映射用户—内核共享内存，以便交换压实数据。每个压缩线程仅初始化一次，以最大限度地提高复用性并尽可能降低调用开销。每个压缩线程都维护一个独立的 io_uring 实例（即文件描述符），用于并行执行相互隔离的压实作业。

RESYSTANCE 的源代码可在以下地址公开获取：https://github.com/DISCOS-LAB/Resystance。

B. 实验设置

平台。所有实验均在一台服务器上进行，该服务器配备 AMD Ryzen 9 7950X @ 4.5 GHz CPU（已禁用同时多线程和 C 状态）、32 GB DDR5 动态随机存取存储器，以及一块 1 TB Samsung 990 PRO NVMe 固态硬盘。系统运行 Ubuntu 24.04.3、Linux 6.14 内核和 ext4 文件系统。

比较。 我们将 RESYSTANCE 与基线 RocksDB 进行比较，因为 RESYSTANCE 仅优化软件 I/O 栈，不会改变 LSM 树结构或压缩算法。因此，RocksDB 是评估 RESYSTANCE 有效性的公平且有意义的基线。此外，为了评估 eBPF 的执行开销和内核内的效率，

算法 1:
NextLinear(数据块，指针，写缓冲区)

```
输入  : 数组 数据块[1..n]; 数组 指针[1..n]; 写缓冲区
输出  : 写缓冲区大小
函数 NextLinear(数据块, 指针, 写缓冲区):
    有效 ← { i ∈ [1..n] | 数据块[i] 有效 }
    当 ∀i ∈ 有效项, 指针[i] <块大小[i] 执行
        索引 ← −1; 最佳键 ← NULL
        对每个 i ∈有效项 执行
            键 ←KeyAt (数据块[i], 指针[i])
            如果 索引 = −1 或 memcmp(键, 最佳键) < 0 则
                索引 ← i; 最佳键 ←键
        adv ←键值对大小获取(数据块[索引], 指针[索引])
        kv ←KVAt (数据块[索引], 指针[索引])
        指针[索引] ← 指针[索引] + adv
        追加(写缓冲区, kv)
    返回 |写缓冲区|
```

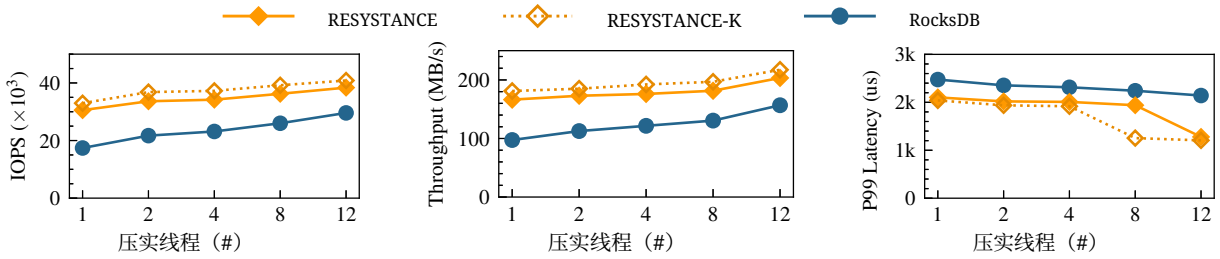
算法 2:
下一最小堆(数据块, 指针，写缓冲区，H)

```
输入  : 数组 数据块[1..n]; 数组 指针[1..n]; 写缓冲区; 最小堆 H
输出  : 写缓冲区大小
函数 下一最小堆(数据块, 指针, 写缓冲区, H):
    有效项 ← { i ∈ [1..n] | 数据块[i] 有效 }
    如果首次调用 则
        对每个 i ∈有效项 执行
            键 ←KeyAt (数据块[i], 指针[i])
            Push(H,(键 i))
    如果 前一个_Pop_块 ≠ 空值 那么
        j ←前一个_Pop_块
        前一个_Pop_块 ← 空值
        键 ←KeyAt (数据块[j], 指针[j])
        Push(H, (键 j))
    当 true 时
        (键, 索引) ← Pop (H)
        键值 ←KVAt (数据块[索引], 指针[索引])
        adv ←键值对大小获取(数据块[索引], 指针[索引])
        指针[索引] ← 指针[索引] + adv
        追加(写缓冲区, 键值)
        如果 指针[索引] < 块大小[索引] 则
            key' ←KeyAt (数据块[索引], 指针[索引])
            Push (H, (键', 索引))
        else
            前一个_Pop_块 ← 索引
            中断
    返回 |写缓冲区|
```

我们还纳入了 RESYSTANCE-K，这是 RESYSTANCE 的一个变体，它将 eBPF 代码完全集成到内核中，并直接在 io_uring 中执行压实。评估目标如下：

- **RocksDB:** 基线配置在压实期间通过内核路径使用传统系统调用。
- **RESYSTANCE:** 我们提出的系统使用 io_uring 和 eBPF，降低了压实期间的软件栈开销。
- **RESYSTANCE-K:** RESYS-TANCE 的内核集成变体，其中 eBPF 逻辑直接嵌入内核，压实完全在 io_uring 中执行。

RocksDB 参数配置。为隔离压实性能的影响，我们禁用预写日志、块缓存和压缩。压实和输入/输出请求均使用直接 I/O。压实策略遵循默认的分层策略，并禁用 L0 到 L0 的内部压实，以消除写入停顿缓解的影响。使用 Twomemta-



(a) I/O 吞吐量　　(b) 压实吞吐量　　(c) 延迟

图 5：FillRandom 工作负载下不同压实线程数量时的性能比较：总体每秒输入输出操作数、压实吞吐量和延迟。

bles，每个大小为 64 MB，并使用一个刷新线程。

基准测试与工作负载。 我们使用三个具有代表性的基准测试评估 RESYSTANCE：首先，使用 db bench– [30]进行合成工作负载分析。其次，我们使用 YCSB 基准测试 [31]评估键值存储环境中的真实读写访问模式。在 db_bench 和 YCSB 中，所有工作负载均使用 16 B 的键和 1 KB 的值。此外，我们还使用 Sysbench 的 OLTP 工作负载，在基于 RocksDB 的关系数据库管理系统（RDBMS）中评估 RESYSTANCE 的有效性。工作负载定义如下：

- **FillRandom：** db bench 中的 100% 写入工作负载。
- **ReadRandomWriteRandom：** db bench 中读写比例可变的混合读写工作负载，在通过 FillRandom 加载数据后执行。
- **ReadWhileWriting：** db bench 中各有一个 100% 读取线程和一个 100% 写入线程并发运行，并在 FillRandom 之后执行。
- **YCSB：** 表 IV中所述的标准 YCSB 工作负载。
- **联机事务处理：** 在基于 RocksDB 的关系数据库管理系统上使用 Sysbench [32]评估联机事务处理工作负载。详细的工作负载描述见表V。

在 FillRandom、ReadRandomWriteRandom 和 YCSB 中，每个输入/输出线程对 10M 个条目执行操作。在 ReadWhileWriting 和 OLTP 工作负载中，操作持续执行 1200 秒。

C. 整体性能

1）写密集型工作负载： 图5 展示了 FillRandom 工作负载在压实线程数量不同时的结果。如图5(a)所示，RESYSTANCE 和 RESYSTANCE-K 的每秒输入输出操作数均高于 RocksDB；在使用单个压实线程时，RESYSTANCE 的吞吐量最高比 RocksDB 高出 75%。

表IV：YCSB 工作负载特征说明。

Type	说明	分布
Load	100% 写入	齐夫分布
A	50% 更新，50% 读取	齐夫分布
B	95% 读取，5% 更新	齐夫分布
C	100% 读取	齐夫分布
D	95% 读取，5% 插入	最新
E	95% 查找，5% 插入；扫描长度：128	齐夫分布
F	50% 读取-修改-写入，50% 读取	齐夫分布

由于 eBPF 机制本身存在调用和执行开销，RESYSTANCE-K 的表现略优于 RESYSTANCE。图5(b)展示了压实吞吐量，其趋势与每秒输入输出操作数基本一致，这证实了吞吐量的提升源于更快的压实处理。平均而言，RESYSTANCE 将系统调用次数减少了 99% 以上，并将压实执行时间缩短了约 50%。图5(c)展示了 p99 写延迟，RESYSTANCE 和 RESYSTANCE-K 的表现均优于 RocksDB。使用 12 个压实线程时，RESYSTANCE 的 p99 延迟最高比 RocksDB 低 40%。在 8 个线程时，RESYSTANCE 与 RESYSTANCE-K 之间的差距更大，这是因为在写入停滞条件下，即使压实方面只有微小差异，也可能导致延迟显著变化。

2) 读写混合负载： 图 6(a)–(c) 展示了采用四个压实线程、改变 I/O 线程数量时 ReadRandomWriteRandom 工作负载的结果。第一行显示每秒输入输出操作数，第二行和第三行分别显示 P99 读延迟和写入延迟。

对于 R10/W90（图6(a)），吞吐量的变化趋势与 FillRandom 类似，但 RESYSTANCE 与 RESYSTANCE-K 之间的性能差距几乎消失，这表明压实频率降低时 eBPF 开销可以忽略不计。在 P99 读延迟方面，在吞吐量相近的情况下，RESYSTANCE 和 RESYSTANCE-K 的延迟均低于 RocksDB。随着吞吐量趋于饱和，延迟略有上升，但仍低于 RocksDB。RESYSTANCE 和 RESYSTANCE-K 的 P99 写入延迟显著低于 RocksDB（尤其是按吞吐量归一化后），这得益于它们有效缓解了写入停顿。

图6(b)展示了 R50/W50 的结果。随着写入比率下降，RESYSTANCE 与 RocksDB 之间的吞吐量差距缩小。在吞吐量较低时，由于更快的压实提高了磁盘访问并发度，RESYSTANCE 和 RESYSTANCE-K 的 P99读延迟略高于 RocksDB。然而，随着吞吐量增加，更快的压实能够稳定读取性能。p99 写延迟差异很小，因为写停顿在混合工作负载中的影响较小。

对于 R90/W10（图6(c)），吞吐量趋势与 R50/W50 类似，RESYSTANCE 在读取和写入方面均保持更高的吞吐量和更低的 p99 延迟，表明即使在读密集型工作负载下，高效压实也能带来益处——

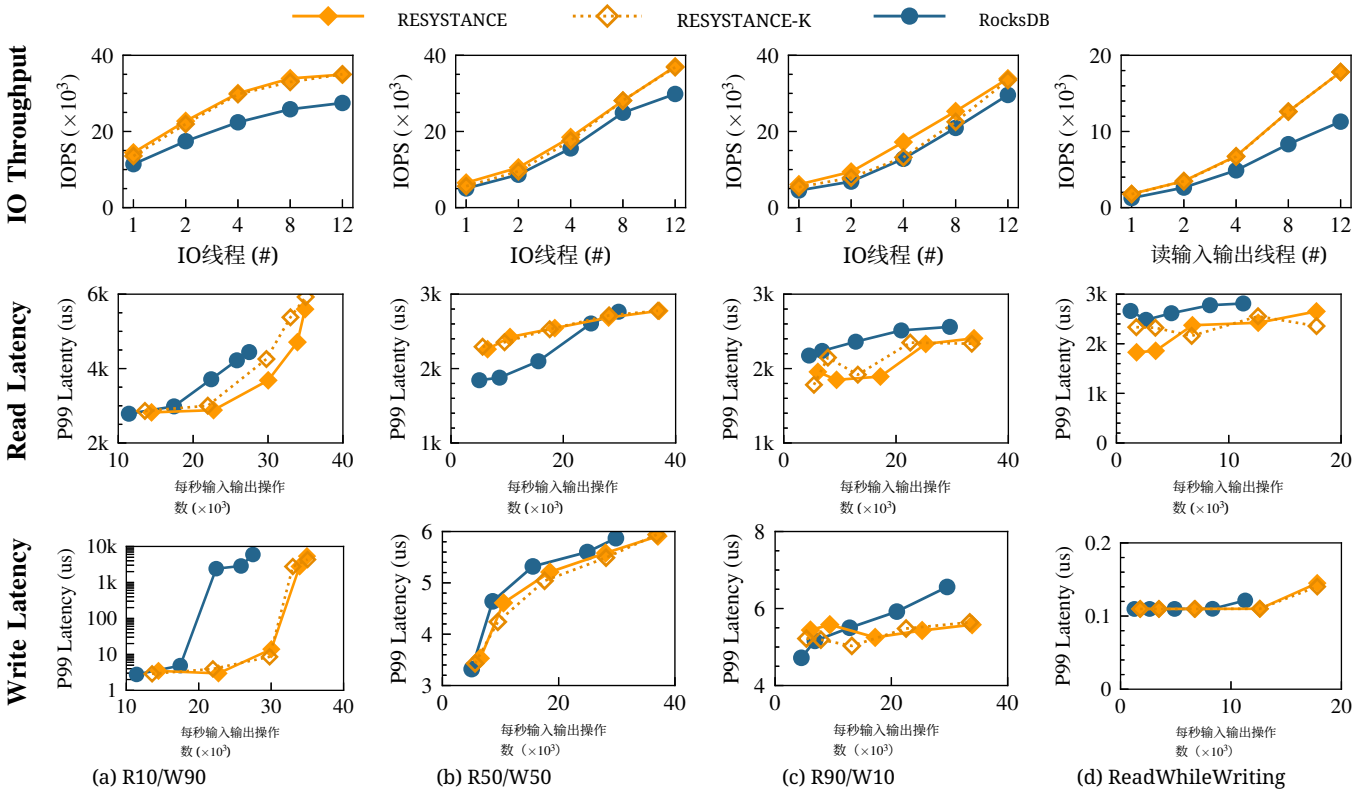


图 6：使用 4 个压实线程时，不同 IO线程数量下的性能对比。(a)–(c) ReadRandomWriteRandom 工作负载下读写比例的变化；(d) ReadWhileWriting 工作负载下读取 IO线程数量的增加。

密集型工作负载。

3) 并发写入下的读密集型工作负载：图6(d) 展示了 ReadWhileWriting 在使用四个压实线程时，随着读取 IO线程数量增加而呈现的性能变化。随着读取 IO线程数量增加，RESYSTANCE 的吞吐量提升愈发显著，最高可提升 57%。这一结果表明，压实吞吐量的提升对读密集型工作负载有积极影响。此外，即使吞吐量增加，RESYSTANCE 的 P99读延迟也比 RocksDB 更稳定。

4）YCSB 工作负载：图7 展示了使用四个压实线程时 YCSB 工作负载下的吞吐量对比。如表IV所示，在 YCSB-Load 中，RESYSTANCE 的吞吐量比 RocksDB 高出最多 85%，甚至超过了 FillRandom 工作负载。这一提升源于齐夫分布访问模式：更高的键丢弃率会放大 RESYSTANCE 内核内处理的优势。在写入比例较高的 YCSB-A 中，RESYSTANCE 的吞吐量比 RocksDB 高出最多 53%。在读取占主导的工作负载和范围扫描工作负载中，由于压实频率降低限制了压实加速的效果，RESYSTANCE 的表现始终优于 RocksDB，但优势幅度较小。尽管如此，随着输入/输出线程数量增加，吞吐量差距也随之扩大，这凸显出 RESYSTANCE 的压实效率能够随着输入/输出并行度有效提升。

5）关系数据库管理系统上的 OLTP 工作负载：图8 展示了使用 Sysbench 对 MariaDB 上 OLTP 工作负载的评估。MariaDB 是一种使用 RocksDB 作为存储引擎的关系数据库管理系统（MyRocks [33]）。事务层仅与 RocksDB 交互，而压实完全由 RocksDB 存储引擎执行；因此，将 RESYSTANCE 集成到 RocksDB 后，无需进一步修改 RocksDB 的上层或 MariaDB 的事务层。与第VI-B节所述的先前实验不同，本次评估启用了预写日志（WAL）和 L0 到 L0 的内部压实，以评估 RESYSTANCE 在关系数据库管理系统环境中的实际效果（使用四个压实线程）。此外，如表V所示，OLTP 工作负载包含多种键分布和值大小。除 oltp 仅写入工作负载（图8(c)）外，随着 IO线程数量增加，RESYSTANCE 与 RocksDB 之间的吞吐量差距扩大，体现了 RESYSTANCE 的有效性。特别是，在 oltp 读写工作负载中，RESYSTANCE 的吞吐量比 RocksDB 高出 22%。

表V：联机事务处理工作负载特征说明。

工作负载类型	每事务操作数（执行顺序）	键（字节）范围／平均值	值（字节）范围 / 平均
OLTP 插入	1 插入	4–42 / 8	2–724 / 722
OLTP 仅写入	2 次更新、1 次删除、1 次插入		
OLTP 读写	14 次查询、2 次更新，		
OLTP 非索引更新	1 次插入、1 次删除 1 更新		

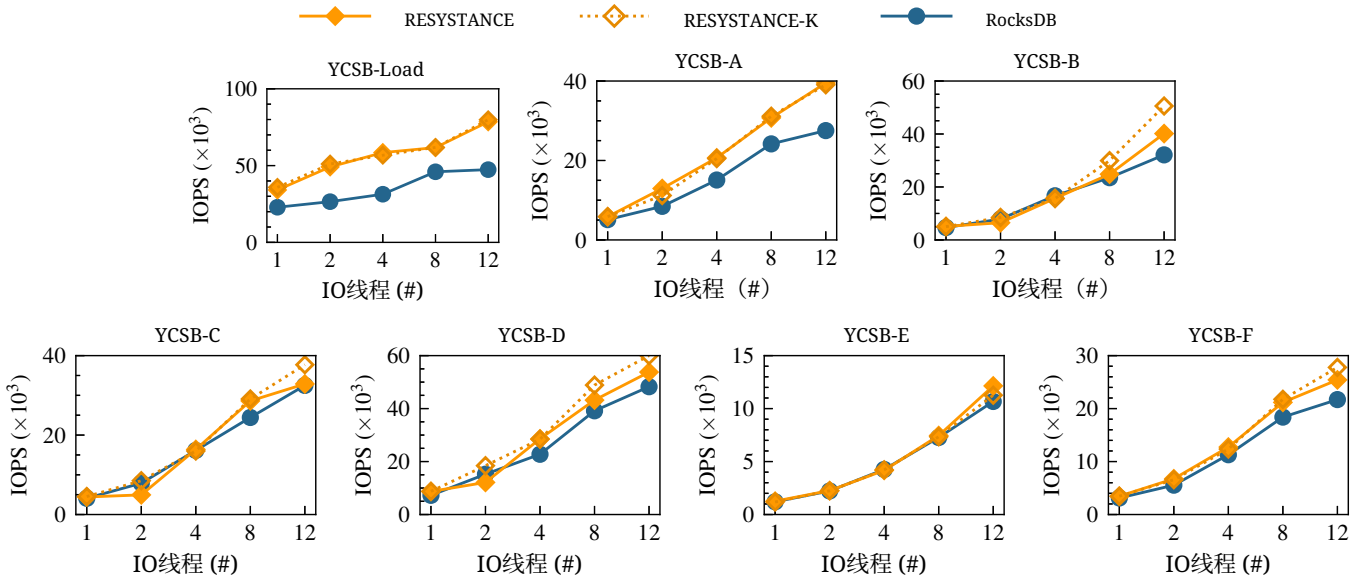
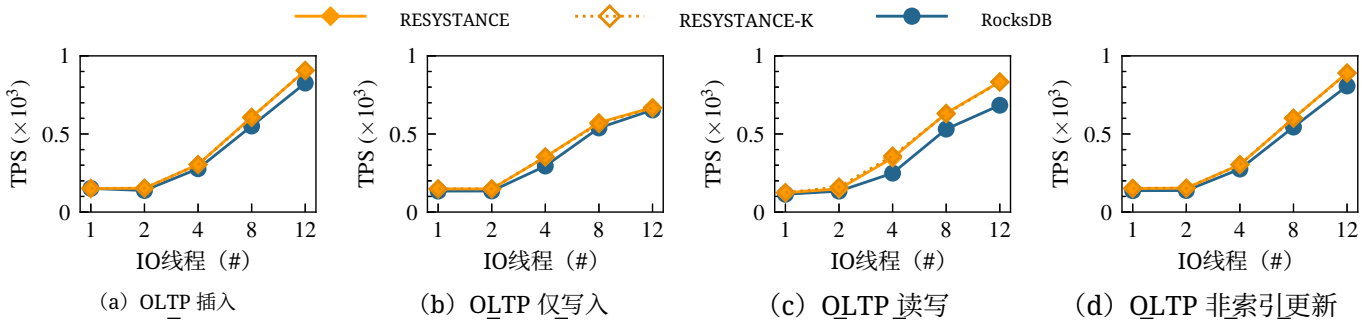


图 7：YCSB 基准测试各工作负载的吞吐量对比。



（a）OLTP 插入　（b）OLTP 仅写入　（c）OLTP 读写　（d）OLTP 非索引更新

图 8：搭配 MyRocks 的 MariaDB 的 Sysbench 评估。

oltp_仅写入工作负载具有读-改-写行为，其性能主要取决于事务处理开销；因此，改进压实带来的收益相对有限。

D. RESYSTANCE 特性分析

1）归并排序算法比较：图9 比较了 RESYSTANCE 中不同归并排序算法的压实执行时间随 SST 文件数量的变化。我们评估了两种代表性的值大小：基准测试中常用的 1千字节，以及在真实 RocksDB 部署中经常观察到的 128字节 [27]。对于这两种值大小，在合并至多 6-8 个 SST 时，线性归并排序略优于最小堆方法，此后最小堆方法变得更快。这些结果为 RESYSTANCE 在第 VI-A 节中采用的归并排序选择策略提供了依据。然而，这些边际性能差异对整体系统吞吐量并不具有主导性影响，因为压实涉及多样的 SST 文件分布，并且大量时间花费在合并以外的阶段。尽管如此，通过评估两种简单算法，我们表明 RESYSTANCE 因其底层架构而仍留有进一步优化的空间。

2）*eBPF* 程序验证开销：我们分析了 eBPF 程序的验证开销，以展示为加载用户定义的合并逻辑内核而进行的验证过程能够正确执行。图 10 展示了将线性归并排序 eBPF 程序加载到内核时，验证开销随允许的最大 SST 文件数变化的情况。图 10(a) 报告了验证时间；由于分支点处验证器的状态空间呈指数增长，验证时间也随最大 SST 文件数呈指数增长。图 10(b) 以对数刻度展示已验证指令数，呈现出类似的指数增长趋势；当 SST 文件数达到 24 时，指令数超过了默认的 1M 验证上限。相比之下，基于最小堆的实现所需的验证成本可以忽略不计：验证时间不超过 0.05 秒，指令数在 20K 至 100K 之间，远低于验证指令上限。这是因为最小堆算法包含的循环迭代次数较少且有静态上界，因此验证状态空间小得多。两种实现所需的栈大小都远低于 eBPF 上限（线性实现为 64B，最小堆实现为 128B），因为归并相关数据存储在内核内存中并通过内核内存访问，而非存储在 eBPF 栈中。

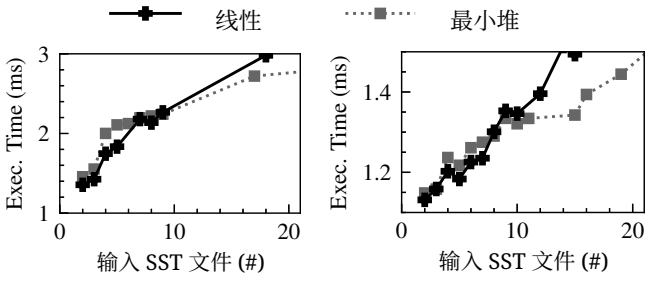


图9：不同值大小下归并排序算法的压实执行时间比较。

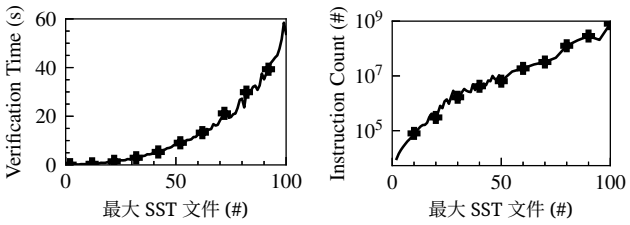


图10：在允许的最大SST数量限制变化时，eBPF程序的验证开销。(a) 验证时间，(b) 验证指令数（对数刻度）。

3）键、值和压实输入大小的影响：图11(a)–(c) 分别展示了 RESYSTANCE 在键大小、值大小和压缩输入大小变化时的效果。 y 轴表示相对于 RocksDB 归一化后的压实时间。图 11(a) 和 (b) 分别展示了键大小变化（值固定为 1千字节）以及值大小变化（键固定为 16B）时的结果。无论键大小还是值大小如何变化，RESYSTANCE 都能将压实时间缩短约 50%。这是因为键和值的大小只会影响数据块中的键值对数量，却不会影响 RESYSTANCE 的 I/O 路径。图11(c) 展示了值为 1千字节时压实时间随压缩输入大小的变化；压缩输入越小，性能提升越明显。这一趋势的原因是，随着压缩开销降低，软件栈开销会变得更加显著，从而进一步提升 RESYSTANCE 的相对效果。

4) 内核旁路与异步 *I/O* 的有效性：为评估内核旁路和异步 I/O 对压实的影响，我们比较了三种配置下的压缩性能：原版RocksDB、使用 io_uring 的 RocksDB，以及基于 SPDK 的 RocksDB。图 12(a) 展示了压实吞吐量随压实线程数的变化。SPDK 通过 BlobFS 实现内核旁路，[16], BlobFS 使用私有的用户空间缓存，而不依赖内核页缓存。该缓存无法禁用，因此我们采用实验确定的最小缓存大小 1GB。由于 BlobFS 在此配置下性能有限，SPDK 带来的压实性能提升较为有限。虽然 io_uring 可通过异步 I/O 加速压实，但压实本身的特性使异步 I/O 的性能收益

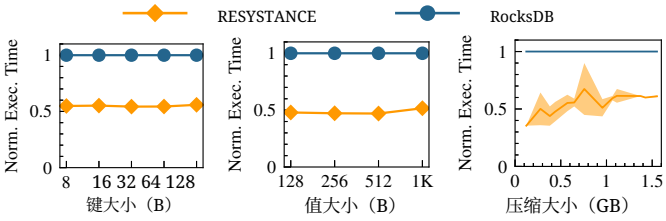


图 11：RESYSTANCE 在不同键大小、值大小和输入大小下的压实效果。 y 轴表示 RocksDB 中归一化的压实处理时间。(a) 按键大小进行比较，(b) 按值大小进行比较，(c) 按压缩输入大小进行比较。

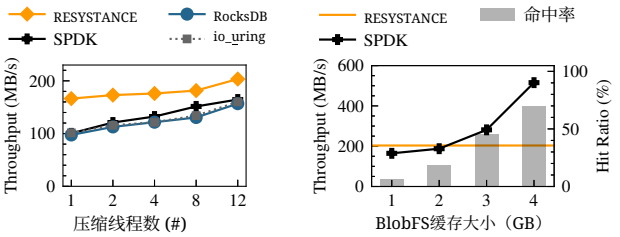


图12：RocksDB、SPDK与io_uring的压实吞吐量对比。(a) 不同压实线程数量下RESYSTANCE、SPDK与io_uring的对比。(b) 不同缓存大小下SPDK与BlobFS的对比。

受到限制，导致压实性能提升微乎其微。图 12(b) 比较了在 12 个压实线程下，BlobFS 缓存大小变化时的吞吐量和 BlobFS 缓存命中率。缓存大小为 1GB 时，缓存命中率仍然很低，表明大多数 I/O 操作都是通过直接 I/O 执行的。随着缓存大小增大，SPDK 吞吐量显著提升，缓存命中率也相应提高。这表明，观察到的性能提升主要源于内存访问增加，而非通过内核旁路降低 I/O 开销。

VII. 相关工作

A. 改进*LSM* 树中的压实

优化压实策略。 Dostoevsky [34]与Spooky [35]通过基于成本模型分析选择混合合并策略，最大限度地减少不必要的压实。旨在降低压实量的研究 [22], [23], [36]–[38]通过细化合并粒度来降低开销。流水线压实 [39], LDC [40], bLSM [41],LSbM树 [42], 与DownForce [43] 改进了压实方法和调度算法。dCompaction [44] 利用元数据减少压实期间的数据I/O。

动态资源管理。 SILK [45], ADOC [25],和 SuccinctKV [46] 动态调节输入/输出压力和后台线程，使数据流与计算资源保持平衡。DLC [47] 通过其调度器延迟 L0压实，并通过突发式压实分散输入/输出消耗。

重新设计 LSM 结构。 WipDB [48]和 PebblesDB [49] 将 LSM 树垂直划分为更小的组件，以减轻内部操作成本。LSM-trie [36]使用哈希键构建唯一的键范围，从而消除键范围重叠。

硬件加速与卸载。 FPGA 加速 [50], [51], GPU 流水线处理 [52]–[54], 近数据处理 [55], [56], DPU 卸载 [57], KV-SSD 协同 [58], 以及云端/边缘卸载 [59]–[62] 将计算转移到设备或外部环境，以减轻主机负载。**键值分离**。WiscKey [63], HashKV [64],和 DiffKV [65] 将键和值分离。尽管这些设计并非专注于优化压实本身，但它们减少了压实所处理的数据量，从而降低了压缩开销。

B. 使用 *eBPF* 优化内核中的 *I/O* 路径

I/O 路径中的内核卸载。 近期研究利用 eBPF 直接优化 I/O 路径，消除用户态–内核态边界处的系统调用和数据复制开销。XRP [7]在 NVMe 驱动程序层插入 eBPF 钩子，以便在内核中执行用户定义的存储函数，从而绕过传统存储栈。用户空间旁路 [66]动态识别连续的系统调用区域，并安全地将相应的用户指令转换为在内核中运行，使程序能够继续执行而无需跨越系统调用边界。这种方法消除了 I/O 密集型工作负载的系统调用开销，且无需修改现有应用程序。

借助 eBPF 扩展内核内数据路径。eBPF 已从输入/输出栈扩展到内存与缓存层。cach ext [67] 允许应用程序将页缓存替换策略定义为在内核中执行的 eBPF 程序。BPF-DB [68]将事务型数据库管理系统嵌入 eBPF，以提供内核级的数据管理一致性，并确保基于 eBPF 的应用程序具备持久性与并发性。基于 eBPF 的缺页处理 [69]进一步通过在 eBPF 中处理缺页，消除了内核态与用户空间之间的数据传输开销，克服了 userfaultfd() 的可扩展性限制。*C*

· 异步输入/输出与系统调用精简

KVell [70]降低系统调用开销，并利用 Linux 异步I/O 批量处理请求，更好地发挥 NVMe 并行性，即使在随机访问工作负载下也能实现高吞吐量。不过，它是通过放宽数据顺序要求并重新设计键值存储架构来实现这些收益的。相比之下，我们的工作保留现有的 LSM-tree 引擎和压实框架，而是借助 eBPF 和 io_uring 消除压实过程中的系统调用开销。

VIII. 讨论

为何仅在 eBPF 中设计合并阶段。如前所述，将 eBPF 程序应用于压实，需要决定是否将读写逻辑与合并逻辑一同纳入 eBPF 即时编译。在本研究中，我们仅将 eBPF 应用于合并阶段。对于写入，压实过程中会先将输出数据累积在内存中，再以大批量的方式写入磁盘。由于这类写入操作发生频率较低，且每次写入的数据量较大，因此 I/O 延迟的影响超过系统调用开销。因此，将 RESYSTANCE 应用于写入路径不太可能带来显著的性能提升。由于 eBPF 目前存在局限，将其应用于读取操作并不可行。首先，eBPF 没有用于发起磁盘 I/O 的辅助函数。若要提供此类支持，就需要对内核进行大量修改，并可能削弱验证器的保证。其次，调用多个 eBPF 程序需要使用尾调用，这会显著增加编程复杂度。这两个问题都会对系统稳定性和可维护性产生负面影响。不过，如果未来能够解决这些局限，就有可能消除读取与合并阶段之间的冗余开销，进而进一步提升性能。

RESYSTANCE 的可维护性。 为实现 RESYSTANCE，我们在 io_uring 中引入了专用的 eBPF 挂钩。此设计确保 RESYSTANCE 所需的验证器修改彼此隔离，不会影响其他 eBPF 程序。此外，io_uring 基本保留了现有架构；RESYSTANCE 扩展了功能，但不会在提交队列条目注册后修改输入/输出执行流程，从而最大限度地减少对其他内核组件的干扰。为增强对 io_uring API 潜在变更的适应能力，RESYSTANCE 只与 io_uring 中的高层抽象交互，例如提交队列注册和完成队列处理，而不依赖内部实现细节。因此，此设计能够保持跨内核版本的兼容性，并在内核演进过程中简化长期维护。

RESYSTANCE 的适应性。 LSM 树中的压实策略大致分为分层策略、分级策略和混合策略；它们在压实触发条件、数据布局和数据移动策略方面各不相同 [13]。这些差异主要体现在准备阶段：在此阶段会触发压实，并根据各策略的布局和移动策略确定输入 SSTable 和合并键范围。然而，一旦输入集合确定，后续运行阶段便遵循通用的基于迭代器的执行流程：块级读取、键比较与合并，以及新 SSTable 的创建（图2）。RESYSTANCE 旨在加速运行阶段反复发生的块读取路径。因此，无论采用何种压实策略，它都适用于分层策略、分级策略和混合策略。不过，系统调用的频率可能因输入 SSTable 的扇入规模或嵌套深度而异。因此，加速效果的幅度可能因策略而有所不同。

九、结论

我们提出了 RESYSTANCE，它集成 eBPF 与 io_uring，将压实 I/O 例程迁移至内核，从而消除不必要的用户与内核切换并降低软件栈开销。基于 db bench 与 YCSB 的实验结果表明，与基线 RocksDB 相比，RESYSTANCE 将压实期间的系统调用次数平均减少 99%，并将压缩时间缩短 50%；在写密集型工作负载下，吞吐量最高提升 75%，p99 延迟降低 40%。

参考文献

[1] 三星电子，“9100 Pro 系列固态硬盘（PCIe 5.0）NVMeTM。” https: // semiconductor.samsung.com/consumer-storage/internal-ssd/9100-pro/，2025年。访问日期：2025年10月28日。[2]英特尔公司，“Optane P5800X 固态硬盘产品简介。” https://www.techradar.com/computing/cloud-computing/an-obscure-4-year-old-intel-ssd-is-still-the-worlds-fastest，2025年。访问日期：2025年10月28日；评测中报告的延迟为“低于10微秒”。[3] 美光科技，“7600 NVMe 固态硬盘（PCIe Gen5）产品简介。” https:// www.micron.com/products/storage/ssd/data-center-ssd/7600-ssd，2025年。访问日期：2025年10月28日；文中称“低且稳定的延迟”。[4] H. He、E. Xu、S. Li、Z. Jia、S. Zheng、Y. Yu、J. Ma 和 X. Liao，“当数据库遇见新型存储设备：通过配置理解并揭示性能不匹配”，VLDB会刊，第16卷，第7期，第1712–1725页，2023年。[5] 铠侠株式会社，“CD8P-R 系列（PCIe 5.0）数据中心 NVMeTM固态硬盘。” https://americas.kioxia.com/en-us/business/ssd/data-center-ssd/cd8p-r-e3s.html，2023年。访问日期：2025年10月28日。[6] SSSTC（固态存储技术公司），“EJ5 U.2（PCIe 5.0）企业级固态硬盘。” https://www.ssstc.com/products-detail/ EJ5-U2-eTLC-pcie-ssd/，2024年。访问日期：2025年10月28日。[7] Y. Zhong、H. Li、Y. J. Wu、I. Zarkadas、J. Tao、E. Mesterhazy、M. Makris、J. Yang、A. Tai、R. Stutsman 等，“XRP：基于 eBPF 的内核内存储函数”，第16届USENIX操作系统设计与实现研讨会（OSDI 22），第375–393页，2022年。[8] Facebook，“Rocksdb：面向高速存储环境的持久化键值存储。” https://github.com/facebook/rocksdb，2013年。访问日期：2025年5月1日。[9] 谷歌，“LevelDB。” https://github.com/google/leveldb，2017年。[10] A. Lakshman 和 P. Malik，“Cassandra：一种去中心化的结构化存储系统”，ACM SIGOPS《操作系统评论》，第44卷，第2期，第35–40页，2010年。[11] P. O’Neil、E. Cheng、D. Gawlick 和 E. O’Neil，“日志结构合并树（LSM树）”，《信息学学报》，第33卷，第351–385页，1996年。[12] Facebook，“压实 · rocksdb。” https://github.com/facebook/ rocksdb/wiki/Compaction，2023年。访问日期：2025年5月1日。[13] S. Sarkar、D. Staratzis、Z. Zhu 和 M. Athanassoulis，“构建并分析LSM压实设计空间（更新版）”，arXiv 预印本 arXiv:2202.04522，2022年。[14] H.-J. Kim、Y.-S. Lee 和 J.-S. Kim，“NVMeDirect：面向NVMe固态硬盘应用特定优化的用户空间I/O框架”，第8届USENIX存储与文件系统热点研讨会（HotStorage 16），（美国科罗拉多州丹佛），USENIX协会，2016年6月。[15] G. Lee、S. Shin、W. Song、T. J. Ham、J. W. Lee 和 J. Jeong，“异步I/O栈：面向超低延迟固态硬盘的低延迟内核 {输入/输出}栈”，2019年USENIX年度技术会议（USENIX ATC 19），第603–616页，2019年。[16] Z. Yang、J. R. Harris、B. Walker、D. Verkamp、C. Liu、C. Chang、G. Cao、J. Stern、V. Verma 和 L. E. Paul，“Spdk：用于构建高性能存储应用的开发套件”，2017年IEEE云计算技术与科学国际会议（CloudCom），第154–161页，IEEE，2017年。[17] K. Latecki 和 M. Wawryk，“SPDK NVMe BDEV性能报告 22.01版”，第11–12页，2022年2月。[18] DPDK项目，“数据平面开发套件（DPDK）。” https://www.dpdk. org/，2025年。访问日期：2025年10月28日。[19] I. Zhang、J. Liu、A. Austin、M. L. Roberts 和 A. Badam，“我还没死！操作系统在内核旁路时代的角色”，操作系统热点研讨会论文集，第73–80页，2019年。[20] M. Fleming，“eBPF全面介绍”，LWN.net，2017年12月。访问日期：2025年10月28日。[21] M. Kerrisk 等，“io_uring(7) — Linux异步I/O API。” man7.org Linux手册页，2024年。[22] T. Yao、Y. Zhang、J. Wan、Q. Cui、L. Tang、H. Jiang、C. Xie 和 X. He，“MatrixKV：利用非易失性内存中的矩阵容器降低基于LSM树的键值存储中的写停顿与写放大”，2020年USENIX年度技术会议（USENIX ATC 20），第17–31页，2020年。[23] C. Ding、T. Yao、H. Jiang、Q. Cui、L. Tang、Y. Zhang、J. Wan 和 Z. Tan，“Trianglekv：利用非易失性内存中的三角形容器降低基于LSM树的键值存储中的写停顿与写放大”，IEEE《并行与分布式系统汇刊》，第33卷，第12期，第4339–4352页，2022年。[24] Z. Wang 和 S. Yin，“RBC：一种用于减少写入停顿和尾延迟的带宽控制器”，载于第52届国际并行处理会议论文集，第213–222页，2023年。[25] J. Yu、S. H. Noh、Y. ri Choi 和 C. J. Xue，“ADOC：通过自动协调日志结构键值存储各组件间的数据流来提升性能”，载于第21届USENIX文件与存储技术会议（FAST 23），（加利福尼亚州圣克拉拉），第65–80页，USENIX协会，2023年2月。[26] Facebook，“写入停顿”。 https://github.com/facebook/rocksdb/wiki/ Write-Stalls，2021年。[27] Z. Cao、S. Dong、S. Vemuri 和 D. H. Du，“Facebook RocksDB 键值工作负载的特征分析、建模与基准测试”，载于第18届USENIX文件与存储技术会议（FAST 20），第209–223页，2020年。[28] S. Volpert、S. Winkelhofer、J. Domaschka 和 S. Wesner，“迈向 eBPF 开销量化：eBPF 与 system-tap 的示例性比较”，载于第16届ACM/SPEC国际性能工程会议伴随论文集，第122–129页，2025年。[29] C. Liu、B. Tak 和 L. Wang，“理解 eBPF 映射的性能”，载于2024年ACM SIGCOMM eBPF与内核扩展研讨会论文集，第9–15页，2024年。[30] Facebook，“DB Bench”。 https://github.com/facebook/rocksdb/wiki/ Benchmarking-tools，2017年。[31] B. F. Cooper、A. Silberstein、E. Tam、R. Ra makrishnan 和 R. Sears，“使用 YCSB 对云服务系统进行基准测试”，载于第一届ACM云计算研讨会论文集，第143–154页，2010年。[32] A. Kopytov，“Sysbench”。https://github.com/akopytov/sysbench，2020年。[33] Y. Matsunobu、S. Dong 和 H. Lee，“MyRocks：服务于 Facebook 社交图的 LSM树数据库存储引擎”，VLDB会刊，第13卷第12期，第3217–3230页，2020年。[34] N. Dayan 和 S. Idreos，“Dostoevsky：通过自适应移除冗余合并，为基于 LSM树的键值存储实现更优的时空权衡”，载于2018年国际数据管理会议论文集，SIGMOD ’18，（美国纽约州纽约市），第505–520页，美国计算机协会，2018年。[35] N. Dayan、T. Weiss、S. Dashevsky、M. Pan、E. Bortnikov 和 M. Twitto，“Spooky：正确地细分LSM树压实”，VLDB会刊，第15卷，第3071–3084页，2022年7月。[36] X. Wu、Y. Xu、Z. Shao 和 S. Jiang，“LSM-Trie：一种面向小数据项、基于 LSM树的超大型键值存储”，载于2015年USENIX年度技术会议（USENIX ATC 15），（加利福尼亚州圣克拉拉），第71–82页，USENIX协会，2015年7月。[37] X. Wang、P. Jin、B. Hua、H. Long 和 W. Huang，“通过块粒度压实降低 LSM树的写放大”，载于2022年IEEE第38届国际数据工程会议（ICDE），第3119–3131页，2022年。[38] H. Sun、G. Chen、Y. Yue 和 X. Qin，“利用细粒度压实机制改进基于 LSM树的键值存储”，IEEE云计算汇刊，第11卷第4期，第3778–3796页，2023年。[39] Z. Zhang、Y. Yue、B. He、J. Xiong、M. Chen、L. Zhang 和 N. Sun，“LSM树的流水线压实”，载于2014年IEEE第28届国际并行与分布式处理研讨会，第777 786页，2014 – 。[40] Y. Chai、Y. Chai、X. Wang、H. Wei 和 Y. Wang，“自适应的低层驱动压实，以优化 LSM树键值存储”，IEEE知识与数据工程汇刊，第34卷第6期，第2595–2609页，2022年。[41] R. Sears 和 R. Ramakrishnan，“bLSM：一种通用日志结构合并树”，载于2012年ACM SIGMOD国际数据管理会议论文集，SIGMOD ’12，（美国纽约州纽约市），第217–228页，美国计算机协会，2012年。[42] D. Teng、L. Guo、R. Lee、F. Chen、Y. Zhang、S. Ma 和 X. Zhang，“一种低成本磁盘解决方案，使 LSM树能够在混合读写工作负载下实现高性能”，ACM存储汇刊，第14卷，2018年4月。[43] H. Byun、H. Yoo 和 S. Park，“重新审视 LSM树中的多线程压实：实现压实流水线化”，载于第54届国际并行处理会议论文集，第794–803页，2025年。[44] F. Pan、Y. Yue 和 J. Xiong，“dCo mpaction：LSM树的延迟压实”，《国际并行程序设计杂志》，第45卷，第1310–1325页，2017年12月。[45] O. Balmau、F. Dinu、W. Z waenepoel、K. Gupta、R. Chandhiramoorthi 和 D. Didona，“SILK：防止日志结构合并键值存储中的延迟尖峰”，载于2019年USENIX年度技术会议（USENIX ATC 19），（华盛顿州伦顿），第753–766页，USENIX协会，2019年7月。

[46] Y. Zhang、S. Yang、H. Hu、C. Yang、P. Cai 和 X. Zhou，“SuccinctKV：一种基于 LSM树、采用基于扫描的压缩的高效 CPU 键值存储”，ACM体系结构、代码优化汇刊，第21卷，2024年11月。[47] P. Jin、J. Li 和 H. Long，“DLC：一种稳定性高、延迟低的新型 LSM树压缩方案”，载于扩展数据库技术国际会议，2021年。[48] X. Zhao、S. Jiang 和 X. Wu，“WipDB：一种模拟桶排序的原地写入键值存储”，载于2021年IEEE第37届国际数据工程会议（ICDE），第1404–1415页，2021年。[49] P. Raju、R. Kadekodi、V. Chidambaram 和 I. Abraham，“PebblesDB：利用碎片化日志结构合并树构建键值存储”，载于第26届操作系统原理研讨会论文集，SOSP ’17，（美国纽约州纽约市），第497–514页，美国计算机协会，2017年。[50] T. Zhang 等，“面向基于 LSM 的键值存储的 FPGA加速压缩”，载于第18届USENIX文件与存储技术会议（FAST ’20），第183–198页，USENIX协会，2020年。另见 USENIX FAST ’20。[51] X. Sun、J. Yu、Z. Zhou 和 C. J. Xue，“用于加速 LSM树键值存储的基于 FPGA 的压缩引擎”，载于2020年IEEE第36届国际数据工程会议（ICDE），第1261 1272页，2020 – 。[52] H. Zhou 等，“面向基于 LSM 的键值存储的 GPU加速压缩策略”，载于2024年MSST会议论文集，2024年。该策略在 LevelDB 上实现，报告的压缩速度提升最高达3。61× 压实速度提升。[53] H. Sun、J. Xu、X. Jiang、G. Chen、Y. Yue 和 X. Qin，“gLSM：使用通用图形处理器加速基于 LSM树的键值存储中的压实”，ACM存储汇刊，第20卷，2024年1月。[54] P. Xu、J. Wan、P. Huang、X. Yang、C. Tang、F. Wu 和 C. Xie，“LUDA：利用 GPU 加速 LSM 键值存储压实”，arXiv预印本 arXiv:2004.03054，2020年。预印本。[55] H. Sun、W. Liu、J. Huang 和 W. Shi，“面向基于 LSM树的键值存储、采用近数据处理的协同压实优化系统”，《并行与分布式计算杂志》，第131卷，第29–43页，2019年9月。[56] H. Sun、B. Lou、C. Zhao、D. Kong、C. Zhang、J. Huang、Y. Yue 和 X. Qin，“面向启用近数据处理的基于 LSM树键值存储的异步压缩加速方案”，ACM嵌入式计算系统汇刊，第23卷，2024年9月。[57] F.-F. Pan、Y.-L. Yue 和 J. Xiong，“dCompaction：通过延迟压实加速 LSM树压实”，《计算机科学与技术学报》，第32卷第1期，第41–54页，2017年。[58] K. Kim、H. Chung、S. Ahn、J. Park、S. Jamil、H. Byun、M. Lee、J. Choi 和 Y. Kim，“KVAccel：一种通过主机与固态硬盘协同、面向基于 LSM树键值存储的新型写入加速器”，载于2025年IEEE国际并行与分布式处理研讨会（IPDPS），第666–677页，IEEE，2025年。[59] J. Li、P. Jin、Y. Lin、M. Zhao、Y. Wang 和 K. Guo，“面向 LSM树的弹性稳定压实：基于函数即服务的 TerarkDB 方案”，CIKM ’21，（美国纽约州纽约市），第3906–3915页，美国计算机协会，2021年。[60] Q. Yu、C. Guo、J. Zhuang、V. Thakkar、J. Wang 和 Z. Cao，“CaaS-LSM：存储解耦基础设施中面向基于 LSM 的键值存储的压缩即服务”，ACM数据管理会刊，第2卷，2024年5月。[61] J. Kim、H. Yoo、S. Lee、H. Byun 和 S. Park，“在边缘联邦中协调基于 LSM树的键值存储之间的压实”，载于2024年IEEE第17届云计算国际会议（CLOUD），第419–429页，IEEE，2024年。[62] J. Kim、H. Byun、S. Lee、M. Kim、Y. Kim、Z. Xie 和 S. Park，“Eco-KVS：面向边缘联邦中基于 LSM树键值存储的能量感知压缩卸载机制”，载于2025年IEEE第25届集群、云与互联网计算国际研讨会（CCGrid），第570–579页，IEEE，2025年。[63] L. Lu、T. S. Pillai、H. Gopalakrishnan、A. C. Arpaci-Dusseau 和 R. H. Arpaci-Dusseau，“WiscKey：在面向固态硬盘的存储中分离键和值”，ACM存储汇刊，第13卷，2017年3月。[64] H. H. W. Chan、Y. Li、P. P. C. Lee 和 Y. Xu，“HashKV：通过哈希实现键值存储中的高效更新”，载于2018年USENIX年度技术会议（USENIX ATC 18），（马萨诸塞州波士顿），第1007–1019页，USENIX协会，2018年7月。[65] Y. Li、Z. Liu、P. P. C. Lee、J. Wu、Y. Xu、Y. Wu、L. Tang、Q. Liu 和 Q. Cui，“面向均衡输入/输出的差异化键值存储管理性能”，载于*2021*年*USENIX*年度技术会议（*USENIX ATC 21*），第673–687页，USENIX协会，2021年7月。[66] Z. Zhou、Y. Bi、J. Wan、Y. Zhou和Z. Li，“用户空间旁路：加速系统调用密集型应用”，载于第*17*届*USENIX*操作系统设计与实现研讨会（*OSDI 23*），（马萨诸塞州波士顿），第33–49页，USENIX协会，2023年7月。[67] T. Zussman、I. Zarkadas、J. Carin、A. Cheng、H. Franke、J. Pfefferle和A. Cidon，“cache ext：使用eBPF定制页缓存”，载于美国计算机协会*SIGOPS*第*31*届操作系统原理研讨会论文集，SOSP ’25，（美国纽约州纽约），第462–478页，美国计算机协会，2025年。[68] M. Butrovich、S. Arch、W. S. Lim、W. Zhang、J. M. Patel和A. Pavlo，“BPF-DB：面向eBPF应用的内核嵌入式事务型数据库管理系统”，美国计算机协会数据管理论文集，第3卷，第3期，第1–27页，2025年。[69] T. Zussman、T. Jiang和A. Cidon，“使用eBPF自定义页面错误处理”，载于*2024*年*ACM SIGCOMM eBPF*与内核扩展研讨会论文集，eBPF ’24，（美国纽约州纽约），第71–73页，美国计算机协会，2024年。[70] B. Lepers、O. Balmau、K. Gupta和W. Zwaenepoel，“Kvell：快速持久化键值存储的设计与实现”，载于第*27*届美国计算机协会操作系统原理研讨会论文集，第447–461页，2019年。