XRP: In-Kernel Storage Functions with eBPF

Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao,
Evan Mesterhazy, Michael Makris, and Junfeng Yang, *Columbia University;*
Amy Tai, *Google;* Ryan Stutsman, *University of Utah;*
Asaf Cidon, *Columbia University*

https://www.usenix.org/conference/osdi22/presentation/zhong

This paper is included in the Proceedings of the
16th USENIX Symposium on Operating Systems
Design and Implementation.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-28-1

XRP: 内核存储功能与eBPF

钟宇红, 李浩宇, 吴宇建, 约翰尼斯·扎卡达斯, 蔡杰 ,
伊万·梅斯特哈齐, 迈克尔·马克里斯, 和杨军锋, 哥伦比亚大学 *rsity;*
艾米·泰, 谷歌; 瑞安·施图茨曼, 犹他大学;
阿萨夫·西顿, 哥伦比亚大学

https://www.usenix.org/conference/osdi22/presentation/zh ong

本文收录在《第16届USENIX操作系统设计与实现
研讨会论文集》中。

7月11日至13日, 2022 • 卡尔斯巴德，CA，美国

978-1-939133-28-1

XRP: In-Kernel Storage Functions with eBPF

Yuhong Zhong[1], Haoyu Li[1], Yu Jian Wu[1], Ioannis Zarkadas[1], Jeffrey Tao[1], Evan Mesterhazy[1], Michael Makris[1], Junfeng Yang[1], Amy Tai[2], Ryan Stutsman[3], and Asaf Cidon[1]

[1]Columbia University, [2]Google, [3]University of Utah

Abstract

With the emergence of microsecond-scale NVMe storage devices, the Linux kernel storage stack overhead has become significant, almost doubling access times. We present XRP, a framework that allows applications to execute user-defined storage functions, such as index lookups or aggregations, from an eBPF hook in the NVMe driver, safely bypassing most of the kernel's storage stack. To preserve file system semantics, XRP propagates a small amount of kernel state to its NVMe driver hook where the user-registered eBPF functions are called. We show how two key-value stores, BPF-KV, a simple B^+-tree key-value store, and WiredTiger, a popular log-structured merge tree storage engine, can leverage XRP to significantly improve throughput and latency.

1 Introduction

With the rise of new high performance memory technologies, such as 3D XPoint and low latency NAND, new NVMe storage devices can now achieve up to 7 GB/s bandwidth and latencies as low as 3 μs [11, 19, 24, 26]. At such high performance, the kernel storage stack becomes a major source of overhead impeding both application-observed latency and IOPS. For the latest 3D XPoint devices, the kernel's storage stack *doubles* the I/O latency, and it incurs an even greater overhead for throughput (§2.1). As storage devices become even faster, the kernel's relative overhead is poised to worsen.

Existing approaches to tackle this problem tend to be radical, requiring intrusive application-level changes or new hardware. Complete kernel bypass through libraries such as SPDK [82] allows applications to directly access underlying devices, but such libraries also force applications to implement their own file systems, to forgo isolation and safety, and to poll for I/O completion which wastes CPU cycles when I/O utilization is low. Others have shown that applications using SPDK suffer from high average and tail latencies and severely reduced throughput when the schedulable thread count exceeds the number of available cores [54]; we confirm this in §6, showing that in such cases applications indeed suffer a 3× throughput loss with SPDK.

In contrast to these approaches, we seek a readily-deployable mechanism that can provide fast access to emerging fast storage devices that requires no specialized hardware and no significant changes to the application while working with existing kernels and file systems. To this end, we rely on BPF (Berkeley Packet Filter [67, 68]) which lets applications offload simple functions to the Linux kernel [8]. Similar to kernel bypass, by embedding application-logic deep in the kernel stack, BPF can eliminate overheads associated with kernel-user crossings and the associated context switches. Unlike kernel bypass, BPF is an OS-supported mechanism that ensures isolation, does not lead to low utilization due to busy-waiting, and allows a large number of threads or processes to share the same core, leading to better overall utilization.

The support of BPF in the Linux kernel makes it an attractive interface for allowing applications to speed up storage I/O. However, using BPF to speed up storage introduces several unique challenges. Unlike existing packet filtering and tracing use cases, where each BPF function can operate in a self-contained manner on a particular packet or system trace — for example, network packet headers specify which flow they below to — a storage BPF function may need to synchronize with other concurrent application-level operations or require multiple function calls to traverse a large on-disk data structure, a workload pattern we call "resubmission" of I/Os (§2.3). Unfortunately the state required for resubmission such as access-control information or metadata on how individual storage blocks fit in the larger data structure they belong to is not available at lower layers.

To tackle these challenges, we design and implement XRP (eXpress Resubmission Path), a high-performance storage data path using Linux eBPF. XRP is inspired by XDP, the recent efficient Linux eBPF networking hook [28]. In order to maximize its performance benefit, XRP uses a hook in the NVMe driver's interrupt handler, thereby bypassing the kernel's block, file system and system call layers. This allows XRP to trigger BPF functions directly from the NVMe driver as each I/O completes, enabling quick resubmission of I/Os that traverse other blocks on the storage device.

XRP: 内核存储功能与eBPF

仲宇红1, 李浩宇1, 建武余1,IoannisZarkadas1,Jeffrey Tao1,Evan Mesterhazy 1 1 2 3 1 1,
Michael Makris , Junfeng Yang , Amy Tai , Ryan Stutsman , and Asaf Cidon

1哥伦比亚大学, 2Google, 3犹他大学

摘要

随着微秒级NVMe存储设备的出现，Linux内核存储栈的开销变得显著，访问时间几乎翻倍。我们介绍了XRP，一个允许应用程序从NVMe驱动中的eBPF钩子执行用户定义的存储功能（如索引查找或聚合）的框架，安全地绕过内核的大部分存储栈。为了保留文件系统语义，XRP将少量内核状态传播到其NVMe驱动器钩子，用户注册的eBPF函数在此处被调用。我们展示了两个键值存储，BPF-KV（一个简单的B^+-树键值存储）和WiredTiger（一个流行的日志结构合并树存储引擎），如何利用XRP显著提高吞吐量和延迟。

1 简介

随着新型高性能内存技术（如3D XPoint和低延迟NAND）的兴起，新的NVMe存储设备现在可以实现高达7 GB/s的带宽和低至3 μs的延迟 [11, 19, 24, 26]。在如此高的性能下，内核存储栈成为主要的开销来源，阻碍了应用程序感知的延迟和IOPS。对于最新的3D XPoint设备，内核的存储栈 使*I/O*延迟翻倍 ，并且对吞吐量的开销更大（§2.1）。随着存储设备变得更快，内核的相对开销可能会恶化。

现有的解决此问题的方法往往比较激进，需要侵入式应用程序级更改或新硬件。通过SPDK等库实现的完全内核绕过允许应用程序直接访问底层设备，但这些库也迫使应用程序实现自己的文件系统，放弃隔离和安全性，并在I/O利用率低时轮询I/O完成以浪费CPU周期。其他人已经表明，当可调度的线程数超过可用核心数时，使用SPDK的应用程序会遭受高平均和尾部延迟以及严重降低的吞吐量 [54]；我们在

§6, showing that in such cases applications indeed suffer a throughput loss with SPDK.

与这些方法不同，我们寻求一种易于部署的机制，该机制能够在无需专用硬件且应用程序无需重大更改的情况下，为新兴的高速存储设备提供快速访问，同时与现有内核和文件系统协同工作。为此，我们依赖于BPF（Berkeley Packet Filter），它允许应用程序将简单功能卸载到Linux内核。类似于内核旁路，通过将应用程序逻辑嵌入到内核栈深处，BPF可以消除与内核-用户跨越相关的开销以及相关的上下文切换。与内核旁路不同，BPF是一种由操作系统支持的机制，它确保隔离性，不会因忙等待而导致低利用率，并允许大量线程或进程共享同一核心，从而提高整体利用率。

Linux内核中的BPF支持使其成为应用程序加速存储I/O的吸引人接口。然而，使用BPF加速存储引入了几个独特的挑战。与现有的数据包过滤和跟踪用例不同，在这些用例中，每个BPF函数可以独立地对特定数据包或系统跟踪进行操作——例如，网络数据包头指定它们属于哪个流——存储BPF函数可能需要与其他并发应用程序级操作同步，或者需要多次函数调用来遍历大型磁盘数据结构，我们称这种工作负载模式为I/O的“重新提交”（§2.3）。不幸的是，重新提交所需的状态信息，例如访问控制信息或有关单个存储块如何适合其所属更大数据结构的元数据，在较低层不可用。

为了应对这些挑战，我们设计并实现了 XRP（eXpress Resubmission Path），一种使用 Linux eBPF 的高性能存储数据路径。XRP 受到 XDP 的启发，XDP 是最近一种高效的 Linux eBPF 网络钩子 [28]。为了最大化其性能优势，XRP 在 NVMe 驱动程序的中断处理程序中使用一个钩子，从而绕过内核的块、文件系统和系统调用层。这允许 XRP 在每个 I/O 完成时直接从 NVMe 驱动程序触发 BPF 函数，从而能够快速重新提交在存储设备上其他块上传输的 I/O。

The key challenge in XRP is that the low-level NVMe driver lacks the context that the higher levels provide. Those layers contain information such as who owns a block (file system layer), how to interpret the block's data, and how to traverse the on-disk data structure (application layer).

Our insight is that many storage-optimized data structures that power real-world databases [10, 12, 20, 27, 44, 66, 70, 80] – such as on-disk B-trees, log-structured merge trees, and log segments – are typically implemented on a small set of large files, and they are updated orders of magnitude less frequently than they are read; we validate this in §3. Hence, we exclusively focus XRP on operations contained within one file and on data structures that have a fixed layout on disk. Consequently, the NVMe driver only requires a minimal amount of the file system mapping state, which we term the *metadata digest*; this information is small enough that it can be passed from the file system to the NVMe driver so it can safely perform I/O resubmissions. This allows XRP to safely support some of the most popular on-disk data structures.

We present a design and implementation of XRP on Linux, with support for ext4, which can easily be extended to other file systems. XRP enables the NVMe interrupt handler to resubmit storage I/Os based on user-defined BPF functions.

We augment two key-value stores with XRP: BPF-KV, a B^+-tree based key-value store that is custom-designed for supporting BPF functions, and WiredTiger's log-structured merge tree, which is used as one of MongoDB's storage engines [27]. With random 512 B object reads on BPF-KV with multiple threads using a B^+-tree that has three index levels on disk, XRP has 47%–94% higher throughput and 20%–34% lower p99 latency than `read()`. XRP also enables more efficient sharing of cores among applications than kernel bypass: it is able to provide 56% better p99 latency than SPDK with two threads sharing the same core. In addition, XRP is able to consistently improve WiredTiger's performance by up to 24% under YCSB [41]. We open source XRP and our changes to BPF-KV and WiredTiger at `https://github.com/xrp-project/XRP`.

We make the following contributions.

1. **New Datapath**. XRP is the first datapath that enables the use of BPF to offload storage functions to the kernel.
2. **Performance**. XRP improves the throughput of a B-tree lookup by up to 2.5× compared to normal system calls.
3. **Utilization**. XRP provides latencies that approach kernel bypass, but unlike kernel bypass, it allows cores to be efficiently shared by the same threads and processes.
4. **Extensibility**. XRP supports different storage use cases, including different data structures and storage operations (e.g., index traversals, range queries, aggregations).

2 Background and Motivation

In this section we show why the Linux kernel is becoming a primary bottleneck with fast NVMe devices and provide a primer on BPF.

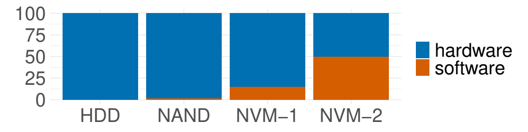


Figure 1: Kernel's latency overhead with 512 B random reads. HDD is Seagate Exos X16, NAND is Intel Optane 750 TLC NAND, NVM-1 is first generation Intel Optane SSD (900P), and NVM-2 is second generation Intel Optane SSD (P5800X).

kernel crossing	351 ns	5.6%
read syscall	199 ns	3.2%
ext4	2006 ns	32.0%
bio	379 ns	6.0%
NVMe driver	113 ns	1.8%
storage device	3224 ns	51.4%
total	6.27 µs	100.0%

Table 1: Average latency breakdown of a 512 B random `read()` syscall using Intel Optane P5800X.

2.1 Software is Now the Storage Bottleneck

New media like 3D Xpoint [1] and low-latency NAND [26], have led to new NVMe storage devices that exhibit single-digit µs latencies and millions of IOPS [11, 19, 24, 26]. The kernel storage stack is becoming a major performance bottleneck when accessing these devices. Figure 1 shows the percentage of time spent in the Linux stack when issuing a 512 B random read I/O on different storage devices. While the software overhead for the first generation of fast NVMe devices (first generation Intel Optane or Z-NAND) was non-negligible (~15%), with the latest generation of devices (Intel Optane SSD P5800X) the software overhead accounts for about half of the latency of each read request. The kernel's relative overhead will only get worse as storage devices become even faster.

Where is the time going? Table 1 shows the time spent in the different storage layers when issuing a random 512 B read with O_DIRECT on Optane P5800X. The experimental setup, which is used throughout this section, is a server with 6-core i5-8500 3 GHz with 16 GB of memory, using Ubuntu 20.04, and Linux 5.8.0. We also disable processor C-states and turbo boost, use the maximum performance governor, and disable KPTI [30]. The experiment shows that the most expensive layer is the file system (ext4), followed by the block layer (bio) and the kernel crossing, and that the total software overhead accounts for 48.6% of the average latency.

Why not just bypass the kernel? One approach to eliminate kernel overhead is to bypass it altogether [7, 65, 82, 83], leaving just the cost to post a request to the NVMe driver and the device's latency. However, kernel bypass is no panacea: each user is entrusted with full access to the device; they must also construct their own user space file systems [73, 74]. This means that there is no mechanism to enforce fine-grained

XRP中的关键挑战在于，低级NVMe驱动程序缺乏高层提供的上下文。这些层包含诸如谁拥有一个块（文件系统层）、如何解释块的数据以及如何遍历磁盘上的数据结构（应用层）等信息。

我们的见解是，许多为真实世界数据库提供动力的存储优化数据结构 [10, 12, 20, 27, 44, 66, 70,80] ——例如磁盘上的B树、日志结构合并树和日志段——通常是在一小组大文件上实现的，并且它们更新的频率比读取的频率低几个数量级；我们在 § 3中验证了这一点。因此，我们专注于XRP中单个文件内的操作以及磁盘上具有固定布局的数据结构。因此，NVMe驱动程序只需要文件系统映射状态的最小部分，我们称之为元数据摘要；这些信息足够小，可以将其从文件系统传递到NVMe驱动程序，以便它可以安全地执行I/O重新提交。这使XRP能够安全地支持一些最受欢迎的磁盘上数据结构。

我们在Linux上展示了XRP的设计和实现，支持ext4文件系统，可以轻松扩展到其他文件系统。XRP使NVMe中断处理程序能够基于用户定义的BPF函数重新提交存储I/O。

我们使用XRP增强了两个键值存储：BPF-KV，一个基于B^+-树的键值存储，专为支持BPF函数而定制，以及WiredTiger的日志结构合并树，它被用作MongoDB的一种存储引擎 [27]。在BPF-KV上使用多个线程进行随机512 B对象读取，使用具有三个磁盘索引级别的B^+-树，XRP的吞吐量比 `read()`高47%–94%，p99延迟低20%–34%。XRP还比内核绕过使应用程序之间的核心共享更高效：它能够提供比SPDK使用两个线程共享同一核心高56%的p99延迟。此外，XRP能够在YCSB下始终如一地提高WiredTiger的性能，最高可达24%。我们开源了XRP以及我们对BPF-KV和WiredTiger的更改 [41]。

我们做出了以下贡献。

1. **新数据路径**。XRP 是第一个支持使用 BPF 将存储功能卸载到内核的数据路径。

2. **性能**。与普通系统调用相比，XRP 将 B 树查找的吞吐量提高了高达 2.5×。3. **利用率**。XRP 提供接近内核绕过的延迟，但与内核绕过不同，它允许核心被同一线程和进程高效共享。

bypass，但不同于内核绕过，它允许核心被同一线程和进程高效共享。

4. **可扩展性**。XRP 支持不同的存储用例，包括不同的数据结构和存储操作（例如，索引遍历、范围查询、聚合）。

2 背景 和 动机

在本节中，我们展示了为什么 Linux 内核在与高速 NVMe 设备一起使用时正成为一个主要瓶颈，并提供 BPF 的简介。

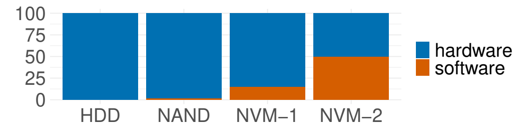


图 1: 内核在 512 B 随机读取下的延迟开销。HDD 是 Seagate Exos X16，NAND 是 Intel Optane 750 TLC NAND，NVM-1 是第一代 Intel Optane SSD（900P），而 NVM-2 是第二代 Intel Optane SSD（P5800X）。

内核跨越	351 ns	5.6%
读取系统调用	199 ns	3.2%
ext4	2006 ns	32.0%
bio	379 ns	6.0%
NVMe driver	113 ns	1.8%
存储设备	3224 ns	51.4%
total	6.27 µs	100.0%

Table 1: Average latency breakdown of a 512 B random `read()` syscall using Intel Optane P5800X.

2.1 软件已成为存储瓶颈

新型媒体如 3D Xpoint [1] 和低延迟 NAND [26],已催生出新的 NVMe 存储设备，这些设备表现出个位数 µs 的延迟和数百万 IOPS [11, 19, 24, 26]。当访问这些设备时，内核存储栈正成为主要的性能瓶颈。图 1 展示了在不同存储设备上执行 512 B 随机读取 I/O 时 Linux 栈所花费的时间百分比。虽然第一代高速 NVMe 设备（第一代 Intel Optane 或 Z-NAND）的软件开销不可忽视（~15%），但随着最新一代设备（Intel Optane SSD P5800X）的推出，软件开销约占每次读取请求延迟的一半。随着存储设备变得更快，内核的相对开销只会变得更糟。

时间都去哪儿了? 表1 展示了在 Optane P5800X 上使用 O_DIRECT 执行随机 512 B 读取时不同存储层所花费的时间。本节中整个实验所使用的实验设置是一台配备 6 核 i5-8500 3 GHz 且内存为 16 GB 的服务器，使用 Ubuntu 20.04 和 Linux 5.8.0。我们还禁用了处理器 C-states 和涡轮增压，使用最大性能调节器，并禁用了 KPTI [30]。实验表明，最昂贵的层是文件系统（ext4），其次是块层（bio）和内核跨越，总软件开销占平均延迟的 48.6%。

为什么不直接绕过内核? 消除内核开销的一种方法是完全绕过它 [7, 65, 82, 83],只留下向 NVMe 驱动器发布请求和设备延迟的成本。然而，内核绕过并非万能药：每个用户都被授予对设备的完全访问权限；他们还必须构建自己的用户空间文件系统 [73, 74]。这意味着没有机制来强制执行细粒度

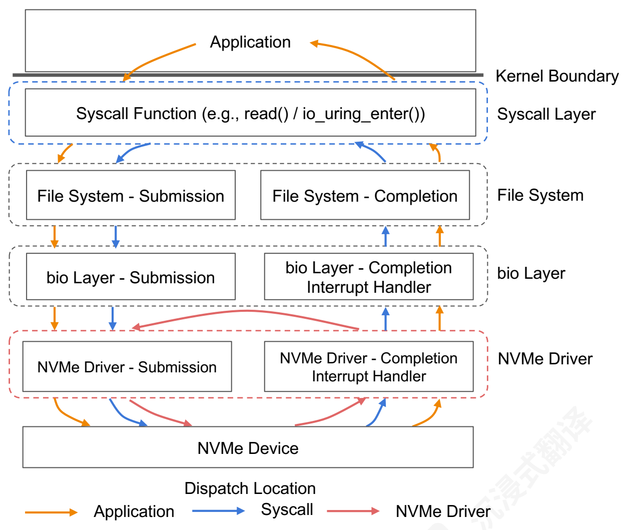


Figure 2: Dispatch paths for the application and two kernel hooks.

isolation or to share capacity among different applications accessing the same device. In addition, there is no efficient way for user space applications to receive interrupts on I/O completions, so applications must directly poll on device completion queues to obtain high performance. Consequently, when I/O is not the bottleneck, cores cannot be shared among processes, which results in significant under-utilization and wasted CPU. Furthermore, when more than one polling thread shares the same processor, the CPU contention between them coupled with the lack of synchronization lead all polling threads to experience degraded tail latency and significantly lower overall throughput. Recent work has highlighted this issue [54] and we reproduce it in §6.2.

2.2 BPF Primer

BPF (Berkeley Packet Filter) is an interface that allows users to offload a simple function to be executed by the kernel. Linux's framework for BPF is called eBPF (extended BPF) [23]. Linux eBPF is commonly used for filtering packets (e.g., TCPdump) [5, 6, 28, 52], load balancing and packet forwarding [5, 18, 25, 60], tracing [2, 4, 50], packet steering [46], network scheduling [53, 58] and network security checks [15]. Functions are verified by the kernel at install-time to ensure they are safe; for example, they are checked to make sure they do not contain too many instructions, unbounded loops, or accesses to out-of-bounds memory addresses [29]. After verification, which typically takes a few seconds or less, the eBPF functions can be called normally.

2.3 The Potential Benefit of BPF

BPF can be a mechanism for avoiding data movement between the kernel and user space in cases when a logical lookup requires a sequence of "auxiliary" I/O requests that generate intermediate data not needed directly by the application, such

	Latency	Speedup	Throughput	Speedup
User Space	78 µs	1×	109K IOPS	1×
Syscall Layer	68 µs	1.15×	130K IOPS	1.2×
NVMe Driver	40 µs	1.95×	276K IOPS	2.5×

Table 2: Average latency and throughput improvement with respect to user space when resubmitting I/O from the given layer; for kernel layers, resubmission is executed with a BPF function. Results shown for lookups on an on-disk B-tree of depth 10 [85].

as in pointer-chasing workloads. For example, to traverse a B-tree index, a lookup at each level traverses the kernel's entire storage stack only to be thrown away by the application once it obtains the pointer to the next child node in the tree. Instead of a sequence of system calls from user space, each of the intermediate pointer lookups could be executed by a BPF function, which would parse the B-tree node to find the pointer to the relevant child node. The kernel would then submit the I/O to fetch the next node. Chaining a sequence of such BPF functions could avoid the cost of traversing kernel layers and moving data to user space.

Other popular on-disk data structures, such as log-structured merge trees (LSM trees) [70], also have such auxiliary pointer lookups which can be accelerated using BPF functions. Other types of operations that would benefit from such an approach include range queries, iterators, and other types of aggregations (e.g., obtain the maximum or average value in a range of key-value pairs). In all of these cases, only a single result or a small subset of the objects that might be accessed by the storage system ultimately need to be returned to the application.

The BPF function that resubmits (dispatches) I/O in auxiliary I/O workloads could be placed at any layer of the kernel. Figure 2 shows the I/O paths for both normal user space dispatch and for two possible locations of BPF resubmission hooks: in the syscall layer and in the NVMe driver. Zhong et al. [85] compared the performance improvement from a resubmission hook in both locations on workloads with auxiliary I/O by measuring the speedup of lookup queries on an on-disk B-tree of depth 10. The baseline for comparison is reading I/O through the `read` system call. Table 2 summarizes the results.

Best Case Acceleration. Dispatching the I/O requests from the NVMe driver provides a significant latency reduction (up to 49%) and corresponding speedup (up to 2.5×), since it bypasses almost the entire kernel software stack. On the other hand, as expected, issuing the BPF functions from the syscall dispatch layer only provides a maximum speedup of 1.25×, since the requests only benefit from eliminating kernel boundary crossings, which only account for 5-6% of the kernel overhead (Table 1). After reaching CPU saturation, the computation savings of reissuing the submissions from the NVMe driver translate into throughput improvements of 1.8-2.5×, depending on the number of threads in the workload [85].

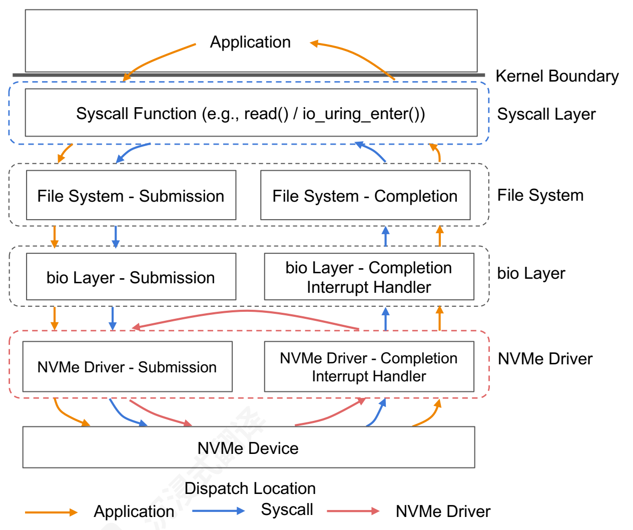


图 2：应用程序和两个内核钩子的调度路径。

隔离或在不同应用程序访问同一设备时共享容量。此外，用户空间应用程序没有有效的方法来接收 I/O 完成中断，因此应用程序必须直接轮询设备完成队列以获得高性能。因此，当 I/O 不是瓶颈时，核心无法在进程之间共享，导致显著的低利用率和 CPU 浪费。此外，当多个轮询线程共享同一处理器时，它们之间的 CPU 竞争加上缺乏同步导致所有轮询线程的尾部延迟降低并显著降低整体吞吐量。最近的工作突出了这个问题 [54]，我们在 § 6.2。

2.2 BPF Primer

BPF（Berkeley Packet Filter）是一个允许用户将简单函数卸载给内核执行的接口。Linux 的 BPF 框架称为 eBPF（扩展 BPF） [23]。Linux eBPF 通常用于过滤数据包（例如，TCPdump） [5, 6, 28, 52], 负载均衡和数据包转发 [5, 18, 25, 60], 跟踪 [2, 4, 50], 数据包重定向 [46], 网络调度 [53, 58] 和网络安全检查 [15]。函数在安装时由内核验证以确保它们是安全的；例如，会检查它们是否包含过多的指令、无界循环或对越界内存地址的访问 [29]。验证后，通常只需几秒钟或更少，eBPF 函数就可以正常调用。

2.3 BPF的潜在益处

BPF可以是一种机制，用于避免内核和用户空间之间的数据移动，在逻辑查找需要一系列"辅助"I/O请求且这些请求生成中间数据而应用程序并不直接需要这些数据的情况下，例如

	延迟	加速	吞吐量	Speedup
用户空间	78 µs	1×	109K IOPS	1×
Syscall Layer	68 µs	1.15×	130K IOPS	1.2×
NVMe 驱动	40 µs	1.95×	276K IOPS	2.5×

表 2：相对于用户空间，在给定层重新提交 I/O 时的平均延迟和吞吐量提升；对于内核层，重新提交是通过 BPF 函数执行的。显示的是在深度 10 [85]的磁盘 B 树上的查找结果。

与指针遍历工作负载类似。例如，要遍历 B 树索引，每个级别的查找都会遍历内核的整个存储堆栈，但一旦应用程序获得树中下一个子节点的指针，就会将其丢弃。而不是用户空间的一系列系统调用，每个中间指针查找都可以由 BPF 函数执行，该函数将解析 B 树节点以找到相关子节点的指针。然后内核会提交 I/O 以获取下一个节点。链式调用一系列这样的 BPF 函数可以避免遍历内核层和将数据移动到用户空间的成本。

其他流行的磁盘数据结构，如日志结构合并树（ LSM树） [70],也具有此类辅助指针查找，可以使用BPF函数加速。其他可以从这种方法中受益的操作包括范围查询、迭代器和其他类型的聚合（例如，在键值对范围内获取最大值或平均值）。在所有这些情况下，最终只需要将单个结果或存储系统可能访问的对象的小部分返回给应用程序。

在辅助I/O工作负载中重新提交（调度）I/O的BPF函数可以放置在内核的任何层。图2 显示了正常用户空间调度和两个可能的BPF重新提交钩子位置（系统调用层和NVMe驱动程序）的I/O路径。Zhong等人 [85] 通过测量深度为10的磁盘B树上的查找查询速度，比较了在两个位置上的重新提交钩子对辅助I/O工作负载的性能提升。比较的基线是通过系统调用读取I/O。`read` 表2 总结了结果。

最佳情况加速。 从 NVMe 驱动程序中调度 I/O 请求可显著降低延迟（高达 49%）并相应提高速度（高达 2。5×），因为它绕过了几乎整个内核软件栈。另一方面，正如预期的那样，从系统调用调度层发出 BPF 函数仅提供最大 1。25× 的加速，因为请求仅受益于消除内核边界跨越，而内核边界跨越仅占内核开销的 5-6%（表 1）。在达到 CPU 饱和后，重新从 NVMe 驱动程序提交的计算节省转化为吞吐量提升 1.8-2。5×，具体取决于工作负载中的线程数量 [85]。

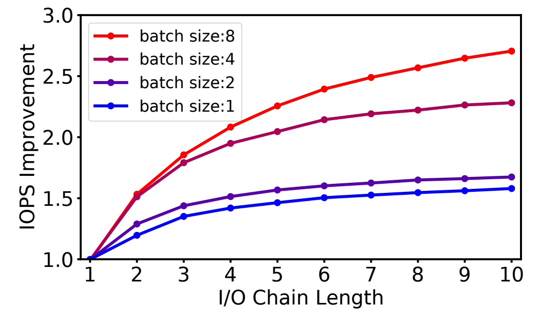


Figure 3: Single-threaded lookups with io_uring syscall, using NVMe driver hook.

Placing an eBPF hook *anywhere* in the kernel may improve throughput between 1.2–2.5×. However, pushing the I/O dispatching as close as possible to the storage device dramatically improves the performance of a traversal. Hence *to obtain the highest possible speedup, XRP's resubmission hook should reside in the NVMe driver.*

What about io_uring? io_uring is a new Linux system call framework [9] that allows processes to submit batches of asynchronous I/O requests, which amortizes user-kernel crossings. However, each I/O submitted with io_uring still passes through all the layers shown in Table 1, so each individual I/O still incurs the full storage stack overhead. In fact, BPF I/O resubmissions are largely complementary to io_uring: io_uring can efficiently submit batches of I/Os that trigger different I/O chains managed by BPF in the kernel.

Figure 3 shows throughput improvements when using io_uring with a BPF hook in the NVMe driver. I/O Chain Length denotes the total number of I/Os, including the initial I/O and the resubmitted I/Os. Figure 3 shows that BPF can increase throughput with respect to io_uring by up to 1.5× for small batch sizes and up to 2.5× as batch sizes increase.

In summary, BPF can benefit both legacy read and io_uring system calls. By placing the hook in the kernel NVMe driver, BPF may increase throughput of both legacy I/O and single-threaded io_uring by up to 2.5×.

3 Design Challenges and Principles

As shown in the previous section, I/O resubmission must occur as close to the device as possible in order to reap the greatest benefits. In the NVMe software stack, this is the NVMe interrupt handler. However, executing the resubmissions from within the NVMe interrupt handler, which lacks the context of the file system layer, introduces two major challenges.

Challenge 1: address translation and security. The NVMe driver has no access to file system metadata. In the example of an index traversal, XRP issues a read I/O to a particular block and executes a BPF function that would extract the offset of the next block it would like to query. However, this offset is meaningless to the NVMe layer, since it cannot tell which physical block the offset corresponds to without having access to the file's metadata and extents. Even if the application developer made the effort to embed physical block addresses to avoid the translation of the file system offset, which would be burdensome, the BPF function could access *any* block on the device, including blocks that belong to a file that the user does not have permissions to access.

Challenge 2: concurrency and caching. It is challenging to enable concurrent reads and writes issued from the file system with XRP. A write issued from the file system will only be reflected in the page cache, which is not visible to XRP. In addition, any writes that modify the layout of the data structure (e.g., modify the pointers to the next block) that are issued concurrently to read requests could lead XRP to accidentally fetch the wrong data. Both of these could be addressed by locking, but accessing locks from within the NVMe interrupt handler may be expensive.

Observation: most on-disk data structures are stable. Both of these challenges would make it difficult to implement arbitrary concurrent BPF storage functions. However, we make the observation that the files of many storage engines (e.g., LSM trees and B-trees) remain relatively stable. Some data structures simply do not modify on-disk storage structures in-place. For example, once an LSM tree writes its index files (called SSTables) to disk, they are immutable until they are deleted [12, 27, 44]. Accessing these immutable on-disk storage structures requires less synchronization effort. Similarly, even though some on-disk B-tree index implementations support in-place updates, their file extents remain stable for long periods of time. We verify this in a 24-hour YCSB [41] (40% reads, 40% updates, 20% inserts, Zipfian 0.7) experiment on MariaDB running TokuDB [20], which uses a fractal tree (an on-disk B-tree variant) as its lookup index. We found the index file's extents only changed every 159 seconds on average, with only 5 extent changes in 24 hours unmapping any blocks, making it possible to cache file system metadata in the NVMe driver without the overhead of frequent updates. We also make the observation that in all of these storage engines, the indices are stored on a small number of large files, and each index does not span multiple files.

Design principles. These observations and experiments inform the following design principles.

- **One file at a time.** We initially restrict XRP to only issue chained resubmissions on a single file. This greatly simplifies address translation and access control, and it minimizes the metadata that we need to push down to the NVMe driver (the *metadata digest*, §4.1.3).
- **Stable data structures.** XRP targets data structures, whose layout (i.e. pointers) remain immutable for a long period of time (i.e. seconds or more). Such data structures include the indices used in many popular commercial storage engines, such as RocksDB [44], LevelDB [12], TokuDB [12] and WiredTiger [27]. Since the cost of im-

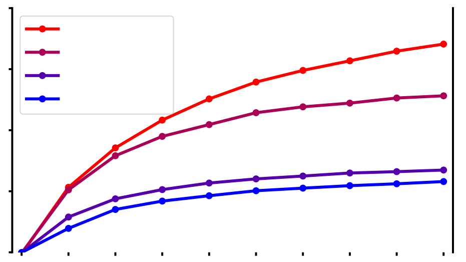


图3： 使用NVMe驱动程序钩子的单线程查找，使用io_uring系统调用。

在内核中的任何位置放置eBPF钩子都可能提高*1.2–2.*之间的吞吐量。然而，将I/O调度尽可能靠近存储设备可以显著提高遍历的性能。因此为了获得最高的加速效果，*XRP*的重提交钩子应该驻留在*NVMe*驱动程序。

那么io_uring呢？ io_uring是一个新的Linux系统调用框架 [9] 它允许进程提交一批异步I/O请求，从而摊销用户-内核交叉。然而，使用io_uring提交的每个I/O仍然会通过表1中所示的所有层，因此每个单独的I/O仍然会产生完整的存储堆栈开销。实际上，BPF I/O重提交与io_uring很大程度上是互补的：io_uring可以高效地提交触发内核中由BPF管理的不同I/O链路的一批I/O。

图 3显示了在使用 NVMe 驱动中的 BPF 钩子时 io_-uring 的吞吐量提升。I/O 链长度表示总的 I/O 数量，包括初始 I/O 和重新提交的 I/O。图 3显示了 BPF 相对于 io_uring 可以将吞吐量提高高达 1。5×对于小批处理大小和高达 2。5×随着批处理大小的增加。

总之，BPF 可以使传统的读取和 io_-uring 系统调用都受益。通过在内核 NVMe 驱动中放置钩子，BPF 可能将传统的 I/O 和单线程 io_uring 的吞吐量提高高达 2.5×。

3 设计挑战和原则

如前一节所示，I/O 重新提交必须尽可能靠近设备发生，以获得最大的收益。在 NVMe 软件栈中，这是 NVMe 中断处理程序。然而，在缺乏文件系统层上下文的情况下，从 NVMe 中断处理程序中执行重新提交，引入了两个主要挑战。

挑战 1：地址转换和安全。 NVMe 驱动程序无法访问文件系统元数据。在索引遍历的示例中，XRP 向特定块发出读取 I/O，并执行一个 BPF 函数，该函数将提取它希望查询的下一个块的偏移量。然而，这个偏移量对 NVMe 层来说没有意义，因为它无法在没有访问文件元数据和扩展的情况下告诉偏移量对应哪个物理块需要访问文件的元数据和扩展。即使应用程序开发人员付出了努力来嵌入物理块地址以避免文件系统偏移量的转换，这将是繁琐的，BPF 函数可以访问任何 设备上的块，包括属于用户没有访问权限的文件的那些块。

挑战 2：并发和缓存。 使用 XRP 启用来自文件系统的并发读取和写入具有挑战性。来自文件系统的写入仅在页面缓存中反映，而 XRP 无法看到。此外，任何并发发出以修改数据结构布局（例如，修改指向下一个块的指针）的写入都可能使 XRP 意外地获取错误的数据。这两者都可以通过锁定来解决，但从 NVMe 中断处理程序中访问锁可能很昂贵。

观察：大多数磁盘上的数据结构是稳定的。这两个挑战都会使实现任意的并发 BPF 存储函数变得困难。然而，我们观察到许多存储引擎（例如 LSM 树和 B 树）的文件保持相对稳定。一些数据结构根本不会原地修改磁盘上的存储结构。例如，一旦 LSM 树将其索引文件（称为 SSTables）写入磁盘，它们就会在删除之前保持不可变 [12, 27, 44]。访问这些不可变的磁盘存储结构需要较少的同步工作。类似地，即使某些磁盘上的 B 树索引实现支持原地更新，它们的文件范围在长时间内保持稳定。我们在 MariaDB 运行 TokuDB 上进行了一项 24 小时的 YCSB [41](40% 读取，40% 更新，20% 插入，Zipfian 0.7) 实验 [20],，该实验使用分形树（一种磁盘上的 B 树变体）作为其查找索引。我们发现索引文件的范围平均每 159 秒才改变一次，24 小时内只有 5 次范围变化会卸载任何块，这使得在没有频繁更新开销的情况下，可以在 NVMe 驱动器中缓存文件系统元数据。我们还观察到，在这些所有存储引擎中，索引存储在少量大文件上，并且每个索引不会跨越多个文件。

设计原则。 这些观察和实验形成了以下设计原则。

- **一次一个文件。** 我们最初将 XRP 限制为仅对单个文件发出链式重提交。这极大地简化了地址转换和访问控制，并最大限度地减少了我们需要向下推送到 NVMe 驱动的元数据（元数据摘要，§4.1.3）。
- **稳定的数据结构。** XRP 目标是那些布局（即指针）在长时间内保持不可变的数据结构（即秒或更长时间）。此类数据结构包括许多流行商业存储引擎中使用的索引，例如 RocksDB [44],LevelDB [12],TokuDB [12] 和 WiredTiger [27]。由于不可变

plementing locks in the NVMe layer is high, we also initially do not plan to support operations that require locks during the traversal or iteration of data structures.

- **User-managed caches.** XRP does not interface with the page cache, so XRP functions cannot safely be run concurrently if blocks are buffered in the kernel page cache. This constraint is acceptable since popular storage engines often implement their own user space caches [20,27,39,44]; Commonly they do this to fine-tune their caching and prefetching policies and to cache data in an application-meaningful way (e.g., cache key-value pairs or database rows instead of physical blocks).
- **Slow path fallback.** XRP is best-effort; if a traversal fails for some reason (e.g., the extent mappings become stale), the application must retry or fall back to dispatching the I/O requests using user space system calls.

4 XRP Design and Implementation

This section presents XRP's design and implementation with Linux eBPF and ext4. We describe the kernel modifications that enable XRP's resubmission logic in the interrupt handler, and how applications are modified to use XRP. We also discuss XRP's synchronization and scheduling limitations.

4.1 Resubmission Logic

The core of XRP augments the NVMe interrupt handler with resubmission logic that consists of a BPF hook, a file system translation step, and the construction and resubmission of the next NVMe request at the new physical offsets (Figure 4). Our modifications to the Linux kernel consist of ~900 lines of code: ~500 lines for the BPF hook and the changes to the NVMe driver, ~400 lines for the file system translation step.

When an NVMe request completes, the device generates an interrupt that causes the kernel to context switch into the interrupt handler. For each NVMe request that is completed in the interrupt context, XRP calls its associated BPF function (`bpf_func_0` in Figure 4), the pointer of which is stored in a field in the kernel I/O request struct (i.e. `struct bio`). After calling the BPF function, XRP invokes the metadata digest, which is usually a digest of file system state that enables XRP to translate the logical address of the next resubmission. Finally, XRP prepares the next NVMe command resubmission by setting the corresponding fields in the NVMe request, and it appends the request to the NVMe submission queue (SQ) for that core.

For a particular NVMe request, the resubmission logic is called as many times as necessary for subsequent completions as determined by the specific BPF function registered with the NVMe request. For example, for traversing a tree-like data structure, the BPF function would resubmit I/O requests for branch nodes and end resubmission whenever a leaf node is found. In our current prototype there is no hard limit on the number of resubmissions before the completion returns control to the application; such a limit would be necessary to

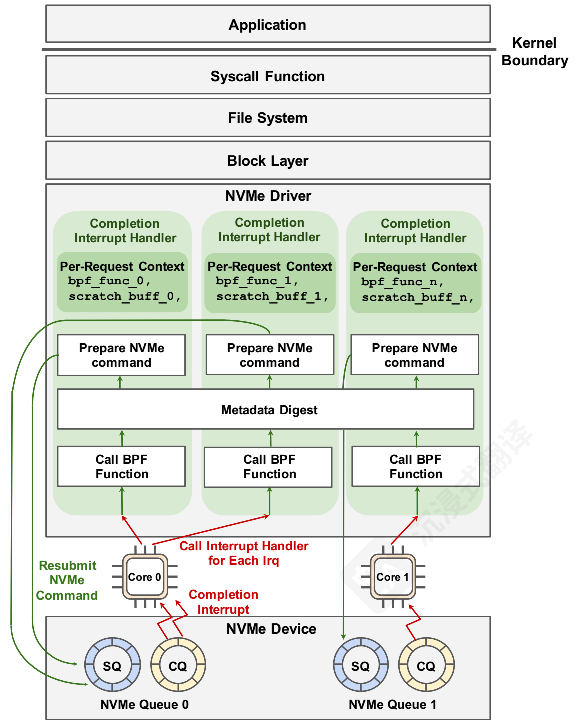


Figure 4: XRP architecture.

```
struct bpf_xrp {
  // Fields inspected outside BPF
  char *data;
  int done;
  uint64_t next_addr[16];
  uint64_t size[16];
  // Field for BPF function use only
  char *scratch;
};
uint32_t BPF_PROG_TYPE_XRP(struct bpf_xrp *ctxt);
```

Listing 1: Signature of BPF programs that can be loaded by XRP.

prevent unbounded execution. A hard limit can be enforced by maintaining a resubmission counter in each I/O request descriptor. Since I/O request descriptors cannot be accessed from user space or from XRP's BPF programs, their hard resubmission limits cannot be overridden by users even if XRP has multiple BPF functions that execute request resubmissions. BPF function contexts are per-request, while the metadata digest is shared across *all* invocations of the interrupt handler across all cores. Safe concurrent access to the metadata digest relies on read-copy-update (RCU) (§4.1.3).

4.1.1 BPF Hook

XRP introduces a new BPF type (`BPF_PROG_TYPE_XRP`) with the signature shown in Listing 1 – any BPF function that matches the signature can be called from the hook. §5 presents one concrete BPF function matching this signature that is used in our application. For example, for on-disk data structure

在 NVMe 层中实现锁的成本很高，因此我们最初也不计划支持在数据结构遍历或迭代过程中需要锁的操作。

- **用户管理的缓存**。 XRP 不与页面缓存交互，因此如果块被缓存在内核页面缓存中，XRP 功能不能安全地并发运行。由于流行的存储引擎通常实现自己的用户空间缓存，因此这个约束是可以接受的 [20,27,39,44]；它们通常这样做是为了微调它们的缓存和预取策略，并以应用程序有意义的方式缓存数据（例如，缓存键值对或数据库行，而不是物理块）。
- **慢速路径回退**。 XRP 是尽力而为的；如果由于某种原因（例如，范围映射变得过时）遍历失败，应用程序必须重试或回退到使用用户空间系统调用来分发 I/O 请求。

4 XRPDesign andImplementation

本节介绍了使用 Linux eBPF 和 ext4 的 XRP 的设计和实现。我们描述了启用中断处理程序中 XRP 重提交逻辑的内核修改，以及应用程序如何修改以使用 XRP。我们还讨论了 XRP 的同步和调度限制。

4.1 重新提交逻辑

XRP 的核心通过在 NVMe 中断处理程序中添加重新提交逻辑来增强，该逻辑由一个 BPF 钩子、一个文件系统转换步骤以及在新物理偏移量处构建和重新提交下一个 NVMe 请求组成（图 4）。我们对 Linux 内核的修改包括 ~900 行代码: ~500 行用于 BPF 钩子和 NVMe 驱动程序的更改， ~400行用于文件系统转换步骤。

当 NVMe 请求完成时，设备生成一个中断，导致内核切换到中断处理程序。对于在每个中断上下文中完成的每个 NVMe 请求， XRP 调用其关联的 BPF 函数（`bpf_func_0` 如图 4所示），其指针存储在内核 I/O 请求结构体的一个字段中（即 `struct bio`）。调用 BPF 函数后， XRP 调用元数据摘要，该摘要通常是文件系统状态的摘要，使 XRP 能够转换下一个重新提交的逻辑地址。最后， XRP 通过在 NVMe 请求中设置相应的字段来准备下一个 NVMe 命令的重新提交，并将请求附加到该核心的 NVMe 提交队列（SQ）。

对于特定的 NVMe 请求，重提交逻辑会根据与 NVMe 请求注册的特定 BPF 函数确定的后续完成次数被调用任意次数。例如，对于遍历树状数据结构， BPF 函数会重提交分支节点的 I/O 请求，并在找到叶节点时停止重提交。在我们的当前原型中，完成返回到应用程序之前没有重提交次数的硬限制；这样的限制是必要的，以

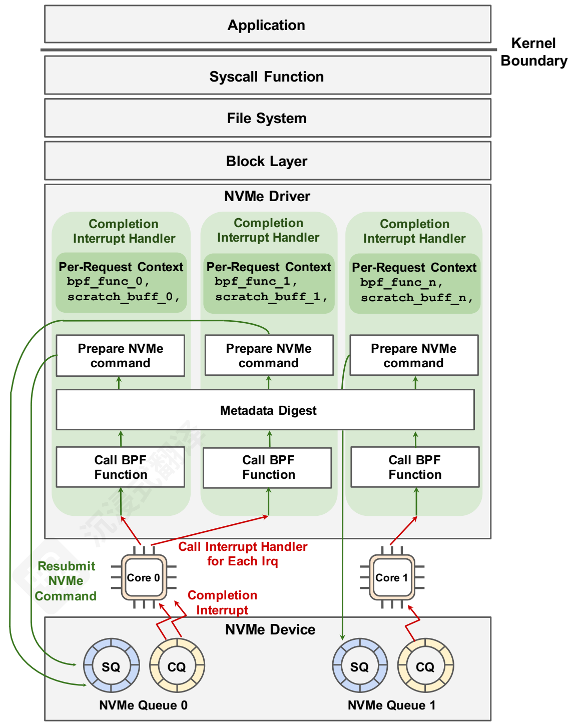


图 4： XRP 架构。

```
struct bpf_xrp {
  // Fields inspected outside BPF
  char *data;
  int done;
  uint64_t next_addr[16];
  uint64_t size[16];
  // Field for BPF function use only
  char *scratch;
};
uint32_t BPF_PROG_TYPE_XRP(struct bpf_xrp *ctxt);
```

清单 1： 可以被 XRP 加载的 BPF 程序的签名。

防止无界执行。可以通过在每个 I/O 请求描述符中维护一个重提交计数器来强制执行硬限制。由于 I/O 请求描述符不能从用户空间或从 XRP 的 BPF 程序中访问，即使 XRP 有多个执行请求重提交的 BPF 函数，用户也无法覆盖它们的硬重提交限制。BPF 函数上下文是按请求的，而元数据摘要是在所有核心的所有中断处理程序调用之间共享的。对元数据摘要的安全并发访问依赖于读-复制-更新 (RCU) (§ 所有 调用 4.1.3)。

4.1.1 BPF Hook

XRP 引入了一种新的 BPF 类型（`BPF_PROG_TYPE_XRP`），其签名如列表 1 所示——任何匹配该签名的 BPF 函数都可以从钩子中调用。 § 5presents one concrete BPF function matching this signature that is used in our application. For example, for on-disk data structure traversal, the BPF function typically contains logic to extract the next offset to fetch from the block.

BPF_PROG_TYPE_XRP programs require a context with five fields, categorized into fields that are inspected or modified by the BPF caller (resubmission logic in the interrupt handler), and fields that should be private to the BPF function. Fields that are accessed externally include data, which buffers data read from the disk (e.g., a B-tree page waiting to be parsed by the BPF function). done is a boolean that notifies the resubmission logic whether to return to the user or continue resubmitting I/O requests. next_addr and size are arrays of logical addresses and their corresponding sizes that indicate the next logical addresses for resubmission.

In order to support data structures with fanout, multiple next_addr values can be supplied. By default we limit fanout to 16; on-disk data structures align their components to small multiples of device pages, so we have not encountered a need for higher fanout per completion. For example, chained hash table buckets are likely implemented as a chain of individual physical pages and the elements of an on-disk linked list are likely implemented at the granularity of physical pages. Setting a corresponding size field to zero issues no I/O.

scratch is a scratch space that is private to the user and the BPF function. It can be used to pass the parameters from the user to the BPF function. Also, the BPF function can use it to store intermediate data in between I/O resubmissions and to return data to the user. For example, in the first BPF invocation, the application can store a search key in the scratch buffer so that the BPF function can compare it with the keys in the disk block in order to find the next offset. When the I/O chain reaches the leaf node of the B-tree, the BPF function then places the key-value pair in the scratch buffer to return it back to the application. For simplicity, we assume that the size of the scratch buffer is always 4 KB. We find that a 4 KB scratch buffer is sufficient to support a BPF function for a production key-value store (§5). BPF functions can also use BPF maps to store more data if their intermediate data cannot fit into the scratch buffer. Each BPF context is private to one NVMe request, so no locking is needed when working with BPF context state. Letting the user supply a scratch buffer (instead of using BPF map) avoids the overhead of processes and functions having to call bpf_map_lookup_elem to access the scratch buffer.

4.1.2 BPF Verifier

The BPF verifier ensures memory safety by tracking the semantics of the value stored in each register [14]. A valid value can either be a scalar or a pointer. SCALAR_TYPE represents a value that cannot be dereferenced. The verifier defines various pointer types; most of them include extra constraints beyond the no out-of-bound access requirement. For example, PTR_TO_CTX is the type for the pointer to a BPF context. It can only be dereferenced using a constant offset so the verifier can identify which context field a memory operation

```
void update_mapping(struct inode *inode);
void lookup_mapping(struct inode *inode,
                    off_t offset, size_t len,
                    struct mapping *result);
```

Listing 2: Metadata digest: XRP exposes an interface to share logical-to-physical-block mappings between the file system and the IRQ handler.

accesses. Each BPF function type also defines a callback function is_valid_access() to perform additional checks on context accesses and to return the value type of the context field. PTR_TO_MEM describes a pointer referring to a fixed-size memory region. It supports dereferencing using a variable offset as long as the access is always within bounds. The data and scratch fields of the BPF_PROG_TYPE_XRP context are PTR_TO_MEM and the rest are SCALAR_TYPE. We augment the verifier to allow the BPF_PROG_TYPE_XRP's is_valid_access() callback to pass the size of the data buffer or scratch buffer to the verifier so that it can perform the boundary check. We discussed our proposed modification to the verifier with the Linux eBPF maintainers, and they think it is sensible.

4.1.3 The Metadata Digest

In the conventional storage stack, the logical block offsets in on-disk data structures are translated by the file system in order to identify the next physical block to read. This translation step also enforces access control and security, preventing reading in regions that are not mapped to the open file. In XRP, the next logical address for a lookup is given by the next_addr field after the BPF call. However, translating this logical address to a physical address is challenging since the interrupt handler has no notion of a file and does not perform physical address translation.

To solve this, we implement the metadata digest, a thin interface between the file system and the interrupt handler that lets the file system share its logical-to-physical-block mappings with the interrupt handler, enabling safe eBPF-based on-disk resubmissions. The metadata digest consists of two functions (Listing 2). The update function is called within the file system when the logical-to-physical mapping is updated. The lookup function is called within the interrupt handler; it returns the mapping for a given offset and length. The lookup function also enforces access control by preventing BPF functions from requesting resubmissions for blocks outside of the open file. The inode address of the open file is passed to the interrupt handler in order to query the metadata digest. If an invalid logical address is detected, XRP returns to user space immediately with an error code. The application can then fall back to normal system calls to attempt its request again.

These two functions are specific to each file system, and even for a particular file system, there may be multiple ways to implement the metadata digest, presenting a tradeoff between ease of implementation and performance. For example, in

遍历时，BPF 函数通常包含逻辑以提取下一个偏移量以从块中获取数据。

BPF_PROG_TYPE_XRP 程序需要一个包含五个字段的上下文，这些字段分为 BPF 调用者检查或修改的字段（中断处理程序中的重新提交逻辑）和应为 BPF 函数私有的字段。外部访问的字段包括 data，它缓冲从磁盘读取的数据（例如，等待 BPF 函数解析的 B 树页）。 done 是一个布尔值，通知重新提交逻辑是返回用户还是继续重新提交 I/O 请求。 next_addr 和 size 是逻辑地址及其对应大小的数组，指示重新提交的下一个逻辑地址。

为了支持具有分叉的数据结构，可以提供多个 next_addr 值。默认情况下，我们将分叉限制为 16；磁盘上的数据结构将其组件对齐到设备页的小倍数，因此我们还没有遇到需要每完成更高的分叉的需求。例如，链式哈希表桶可能实现为单个物理页面的链，磁盘上链表的元素可能以物理页面的粒度实现。将相应的尺寸字段设置为零则不发出 I/O。

scratch 是一个仅用户和BPF函数私有的临时空间。它可用于将用户传递给BPF函数的参数。此外，BPF函数还可以使用它来在I/O重新提交之间存储中间数据，并将数据返回给用户。例如，在第一次BPF调用中，应用程序可以将搜索键存储在临时缓冲区中，以便BPF函数可以将其与磁盘块中的键进行比较，以找到下一个偏移量。当I/O链到达B树的叶节点时，BPF函数然后将键值对放置在临时缓冲区中以将其返回给应用程序。为简单起见，我们假设临时缓冲区的大小始终为4 KB。我们发现4 KB的临时缓冲区足以支持生产键值存储的 BPF函数（§5）。BPF函数还可以使用BPF映射来存储更多数据，如果它们的中间数据无法适应临时缓冲区。每个BPF上下文仅属于一个NVMe请求，因此在处理BPF上下文状态时无需锁定。让用户提供临时缓冲区（而不是使用BPF映射）避免了进程和函数必须调用 bpf_map_lookup_elem 来访问临时缓冲区的开销。

4.1.2 BPF 验证器

BPF 验证器通过跟踪每个寄存器中存储的值的语义来确保内存安全 [14]。一个有效的值可以是标量或指针 SCALAR_TYPE 表示一个无法解引用的值。验证器定义了各种指针类型；它们中的大多数除了没有越界访问要求之外还包括额外的约束。例如 PTR_TO_CTX 是 BPF 上下文指针的类型。它只能使用常量偏移量进行解引用，因此验证器可以识别内存操作访问的是哪个上下文字段

```
void update_mapping(struct inode *inode);
void lookup_mapping(struct inode *inode,
                    off_t offset         , size_t len,

struct mapping *result);
```

表2: 元数据摘要：XRP 提供了一个接口，用于在文件系统和 IRQ 处理器之间共享逻辑到物理块映射

访问。每个 BPF 函数类型还定义了一个回调函数 is_valid_access() 来对上下文访问进行额外检查并返回上下文字段的值类型。 PTR_TO_MEM 描述了一个指向固定大小内存区域的指针。只要访问始终在边界内，它支持使用可变偏移量进行解引用。data 和 scratch 字段的 BPF_PROG_TYPE_XRP 上下文是 PTR_TO_MEM ，其余的是 SCALAR_TYPE。我们增强了验证器，允许 BPF_PROG_TYPE_XRP 的 is_valid_ac-cess() 回调将数据缓冲区或暂存缓冲区的大小传递给验证器，以便它执行边界检查。我们与 Linux eBPF 维护人员讨论了我们提出的验证器修改，他们认为这是合理的。

4.1.3 元数据摘要

在传统的存储堆栈中，磁盘数据结构中的逻辑块偏移量由文件系统进行转换，以识别下一个要读取的物理块。此转换步骤还执行访问控制和安全性检查，防止读取未映射到打开文件的区域。在XRP中，查找的下一个逻辑地址由BPF调用后的next_addr 字段给出。然而，将此逻辑地址转换为物理地址具有挑战性，因为中断处理程序没有文件的概念，并且不执行物理地址转换。

为了解决这个问题，我们实现了元数据摘要，这是文件系统和中断处理程序之间的一个薄接口，允许文件系统与中断处理程序共享其逻辑到物理块的映射，从而实现基于eBPF的安全磁盘重新提交。元数据摘要由两个函数组成（列表 2）。更新函数在文件系统内调用，当逻辑到物理映射更新时。查找函数在中断处理程序内调用；它返回给定偏移量和长度的映射。查找函数还通过防止BPF函数请求打开文件外块的重新提交来执行访问控制。打开文件的inode地址传递给中断处理程序，以便查询元数据摘要。如果检测到无效的逻辑地址，XRP会立即返回用户空间并带有错误代码。然后应用程序可以回退到正常系统调用以再次尝试其请求。

这两个函数是针对每个文件系统的，即使对于特定的文件系统，元数据摘要也可能存在多种实现方式，这需要在实现的简便性和性能之间进行权衡。例如，在

our implementation for ext4, the metadata digest consists of a cached version of the extent status tree, which stores the physical-to-logical block mappings. This cached tree is accessed by the update and lookup function of the interface, and it uses read-copy-update (RCU) for concurrency control. RCU enables the lookup function to be lockless and fast (96 ns on average).

To keep the cached tree up-to-date with the extents in ext4, the update function is called in two places in ext4: whenever extents are inserted or removed from the main extent tree. To prevent a race condition where an extent is modified while there is an inflight read on it, we maintain a version number for each extent to track its changes. After data is read, but before it is passed to the BPF function, a second metadata digest lookup is performed. If the corresponding extent no longer exists or its version number has changed, XRP will abort the operation. Since application-level synchronization usually prevents concurrent modifications and lookups on the same region of a file at the same time, version mismatches should only occur if the application is buggy or malicious.

An alternative, simpler implementation of the metadata digest for ext4 could simply pass through to existing update and access functions of the extent tree in ext4. In this case, the update function would be a no-op, because ext4 already keeps its extent tree up-to-date. However, such an implementation would be much slower on the lookup path, because the extent lookup function in ext4 acquires a spinlock, which would be prohibitively expensive in the interrupt handler.

For now, XRP only supports the ext4 file system, but the metadata digest can be easily implemented for other file systems. For example, in f2fs [64], logical-to-physical-block mappings are stored in the node address table (NAT). Similar to the ext4 implementation, an implementation of its metadata digest could cache a local copy of the NAT, which would be consulted in `lookup_mapping`. Then `update_mapping` would need to be called anywhere in f2fs where the NAT is updated.

4.1.4 Resubmitting NVMe Requests

After looking up the physical block offsets, XRP prepares the next NVMe request. Because this logic occurs in the interrupt handler, to avoid the (slow) `kmalloc` calls needed to prepare NVMe requests, XRP reuses the existing NVMe request struct (i.e. `struct nvme_command`) of the just-completed request. XRP simply updates the physical sector and block addresses of the existing NVMe request to the new offsets derived from the mapping lookup. Reusing NVMe request structs for immediate resubmission is safe because neither user space nor XRP BPF programs can access the raw NVMe request structs.

While `struct bpf_xrp` supports a maximum fanout of 16, in the current implementation a resubmitted I/O request can only fetch as many physical segments as the initial NVMe request. For example, if an initial NVMe request only fetches a single block, then all subsequent resubmissions for that request can only fetch a single physical segment. During a resubmission chain, if the BPF call returns multiple valid addresses in `next_addr`, XRP will abort the request. This limitation can be worked around by allocating and setting up 16 dummy NVMe commands in the first I/O request so that subsequent resubmissions can express fanout if necessary.

4.2 Synchronization Limitations

BPF currently only supports a limited spinlock for synchronization. The verifier only allows BPF programs to acquire one lock at a time, and they must release the lock before returning. Also, user space applications do not have direct access to these BPF spinlocks. Instead, they must invoke the `bpf()` syscall; the syscall can read or write the lock-protected structure while holding the lock for the duration of that operation. Hence, complex modifications that require synchronizing across multiple reads and writes cannot be accomplished in user space.

Users can implement custom spinlocks using BPF atomic operations. This allows both BPF functions and user space programs to acquire any spinlock directly. However, the termination constraint prohibits BPF functions from spinning to wait for a spinlock infinitely. Another option for synchronization is RCU. Since XRP BPF programs are run in the NVMe interrupt handler, which cannot be preempted, de-facto they are already in an RCU read-side critical section.

4.3 Interaction with Linux Schedulers

Process scheduler. Interestingly, we observed that a microsecond-scale storage device like Optane SSD interferes with Linux's CFS when multiple processes share the same core, *even when all I/O is issued from user space*. For example, in the case where an I/O-heavy and compute-heavy process share the same core, the I/O interrupts generated by the I/O-heavy process will be handled in the timeslice of the compute-heavy process. This may cause the compute-heavy process to be starved of CPU; in the worst case in our experiments, the compute-heavy process only received about 34% of what would be a "fair" allocation of CPU time. We experimentally verified this does not occur when using a slower storage device, which generates interrupts much less frequently. While XRP exarcerbates this problem by generating chains of interrupts, this issue is not specific to eBPF, and can also be caused by network-driven interrupts [59]. We leave this problem for future work.

I/O scheduler. XRP bypasses Linux's I/O scheduler, which sits at the block layer. However, the noop scheduler is already the default I/O scheduler for NVMe devices, and the NVMe standard supports arbitration at hardware queues if fairness is a requirement [17].

5 Case Studies

To use XRP, applications use the interface shown in Listing 3. Applications call libbpf [13] function `bpf_prog_load` to load a BPF function of type `BPF_PROG_TYPE_XRP` to be offloaded

我们为 ext4 的实现，元数据摘要由一个缓存的扩展状态树组成，该树存储了物理到逻辑块的映射。此缓存树由接口的更新和查找函数访问，并使用读-复制-更新 (RCU) 进行并发控制。RCU 使查找函数能够无锁且快速（平均 96 ns）。

为了使缓存树与 ext4 中的扩展保持最新，更新函数在 ext4 中有两个调用位置：每当扩展被插入或从主扩展树中移除时。为了防止在扩展被修改时存在 inflight 读取的情况，我们为每个扩展维护一个版本号以跟踪其变化。在数据读取后但在传递给 BPF 函数之前，会进行第二次元数据摘要查找。如果相应的扩展不再存在或其版本号已更改，XRP 将中止操作。由于应用级别的同步通常防止在同一时间对文件同一区域进行并发修改和查找，版本不匹配的情况应仅发生在应用程序有错误或恶意行为时。

ext4 的元数据摘要的另一种更简单的实现方案可以直接通过到 ext4 扩展树现有的更新和访问函数。在这种情况下，更新函数将是一个空操作，因为 ext4 已经保持其扩展树是最新的。然而，这种实现方案在查找路径上会慢得多，因为 ext4 中的扩展查找函数会获取一个自旋锁，这在中断处理程序中将是极其昂贵的。

目前，XRP 仅支持 ext4 文件系统，但元数据摘要可以很容易地为其他文件系统实现。例如，在 f2fs [64],逻辑到物理块映射存储在节点地址表 (NAT) 中。类似于 ext4 的实现，其元数据摘要的实现可以缓存 NAT 的本地副本，该副本将在 `lookup_mapping`中查询。然后 `update_mapping` 需要在 f2fs 中任何更新 NAT 的地方调用它。

4.1.4 重新提交 NVMeRequests

在查询物理块偏移量后，XRP 准备下一个 NVMe 请求。由于此逻辑发生在中断处理程序中，为了避免准备 NVMe 请求所需的（较慢的） `kmalloc` 调用，XRP 重用刚刚完成的请求的现有 NVMe 请求结构（即 `struct nvme_command`）。XRP 仅将现有 NVMe 请求的物理扇区和块地址更新为从映射查询中派生的新偏移量。由于用户空间或 XRP BPF 程序都无法访问原始 NVMe 请求结构，因此立即重新提交 NVMe 请求结构是安全的。

虽然 `struct bpf_xrp` 支持最大 16 个 fanout，但在当前实现中，重新提交的 I/O 请求只能获取与初始 NVMe 请求相同的物理段数量。例如，如果初始 NVMe 请求仅获取单个块，则针对该请求的所有后续重新提交都只能获取单个物理段。在重新提交链中，如果 BPF 调用在 `struct bpf_xrp` 中返回多个有效地址，XRP 将中止请求。此限制可以通过在第一个 I/O 请求中分配并设置 16 个虚拟 NVMe 命令来绕过，以便后续重新提交在必要时可以表达 fanout。

在重新提交链中，如果 BPF 调用在 `next_addr`中返回多个有效地址，XRP 将中止请求。此限制可以通过在第一个 I/O 请求中分配并设置 16 个虚拟 NVMe 命令来绕过，以便后续重新提交在必要时可以表达 fanout。

4.2 同步限制

BPF 目前仅支持有限的自旋锁用于同步。验证器仅允许 BPF 程序一次获取一个锁，并且必须在返回前释放该锁。此外，用户空间应用程序不能直接访问这些 BPF 自旋锁。相反，它们必须调用`bpf()` 系统调用；该系统调用可以在持有锁期间读取或写入受锁保护的结 构。因此，需要跨多个读和写进行同步的复杂修改无法在用户空间完成。

用户可以使用 BPF 原子操作实现自定义自旋锁。这允许 BPF 函数和用户空间程序直接获取任何自旋锁。然而，终止约束禁止 BPF 函数无限期地自旋等待自旋锁。同步的另一个选项是 RCU。由于 XRP BPF 程序在 NVMe 中断处理程序中运行，该处理程序不能被抢占，实际上它们已经处于 RCU 读侧临界区。

4.3 与 Linux 调度器的交互

进程调度器。 有趣的是，我们观察到像 Optane SSD 这样的微秒级存储设备在多个进程共享同一核心时会干扰 Linux 的 CFS，即使所有 *I/O* 都是从用户空间发出的。例如，在 I/O 密集型和计算密集型进程共享同一核心的情况下，I/O 密集型进程产生的 I/O 中断将在计算密集型进程的时间片内被处理。这可能导致计算密集型进程缺乏 CPU；在我们的实验中最坏的情况下，计算密集型进程仅获得了大约 34% 的"公平"分配的 CPU 时间。我们通过实验验证了当使用更慢的存储设备时，该问题不会发生，因为更慢的设备产生的中断频率要低得多。虽然 XRP 通过生成中断链加剧了这个问题，但这个问题并非 eBPF 特有，也可能由网络驱动中断 [59] 引起。我们将此问题留待未来研究。

I/O 调度器。 XRP 绕过 Linux 的 I/O 调度器，后者位于块层。然而，noop 调度器已经是 NVMe 设备的默认 I/O 调度器，并且 NVMe 标准在硬件队列中支持仲裁，如果需要公平性 [17]。

5 案例研究

要使用 XRP，应用程序使用列表中所示的接口 3。应用程序调用 libbpf [13] 函数 `bpf_prog_load` 来加载类型为 `BPF_PROG_TYPE_XRP` 的 BPF 函数以进行卸载

```
int bpf_prog_load(const char *file,
                  enum bpf_prog_type type,
                  struct bpf_object **pobj,
                  int *prog_fd);
int read_xrp(int fd, void *buf, size_t count,
             off_t offset, int bpf_fd,
             void *scratch);
```

Listing 3: The XRP application interface consists of a libbpf function to load BPF functions into the kernel and a read syscall that requests that a BPF function be used. `bpf_prog_load` is an existing function in libbpf. `bpf_prog_load` returns a file descriptor for the loaded function, which must be passed to `read_xrp`. `read_xrp` adds two arguments to the standard `pread` [21] syscall: this file descriptor and a pointer to a 4 KB scratch space that is passed to the BPF context.

in the driver and call `read_xrp` to apply a specific BPF function to the read request. Applications can load multiple BPF functions with XRP. For example, a database can load a function for filtering and calculating aggregations from values on-disk and a function for `GET` point lookups. XRP allows the application to load multiple BPF functions into the kernel and to specify the BPF function to use in each `read_xrp` syscall. We present two case studies on how applications should be modified to use XRP.

5.1 BPF-KV

We built a simple key-value store, called BPF-KV, with which we can evaluate XRP against other baselines: Linux's synchronous and asynchronous system calls and kernel bypass (SPDK [82]). BPF-KV is designed to store a large number of small objects and to provide good read performance even under uniform access patterns. BPF-KV uses a B^+-tree index to find the location of objects, and the objects themselves are stored in an unsorted log. For simplicity, BPF-KV uses fixed-sized keys (8 B) and values (64 B). The index and the log are both stored in one large file. The index nodes use a simple page format with a header followed by keys followed by values. Leaf nodes contain a file offset pointing to the next leaf node, enabling efficient index traversal for range queries and aggregation. Object sizes are fixed, so updates occur in-place in the unsorted log. Newly inserted items are appended to the log; their index is initially stored in an in-memory hash table. Once the hash table fills, BPF-KV merges it with the on-disk B^+-tree file.

Caching. BPF-KV implements a user space DRAM cache for index blocks and objects. To reduce the number of I/Os it needs to issue for lookups, BPF-KV caches the top *k* levels of the B^+-tree index. With a sufficiently large number of objects, it is not possible to fit the entire index in the cache. Consider the case where BPF-KV is used to store 10 billion 64 B objects. In BPF-KV's index, each node is 512 B (matching the access granularity of the Optane SSD); hence, the tree has a fanout of 31 (i.e. each internal node can store pointers to 31 children). Therefore, 10 billion objects would require an index with 8 levels. Fitting 6 index levels in DRAM is expensive and would require 14 GB, while fitting 7 levels or more becomes prohibitively expensive (437 GB of DRAM or more). So, to support a large number of keys, BPF-KV would require at the minimum 3-4 I/Os from storage for each lookup, including a final I/O to fetch the actual key-value pair from disk. Also note that having a hard memory budget for caching the index is common in many real-world key-value stores (e.g., RocksDB [45], DocumentDB [78], SplinterDB [40], TokuDB [20]), since the index cache often competes with other parts of the system that need memory, such as filters and the object cache.

BPF-KV also maintains a least recently used (LRU) object cache of the most popular key-value pairs. Before looking up an object on disk, BPF-KV first checks whether it is stored in the object cache. If not, it checks whether it is indexed in the in-memory hash table. If the item is not found in the in-memory hash table, it looks up the object by accessing the first *k* cached levels of the index. Once it encounters an index node that is not cached, it completes the index and the final lookup on disk.

To find an object without XRP, BPF-KV traverses the B-tree until the desired value is found using an I/O request per level. For example, if the index contains 7 levels and the first 3 are cached and read from DRAM, then the traversal will issue 4 I/Os to navigate the rest of the tree, followed by a final I/O to fetch the object from the log.

BPF function. Listing 4 shows the BPF function used in BPF-KV to lookup a key-value pair. We omit the code to handle the final lookup in the log for simplicity. `struct node` defines the layout of B^+-tree index nodes whose size is 512 B. The BPF function `bpfkv_bpf` first extracts the target key stored in the scratch buffer, and then it linearly searches the slots in the current node to find the next node to read.

Interface modifications. We replace `read` calls with `read_xrp`. Before calling into `read_xrp`, BPF-KV first allocates a buffer for the scratch space and calculates the offset at which to start the lookup.

Range queries. BPF-KV supports range queries returning a variable number of objects. We implement a BPF function that runs as a state machine, allowing the operation to be suspended and resumed when objects are returned to the application for processing. The BPF function state, including the beginning and end of the range, and the retrieved objects, are stored in the scratch space (up to 32 72-byte key-value pairs). On the initial invocation, the function traverses to the leaf node that contains the starting key. Once the first key in the range is found, the function stores the leaf node in the scratch space and requests the block containing the corresponding value. On the next BPF invocation, the function stores the value in the scratch space and it continues the index scan

```
int bpf_prog_load(const char *file,
                  int *prog_fd);
enum read_bpf_xrpog(int type fd, type, void struct *buf, bpf size_object count **pobj,
             off_t offset  int bpf_fd         ,,


void *scratch);
```

列表 3: XRP 应用接口由一个用于将 BPF 函数加载到内核的 libbpf 函数和一个请求使用 BPF 函数的读取系统调用组成。 `bpf_prog_load` 是 libbpf 中现有的一个函数。 `bpf_prog_load` 返回加载函数的文件描述符，该文件描述符必须传递给 `read_xrp`。 `read_xrp` 向标准 `pread` [21] 系统调用添加了两个参数：这个文件描述符和一个指向 4 KB 临时空间的指针，该临时空间传递给 BPF 上下文。

在驱动程序中，并调用 `read_xrp` 将特定的 BPF 函数应用于读取请求。应用程序可以使用 XRP 加载多个 BPF 函数。例如，数据库可以加载一个用于过滤和计算磁盘上值的聚合函数，以及一个用于 `GET` 点查找的函数。XRP 允许应用程序将多个 BPF 函数加载到内核中，并指定在每个 `read_xrp` syscalls 中使用的 BPF 函数。我们介绍了两个案例研究，说明应用程序应如何修改以使用 XRP。

5.1 BPF-KV

我们构建了一个简单的键值存储，称为BPF-KV，通过它可以评估XRP相对于其他基线的性能：Linux的同步和异步系统调用以及内核绕过（SPDK [82]）。BPF-KV的设计目的是存储大量小对象，即使在均匀访问模式下也能提供良好的读取性能。BPF-KV使用B^+-树索引来查找对象的位置，而对象本身则存储在一个无序日志中。为了简化，BPF-KV使用固定大小的键（8 B）和值（64 B）。索引和日志都存储在一个大文件中。索引节点使用简单的页面格式，包含一个头部，后面跟着键，再跟着值。叶节点包含一个指向下一个叶节点的文件偏移量，从而能够高效地进行范围查询和聚合。对象大小是固定的，因此更新是在无序日志中就地进行的。新插入的项目被追加到日志中；它们的索引最初存储在内存中的哈希表中。一旦哈希表填满，BPF-KV就会将其与磁盘上的B^+-树文件合并。

缓存。 BPF-KV 实现了一个用户空间的 DRAM 缓存，用于索引块和对象。为了减少查找时需要发出的 I/O 次数，BPF-KV 缓存了 B^+-树索引的顶部 *k* 层。当对象数量足够多时，不可能将整个索引放入缓存中。考虑 BPF-KV 用于存储 10 亿个 64 B 对象的情况。在 BPF-KV 的索引中，每个节点是 512 B（与 Optane SSD 的访问粒度匹配）；因此，树的扇出为 31（即每个内部节点可以存储指向 31 个子节点的指针）。因此，10 亿个对象需要一个 8 层的索引。将 6 个索引级别放入 DRAM 是昂贵的，需要 14 GB，而将 7 层或更多级别放入则变得非常昂贵（437 GB 的 DRAM 或更多）。因此，为了支持大量的键，BPF-KV 至少需要每个查找 3-4 次来自存储的 I/O，包括最终从磁盘获取实际键值对的 I/O。还请注意，为索引缓存设置硬性内存预算在许多现实世界的键值存储中很常见（例如，RocksDB [45], DocumentDB [78],SplinterDB [40], TokuDB [20]），因为索引缓存经常与其他需要内存的系统部分竞争，例如过滤器和对 tượng 缓存。

BPF-KV 还维护一个最近最少使用（LRU）的对象缓存，其中包含最常用的键值对。在从磁盘查找对象之前，BPF-KV 首先检查它是否存储在对象缓存中。如果没有，它检查是否在内存哈希表中索引。如果该条目未在内存哈希表中找到，它通过访问索引的前 *k* 个缓存级别来查找对象。一旦遇到未缓存的索引节点，它就会完成索引并在磁盘上完成最终查找。

要在没有 XRP 的情况下查找对象，BPF-KV 会遍历 B-树，每层使用一个 I/O 请求直到找到所需值。例如，如果索引包含 7 层，其中前 3 层被缓存并从 DRAM 中读取，那么遍历将发出 4 个 I/O 来导航树的其余部分，然后发出一个最终的 I/O 来从日志中获取对象。

BPF函数。 列表 4 显示了BPF-KV中用于查找键值对的BPF函数。为了简化，我们省略了在日志中处理最终查找的代码。 `structnode` 定义了大小为512 B的B树索引节点的布局。BPF函数 `bpfkv_bpf` 首先提取存储在暂存缓冲区中的目标键，然后它线性搜索当前节点的槽以找到下一个要读取的节点。

接口修改。 我们用`read_xrp`替换 `read` 调用。在调用 `read_xrp`之前，BPF-KV首先为暂存空间分配一个缓冲区，并计算查找的起始偏移量。

范围查询。 BPF-KV支持返回可变数量对象的范围查询。我们实现了一个BPF函数，该函数作为状态机运行，允许在对象返回应用程序进行处理时暂停和恢复操作。BPF函数状态，包括范围的开始和结束，以及检索到的对象，都存储在暂存空间中（最多32个72字节的键值对）。在初始调用时，函数遍历包含起始键的叶节点。一旦找到范围内的第一个键，函数将叶节点存储在暂存空间中，并请求包含相应值的块。在下一个BPF调用中，函数将值存储在暂存空间中，并继续索引扫描

```
struct node {
  uint64_t num; uint64_t type;
  uint64_t key[31]; uint64_t ptr[31];
};
uint32_t bpfkv_bpf(struct bpf_xrp *ctxt) {
  uint64_t key = *((uint64_t*)ctxt->scratch);
  struct node *n = (struct node *)ctxt->data;
  if (n->type == LEAF_NODE) {
    ctxt->done = true;
    return 0;
  }
  int i;
  for (i = 1; i < n->num; i++)
    if (key < n->key[i]) break;
  ctxt->done = false;
  ctxt->next_addr[0] = n->ptr[i - 1];
  ctxt->size[0] = 512;
  return 0;
}
```

Listing 4: BPF function for BPF-KV.

on the cached leaf node. When the leaf node has been read completely, the function submits a request for the next leaf node using the node's next-leaf file offset. The function returns to the application in three cases: 1) the function reaches a key past the end of the range; 2) the function reaches the end of the index; 3) the function fills the scratch space with values read from the log. In the last case, the application can process the values and re-invoke the BPF function with the range query state, allowing the range query to resume from where it left off.

Aggregations. BPF-KV also supports aggregation operations, such as `SUM`, `MAX` and `MIN`. We implement these operations on top of the BPF range query function by setting a bit that causes the function to perform the corresponding aggregation instead of returning the individual values. Since aggregation queries return a single answer, storing values in the scratch space does not limit the number of I/O resubmits the BPF function can request.

5.2 WiredTiger

WiredTiger is a popular key-value store that is the default backend for MongoDB [27]. We use it as a case study since it is a relatively simple and open key-value store that is used in production. WiredTiger provides an option to use an LSM tree where data is split into different levels; each level contains a single file. Each file uses a B-tree index with the key-value pairs embedded in the tree's leaf nodes. The files are read-only; updates and inserts are written into a buffer in memory. When the buffer is full, the data is written out in a new file. We configure the B-tree page size to be the same as our Optane SSD's block size (512 B). Our modification to WiredTiger is around 500 lines of code, which mainly consist of buffer allocation, extending function signatures and wrapping the XRP syscall. XRP helps accelerate reads that are serviced from disk, and it does not affect updates or inserts, which are always absorbed by WiredTiger's in-memory buffer.

BPF function. To use XRP, WiredTiger installs a BPF function similar to the one shown in Listing 4. The difference is in order to find the next lookup address from the current page, the BPF function contains a port of WiredTiger's B-tree page parsing code. This parsing logic replaces the `for` loop in Listing 4.

The WiredTiger BPF function also makes several modifications to make the BPF program compile correctly and pass the BPF verifier. The modifications mainly consist of adding bounds on loops to avoid infinite loops, masking pointers to eliminate out-of-bound access, and initializing local variables to prevent access to uninitialized registers. We also use the BPF function-by-function verification feature [3] to break a complex function into several simple sub-functions. This allows BPF functions to be verified independently, so the functions that have been verified do not need another round of verification when being called by other functions. The function-by-function verification feature also supports more complex BPF programs without exceeding the verifier's restrictions on function length.

Caching. WiredTiger maintains a least recently used (LRU) cache for its B-tree internal pages and leaf pages. When looking up a new key-value pair, WiredTiger caches the entire lookup path including the leaf page in the cache. In order to comply with WiredTiger caching semantics, the BPF function described in the previous section also returns all traversed pages so that WiredTiger can cache them. The BPF function stores traversed pages in the scratch buffer of its context. When the scratch buffer is exhausted, the BPF function will stop resubmitting requests and return to user space immediately. After WiredTiger adds those pages into its cache, it will call `read_xrp` again to continue the lookup starting at the previous page. Since we set the size of the scratch buffer to 4 KB, a BPF function can store up to 6 traversed 512 B pages in the scratch buffer, which leaves room for necessary metadata such as the search key.

Interface modifications. To integrate WiredTiger with XRP, we replace normal read calls with `read_xrp`. `read_-xrp` is called when the next page is not in the cache and needs to be read from disk. The eviction policy of WiredTiger enforces that only the pages without any cached children pages can be evicted, so any uncached page will not have cached descendants. Therefore, it is safe to call `read_xrp` to read all of the remaining path from disk without checking the application-level cache again. If `read_xrp` fails for any reason, WiredTiger falls back to the normal lookup path. We allocate a data and scratch buffer for each WiredTiger session to avoid the overhead of allocating and freeing buffers for every request. WiredTiger sessions synchronously process

```
struct node {uint64_t num; uint64_t type;
uint64_t key[31]; uint64_t ptr[31];};
uint32_t bpfkv_bpf(struct bpf_xrp *ctxt) {
uint64_t key = *((uint64_t*)ctxt->scratch);
struct node *n = (struct node *)ctxt->data;
if (n->type == LEAF_NODE) {

ctxt->done = true;return 0;}int i;
for (i = 1; i < n->num; i++)
if (key < n->key[i]) break;
ctxt->done = false;
ctxt->next_addr[0] = n->ptr[i - 1];
ctxt->size[0] = 512;return 0;}
```
Listing 4:BPF-KV的BPF函数。

在缓存的叶节点上。当叶节点被完全读取时，该函数使用节点的next-leaf文件偏移量提交对下一个叶节点的请求。该函数在三种情况下返回给应用程序：1) 函数达到范围末尾之后的一个键；2) 函数达到索引末尾；3) 函数用从日志中读取的值填充了暂存空间。在最后一种情况下，应用程序可以处理这些值并重新调用BPF函数，带有范围查询状态，允许范围查询从上次停止的地方继续。

聚合。 BPF-KV 还支持聚合操作，例如 `SUM`, `MAX` 和 `MIN`。我们通过设置一个位来在 BPF 范围查询函数上实现这些操作，使函数执行相应的聚合而不是返回单个值。由于聚合查询返回单个结果，将值存储在临时空间中不会限制 BPF 函数请求的 I/O 重提交次数。

5.2 WiredTiger

WiredTiger 是一个流行的键值存储，是 MongoDB 的默认后端 [27]。我们将其作为一个案例研究，因为它是一个相对简单且开源的键值存储，用于生产环境。WiredTiger 提供了一个使用 LSM 树的选项，其中数据被分成不同的级别；每个级别包含一个文件。每个文件使用一个 B 树索引，键值对嵌入在树的非叶子节点中。文件是只读的；更新和插入操作写入内存中的缓冲区。当缓冲区满时，数据会写入一个新文件。我们将 B 树页面大小配置为与我们的 Optane SSD 的块大小相同（512 B）。我们对 WiredTiger 的修改大约有 500 行代码，主要涉及缓冲区分配，扩展函数签名并包装 XRP 系统调用。XRP 帮助加速从磁盘服务的读取，并且它不会影响更新或插入，这些操作始终由 WiredTiger 的内存缓冲区吸收。

BPF 函数。 要使用 XRP，WiredTiger 安装一个类似于列表中所示的 BPF 函数 4。区别在于为了从当前页面找到下一个查找地址，BPF 函数包含 WiredTiger 的 B 树页面解析代码的一个端口。这个解析逻辑替换了 `for` 列表 44 中的循环。

WiredTiger 的 BPF 函数还对 BPF 程序进行了几个修改，以确保其正确编译并通过 BPF 验证器。这些修改主要包括在循环上添加边界以避免无限循环，掩码指针以消除越界访问，以及初始化局部变量以防止访问未初始化的寄存器。我们还使用 [3] 逐函数验证功能 将复杂函数拆分为多个简单子函数。这使得 BPF 函数可以独立验证，因此已被验证的函数在被其他函数调用时不需要再次进行验证。逐函数验证功能还支持更复杂的 BPF 程序，而不会超过验证器对函数长度的限制。

缓存。 WiredTiger为其B树内部页面和叶页面维护一个最近最少使用（LRU）缓存。在查找新的键值对时，WiredTiger将整个查找路径（包括叶页面）缓存起来。为了符合WiredTiger的缓存语义，上一节中描述的BPF函数也会返回所有遍历的页面，以便WiredTiger可以缓存它们。BPF函数将其遍历的页面存储在其上下文的临时缓冲区中。当临时缓冲区用尽时，BPF函数将停止重新提交请求并立即返回用户空间。在WiredTiger将这些页面添加到其缓存后，它将再次调用 `read_xrp` 以从上一页继续查找。由于我们将临时缓冲区的大小设置为4 KB，一个BPF函数可以在临时缓冲区中存储多达6个遍历的512 B页面，这为必要的元数据（如搜索键）留出了空间。

接口修改。 为了将 WiredTiger 与 XRP 集成，我们用 `read_xrp` 替换了正常的读取调用。 `read_-xrp` 在下一页不在缓存中并且需要从磁盘读取时被调用。WiredTiger 的驱逐策略强制要求只有没有缓存子页面的页面才能被驱逐，因此任何未缓存的页面都不会有缓存的子页面。因此，可以安全地调用 `read_xrp` 从磁盘读取剩余路径的所有内容，而无需再次检查应用级缓存。如果 `read_xrp` 因任何原因失败，WiredTiger 会回退到正常的查找路径。我们为每个 WiredTiger 会话分配一个数据和临时缓冲区，以避免为每个请求分配和释放缓冲区的开销。WiredTiger 会话同步处理

	Average Lookup Latency (μs)			
# Ops	SPDK	io_uring	`read()`	XRP
1	5.2	13.6	13.4	10.7
2	7.8	20.2	20.6	14.2
3	11.2	28.0	27.4	18.0
4	14.3	35.0	34.0	21.7
5	17.2	42.4	41.5	25.4
6	20.2	49.3	48.8	29.3

Table 3: Average latency of a random key lookup with BPF-KV as a function of the depth of the B$^+$-tree stored on-disk. # ops is the number of index I/Os per lookup.

one request at a time, which avoids concurrency issues.

6 Evaluation

In this section we seek to answer the following questions:

1. What are the overheads of using BPF for storage (§6.1)?
2. How does XRP scale to multiple threads (§6.2)?
3. What types of operations can XRP support (§6.3)?
4. Can XRP accelerate a real-world key-value store (§6.4)?

Experimental setup. All experiments are conducted on a 6-core i5-8500 3 GHz server with 16 GB of memory, using Ubuntu 20.04, and Linux 5.12.0 with an Intel Optane 5800X prototype. All experiments use O_DIRECT, turn off hyper-threading, disable processor C-states and turbo boost, use the maximum performance governor, and enable KPTI [30]. We use WiredTiger 4.4.0 in the experiments.

Baselines. We compare the following configurations: (a) XRP, (b) SPDK (a popular kernel-bypass library), (c) standard `read()` system calls, and (d) standard io_uring system calls.

6.1 BPF-KV

Latency. To answer the first evaluation question, we measure the performance of BPF-KV on a benchmark that performs a million read operations with keys drawn randomly with uniform probability. The experiment varies the number of levels of the tree that are stored on-disk. In this subsection, we disable caching of data objects and index nodes to focus on the overhead of looking up on-disk items. The measured average latency is shown in Table 3. The leftmost column represents the number of chained I/Os that are required to lookup the key in the index (not including the final data lookup). For example, if the number of operations is 4, then BPF-KV is configured with an on-disk tree of depth 4, and it also needs to issue one more I/O to fetch the key-value pair from the log.

There are a few takeaways from this experiment. First, XRP improves latency over `read()`, because XRP saves one or more storage layer traversals when it traverses the index or moves from the index to the log. Indeed, one can see that XRP's latency increases by about 3.5-3.9 μs for each additional I/O operation, which is close to the device's latency (Table 1). This means that XRP achieves close to optimal latency for resubmitted requests. The same is true for io_uring: in the case of submitting I/O requests synchronously without batching, `read()` and io_uring are almost equivalent. Second, SPDK exhibits better latency than XRP since XRP must pass through the kernel's storage stack once to initiate the index traversal, while SPDK completely bypasses the kernel. Nonetheless, XRP's marginal added latency when the depth of the B$^+$-tree is increased is close to SPDK's (2.6 μs-3.4 μs). For this reason, in the case of a 6-level index, XRP is only 45% slower than SPDK while `read()` is 142% slower than SPDK. Importantly, XRP achieves this without resorting to polling. This means that, unlike with SPDK, processes can continue to use CPU cores efficiently for other work; XRP's use of CPU time is limited to what is specifically needed to resubmit I/Os in the background and to keep I/O device utilization high.

Figure 5a and Figure 5b present the 99th-percentile latency and 99.9th-percentile latency of XRP, respectively. When running with a single thread, similar to the average latency results, XRP reduces both 99th-percentile latency and 99.9th-percentile latency by up to 30% compared to `read()` and io_uring. Note that our experiment runs as a closed loop, so XRP is running at a higher throughput than `read()` and io_uring. At identical throughput XRP would show additional improvement over these baselines. Interestingly, when the number of threads exceeds the number of cores (6) by more than 3, SPDK's 99.9th-percentile latency increases significantly. This is due to the fact that with SPDK all threads are busy-polling, and cannot effectively share the same core with other threads. To this end, we measure the percentage of requests whose latencies are greater than or equal to 1 ms and present the data in Figure 5c. The results show that SPDK has 0.03% of such requests with 7 threads, and this percentage increases to 0.28% when the number of threads reaches 24. In contrast, io_uring, `read()`, and XRP always have fewer than 0.01% of such requests.

Throughput. Figure 6a shows the throughput of XRP. As expected, as the index depth increases, XRP's speedup is higher compared to standard system calls. Figures 6b and 6c show the throughput speedups with a varying number of threads with an index of depth 3 and 6, respectively. Both figures show the speedup of XRP relative to issuing standard system calls does not decrease even as I/O and XRP BPF functions are scaled across several cores. Once again, XRP provides equal to or higher throughput compared to SPDK once the number of threads is 9 or higher.

6.2 Thread Scaling

Since storage applications often use a large number of concurrent threads that access I/O devices, for example in order to process concurrent requests and to perform background garbage collection [12, 20, 27, 44], XRP needs to be able to provide good tail latency and throughput under a large number threads. We analyze how XRP scales as a function of the num-

	平均查找延迟（μs）			
# Ops	SPDK	io_uring	`read()`	XRP
1	5.2	13.6	13.4	10.7
2	7.8	20.2	20.6	14.2
3	11.2	28.0	27.4	18.0
4	14.3	35.0	34.0	21.7
5	17.2	42.4	41.5	25.4
6	20.2	49.3	48.8	29.3

Table 3: Average latency of a random key lookup with BPF-KV as a function of the depth of the B$^+$-tree stored on-disk. # ops is the number of index I/Os per lookup.

一次只处理一个请求，这避免了并发问题。

6 Evaluation

在本节中，我们试图回答以下问题：

1. 使用 BPFforstorage 的开销是什么 (§6.1)? 2. XRP 如何扩展到多个线程 (§6.2)? 3. XRP 可以支持哪些类型的操作 (§6.3)? 4. XRP 能否加速一个现实世界的键值存储 (§6.4)?

实验设置。 所有实验都在一台 6 核 i5-8500 3 GHz 服务器上运行，内存为 16 GB，使用 Ubuntu 20.04，以及带有 Intel Optane 5800X 原型的 Linux 5.12.0。所有实验使用 O_DIRECT，关闭超线程，禁用处理器 C-states 和涡轮增压，使用最大性能策略，并启用 KPTI [30]。实验中使用 WiredTiger 4.4.0。

基准。 我们比较以下配置：(a) XRP，(b) SPDK（一个流行的内核绕过库），(c) 标准`read()`系统调用，以及 (d) 标准 io_uring 系统调用。

6.1 BPF-KV

延迟。 为了回答第一个评估问题，我们在一个执行百万次随机均匀概率读取操作的基准测试上测量了 BPF-KV 的性能。实验变化了树中存储在磁盘上的层数。在本小节中，我们禁用数据对象和索引节点的缓存，以专注于查找磁盘项的开销。测量的平均延迟显示在表 3。最左边的列表示在索引中查找键所需的链式 I/O 数量（不包括最终的最终数据查找）。例如，如果操作次数是 4，那么 BPF-KV 配置为具有深度 4 的磁盘树，并且它还需要发出一个额外的 I/O 来从日志中获取键值对。

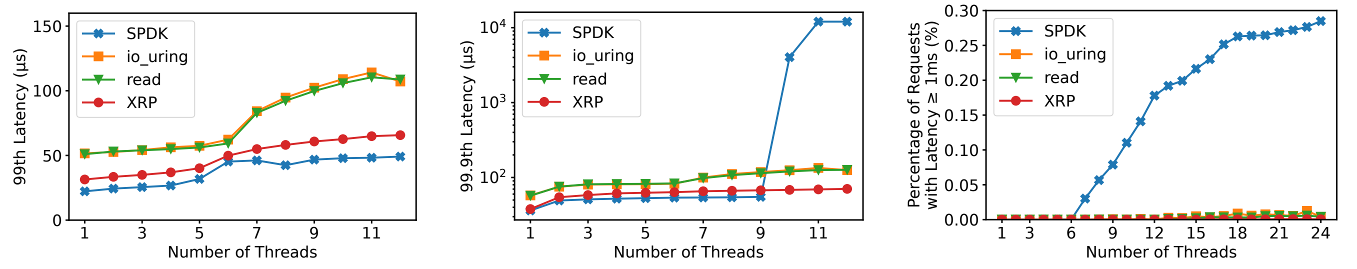
从这个实验中可以得出一些结论。首先，XRP 相对于`read()`提高了延迟，因为 XRP 在遍历索引或从索引移动到日志时节省了一个或多个存储层遍历。实际上，可以看到 XRP 的延迟每增加一个 I/O 操作大约增加 3.5-3.9 μs，这接近设备的延迟（表 1）。这意味着 XRP 实现了接近最优的重新提交请求的延迟。io_uring 也是如此：在同步提交 I/O 请求且不进行批处理的情况下，`read()` 和 io_uring 几乎等效。其次，SPDK 的延迟表现优于 XRP，因为 XRP 必须通过内核的存储堆栈一次来启动索引遍历，而 SPDK 完全绕过了内核。尽管如此，当 B$^+$-树的深度增加时，XRP 的边际附加延迟接近 SPDK（2.6 μs- 3.4 μs）。因此，在 6 级索引的情况下，XRP 仅比 SPDK 慢 45%，而 `read()` 比 SPDK 慢 142%。重要的是，XRP 在此过程中无需依赖轮询。这意味着，与 SPDK 不同，进程可以继续高效地使用 CPU 核心进行其他工作；XRP 的 CPU 使用时间仅限于在后台重新提交 I/O 所需的部分，以及保持 I/O 设备利用率高。

图 5a 和图 5b展示了 99th-百分位延迟和 99。9th-百分位延迟。在单线程运行时，与平均延迟结果类似，XRP 将 99th-百分位延迟和 99。9th-百分位延迟降低了高达 30%，与 `read()` 和 io_uring 相比。请注意，我们的实验以闭环方式运行，因此 XRP 的吞吐量高于 `read()` 和 io_- uring。在相同的吞吐量下，XRP 在这些基线之上会表现出额外的改进。有趣的是，当线程数量超过核心数量（6）超过 3 个时，SPDK 的 99。9th-百分位延迟显著增加。这是由于 SPDK 中所有线程都在轮询，并且无法有效地与其他线程共享同一个核心。为此，我们测量了延迟大于或等于 1 ms 的请求的百分比，并在图 5c中展示了数据。结果表明，在 7 个线程时，SPDK 有 0.03% 的此类请求，而当线程数量达到 24 时，这一百分比增加到 0.28%。相比之下，io_uring `read()`、和 XRP 始终少于 0.01% 的此类请求。

吞吐量。 图6a 显示了XRP的吞吐量。正如预期的那样，随着索引深度的增加，XRP的速度提升高于标准系统调用。图6b和 6c分别显示了具有不同线程数量的吞吐量速度提升，索引深度为3和6。这两个图都显示，即使I/O和XRP BPF函数在多个核心上扩展，XRP相对于发出标准系统调用的速度提升也不会降低。再次，一旦线程数量达到9个或更高，XRP提供的吞吐量等于或高于SPDK。

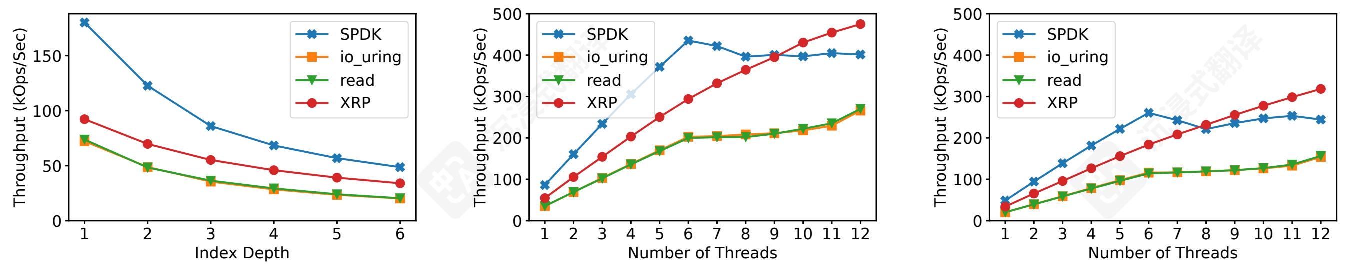
6.2 线程扩展

由于存储应用程序通常使用大量并发线程来访问I/O设备，例如为了处理并发请求和执行后台垃圾回收 [12, 20, 27, 44], XRP需要能够在大量线程下提供良好的尾部延迟和吞吐量。我们分析XRP如何作为线程数量的函数进行扩展



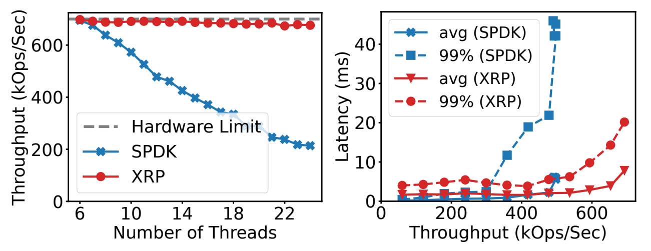
(a) 99th-percentile latency. **(b)** (Log scale) 99.9th-percentile latency. **(c)** Percentage of requests with latency ≥ 1 ms.

Figure 5: Tail latency and percentage of requests with extreme latency of XRP and SPDK against read and io_uring with BPF-KV with index depth 6, random key lookups, and closed-loop load generator.



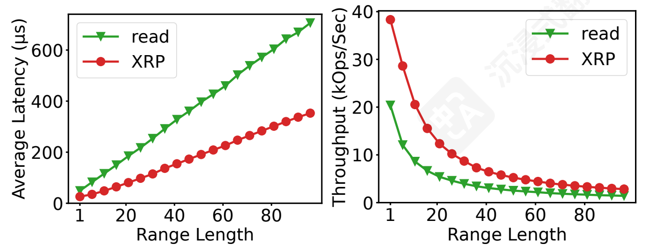
(a) Single thread with varying index depth. **(b)** Multiple threads with index depth 3. **(c)** Multiple threads with index depth 6.

Figure 6: Throughput of XRP and SPDK against read and io_uring with BPF-KV with random key lookups and closed-loop load generator.



(a) Scalability of XRP and SPDK. **(b)** Latency-throughput graph of XRP and SPDK with 12 threads.

Figure 7: XRP vs. SPDK with open-loop load generator.



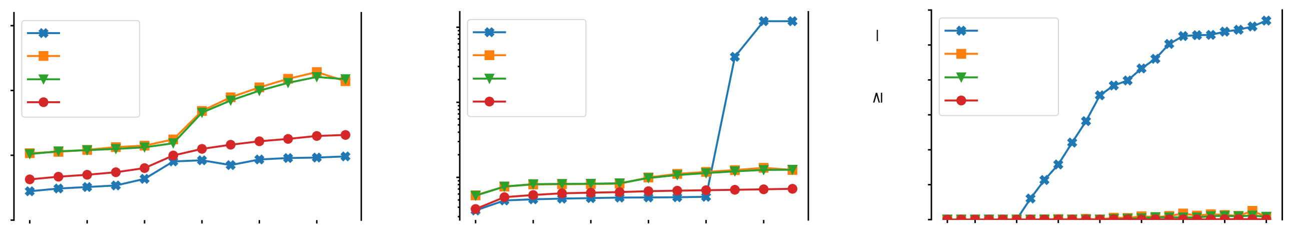
(a) Average Latency **(b)** Throughput

Figure 8: Average read latency and throughput of BPF-KV with XRP vs. `read()` when performing a range query over a varying number of objects.

ber of threads and compare it to SPDK. We run an open loop experiment, where the amount of load matches the maximum bandwidth of the Intel device (5M IOPS for 512 B random reads). Figure 7a compares the throughput of XRP (integrated with io_uring) to SPDK with BPF-KV using 6 on-disk index levels, where each thread represents a different tenant. Two major observations are: 1) when using 6 working threads (the number of CPU cores on the machine) both SPDK and XRP can achieve a throughput close to the hardware limit (the grey dashed line); 2) once the thread count exceeds the CPU cores, SPDK's throughput steadily decreases while XRP still provides stable throughput. SPDK's throughput collapse stems from its polling-based approach; SPDK threads never yield, leaving scheduling up to Linux's CFS which works in coarse 6 ms timeslices. However, idle XRP threads will voluntarily yield the CPU to busy threads, so more CPU cycles are spent on actual work. Figure 7b presents the throughput-latency relationship under 12 working threads as a function of the load. With more threads than CPU cores, both average and tail latencies also increase more significantly in SPDK, as each thread waits longer to be scheduled than in XRP.

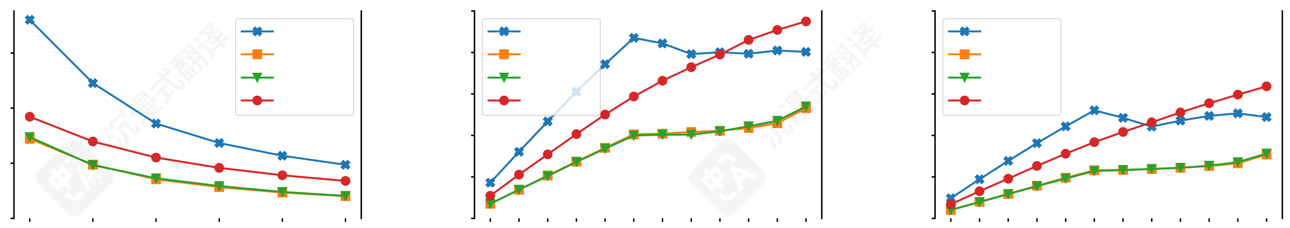
6.3 Range Query

Figure 8 compares the average latency and the throughput of running a range query with XRP against performing the query with `read()` system calls. In both cases the range query performs a single index traversal to find the first object, and traverses the leaf nodes of the index to find the address of subsequent objects. The index depth is 6 in this experiment. Even though the XRP range query can only retrieve 32 objects per syscall, the results show this adds negligible overhead. XRP's performance speedup remains relatively constant as a function of the length of the aggregation, since XRP performs only one storage stack traversal for every 32 values retrieved.



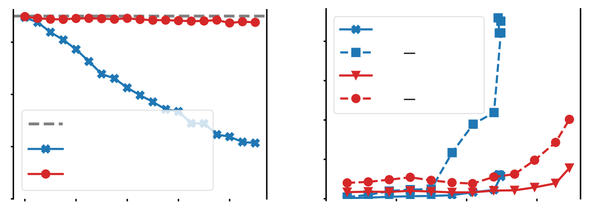
(a) 99th-percentile latency. **(b)** (Log scale) 99.9th-percentile latency. **(c)** Percentage of requests with latency ≥ 1 ms.

图 5: 尾延迟和极端延迟请求的百分比，XRP 和 SPDK 对比 `read` 和 io_uring，使用 BPF-KV，索引深度为 6，随机键查找，以及闭环负载生成器。



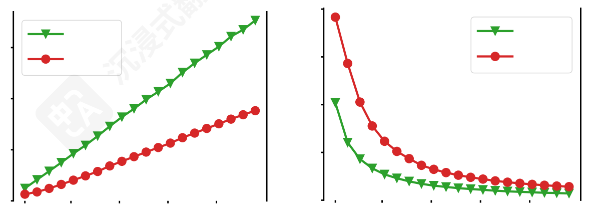
(a)单线程，索引深度变化。 **(b)** Multiple threads with index depth 3. **(c)** Multiple threads with index depth 6.

Figure 6: Throughput of XRP and SPDK against `read` and io_uring with BPF-KV with random key lookups and closed-loop load generator.



(a) Scalability of XRP and SPDK. **(b)** Latency-throughput graph of XRP and SPDK with 12 threads.

图 7： XRP vs. SPDK，使用开环负载生成器。



(a) Average Latency **(b)** Throughput

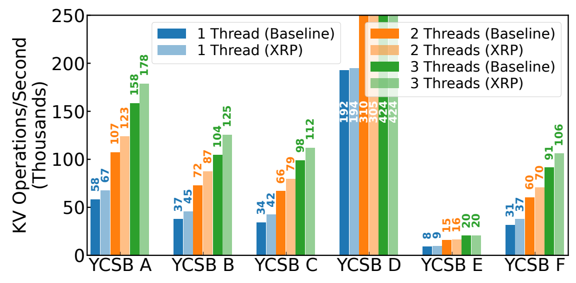
图 8： BPF-KV 的平均读取延迟和吞吐量，XRP vs. `read()`，在跨越不同数量对象的范围查询中执行时。

将线程数量与 SPDK 进行比较。我们进行了一个开环实验，其中负载量等于英特尔设备的最大带宽（512 B 随机读取为 5M IOPS）。图 7a 将 XRP 的吞吐量（集成）与其他…进行比较

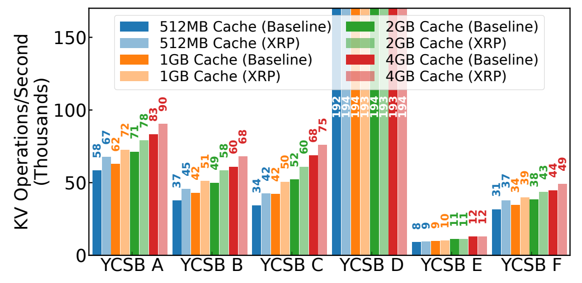
使用io_uring)到SPDK，通过BPF-KV使用6个磁盘索引级别，其中每个线程代表一个不同的租户。有两个主要的观察结果：1)当使用6个工作线程（机器上的CPU核心数）时，SPDK和XRP都可以实现接近硬件极限（灰色虚线）的吞吐量；2)一旦线程数超过CPU核心数，SPDK的吞吐量就会稳步下降，而XRP仍然提供稳定的吞吐量。SPDK的吞吐量崩溃源于其基于轮询的方法；SPDK线程从不放弃，将调度交给Linux的CFS，CFS工作在粗粒度的6毫秒时间片。然而，空闲的XRP线程会自愿放弃CPU给繁忙的线程，因此更多的CPU周期被用于实际工作。图7b 展示了在12个工作线程下，吞吐量-延迟关系作为负载的函数。当线程数超过CPU核心数时，SPDK的平均和尾部延迟也显著增加，因为每个线程等待调度的时间比XRP更长。

6.3 范围查询

图 8 比较了使用 XRP 执行范围查询的平均延迟和吞吐量与使用 `read()` 系统调用执行查询。在这两种情况下，范围查询执行一次索引遍历来查找第一个对象，并遍历索引的叶节点来查找后续对象的地址。在此实验中，索引深度为 6。尽管 XRP 范围查询每次系统调用只能检索 32 个对象，但结果表明这增加了可忽略的开销。XRP 的性能提升随着聚合长度的增加而保持相对稳定，因为 XRP 每次检索 32 个值时只执行一次存储栈遍历。

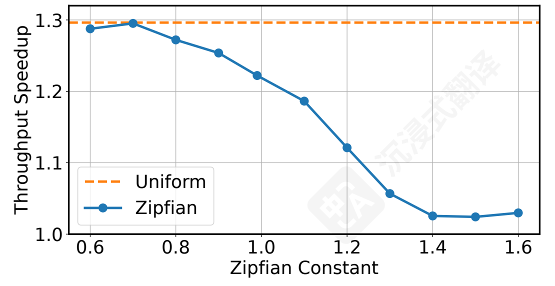


(a)

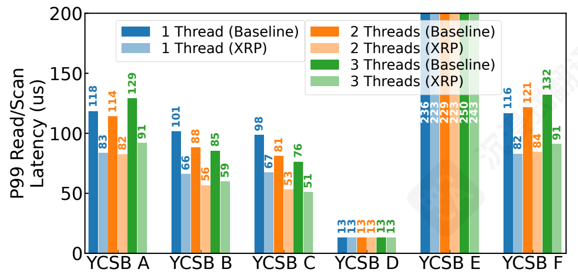


(b)

Figure 9: Throughput of reads/scans in WiredTiger with **(a)** varying client threads with a 512 MB cache and **(b)** varying cache size.



(a)



(b)

Figure 10: **(a)** Throughput speedup of WiredTiger on YCSB C with a varying Zipfian constant and with a uniform distribution. **(b)** 99th-percentile latency of reads/scans in WiredTiger with varying number of threads with 512 MB cache.

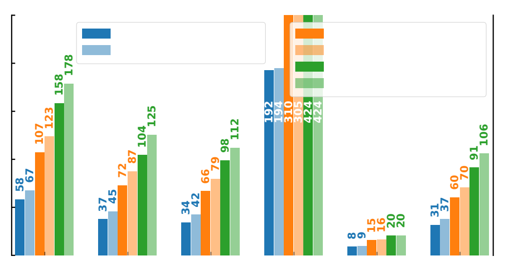
6.4 WiredTiger

To understand whether XRP can benefit a real-world database, we evaluate the performance of WiredTiger with and without XRP on YCSB [41]. We run the different YCSB workloads so that their runtime takes more or less the same time: YCSB A, B, C and E use 10M operations, D uses 50M operations and E uses 3M. The baseline WiredTiger uses `pread()` to read B-tree pages, while the WiredTiger with XRP uses `read_xrp()`. We populate the database with 1 billion key-value pairs and set the size of both key and value to 16 B. The total size of the database is 46 GB. WiredTiger runs eviction threads to evict pages when its cache usage is close to full, and we set the number of eviction threads to 2.

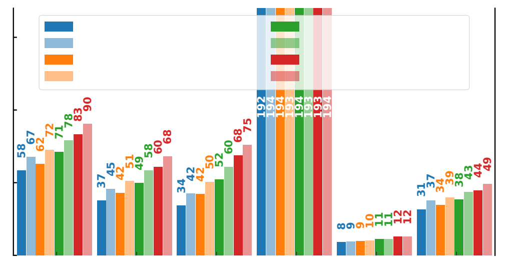
Throughput. Figure 9 shows the total throughput of WiredTiger with different cache sizes and different numbers of client threads. We configure WiredTiger with 512 MB, 1 GB, 2 GB, and 4 GB cache sizes to ensure that WiredTiger can cache at least 1% of its database while not exhausting all the available memory on the machine. We run up to 3 client threads to avoid context switches. The results show that XRP speeds up most workloads consistently by up to 1.25×. The throughput improvements are mostly affected by the cache size. The speedup generally goes down when the cache size becomes larger. In general, XRP provides a lower speedup on WiredTiger than on BPF-KV, because WiredTiger is less optimized than BPF-KV for reading from fast NVM storage, and only spends 63% of its total time on I/O. In particular, XRP does not provide significant improvements on YCSB D and YCSB E. This is because YCSB D follows a latest distribution where the newly inserted items are the most popular ones. Since new inserts are always written into in-memory buffers, most read operations read from those buffers in YCSB D. On the other hand, YCSB E only has inserts and scans. WiredTiger supports scans via an iterator interface, which only looks up one key-value pair at a time. XRP can only benefit the lookup of the first key-value pair of a scan operation, since the rest of the key-value pairs mostly either reside on the same leaf node or require only one additional I/O to fetch the next leaf node.

To study the effect of access distribution on XRP, we run YCSB C with a varying Zipfian constant and with a uniform distribution. Figure 10a shows that XRP's benefit decreases when the Zipfian constant becomes larger (i.e. , the distribution is more skewed) because of the increased cache hit ratio. Note that skews greater than 0.99 represent very high skew levels. We also see that the throughput gain on WiredTiger is lower than that on BPF-KV with the uniform YCSB C. This is again because WiredTiger spends 37% of its total time on non-I/O operations.

Tail latency. We measure the tail read latency of WiredTiger with and without XRP under a fixed load: 20 kop/s per client thread for YCSB A, B, C, D, F, and 5 kops/s per client thread for YCSB E. Since YCSB E has scans instead of reads, we set a lower load for it and measure the tail scan latency instead of the tail read latency. Figure 10b shows that XRP can reduce the 99th-percentile latency by up to 40%. Similar to the throughput, the 99th-percentile latency improve-

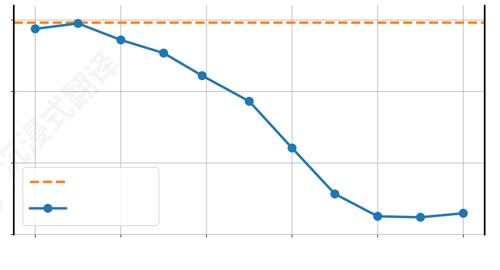


(a)

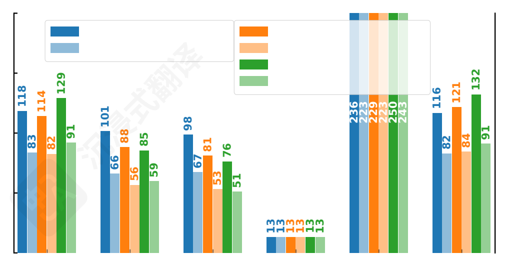


(b)

图 9: WiredTiger 中的读/扫描吞吐量，带有**(a)** 不同客户端线程和 512 MB 缓存，以及 **(b)** 不同缓存大小



(a)



(b)

图 10: (a) WiredTiger 在 YCSB C 上的吞吐量加速，带有不同 Zipfian 常数和均匀分布。 (bercentile 延迟，WiredTiger 中的读/扫描，带有) 99th-p不同线程数和 512 MB 缓存。

6.4 WiredTiger

为了了解 XRP 是否能提升真实世界数据库的性能，我们评估了 WiredTiger 在 YCSB 上启用和禁用 XRP 的性能 [41]。我们运行不同的 YCSB 工作负载，使它们的运行时间大致相同：YCSB A、B、C 和 E 使用 10M 操作，D 使用 50M 操作，E 使用 3M。基准 WiredTiger 使用 `pread()` 来读取 B-树页面，而带有 XRP 的 WiredTiger 使用 `read_xrp()`。我们将数据库填充了 1 亿个键值对，并将键和值的大小设置为 16 B。数据库的总大小为 46 GB。WiredTiger 运行驱逐线程，当其缓存使用接近满时驱逐页面，我们将驱逐线程的数量设置为 2。

吞吐量。 图 9 显示了 WiredTiger 在不同缓存大小和不同客户端线程数下的总吞吐量。我们配置 WiredTiger 使用 512 MB、1 GB、2 GB 和 4 GB 的缓存大小，以确保 WiredTiger 可以缓存其数据库至少 1% 的数据，同时不会耗尽机器上的所有可用内存。我们运行最多 3 个客户端线程以避免上下文切换。结果表明，XRP 一致地加速了大多数工作负载，最高可达 1。25×。 吞吐量的提升主要受缓存大小的影响。当缓存大小变大时，加速效果通常会下降。一般来说，XRP 在 WiredTiger 上的加速效果低于在 BPF-KV 上的效果，因为 WiredTiger 在从快速 NVM 存储读取方面不如 BPF-KV 优化，并且只花费了其总时间的 63% 在 I/O 上。特别是，XRP 在 YCSB D 和 YCSB E 上没有显著提升。这是因为 YCSB D 遵循最新的分布，其中新插入的项目是最受欢迎的。由于新插入的数据总是写入内存缓冲区，YCSB D 中的大多数读操作都从这些缓冲区读取。另一方面，YCSB E 只有插入和扫描操作。WiredTiger 通过迭代器接口支持扫描，该接口一次只查找一个键值对。XRP 只能受益于扫描操作中第一个键值对的查找，因为其余的键值对大多位于同一个叶节点上，或者只需要一个额外的 I/O 操作来获取下一个叶节点。

为了研究访问分布对 XRP 的影响，我们使用不同的 Zipfian 常数和均匀分布运行 YCSB C。图 10a 显示，当 Zipfian 常数变大时（即分布更加偏斜），XRP 的收益会降低，因为缓存命中率增加了。请注意，大于 0.99 的偏斜度代表非常高的偏斜水平。我们还发现，在均匀 YCSB C 下，WiredTiger 的吞吐量提升低于 BPF-KV。这再次是因为 WiredTiger 将其总时间的 37% 用于非 I/O 操作。

尾部延迟。 我们在固定负载下测量了 WiredTiger 在有和没有 XRP 时的尾部读取延迟：对于 YCSB A、B、C、D、F，每个客户端线程为 20 kop/s；对于 YCSB E，每个客户端线程为 5 kops/s。由于 YCSB E 使用扫描而不是读取，我们为其设置了较低的负载，并测量尾部扫描延迟而不是尾部读取延迟。图 10b 显示 XRP 可以将 99th-百分位延迟降低高达 40%。与吞吐量类似， 99th-百分位延迟提升

ment mostly decreases with a larger cache size, and XRP does not have significant effect on YCSB D and E.

7 Related Work

There are four areas of related work: (a) using BPF to accelerate I/O (typically networking), (b) kernel-bypass systems, (c) near-storage compute, and (d) extensible operating systems and library file systems.

BPF for I/O. There is a large number of systems and frameworks that use BPF to accelerate I/O processing, primarily focused on networking and tracing use cases [2,4–6,15,18,25,28,37,46,49,50,52]. Most closely related to XRP, XDP [28] accelerates networking I/O by adding a hook in the NIC driver's RX path. It then provides an interface for eBPF programs that either filter, redirect, or bounce the packet.

There are no existing systems that use BPF to resubmit storage requests from within the kernel. Kourtis et al. [62] propose a system that uses eBPF functions as an interface to submit disaggregated storage requests in order to avoid crossing the network. In their system, resubmissions occur from a user space service sitting at the host and are not serviced by the kernel itself, since the network is the primary bottleneck (not the kernel software stack). ExtFUSE [36] allows user space file systems on Linux to load BPF functions into the kernel to serve low-level file system requests and thus eliminates unnecessary context switches. While ExtFUSE accelerates user space file systems, it provides no performance benefits for an application that already uses a standard kernel file system (e.g., ext4), since it does not allow applications to bypass the kernel's storage stack. BMC [49] uses BPF to accelerate memcached by intercepting packets on the network path at the host. The BPF functions can then access a separate small kernel-based cache, which serves as a first-level cache and is not synchronized with the user space memcached application. Zhong et al. [85] provide motivation for using BPF for accelerating storage from within the kernel, but do not provide a concrete design, implementation or evaluation.

Kernel bypass. In order to reduce the kernel's overhead when processing I/O, several libraries and operating systems have been designed to let users directly access I/O devices [7,33,34,42,47,57,65,69,71,72,82–84]. Most relevant to our work, Intel's SPDK [82] is a popular kernel-bypass library for storage. In general, the downside of allowing users to access I/O directly is that applications must directly poll for I/O to obtain high performance. This means that cores cannot be shared among processes, which leads to significant underutilization when I/O is not the bottleneck.

Near-storage compute. There are several systems that allow applications to offload their storage functions to the processor embedded within or attached to a storage device [16, 22, 31, 38, 43, 51, 55, 61, 63, 74, 75, 77, 81]. The downside of this approach is that it requires specialized storage devices, dedicated hardware, or both.

Extensible operating systems and library file systems. Our approach is reminiscent of extensible operating systems and library file systems from the 1990s. Extensible operating systems (e.g., SPIN [35] and VINO [76, 79]) allow extension of kernel functionality via user-defined functions. For example, a client can write kernel extensions that read and decompress video frames from disk. Another related approach is library file systems, such as XN [48,56]. Similar to XRP, XN allows userspace library file systems to load untrusted metadata translation functions into the kernel, while guaranteeing disk block protection without understanding file systems' data structures. These approaches required using dedicated operating and file systems, while XRP is compatible with Linux and its standard file systems. ExtOS [32], a more recent extensible OS, minimizes data movements in `read()` and `splice()` by using BPF functions to filter data before copying them to user space or another file, but it still incurs the full storage stack overhead and does not allow I/O request resubmissions.

8 Conclusions and Future Work

BPF has the potential to accelerate applications using fast NVMe devices by moving computation closer to the device. XRP lets applications write functions that can resubmit dependent storage requests to achieve speedups close to kernel-bypass while retaining the advantages of being OS-integrated. Beyond fast lookups, we envision XRP can be used for many types of functions such as compaction, compression and deduplication. In addition, XRP in the future can be developed as a common interface for other use cases where computation needs to be moved closer to storage, such as programmable storage devices and networked storage systems. For example, XRP could be used as an interface that can dynamically support both in-kernel offloading, as well as offloading functions to a smart storage device or an FPGA. Another direction we plan to explore is networked storage. XRP storage functions could be chained with XDP networking functions to create a datapath that bypasses both the kernel's networking and storage paths.

9 Acknowledgments

We would like to thank our shepherd, Ymir Vigfusson, and the anonymous reviewers for their helpful comments. We also thank Kostis Kaffes and Tom Anderson for providing valuable feedback on earlier versions of our work. This work was supported by NSF grants CNS-2143868, CNS-2104292, and CNS-1750558, and an equipment gift from Intel.

References

[1] 3D Xpoint: A Breakthrough in Non-Volatile Memory Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html.

[2] bcc. https://github.com/iovisor/bcc.

随着缓存大小的增加，ment 主要减少，并且 XRP 对 YCSB D 和 E 没有显著影响。

7 相关工作

相关工作分为四个领域：(a) 使用 BPF 加速 I/O（通常用于网络），(b) 内核绕过系统，(c) 近存储计算，以及 (d) 可扩展操作系统和库文件系统。

BPF 用于 I/O。 有许多系统和框架使用 BPF 加速 I/O 处理，主要集中于网络和跟踪用例 [2,4–6,15,18,25,28,37,46,49,50,52]。与 XRP 最密切相关的 XDP [28]通过在网卡驱动程序的 RX 路径中添加钩子来加速网络 I/O。然后它为 eBPF 程序提供一个接口，该接口可以过滤、重定向或反弹数据包。

没有现有的系统使用 BPF 从内核中重新提交存储请求。Kourtis 等人 [62] 提出了一种系统，该系统使用 eBPF 函数作为接口来提交解耦合的存储请求，以避免跨越网络。在其系统中，重新提交是从主机上的用户空间服务发生的，而不是由内核本身提供服务，因为网络是主要的瓶颈（不是内核软件堆栈）。ExtFUSE [36] 允许 Linux 上的用户空间文件系统将 BPF 函数加载到内核中以服务低级文件系统请求，从而消除了不必要的上下文切换。虽然 ExtFUSE 加速了用户空间文件系统，但它为已经使用标准内核文件系统（例如 ext4）的应用程序提供了没有性能优势，因为它不允许应用程序绕过内核的存储堆栈。BMC [49] 使用 BPF 通过在主机上的网络路径上拦截数据包来加速 memcached。然后 BPF 函数可以访问一个单独的小型基于内核的缓存，该缓存作为一级缓存，并且与用户空间 memcached 应用程序不同步。Zhong 等人 [85] 提供了使用 BPF 从内核中加速存储的动机，但没有提供具体的设计、实现或评估。

内核绕过。 为了减少内核在处理 I/O 时的开销，一些库和操作系统被设计为允许用户直接访问 I/O 设备 [7,33,34,42,47,57,65,69,71,72,82–84]. 与我们的工作最相关的是，Intel 的 SPDK [82] 是一个流行的存储内核绕过库。总的来说，允许用户直接访问 I/O 的缺点是应用程序必须直接轮询 I/O 以获得高性能。这意味着核心不能在进程之间共享，当 I/O 不是瓶颈时会导致显著的低利用率。

近存储计算。 有几种系统允许应用程序将其存储功能卸载到嵌入在存储设备内部或连接到存储设备的处理器中 [16, 22, 31, 38, 43, 51, 55, 61, 63, 74, 75, 77, 81]。这种方法的缺点是它需要专用存储设备、专用硬件或两者兼有。

可扩展操作系统和库文件系统。 我们的方法让人联想到20世纪90年代的可扩展操作系统和库文件系统。可扩展操作系统（例如，SPIN [35] 和 VINO [76, 79]）允许通过用户定义的函数扩展内核功能。例如，客户端可以编写内核扩展，从磁盘读取和解压缩视频帧。另一种相关方法是库文件系统，例如 XN [48,56]。类似于 XRP，XN 允许用户空间库文件系统将不受信任的元数据转换函数加载到内核中，同时在不理解文件系统数据结构的情况下保证磁盘块保护。这些方法需要使用专用的操作系统和文件系统，而 XRP 兼容 Linux 及其标准文件系统。ExtOS [32],是一个更近期的可扩展操作系统，通过使用 BPF 函数在将数据复制到用户空间或另一个文件之前过滤数据，以在 `read()` 和 `splice()` 中尽量减少数据移动，但它仍然会产生完整的存储栈开销，并且不允许 I/O 请求重提交。

8 结论与未来工作 k

BPF 有潜力通过将计算移近设备来加速使用快速 NVMe 设备的应用。XRP 允许应用程序编写函数，这些函数可以重新提交相关的存储请求，以实现接近内核绕过的加速，同时保留作为操作系统集成的优势。除了快速查找之外，我们设想 XRP 可以用于多种类型的函数，如压缩、压缩和重复数据删除。此外，XRP 在未来可以开发为一种通用接口，用于其他需要将计算移近存储的使用案例，例如可编程存储设备和网络存储系统。例如，XRP 可以用作一个接口，可以动态支持内核卸载，以及将功能卸载到智能存储设备或 FPGA。我们计划探索的另一个方向是网络存储。XRP 存储功能可以与 XDP 网络功能串联起来，创建一个绕过内核网络和存储路径的数据路径。

9 致谢

我们感谢我们的指导老师 Ymir Vigfusson 和匿名审稿人为他们的有益评论。我们也感谢 Kostis Kaffes 和 Tom Anderson 为我们早期的工作提供了宝贵的反馈。这项工作得到了 NSF 授予的 CNS-2143868、CNS-2104292 和 CNS-1750558 的资助，以及英特尔公司提供设备捐赠。

参考文献

[1] 3D Xpoint：一种突破性的非易失性存储技术。https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html。

[2] bcc. https://github.com/iovisor/bcc.

[3] bpf: Introduce function-by-function verification. https://lore.kernel.org/bpf/20200109063745.3154913-4-ast@kernel.org/.

[4] bpftrace. https://github.com/iovisor/bpftrace.

[5] Cilium. https://github.com/cilium/cilium.

[6] Cloudflare architecture and how BPF eats the world. https://blog.cloudflare.com/cloudflare-architecture-and-how-bpf-eats-the-world/.

[7] DPDK Data Plane Development Kit. https://www.dpdk.org/.

[8] eBPF. https://ebpf.io/.

[9] Efficient io with io_uring. https://kernel.dk/io_uring.pdf.

[10] HDFS Architecture Guide. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[11] Intel® Optane™ SSD DC P5800X Series. https://ark.intel.com/content/www/us/en/ark/products/201859/intel-optane-ssd-dc-p5800x-series-1-6tb-2-5in-pcie-x4-3d-xpoint.html.

[12] LevelDB. https://github.com/google/leveldb.

[13] libbpf. https://github.com/libbpf/libbpf.

[14] Linux Socket Filtering Documentation. https://www.kernel.org/doc/Documentation/networking/filter.txt.

[15] MAC and Audit policy using eBPF. https://lkml.org/lkml/2020/3/28/479.

[16] NGD systems newport platform. https://www.ngdsystems.com/technology/computational-storage.

[17] NVMe base specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4b-2020.09.21-Ratified.pdf.

[18] Open-sourcing katran, a scalable network load balancer. https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/.

[19] Optimizing Software for the Next Gen Intel Optane SSD P5800X. https://www.intel.com/content/www/us/en/events/memory-and-storage.html?videoId=6215534787001.

[20] Percona TokuDB. https://www.percona.com/software/mysql-database/percona-tokudb.

[21] pread(2) - Linux manual page. https://man7.org/linux/man-pages/man2/pread.2.html.

[22] SmartSSD computational storage drive. https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html.

[23] A thorough introduction to eBPF. https://lwn.net/Articles/740157/.

[24] Toshiba memory introduces XL-FLASH storage class memory solution. https://business.kioxia.com/en-us/news/2019/memory-20190805-1.html.

[25] udplb. https://github.com/moolen/udplb.

[26] Ultra-Low Latency with Samsung Z-NAND SSD. https://www.samsung.com/semiconductor/global.semi.static/Ultra-Low_Latency_with_Samsung_Z-NAND_SSD-0.pdf.

[27] WiredTiger storage engine. https://docs.mongodb.com/manual/core/wiredtiger/.

[28] XDP. https://www.iovisor.org/technology/xdp.

[29] Nadav Amit and Michael Wei. The design and implementation of hyperupcalls. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 97–112, 2018.

[30] Nadav Amit, Michael Wei, and Dan Tsafrir. Dealing with (some of) the fallout from meltdown. In *Proceedings of the 14th ACM International Conference on Systems and Storage*, pages 1–6, 2021.

[31] Antonio Barbalace, Anthony Iliopoulos, Holm Rauchfuss, and Goetz Brasche. It's time to think about an operating system for near data processing architectures. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 56–61, New York, NY, USA, 2017. Association for Computing Machinery.

[32] Antonio Barbalace, Javier Picorel, and Pramod Bhatotia. Extos: Data-centric extensible os. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '19, page 31–39, New York, NY, USA, 2019. Association for Computing Machinery.

[33] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 335–348, Hollywood, CA, October 2012. USENIX Association.

[34] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high

[3] bpf: 引入逐函数验证。 https://lore.kernel.org/bpf/20200109063745.3154913-4-ast@kernel.org/。

[4] bpftrace. https://github.com/iovisor/bpftrace.

[5] Cilium. https://github.com/cilium/cilium.

[6]Cloudflare 架构以及 BPF 如何改变世界。 https://blog.cloudflare.com/cloudflare-architecture-and-how-bpf-eats-the-world/.

[7]DPDK 数据平面开发套件。 https://www.dpdk.org/.

[8] eBPF. https://ebpf.io/.

[9]使用 io_uring 实现高效的 io。 https://kernel.dk/io_uring.pdf.

[10]HDFS 架构指南。 https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html。

[11]Intel® OptaneTM SSD DC P5800X 系列。 https://ark.intel.com/content/www/us/en/ark/products/201859/intel-optane-ssd-dc-p5800x-series-1-6tb-2-5in-pcie-x4-3d-xpoint.html。

[12] LevelDB。 https://github.com/google/leveldb。

[13] libbpf。 https://github.com/libbpf/libbpf。

[14]Linux 套接字过滤文档。 https://www.kernel.org/doc/Documentation/networking/filter.txt。

[15] 使用 eBPF 的 MAC 和审计策略。 https://lkml.org/lkml/2020/3/28/479。

[16]NGD系统新港平台。https://www.ngdsystems.com/technology/computational-storage。

[17] NVMe基础规范。 https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4b-2020.09.21-Ratified.pdf。

[18] 开源可扩展网络负载均衡器katran。 https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/。

[19] 优化下一代英特尔Optane SSDP5800X的软件。 https://www.intel.com/content/www/us/en/events/memory-and-storage.html?videoId=6215534787001。

[20]Percona TokuDB。https://www.percona.com/software/mysql-database/percona-tokudb。

[21] pread(2) - Linux手册页。 https://man7.org/linux/man-pages/man2/pread.2.html。

[22]SmartSSD计算存储驱动器。https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html。

[23] eBPF的全面介绍。 https://lwn.net/Articles/740157/。

[24] Toshibamemory介绍了XL-FLASH存储类内存解决方案。 https://business.kioxia.com/en-us/news/2019/memory-20190805-1.html。

[25] udplb. https://github.com/moolen/udplb。

[26]使用Samsung Z- NAND SSD实现超低延迟。 https://www.samsung.com/semiconductor/global.semi.static/Ultra-Low_Latency_with_Samsung_Z-NAND_SSD-0.pdf。

[27]WiredTiger存储引擎。https://docs.mongodb.com/manual/core/wiredtiger/。

[28] XDP. https://www.iovisor.org/technology/xdp.

[29] Nadav Amit 和 Michael Wei. 超级调用的设计与实现. 在 *2018 USENIX* 年度技术会议 *(USENIX ATC 18)*, 第 97–112 页, 2018.

[30] Nadav Amit, Michael Wei, 和 Dan Tsafrir. 处理熔断效应 (部分) 的后果. 在 第 *14* 届 *ACM* 国际系统与存储会议论文集, 第 1–6 页, 2021.

[31] Antonio Barbalace, Anthony Iliopoulos, Holm Rauchfuss, 和 Goetz Brasche. 是时候为近数据处理架构考虑操作系统了. 在 第 *16* 届操作系统热点技术研讨会论文集, HotOS' 17, 第 56–61 页, 纽约, NY, USA, 2017. 计算机协会.

[32] Antonio Barbalace、Javier Picorel 和 Pramod Bhatotia。Extos：以数据为中心的可扩展操作系统。在第*10*届*ACM SIGOPS*亚太系统研讨会，APSys' 19，第31-39页，纽约，纽约，美国，2019年。计算机协会。

[33] Adam Belay、Andrea Bittau、Ali Mashtizadeh、David Terei、David Mazières 和 Christos Kozyrakis。Dune：安全地访问特权CPU功能。在第*10*届*USENIX*操作系统设计与实施研讨会（*OSDI 12*），第335-348页，好莱坞，加利福尼亚州，2012年10月。USENIX协会。

[34] Adam Belay、George Prekas、Ana Klimovic、Samuel Grossman、Christos Kozyrakis 和 Edouard Bugnion。IX：一个用于高速的受保护数据平面操作系统

throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, October 2014. USENIX Association.

[35] Brian N Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility safety and performance in the SPIN operating system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 267–283, 1995.

[36] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 121–134, Renton, WA, July 2019. USENIX Association.

[37] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient software packet processing on FPGA NICs. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 973–990. USENIX Association, November 2020.

[38] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, Zhenjun Liu, Feng Zhu, and Tong Zhang. POLARDB meets computational storage: Efficiently support analytical workloads in cloud-native relational database. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 29–41, Santa Clara, CA, February 2020. USENIX Association.

[39] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. FASTER: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 275–290, New York, NY, USA, 2018. Association for Computing Machinery.

[40] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. SplinterDB: Closing the bandwidth gap for NVMe key-value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 49–63, 2020.

[41] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[42] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. When idling is ideal: Optimizing tail-latency for heavy-tailed datacenter workloads with perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 621–637, New York, NY, USA, 2021. Association for Computing Machinery.

[43] Jaeyoung Do, Sudipta Sengupta, and Steven Swanson. Programmable solid-state storage in future cloud datacenters. *Communications of the ACM*, 62(6):54–62, 2019.

[44] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing space amplification in RocksDB. In *CIDR*, volume 3, page 3, 2017.

[45] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing DRAM footprint with NVM in Facebook. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–13, 2018.

[46] Pekka Enberg, Ashwin Rao, and Sasu Tarkoma. Partition-aware packet steering using XDP and eBPF for improving application-level parallelism. In *Proceedings of the 1st ACM CoNEXT Workshop on Emerging In-Network Computing Paradigms*, ENCP '19, page 27–33, New York, NY, USA, 2019. Association for Computing Machinery.

[47] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020.

[48] Gregory R Ganger and M Frans Kaashoek. Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files. In *USENIX Annual Technical Conference*, pages 1–17, 1997.

[49] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 487–501. USENIX Association, April 2021.

[50] Brendan Gregg. *BPF Performance Tools*. Addison-Wesley Professional, 2019.

[51] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon

通过吞吐量和低延迟。在 第*11*届*USENIX*操作系统设计与实现研讨会（*OSDI 14*），第49-65页，科罗拉多州布鲁姆菲尔德，2014年10月。USENIX协会。

[35] Brian N Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E Fiuczynski, David Becker, Craig Chambers, 和 Susan Eggers。SPIN操作系统的可扩展性、安全性和性能。在 第*15*届 *ACM*操作系统原理研讨会，第267-283页，1995年。

[36] Ashish Bijlani 和 Umakishore Ramachandran。用户空间文件系统的扩展框架。在 *2019 USENIX*年度技术会议（*USENIX ATC 19*），第121-134页，华盛顿州雷顿，2019年7月。USENIX协会。

[37] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, 和 Roberto Bifulco. hXDP: 在 FPGA 网卡上高效进行软件数据包处理. 在 第 *14* 届 *USENIX* 操作系统设计与实现研讨会 *(OSDI 20)*, 页面 973–990. USENIX 协会, 2020 年 11 月.

[38] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, Zhenjun Liu, Feng Zhu, 和 Tong Zhang. POLARDB 满足计算存储：在云原生关系型数据库中高效支持分析工作负载. 在 第 *18* 届 *USENIX* 文件与存储技术会议 *(FAST 20)*, 页面 29–41, 圣克拉拉, CA, 2020 年 2 月. USENIX 协会.

[39] Badrish Chandramouli, Guna Prasaad, Donald Koss- mann, Justin Levandoski, James Hunter, and Mike Bar- nett. FASTER: 一个具有原地更新的并发键值存储. 在 *2018*年数据管理国际会议论文集, SIGMOD ' 18, 第275–290页, 纽约, NY, USA, 2018. 计算机协会.

[40]Alexander Conway, Abhishek Gupta, Vijay Chi-dambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. SplinterDB: 关闭NVMe键值存储的带宽差距. 在 *2020 USENIX*年度技术会议 *(USENIX ATC 20)*, 第49–63页, 2020.

[41] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 使用YCSB基准测试云服务系统. 在 第一届*ACM*云计算研讨会论文集, 第143–154页, 2010.

[42] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. 当空闲是理想状态时：使用 perséphone 优化重尾数据中心工作负载的尾部延迟。在 *ACM SIGOPS* 第 *28* 届操作系统原理研讨会论文集, SOSP ' 21, 第 621–637 页，纽约, NY, USA, 2021。计算机协会。

[43] Jaeyoung Do,Sudipta Sengupta, 和 Steven Swanson. 未来云数据中心的可编程固态存储。 *AC M* 通讯, 62(6):54–62, 2019。

[44]Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, 和 Michael Strum. 优化 RocksDB 的空间放大。在 *CIDR*, 第 3 卷 , 第 3 页 , 2017。

[45] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 使用 NVM 减少Facebook的 DRAM 占用。在 第十三届 *EuroSys* 会议论文集，第 1–13 页，2018 年。

[46]Pekka Enberg, Ashwin Rao, and Sasu Tarkoma. 使用 XDP 和 eBPF 进行分区感知数据包调度以提升应用级并行性。在 第一届 *ACM CoNEXT* 网络计算范式新兴研讨会论文集，ENCP ' 19，第 27–33 页，纽约，纽约州，美国，2019年。美国计算机协会。

[47] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan：在毫秒时间尺度上缓解干扰。在 第*14*届 *USENIX* 操作系统设计与实现研讨会（*OSDI 20*），第 281–297 页。USENIX 协会，2020年11月。

[48] Gregory R Ganger 和 M Frans Kaashoek. 嵌入式节点和显式分组：利用磁盘带宽优化小文件。在 *USENIX* 年度技术会议，第 1–17 页，1997 年。

[49] Yoann Ghigoff、Julien Sopena、Kahina Lazri、Antoine Blin 和 Gilles Muller。BMC：使用安全的内核缓存和预堆叠处理加速 memcached。在 第 *18* 届 *USENIX* 网络系统设计与实现会议（*NSDI 21*），第 487–501 页。USENIX 协会，2021 年 4 月。

[50] Brendan Gregg。*BPF* 性能工具。Addison-Wesley 专业出版社，2019 年。

[51] Boncheol Gu、Andre S. Yoon、Duck-Ho Bae、Insoon Jo、Jinyoung Lee、Jonghyun Yoon、Jeong-Uk Kang、Moon-sang Kwon、Chanho Yoon、Sangyeun Cho、Jaeheon

Jeong, and Duckhyun Chang. Biscuit: A framework for near-data processing of big data workloads. *SIGARCH Comput. Archit. News*, 44(3):153–165, jun 2016.

[52] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th international conference on emerging networking experiments and technologies*, pages 54–66, 2018.

[53] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. GhOSt: Fast & flexible user-space delegation of Linux scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 588–604, New York, NY, USA, 2021. Association for Computing Machinery.

[54] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. Rearchitecting linux storage stack for µs latency and high throughput. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 113–128. USENIX Association, July 2021.

[55] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. KAML: A flexible, high-performance key-value SSD. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 373–384. IEEE, 2017.

[56] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, page 52–65, New York, NY, USA, 1997. Association for Computing Machinery.

[57] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for *µ*second-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, 2019.

[58] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. Syrup: User-defined scheduling across the stack. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 605–620, New York, NY, USA, 2021. Association for Computing Machinery.

[59] Junaid Khalid, Eric Rozner, Wesley Felter, Cong Xu, Karthick Rajamani, Alexandre Ferreira, and Aditya Akella. Iron: Isolating network-based CPU in container environments. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 313–328, Renton, WA, April 2018. USENIX Association.

[60] Marios Kogias, Rishabh Iyer, and Edouard Bugnion. Bypassing the load balancer without regrets. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 193–207, 2020.

[61] Gunjae Koo, Kiran Kumar Matam, I Te, HV Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. Summarizer: trading communication with computing near storage. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 219–231. IEEE, 2017.

[62] Kornilios Kourtis, Animesh Trivedi, and Nikolas Ioannou. Safe and efficient remote application code execution on disaggregated NVM storage with eBPF. *arXiv preprint arXiv:2002.11528*, 2020.

[63] Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. Cosmos+ OpenSSD: Rapid prototype for flash storage systems. *ACM Transactions on Storage (TOS)*, 16(3):1–35, 2020.

[64] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, 2015.

[65] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 399–413, New York, NY, USA, 2019. Association for Computing Machinery.

[66] Yoshinori Matsunobu, Siying Dong, and Herman Lee. MyRocks: LSM-tree database storage engine serving Facebook's social graph. *Proceedings of the VLDB Endowment*, 13(12):3217–3230, 2020.

[67] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter 1993 Conference (USENIX Winter 1993 Conference)*, San Diego, CA, January 1993. USENIX Association.

Jeong，和 Duckhyun Chang。Biscuit：一个用于大数据工作负载近数据处理的框架。*SIGARCH* 计算机架构新闻，44(3)：153–165，2016年6月。

[52] Toke Høiland-Jørgensen，Jesper Dangaard Brouer，Daniel Borkmann，John Fastabend，Tom Herbert，David Ahern，和 David Miller。快速数据路径：操作系统内核中的快速可编程数据包处理。在第*14*届新兴网络实验与技术国际会议论文集，第54–66页，2018年。

[53] Jack Tigar Humphries，Neel Natu，Ashwin Chaugule，Ofir Weisse，Barret Rhoden，Josh Don，Luigi Rizzo，Oleg Rombakh，Paul Turner，和 Christos Kozyrakis。GhOSt：快速且灵活的用户空间 Linux调度委托。在 第*28*届*ACM SIGOPS*操作系统原理研讨会论文集，SOSP '21，第588–604页，纽约，纽约州，美国，2021年。计算机协会。

[54] 黄哲勋, Midhul Vuppalapati, Simon Peter, 和 Rachit Agarwal. 重新架构 Linux 存储栈以实现 µs 级延迟和高吞吐量. 在 第 *15* 届 *USENIX* 操作系统设计与实现研讨会 *(OSDI 21)*, 第 113–128 页. USENIX 协会, 2021 年 7 月.

[55] 金艳琴, 汤鸿伟, Yannis Papakonstantinou, 和 Steven Swanson. KAML: 一种灵活、高性能的键值 SSD. 在 *2017* 年 *IEEE* 高性能计算架构国际研讨会 *(HPCA)*, 第 373–384 页. IEEE, 2017.

[56] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, 和 Kenneth Mackenzie. 外核系统上的应用性能和灵活性. 在 第十六届 *ACM* 操作系统原理研讨会 *(SOSP '97)*, 第 52–65 页, 纽约, NY, USA, 1997. 计算机协会.

[57] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, 和 Christos Kozyrakis. Shinjuku: 针对微秒级尾部延迟的抢占式调度. 在 第*16*届*USENIX*网络系统设计与实现会议 *(NSDI 19)*, 第345-360页, 2019年.

[58] Kostis Kaffes, Jack Tigar Humphries, David Mazières, 和 Christos Kozyrakis. Syrup: 跨层级的用户定义调度. 在 *ACM SIGOPS*第*28*届操作系统原理会议, SOSP '21, 第605-620页, 纽约, NY, USA, 2021年. 计算机协会 .

[59] Junaid Khalid, Eric Rozner, Wesley Felter, Cong Xu, Karthick Rajamani, Alexandre Ferreira, 和 Aditya Akella. Iron: 在容器环境中隔离基于网络的 CPU. 在 第*15*届*USENIX*网络系统设计与实现会议 *(NSDI 18)*, 第313-328页, Renton, WA, 2018年4月. USENIX协会.

[60] Marios Kogias、Rishabh Iyer和Edouard Bugnion。绕过负载均衡器而不后悔。在第*11*届*ACM*云计算会议论文集，第193-207页，2020年。

[61] Gunjae Koo、Kiran Kumar Matam、I Te、HV Krishna Giri Narra、Jing Li、Hung-Wei Tseng、Steven Swanson和Murali Annavaram。Summarizer：用近存储的计算换取通信。在*2017*年*IEEE/ACM*第*50*届国际微架构会议（*MICRO*）论文集，第219-231页。IEEE，2017年。

[62] Kornilios Kourtis、Animesh Trivedi和Nikolas Ioannou。在解耦合NVM存储上安全高效的远程应用程序代码执行，使用eBPF。*arXiv*预印本*arXiv:2002.11528*，2020年。

[63] Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. Cosmos+ OpenSSD: 快速原型闪存系统. *ACM Transactions onStorage (TOS)*, 16(3):1–35, 2020.

[64] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: 一种新的闪存文件系统. In *13thUSENIX Conference on File andStorageTechnologies (FAST 15)*, 页面 273–286, 2015.

[65] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: 一种微内核方法实现主机网络. In*Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, 页面 399–413, 纽约, NY, USA, 2019. Association for Computing Machinery.

[66] Yoshinori Matsunobu, Siying Dong, 和 Herman Lee. MyRocks：为 Facebook 社交图谱服务的 LSM 树数据库存储引擎。*VLDB* 杂志,13(12):3217–3230, 2020。

[67] Steven McCanne 和 Van Jacobson. BSD 数据包过滤器：一种新的用户级数据包捕获架构。在 *USENIX* 冬季 *1993* 会议 *(USENIX* 冬季 *1993* 会议*)*, 圣地亚哥，CA，1993 年 1 月。USENIX 协会。

[68] J. Mogul, R. Rashid, and M. Accetta. The packer filter: An efficient mechanism for user-level network code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, SOSP '87, page 39–51, New York, NY, USA, 1987. Association for Computing Machinery.

[69] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, February 2019. USENIX Association.

[70] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[71] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, October 2014. USENIX Association.

[72] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 325–341, New York, NY, USA, 2017. Association for Computing Machinery.

[73] Weikang Qiao, Jieqiong Du, Zhenman Fang, Michael Lo, Mau-Chung Frank Chang, and Jason Cong. High-throughput lossless compression on tightly coupled CPU-FPGA platforms. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 37–44. IEEE, 2018.

[74] Zhenyuan Ruan, Tong He, and Jason Cong. INSIDER: Designing in-storage computing system for emerging high-performance drive. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 379–394, 2019.

[75] Robert Schmid, Max Plauth, Lukas Wenzel, Felix Eberhardt, and Andreas Polze. Accessible near-storage computing with FPGAs. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–12, 2020.

[76] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, page 213–227, New York, NY, USA, 1996. Association for Computing Machinery.

[77] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A user-programmable SSD. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 67–80, Broomfield, CO, October 2014. USENIX Association.

[78] Dharma Shukla, Shireesh Thota, Karthik Raman, Madhan Gajendran, Ankur Shah, Sergii Ziuzin, Krishnan Sundaram, Miguel Gonzalez Guajardo, Anna Wawrzyniak, Samer Boshra, et al. Schema-agnostic indexing with Azure DocumentDB. *Proceedings of the VLDB Endowment*, 8(12):1668–1679, 2015.

[79] Christopher A Small and Margo I Seltzer. Vino: An integrated platform for operating system and database research. 1994.

[80] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. CockroachDB: The resilient geo-distributed SQL database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1493–1509, 2020.

[81] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. Recssd: Near data processing for solid state drive based recommendation inference. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 717–729, New York, NY, USA, 2021. Association for Computing Machinery.

[82] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. SPDK: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161. IEEE, 2017.

[83] Irene Zhang, Jing Liu, Amanda Austin, Michael Lowell Roberts, and Anirudh Badam. I'm not dead yet! the role of the operating system in a kernel-bypass era. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 73–80, 2019.

[84] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay

[68] J. Mogul, R. Rashid, 和 M. Accetta. 打包过滤器：一种高效的用户级网络代码机制。在 **第*11*届*ACM*操作系统原理研讨会论文集**，SOSP’87，第39–51页，纽约，纽约州，美国，1987年。美国计算机协会。

[69] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, 和 Hari Balakrishnan. Shenango：为延迟敏感型数据中心工作负载实现高CPU效率。在**第*16*届*USENIX*网络系统设计与应用研讨会**（***NSDI 19***），第361–378页，波士顿，马萨诸塞州，2019年2月。USENIX协会。

[70] Patrick O’Neil, Edward Cheng,Dieter Gawlick, 和 Elizabeth O’Neil. 日志结构合并树（LSM树）。信息学学报，33(4):351–385，1996年。

[71] 西蒙·彼得、李佳林、张怡、丹·R·K·波茨、道格·沃斯、阿维德·克里希南穆提、托马斯·安德森和蒂莫西·罗斯科。Arrakis：操作系统是控制平面。在 **第*11*届*USENIX*操作系统设计与实施研讨会**（***OSDI 14***），第1-16页，科罗拉多州布鲁姆菲尔德，2014年10月。USENIX协会。

[72] 乔治·普雷卡斯、马arios·科吉亚斯和爱德华·布吉翁。Zygos：实现微秒级网络任务的低尾部延迟。在 **第*26*届操作系统原理研讨会**，SOSP’17，第325-341页，纽约，纽约州，美国，2017年。计算机协会。

[73] 乔伟康、杜洁琼、方振曼、迈克尔·洛、张茂中·弗兰克·昌和金川。在紧密耦合的CPU-FPGA平台上实现高吞吐量无损压缩。在 ***2018*年*IEEE*第*26*届年度现场可编程定制计算机研讨会（*FCCM*）**，第37-44页。IEEE，2018年。

[74] 阮振元, 何通, 和郑重. INSIDER: 为新兴高性能驱动器设计存储内计算系统. 在 ***2019 USENIX*年度技术会议 *(USENIX ATC 19)***, 页面 379–394, 2019.

[75] 罗伯特·施密德, 马克斯·普劳夫, 卢卡斯·温策尔, 菲利克斯·埃伯哈德特, 和安德烈亚斯·波尔策. 使用FPGA的近存储计算. 在 **第十五届欧洲计算机系统会议论文集**, 页面 1– 12, 2020.

[76] 马戈·I·塞尔策, 汤川康弘, 克里斯托弗·斯莫尔, 和基思·A·史密斯. 应对灾难：生存恶性行为的内核扩展. 在第二届**操作系统设计与实现*USENIX*研讨会**, OSDI’96, 页面 213–227, 纽约, NY, USA, 1996. 计算机协会.

[77]Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, 和 Steven Swanson. Willow: 一个用户可编程的SSD. 在 **第*11*届*USENIX*操作系统设计与实现研讨会 *(OSDI 14)***, 第67-80页, Broomfield, CO, 2014年10月. USENIX协会.

[78] Dharma Shukla, Shireesh Thota, Karthik Raman, Madhan Gajendran, Ankur Shah, Sergii Ziuzin, Krishnan Sundaram, Miguel Gonzalez Guajardo, Anna Wawrzyniak, Samer Boshra, 等人. Azure DocumentDB的Schema无关索引. ***VLDB*基金会会议录**, 8(12):1668-1679, 2015.

[79] Christopher A Small 和 Margo I Seltzer. Vino: 一个用于操作系统和数据库研究的集成平台. 1994.

[80] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan Van- Benschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, 等. Cock- roachDB: 弹性地理分布式SQL数据库. 在***ACM SIGMOD*国际数据管理会议论文集**, 第1493–1509页, 2020.

[81] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, 和 Gu-Yeon Wei. Recssd: 基于固态硬盘的推荐推理近数据处理. 在 **第*26*届*ACM*国际编程语言与操作系统架构支持会议论文集**, ASPLOS 2021, 第717–729页, 纽约, NY, USA, 2021. ACM协会.

[82] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, 和 Luse E Paul. SPDK: 用于构建高性能存储应用程序的开发工具包. 在 ***2017 IEEE* 云计算技术与科学国际会议（*CloudCom*）论文集**, 第154–161页. IEEE, 2017.

张 Irene，刘 Jing，奥斯汀 Amanda，罗伯茨 Michael Lowell，巴达姆 Anirudh。我还没死呢！操作系统在内核绕过时代的作用。在《操作系统热点专题研讨会论文集》中，第73-80页，2019年。

[84] 张薇，雷巴克，帕特尔，奥利尼克，尼尔森，奥马尔·S·纳瓦罗·利哈，阿什莉·马丁内斯，刘静，科恩菲尔德·辛普森，苏贾伊

Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 195–211, New York, NY, USA, 2021. Association for Computing Machinery.

[85] Yuhong Zhong, Hongyi Wang, Yu Jian Wu, Asaf Cidon, Ryan Stutsman, Amy Tai, and Junfeng Yang. BPF for storage: An exokernel-inspired approach. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 128–135, New York, NY, USA, 2021. Association for Computing Machinery.

Jayakar、Pedro Henrique Penna、Max Demoulin、Piali Choudhury 和 Anirudh Badam。The demikernel dat- apath os 架构用于微秒级数据中心系统。在 *ACM SIGOPS 第 28* 届操作系统原理研讨会论文集，SOSP ' 21，第 195–211 页，纽约，纽约州，美国，2021 年。Association for Computing Machinery。

[85] Yuhong Zhong、Hongyi Wang、Yu Jian Wu、Asaf Cidon、Ryan Stutsman、Amy Tai 和 Junfeng Yang。BPF for storage: An exokernel-inspired approach。在 操作系统热点问题研讨会论文集，HotOS ' 21，第 128–135 页，纽约，纽约州，美国，2021 年。Association for Computing Machinery。

A Artifact Appendix

Abstract

We open-source XRP, a high-performance storage data path using Linux eBPF. The artifact includes the implementation of XRP in the Linux kernel and two key-value stores that leverage XRP to significantly improve throughput and latency.

Scope

The artifact allows readers to run all the experiments in §6 and generate Table 3, Figure 5, Figure 6, Figure 7, Figure 8, Figure 9, and Figure 10.

Contents

The artifact provides the following parts.

1. XRP: the implementation of XRP in the Linux kernel v5.12.0.
2. BPF-KV: a simple key-value store that uses XRP to accelerate both point and range lookups.
3. WiredTiger: a modified WiredTiger (based on v4.4.0) that integrates with XRP to speed up index lookups.
4. My-YCSB: an efficient YCSB benchmark written in C++ for WiredTiger.

Test scripts and drawing scripts are also provided for all the experiments and results in §6.

Hosting

The artifact is hosted on the main branch (commit `fae90c5`) of the Github repository https://github.com/xrp-project/XRP.

Requirements

XRP requires a low latency NVMe SSD on which the overhead of the Linux storage stack is significant. We use Intel Optane SSD P5800X in all the experiments. In the test scripts, we assume that the operating system is Ubuntu 20.04, and there are 6 physical CPU cores on the machine. Other configurations may require changing the scripts accordingly.

A 资产 附录

摘要

我们开源了 XRP，一种使用 Linux eBPF 的高性能存储数据路径。该资产包括 XRP 在 Linux 内核中的实现以及两个利用 XRP 的键值存储，它们显著提高了吞吐量和延迟。

范围

该资产允许读者运行 §6中的所有实验并生成表3、图5、图6、图7、图8、图9、图10。

目录

该工件提供以下部分。

1. XRP：Linux内核v5.12.0中的XRP实现。

2. BPF-KV：一个简单的键值存储，使用XRP来加速点和范围查找。3. WiredTiger：一个修改后的WiredTiger（基于v4.4.0），与XRP集成以加快索引查找。

4. My-YCSB：一个用C++编写的针对WiredTiger的高效YCSB基准测试。所有测试脚本和绘图脚本也提供。

§中的实验和结果6。

Hosting

该工件托管在Github仓库的主分支（提交 `fae90c5`）上 https://github.com/xrp-project/XRP。

Requirements

XRP需要一个低延迟的NVMe SSD，Linux存储堆栈的开销在该SSD上非常显著。我们在所有实验中都使用Intel Optane SSD P5800X。在testscripts中，我们假设操作系统是Ubuntu 20.04，并且机器上有6个物理CPU核心。其他配置可能需要相应地更改脚本。