

Optimizing Memory and Storage Performance in Cloud Datacenters

Ioannis Zarkadas

Submitted in partial fulfillment of the  
requirements for the degree of  
Doctor of Philosophy  
under the Executive Committee  
of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2025

© 2025

Ioannis Zarkadas

All Rights Reserved

## **Abstract**

Optimizing Memory and Storage Performance in Cloud Datacenters

Ioannis Zarkadas

Data-intensive applications increasingly dominate modern datacenters. This surge is propelled by a number of applications including AI/ML training and inference, HPC, data lakes, and large-scale cloud storage systems. However, software overhead remains a persistent challenge, with studies showing nearly half of cloud computing cycles wasted. As storage speeds outpace traditional system architectures, operating system overhead has emerged as a dominant bottleneck constraining overall system performance.

This thesis addresses these critical inefficiencies through a three-pronged approach. First, we revisit extensible operating systems as an effective way to optimize the storage and memory stack in cloud datacenters. Second, we uncover new sources of memory usage inefficiencies in ML accelerators, by constructing novel performance profiling tools. Third, we revisit and optimize traditional distributed protocols.

This thesis addresses these critical inefficiencies through a three-pronged approach that targets key sources of storage and memory overhead. First, I focus on developing efficient storage and memory stacks through OS extensions, enhancing system efficiency through bypassing OS layers and by enabling flexible policies for key OS components.

Second, I target suboptimal memory utilization in ML accelerators by developing novel performance debugging tools to analyze the low-level interaction of model code with the micro-architecture. This unlocks new insights into model inefficiencies and how to address them.

Third, I address storage efficiency through revisiting traditional storage protocols like replication. More specifically, I enhance async replication with strong staleness guarantees and fast failover, enabling better application performance and storage utilization.

Collectively, this thesis aims to significantly reduce storage and memory overhead in cloud datacenters, ultimately enabling more sustainable and cost-effective cloud computing infrastructure.

# Table of Contents

Acknowledgments . . . . .	x
Dedication . . . . .	xii
Chapter 1: Introduction . . . . .	1
1.1 Related work . . . . .	2
1.1.1 BPF-oF . . . . .	2
1.1.2 cache_ext . . . . .	3
1.1.3 xPU-Shark . . . . .	4
1.1.4 Rosé . . . . .	5
Chapter 2: BPF-oF: Storage Function Pushdown Over the Network . . . . .	7
2.1 Introduction . . . . .	7
2.2 Background and Motivation . . . . .	11
2.2.1 Benefits of Networked Storage . . . . .	11
2.2.2 NVMe-oF Primer . . . . .	12
2.2.3 Improving NVMe-oF CPU Efficiency . . . . .	14
2.2.4 eBPF and XRP . . . . .	15
2.3 Challenges . . . . .	16
2.4 BPF-oF Design . . . . .	17

2.4.1	Principles . . . . .	17
2.4.2	Architecture . . . . .	18
2.5	Supporting In-Memory Data Structures . . . . .	22
2.5.1	RocksDB Architecture . . . . .	23
2.5.2	Integration with BPF-oF . . . . .	24
2.6	Evaluation . . . . .	27
2.6.1	Experimental Setup . . . . .	27
2.6.2	BPF-oF With Different Setups (Q1) . . . . .	30
2.6.3	CPU and Network Savings (Q2) . . . . .	31
2.6.4	Sampling Rate (Q3) . . . . .	32
2.6.5	Version Mismatches (Q4) . . . . .	33
2.6.6	WiredTiger (Q5) . . . . .	33
2.6.7	BPF-KV (Q5) . . . . .	34
2.6.8	Takeaways . . . . .	35
2.7	Conclusion . . . . .	36
<b>Chapter 3: <code>cache_ext</code>: Customize Page Eviction With eBPF . . . . .</b>		<b>37</b>
3.1	Introduction . . . . .	37
3.2	Background and Motivation . . . . .	40
3.2.1	Linux Page Cache . . . . .	41
3.2.2	eBPF . . . . .	43
3.3	Challenges . . . . .	43
3.4	Design and Implementation . . . . .	44

3.4.1	Policies in Kernel or Userspace? . . . . .	45
3.4.2	Interface . . . . .	46
3.4.3	Isolation . . . . .	52
3.4.4	Security . . . . .	53
3.4.5	Kernel Implementation Complexity . . . . .	54
3.5	Policies . . . . .	54
3.5.1	S3-FIFO . . . . .	55
3.5.2	Least Hit Density (LHD) . . . . .	55
3.5.3	Multi-Generational LRU (MGLRU) . . . . .	57
3.5.4	Classic Policies: LFU, MRU, FIFO . . . . .	57
3.5.5	Application-Informed Eviction (GET-SCAN) . . . . .	58
3.5.6	Application-Informed Admission Filter . . . . .	59
3.6	Evaluation . . . . .	60
3.6.1	Custom Policies (Q1) . . . . .	61
3.6.2	Isolation (Q2) . . . . .	65
3.6.3	Memory and CPU Overhead (Q3) . . . . .	66
3.7	Conclusions . . . . .	68
Chapter 4:	xPU-Shark: A new way to analyze uarch-scale performance bottlenecks for ML accelerators . . . . .	69
4.1	Introduction . . . . .	69
4.2	Background and Related Work . . . . .	72
4.2.1	ML Software Stack . . . . .	72
4.2.2	Profilers . . . . .	73

4.3	Design Requirements . . . . .	76
4.4	Design and Implementation . . . . .	77
4.4.1	Execution Recorder . . . . .	78
4.4.2	Replayer . . . . .	79
4.4.3	Analyzer . . . . .	80
4.4.4	Workflow . . . . .	81
4.5	Evaluating Models With xPU-Shark . . . . .	82
4.5.1	Experimental Setup . . . . .	82
4.5.2	DMA Stall Analysis . . . . .	82
4.5.3	Detailed Microarchitectural Utilization . . . . .	87
4.5.4	Dependency Analysis . . . . .	91
4.5.5	xPU-Shark's Overhead . . . . .	94
Chapter 5: Rosé: Flexible Replication With Strong Semantics		
	For Partitioned Databases . . . . .	95
5.1	Introduction . . . . .	95
5.2	Background and Related Work . . . . .	97
5.2.1	Replication . . . . .	97
5.2.2	Chablis . . . . .	98
5.3	Challenges . . . . .	99
5.4	Rosé Protocol . . . . .	100
5.4.1	Maintaining Monotonic Prefix Consistency . . . . .	101
5.4.2	Bounding the Replication Lag . . . . .	101
5.4.3	Minimizing Time to Recovery . . . . .	104

5.5	Evaluation . . . . .	106
5.5.1	Setup . . . . .	106
5.5.2	Q1: Capping the replication lag . . . . .	106
5.5.3	Q2: Recovery with Rosé . . . . .	107
	References . . . . .	109

## List of Figures

2.1	(a) Default: each server has its own storage. (b) Networked storage: storage is centralized in storage nodes and accessed over the network. . . . .	12
2.2	NVMe-oF overview. . . . .	13
2.3	RocksDB run locally compared to NVMe/TCP on a NAND SSD. . . . .	14
2.4	BPF-oF architecture. . . . .	18
2.5	BPF-oF eBPF functions control resubmission via this struct. . . . .	22
2.6	BPF-oF read system call. . . . .	23
2.7	RocksDB LSM-tree traversal and query splitting. . . . .	25
2.8	RocksDB (a) uniform read throughput-latency and throughput (b) without and (c) with data blocks cached. BPF-oF vs. NVMe/TCP on NAND SSD. . . . .	28
2.9	RocksDB throughput under different configurations. . . . .	28
2.10	RocksDB throughput with different sampling rates with cache under BPF-oF using TCP and NAND SSD. . . . .	32
2.11	WiredTiger and BPF-KV evaluations with TCP. . . . .	33
2.12	Tail latency and throughput of BPF-oF vs. NVMe-oF on BPF-KV with random 512B reads. . . . .	35
3.1	Overview of the current Linux page cache eviction policy. . . . .	43
3.2	Overview of <code>cache_ext</code> . Eviction lists hold pointers to folios. . . . .	45
3.3	<code>struct_ops</code> for <code>cache_ext</code> and eviction context. . . . .	48

3.4	Simplified LFU implementation with <code>cache_ext</code> . . . . .	51
3.5	Overview of GET-SCAN policy implementation. . . . .	59
3.6	YCSB workload results. . . . .	60
3.7	Twitter workload results (LHD, S3-FIFO, and LFU policies) using LevelDB. No one policy performs best across the different clusters. . . . .	62
3.8	Mixed GET-SCAN workload results. . . . .	63
3.9	File search workload results (MRU policy). . . . .	65
3.10	Using <code>cache_ext</code> , multiple applications can run different eviction policies, yielding better performance for all. . . . .	65
4.1	Overview of xPU-Shark. . . . .	72
4.2	A toy example showing a coarse-grained vs fine-grained analysis. Many existing tools provide coarse-grained information at the kernel or HLO level, while deep optimization requires a fine-grained view. . . . .	72
4.3	Example machine-code instructions for handling memory transfers in an ML model. The ISSUE command starts the DMA, while the WAIT command blocks until it is complete and is typically inserted close to the command that needs the memory. The compiler can hide the DMA latency by inserting instructions between ISSUE and WAIT. . . . .	84
4.4	Lifetime of a DMA. A DMA is comprised of a base latency (constant) and a transfer latency (variable). DMAs begin with the ISSUE command. If the DMA has not completed by the time the WAIT command executes, the accelerator will stall until the DMA completes. We highlight three distinct scenarios. (1) WAIT comes before the base latency is fulfilled. The DMA incurs stalls first because of the base latency (green) and then because of the transfer latency (purple). (2) WAIT comes after the base latency but before the transfer latency is fulfilled. The DMA incurs stalls only because of the transfer latency (purple). (3) WAIT comes after the DMA finishes. The time between the DMA completion and the WAIT is slack (gray cross pattern). . . . .	85
4.5	All-Gather DMA Pattern. Memory accesses are performed in two phases, the setup phase, and data transfer phase. Dependencies for the setup phase are shown with red arrows. These dependencies were manually discovered by reading the machine-code, a laborious process. . . . .	85

4.6	Reduction in runtime for the All-Gather collective operation after eliminating unnecessary DMAs. Runtime is normalized to the median value before optimization. . . . .	87
4.7	xPU-Shark can make it easy to diagnose low matrix multiplication utilization. We annotate the completion of each DMA, and the corresponding cycle on the MU utilization (Figure 4.7b). We see that the matrix multiplier is waiting on data transfers, resulting in low utilization. At the same time, the data-cache has enough is not fully utilized, so we might be able to prefetch more data. . . . .	88
4.8	Figure 4.8a shows the percentage of data-cache space which is unused, and the percentage which is unused and contiguous. The largest contiguous region is consistently much smaller than the total unused portion. Figure 4.8b is the data-cache fragmentation analysis. The total memory space is divided into pages, where each read and write is tracked. Pages are then grouped into “buckets” of 256 pages. Each bucket is a horizontal line in the plot and its color denotes the number of used pages in the bucket for each cycle. We see considerable fragmentation of our data-cache, which is critical for performance. . . . .	89
4.9	Different models of dependency tracking. In the conservative models, an instruction depends on the immediate instructions that shape its input. In the relaxed model, dependencies are propagated until the dependent memory is in the data-cache. . . . .	92
4.10	Example dependency analysis with xPU-Shark, showing the DMA analysis of section 4.5.3 with dependency information overlaid as "backtails". Thanks to that, the user can immediately conclude the DMAs can be issued earlier, without needing to manually analyze the machine-code. . . . .	93
5.1	Chablis architecture. . . . .	99
5.2	Rosé overview. . . . .	100
5.3	Yugabyte recovery of a single partition at t=3. . . . .	105
5.4	Rosé backpressure to cap replication lag. . . . .	107
5.5	Coordinated apply and its impact on failover. . . . .	108

## List of Tables

2.1	Resource consumption with uniform read and 1 GB data cache. . . . .	32
3.1	Performance of workloads without and with userspace-dispatch. . . . .	46
3.2	cache_ext eviction list API. . . . .	49
3.3	Lines of eBPF and userspace loader code in cache_ext policies. . . . .	65
3.4	cache_ext $\mu$ CPU per I/O operation using fio. . . . .	67
3.5	Relative performance of cache_ext vs. baseline MGLRU with YCSB. The harmonic mean is 0.99, indicating a 1% slowdown on average. . . . .	68
4.1	Model descriptions. We evaluate three important in-house LLMs. . . . .	82

## **Acknowledgements**

I would like to thank my advisor, Asaf Cidon, for all his guidance and support during this PhD. He taught me how to conduct research and think about problems, how to present and effectively communicate my ideas. Equally important, he was always a voice of optimism and encouragement, helping me reframe doubt into excitement. He deeply cares about his students and I thank him for fostering a safe and welcoming environment for us during these years. I want to also thank Ofir Weisse, a mentor and core collaborator, for taking a chance on me and teaching me to persist through the most challenging problems.

Second, I want to thank my family for all their love and support. My parents, Christos and Lefki, for all their sacrifices throughout the years so we could have the best education. My sister, Simoni, for always supporting me. My grandmother, Rodoula, for always believing in us. I am immensely lucky and grateful to have you. My late uncle, Ilias, for his love and dedication.

Moreover, I want to thank my girlfriend, Marousa, for supporting me and believing in me. We met during this PhD in turbulent times, and I couldn't have made it without her love and encouragement.

I am immensely grateful to all my lifelong friends. We have grown up together, been through good and hard times and share precious memories. I treasure all of you and your friendship. To Orfeas and Panagiotis, who ended up in the US with me: thank you for making my time here so much better.

Finally, I'm thankful for all the people I met during my time at Columbia and New York. My friends and collaborators: Tal, Jeremy, Kelly, Tamer, Kostis, Hubertus, Ryan and Amanda. My

labmates Wei, Yuhong and Haoyu who I really appreciate.

Switching to authorship acknowledgments, I note that much of this work is the result of academic collaboration. Tal Zussman is an equal co-author in `BPF-oF` and `cache_ext`. Amanda Tomlinson is an equal co-author in `xPU-Shark`.

## **Dedication**

Dedicated to my parents, Christos and Lefki.

## Chapter 1: Introduction

Data-intensive applications increasingly dominate modern datacenters. This surge is propelled by a number of applications including AI/ML training and inference, HPC, data lakes, and large-scale cloud storage systems. However, software overhead remains a persistent challenge, with studies showing nearly half of cloud computing cycles wasted. As storage speeds outpace traditional system architectures, operating system overhead has emerged as a dominant bottleneck constraining overall system performance.

However, significant overhead remains a challenge, with studies showing nearly half of cloud computing cycles wasted. As storage speeds outpace traditional system architectures, operating system overhead has become a dominant bottleneck.

This thesis proposal addresses these inefficiencies through a three-pronged approach:

1. **Efficient Storage and Memory Stack through OS Extensions:** Enhancing efficiency by enabling applications to safely and dynamically extend OS functionality.
2. **Low-level Memory Performance Profiling:** Developing new debugging tools for accelerators like TPUs, optimizing memory usage, identifying inefficiencies, and guiding targeted optimizations.
3. **Improved Distributed Storage Protocols:** Creating replication protocols that guarantee strong data consistency and enable fast failover, addressing issues where current practices lead to unnecessary storage overhead and potential data contamination.

The rest of the thesis is structured as follows. First, I show how storage and memory efficiency can be improved through OS extensions. `BPF-OF` (Chapter 2) increases storage efficiency through bypassing OS layers, while `cache_ext` (Chapter 3) increases memory efficiency by enabling applications to use flexible page cache eviction policies. Second, `xPU-Shark` (Chapter 4) increases memory efficiency in machine learning accelerators. More specifically, it combines

hardware simulation with performance analysis in a real accelerator, revealing new microarchitectural inefficiencies. Third, in *ROSÉ* (Chapter 5), I target storage efficiency by revisiting and enhancing replication protocols. The rest of this chapter covers the related work.

## 1.1 Related work

### 1.1.1 BPF-OF

**eBPF for pushdown** Several recent systems leverage eBPF to move computation closer to storage. Zhong et al. propose XRP [1], an eBPF-based mechanism for resubmitting storage operations on locally-attached Optane SSDs. XRP focuses on local environments and simple key-value stores, whereas BPF-OF targets networked environments and integrates with production storage systems like RocksDB. Kourtis et al. explore executing user-defined functions on remote NVM storage servers using eBPF [2]. Their approach runs functions in userspace on the remote server without integrating with standard networked storage protocols. However, their work remains preliminary and does not present a complete system design, implementation, or evaluation. BMC [3] uses eBPF to implement a remote kernel-level cache for memcached requests, aiming to reduce kernel-userspace crossings. Unlike BPF-OF, BMC does not support custom user-defined functions at the server. ExtFUSE [4] allows bypassing parts of the kernel and file system to reduce overhead and avoid context switches, while  $\lambda$ -IO [5] offers a BPF framework to run custom functions inside computational storage devices. Neither system addresses networked storage scenarios. Finally, Electrode [6] and DINT [7] apply eBPF extensions to accelerate distributed protocols like Multi-Paxos. Their work on distributed protocol acceleration is complementary to BPF-OF.

**Other pushdown frameworks** Splinter [8], ASFP [9], and Kayak [10] are in-memory key-value stores that allow users to ship lightweight Rust functions, executed as co-routines on a remote in-memory server. Redis [11] offers similar functionality using Lua. Unlike BPF-OF, these systems perform pushdown within the context of a specific in-memory key-value store, rather than as part of a general-purpose networked storage protocol. Additionally, operating purely in-memory avoids

the challenges associated with storage systems, such as file metadata synchronization. Some production storage systems support limited application-specific pushdowns. Amazon S3 Select [12] enables users to apply SQL filters directly within S3 to reduce network transfer costs. Ceph [13] offers similar functionality through Ceph extensions. However, these approaches do not provide a general-purpose pushdown mechanism for storage.

### 1.1.2 `cache_ext`

**Page cache customization in Linux.** Recent work has explored using eBPF to customize memory management policies, including huge page placement, page fault handling, and page table designs [14, 15, 16]. P2Cache [17] is a recent proposal for customizing page cache policies with eBPF. Although conceptually similar to `cache_ext`, P2Cache is limited to LRU or MRU ordering with a single queue, restricting the range of policies it can support. Furthermore, P2Cache overrides the global page cache policy for all processes, whereas `cache_ext` supports per-cgroup multi-tenancy. P2Cache is also a work-in-progress workshop paper, closed-source, and lacks detailed design, implementation, and evaluation information. FetchBPF [18] provides a mechanism for customizing Linux’s memory prefetching policy and could be integrated into `cache_ext` as an additional hook. However, prefetching typically relies on simpler data structures than eviction policies, and FetchBPF does not address multi-tenant isolation or application-aware customization. Interestingly, the developers of Linux’s MGLRU have proposed adding eBPF hooks to control page generation placement [19], but no design or implementation has yet been presented.

**Extensible kernels.** In the 1980s and 1990s, page cache customization emerged as a use case for extensible kernels [20, 21, 22, 23], which aimed to allow applications to modify kernel interfaces and policies. Systems like VINO [21, 24] and SPIN [20] enabled applications to customize buffer cache eviction, admission, and prefetching behavior. Although these designs did not achieve widespread adoption, their core ideas have resurfaced with the rise of eBPF, which now provides similar extensibility within monolithic kernels like Linux and Windows.

**Userspace caches.** An alternative approach is for applications to implement their own userspace cache and bypass the OS page cache using direct I/O. Many data systems adopt this strategy [25, 26, 27]. TriCache [28] is a recent framework designed to help applications build customizable userspace caches. However, despite these efforts, many widely-used data systems still rely on the OS page cache, often alongside their userspace caches [25, 29, 30, 31, 32].

### 1.1.3 xPU-Shark

**PMU-based Profiling.** Vendors provide hardware support for recording performance events through performance monitoring units (PMUs), along with profilers that leverage them. The PMU consists of a set of registers (performance counters) that can track events such as cache hits or measure statistics like the number of cycles spent in a function. Profilers that use PMU data include Nsight Systems [33], nvprof [34], Intel VTune [35], AMD ROC-profiler [36], and Google Tensorboard [37]. For example, Nsight provides utilization metrics at the CUDA kernel level, while Tensorboard offers metrics at the HLO level. These tools are lightweight, but PMUs are constrained by limited buffer sizes and can only track a small set of events simultaneously [38].

**PC Sampling.** Beyond PMUs, some vendors (e.g., Nvidia) offer program counter (PC) sampling and stall attribution. PC sampling requires additional hardware support to capture snapshots of the PC during execution.

If a stall occurs at sampling time, the hardware can attribute the stall reason (e.g., memory stall, synchronization stall).

Tools such as CUPTI [39], VTune [35], HPCToolkit [40], and DrGPU [41] use PC sampling to gather stack traces, coalesce stall reasons, and suggest optimization strategies. While useful for identifying performance bottlenecks, PC sampling has limitations: it requires specialized hardware (not available on all accelerators, including ours), pinpoints symptoms rather than root causes (e.g., memory transfer stalls), and provides no insights into overall hardware utilization.

**Instrumentation-Based Profiling.** Another approach is binary instrumentation, which enables fine-grained analysis at the cost of high overhead. Instrumentation engines like Nvidia NVBit [42],

SASSI [43], Sanitizer API [44], Intel GTPin [45], and LLVM [46] can modify binaries to record every instruction’s behavior. This technique is used to deeply analyze software behavior. For instance, VALUEEXPECT [47] traces loads and stores to uncover hidden sparsity or redundant computation, and CUDAAdvisor [48] tracks memory accesses to compute reuse distances. However, instrumentation changes the program’s execution characteristics, rendering PMU data unreliable, and typically requires recompilation—making it impractical in complex, frequently-changing hyperscale environments.

**Simulation.** Hardware simulators can measure every aspect of hardware behavior without PMU constraints but incur significant performance overhead. Simulators like GPGPU-Sim [49, 50, 51] attempt to replicate architectures like Nvidia’s using publicly available information. While useful for exploring hardware design choices (e.g., interconnect networks), simulators struggle with analyzing real-world workloads due to their approximate modeling of hardware.

**Cross-cutting Techniques.** Some tools combine multiple profiling methods to achieve deeper analysis. GPA [52] integrates PC sampling and instrumentation to locate inefficient code regions, analyze dependencies, and suggest root causes, although it does not offer precise optimization advice (i.e. relocate specific instructions). It also relies on hardware PC sampling support, which is not universally available. Nsight Compute [33] is the most comprehensive tool available for CUDA performance analysis, providing PTX-level insights, detecting instruction dependencies, and flagging inefficiencies. However, it analyzes one kernel at a time, lacks low-level SASS insights, and is proprietary and opaque.

#### 1.1.4 Rosé

**Replication** Distributed databases seek to support high-availability and durability, such that when some servers fail, data is not lost and the rest of the servers can continue providing the db functionality. These objectives are commonly satisfied using replication. Replication can be broadly categorized as (1) synchronous and (2) asynchronous. Synchronous replication ensures strong consistency by requiring that transactions do not commit until their writes have been repli-

cated across all designated replicas. This approach is well-suited for single-region deployments where fast network connectivity minimizes overheads, but becomes problematic for cross-regional replication where network delays significantly inflate transaction latency. Two primary approaches dominate the landscape: state machine replication and primary-backup protocols. State machine replication, exemplified by consensus protocols such as Paxos [53] and Raft [54], integrates consensus and replication into a unified protocol, offering very high availability guarantees at the cost of resource efficiency. In contrast, approaches like simple primary-backup, chain replication [55], GFS, CRAQ [56], and Hermes [57], rely on fixed replica sets with specific role assignments coordinated by an external consistent service, trading some availability for improved resource efficiency. Modern distributed databases such as CockroachDB [58] and Spanner [59] often employ synchronous replication across regions to support cross-region durability and availability. However, this design choice imposes a substantial latency penalty on all transactions, even those whose write-sets exhibit strong regional locality. This high latency compounds into increased lock contention and higher abort rates, particularly for transactions involving frequently accessed keys.

## Chapter 2: NVMe-oF: Storage Function Pushdown Over the Network

### 2.1 Introduction

In many datacenter and enterprise settings, storage is accessed over the network, often in the form of a storage area network (SAN). In these settings, as shown in Figure 2.1, storage devices are grouped into big storage servers (i.e. storage appliances) and multiple applications or clients can access the same set of storage devices over a shared or exclusive network, such as Ethernet or Fibre Channel. The benefit of this approach is the storage drives can be scaled, maintained, and backed up independently of the application [60, 61, 62, 63, 64, 65, 66, 67, 68]. As datacenter networks have become lower latency ( $\sim 10 \mu\text{s}$  latency) and higher bandwidth, to the point where they can saturate the I/O bandwidth of storage devices, networked storage becomes even more attractive [63, 69, 65]. Consequently, a variety of data-intensive applications (e.g., transactional databases [70, 71], data analytics [72], key-value stores [73], and data warehouses [74]) in both cloud and on-prem settings use networked storage.

NVMe-oF (NVMe over Fabrics) is the state-of-the-art protocol for networked storage, supported by all SAN major products [60, 61, 62]. NVMe-oF allows an application to transparently access an NVMe block device on a remote server. NVMe-oF supports TCP (NVMe/TCP) and RDMA (NVMe/RDMA), is fully supported by Linux, and can be used without specialized hardware. The major downsides of NVMe-oF (compared to accessing a device locally via NVMe) are the added network latency, the CPU cost of processing the network packets and the increased network bandwidth consumption. The CPU cost is especially substantial in the case of TCP, incurring a nearly 50% throughput decrease compared to local performance in our experiments (§2.2.3). We later show that even in the case of RDMA, where this cost can be partially mitigated by offloading network processing to NICs, network processing still incurs a significant toll.

With the goal of reducing this cost, we observe that in modern key-value storage systems, many queries, such as lookups and aggregations, involve a series of dependent I/O accesses (e.g., a B-tree index lookup), which translate into multiple network round trips for each query in the networked storage setup. Therefore, one approach to amortize the network processing cost of networked storage is to “pushdown” dependent I/O accesses to the remote storage server, thereby eliminating the need to process multiple round-trip network requests for each logical query. This idea itself is not new, and there are many examples of academic and commercial systems that push down functions over the network to operate closer to the data [2, 8, 10, 9, 13, 12]. However, these systems all implement this capability as an *application-specific feature*. Restructuring an application or database to allow it to run its operations on a remote server may entail many intrusive modifications, and it requires developers to contend with how to implement such a capability while maintaining security and concurrency guarantees.

We take a cue from recent work [1, 2, 5, 75] that has employed eBPF (extended Berkeley Packet Filter) for safely executing user-defined storage functions closer to the storage device in Linux [1, 75] or a programmable storage device [5], thereby reducing the CPU overhead of traversing the kernel storage stack. Specifically, XRP [1] executes storage functions within the device driver and primarily benefits workloads that require dependent I/O requests. However, these prior works assume pushdown occurs locally (i.e. on the same server) or on simple storage systems.

Two significant challenges arise when trying to adopt a similar approach in a networked storage setting with a modern storage system. First, due to the design of block-level protocols like NVMe-oF, the programmable function that executes on the remote server operates below the file system layer (i.e. at the NVMe driver). However, user-defined storage functions, such as index traversals, operate on files, which the remote device driver may not be aware of. Second, in modern storage systems, data is stored in large data structures (e.g., LSM-trees or B-trees), and parts of these data structures may be partially (or fully) cached in memory and are frequently updated. However, in a networked setting, these in-memory structures would not be accessible at the remote server, which runs the I/O functions. Thus, if we use remote-storage pushdown, we may lose the benefit of these

in-memory caches, hurting performance.

We solve these challenges by designing BPF-oF, a new networked-storage pushdown protocol for TCP and RDMA that is built on top of NVMe-oF, allowing a client to submit custom eBPF storage functions and safely trigger them at a remote storage server. BPF-oF builds upon XRP, extending it to support multi-file queries, while also addressing networked storage-specific challenges. In order to tackle the first challenge, BPF-oF synchronizes various file system metadata across the host and client machines. BPF-oF uses this information to perform the file-descriptor-to-inode translation at the client. To keep the file-to-block mappings up to date, BPF-oF updates them *asynchronously*, thereby avoiding blocking concurrent storage functions. To avoid conflicts when storage functions execute concurrently with metadata synchronization, BPF-oF sets a version for each inode’s file-to-block mapping, and checks for modified versions before and after a request. We implement BPF-oF as a new NVMe command that is triggered by a system call from userspace.

We tackle the second challenge using a new technique called *query splitting*, which divides queries into in-memory and on-disk portions. Accessing in-memory data structures is often orders of magnitude cheaper than a single storage I/O. Therefore, the in-memory portion of the query speculatively checks all the potentially-required in-memory (cached) data structures at the client (some of which may eventually not be needed), and then, based on the results, executes the remaining on-disk portion of the query at the remote storage server. While this approach may result in “superfluous” accesses to in-memory data structures (e.g., looking up bloom filters for files that will never be used), it makes the on-disk portion of the query much more efficient by grouping all I/O accesses together. Another caching-related challenge is that the client cache may become stale, as it is not updated with the data that is accessed by the remote function. One potential solution to this problem is to simply return the intermediate data accessed by the remote storage query, but this would increase the network consumption between the client and remote server, negating much of the benefit of BPF-oF. Instead, we solve this problem using a lightweight technique of *cache sampling*, where a small percentage (by default 1%) of storage functions do not get offloaded, and

fully update the client cache, keeping it fresh.

Our experiments yield another surprising result: for some workloads, maintaining a client data cache with BPF-oF actually *hurts* performance, because BPF-oF is very CPU efficient and the cost of maintaining the client cache is higher than its benefit. For example, when we run BPF-oF on fast modern NVMe devices, it is often better to use BPF-oF with no cache (beyond indices) at the client!

We focus primarily on accelerating key-value stores (e.g., RocksDB, WiredTiger), which are widely used as storage engines in large-scale production storage systems (e.g., MySQL [76], ZipyDB [77], CockroachDB [78], MongoDB [79]). While BPF-oF requires making some modifications to the key-value stores that use it, it does not require any changes to the applications or databases which build on top of these key-value stores, thereby enabling a wide range of systems to utilize storage pushdown over the network.

We integrate BPF-oF with three storage applications: RocksDB [25] a popular and performance-optimized storage system used by Meta and many other companies [80], BPF-KV [1], a custom eBPF-friendly key-value store, and WiredTiger [30], a storage engine used by MongoDB. Our integration required a relatively modest change to RocksDB and WiredTiger: about 1,900 and 500 lines of code, respectively. BPF-oF provides significant performance boosts to all three systems: under the vast majority of workloads BPF-oF accelerates RocksDB by up to 2.8× and reduces 99<sup>th</sup>-percentile latency by up to 2.6×; it accelerates BPF-KV by up to 8×; and it improves WiredTiger’s throughput by up to 30%. These gains are most apparent in read-heavy workloads and workloads where the working set does not fit in cache, but they translate to other workloads as well. However, as expected, the utility of storage pushdown is lower in write-heavy workloads or when almost all requests are serviced from in-memory data structures. We will make BPF-oF and our key-value storage system integrations open source upon publication. Our major contributions are:

- BPF-oF, an efficient mechanism to safely enable storage pushdown for storage applications running on networked storage. BPF-oF is the first implementation of storage pushdown in a general networked-storage protocol, and its key ideas are an asynchronous file metadata syn-

chronization mechanism and a versioning scheme for inode mapping to protect concurrent access to files. BPF-oF greatly reduces the CPU and network overhead of NVMe-oF while requiring no custom hardware.

- Novel query splitting and cache sampling techniques, which let BPF-oF support local in-memory caching, addressing a significant challenge in storage pushdown. We integrate BPF-oF with modern storage systems, including the highly-optimized RocksDB.

## 2.2 Background and Motivation

This section surveys networked storage (§2.2.1) and NVMe-oF (§2.2.2). It then discusses NVMe-oF’s CPU overhead, providing motivation for remote storage pushdown (§2.2.3), and reviews eBPF as a framework for storage pushdown (§2.2.4).

### 2.2.1 Benefits of Networked Storage

Networked storage (Figure 2.1), where storage devices are managed separately from the application and accessed over the network, has been a popular architecture for datacenter and enterprise systems for many decades. For example, NetApp storage appliances and dedicated storage networks are used in many on-prem settings [60], while vendors like Lightbits Labs provide a cloud-oriented networked storage abstraction over NVMe-oF/TCP [62]. Similarly, hyperscalers like Meta [81, 73] and Alibaba [82] employ a networked storage architecture for their internal applications. Networked storage enables scaling and maintaining the storage tier independently of the application and workload. For example, consider deploying a key-value store such as RocksDB [25], which serves as the storage engine for a wide variety of applications in companies such as Meta [73, 80]. For some applications, which have relatively skewed access, it may make sense to provision a RocksDB instance with a relatively large number of cores, but a small amount of SSD capacity. In contrast, applications that do not experience skewed workloads perhaps require higher DRAM and SSD capacities. It is possible to provision the servers running RocksDB with local SSDs, but doing so efficiently requires anticipating the appropriate ratios of

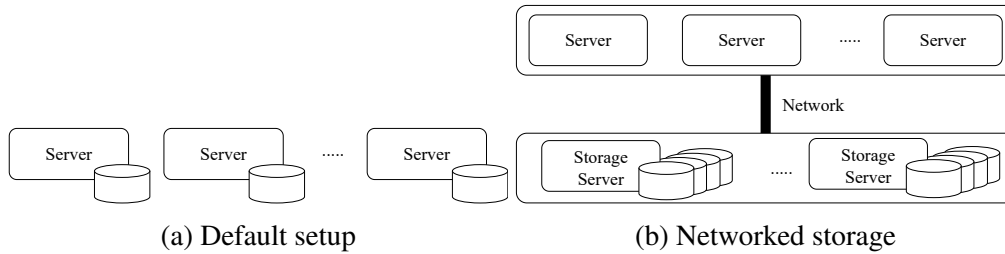


Figure 2.1: (a) Default: each server has its own storage. (b) Networked storage: storage is centralized in storage nodes and accessed over the network.

per-server CPU, DRAM and SSD in advance, which is difficult, especially at scale. On the other hand, in a networked-storage setup with a remote set of dedicated SSD appliances, RocksDB can easily access additional (or fewer) SSDs when extra capacity or I/O bandwidth is needed.

A key enabler of networked storage is the rise of relatively low-latency (single to double-digit  $\mu$ s) and high-bandwidth (100 Gbps and higher) networks. With typical access latencies of SSDs on the order of 100s of  $\mu$ s, and enterprise SSDs providing I/O bandwidths on the order of a million IOPS [83], the extra latency due to the network when issuing a remote I/O is relatively modest, and network bandwidth is able to saturate disk I/O [84, 63].

### 2.2.2 NVMe-oF Primer

NVMe-oF has emerged as the main networked storage protocol, replacing iSCSI due to its better performance [64, 85]. NVMe-oF allows an application to directly access a block storage device connected to a remote server using NVMe. Figure 2.2 shows the flow of an NVMe-oF read request. The *host* (i.e. the client, left) initiates the request, and the *target* (i.e. the server, right) contains the SSD and performs the actual disk I/O. To initiate an NVMe-oF request, an application at the host issues a storage system call, such as READ (step 1), which then traverses its local OS storage stack (steps 2-3). The request is treated as a regular NVMe request until it reaches the local NVMe driver. The *host* and *target* drivers maintain I/O queues for exchanging the NVMe-oF *capsule* (the unit of communication between the host and the target). The NVMe driver handles the request by constructing an NVMe-oF command within a capsule and submitting it to an NVMe I/O queue. The capsule is then forwarded to the relevant network stack (step 4) depending on the

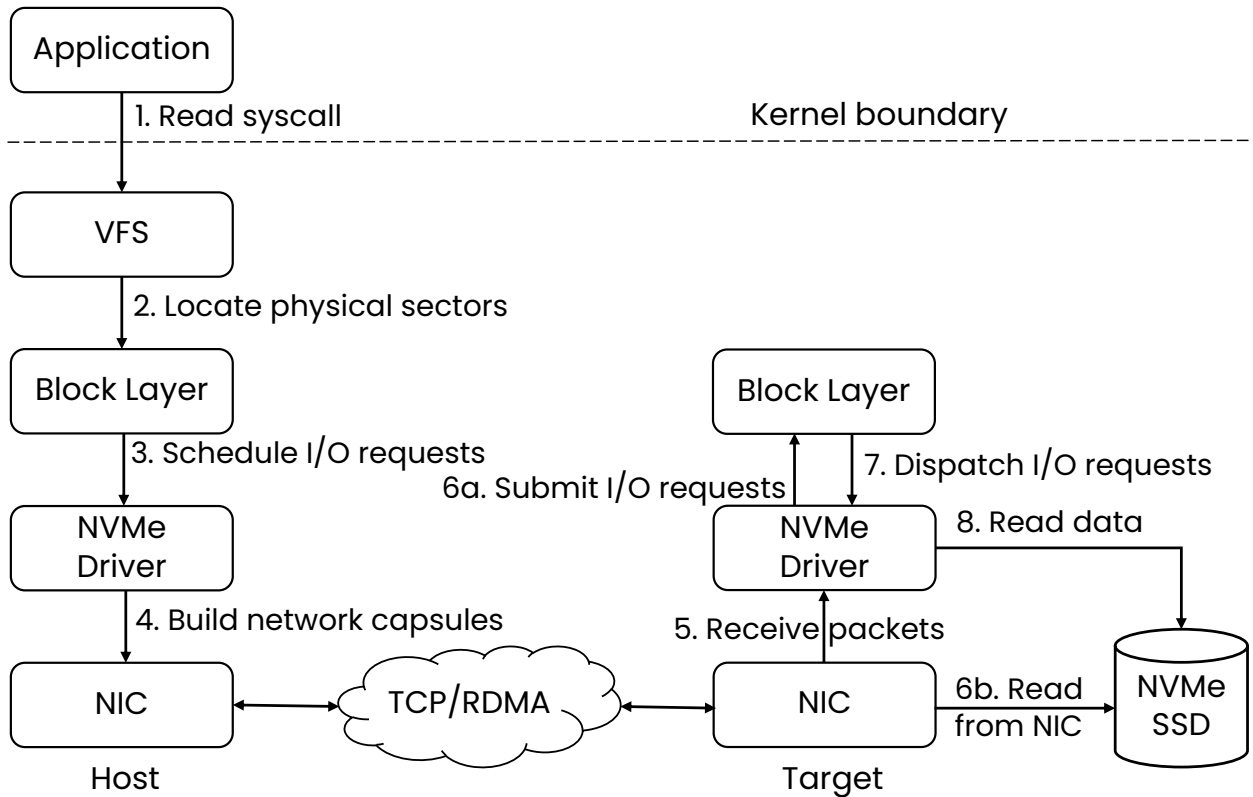
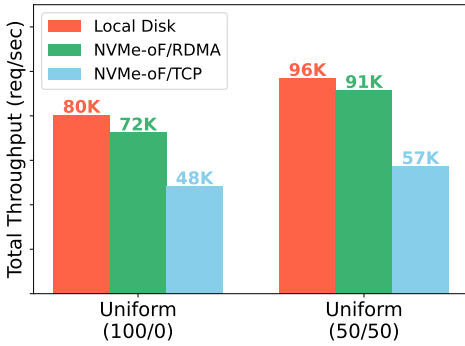
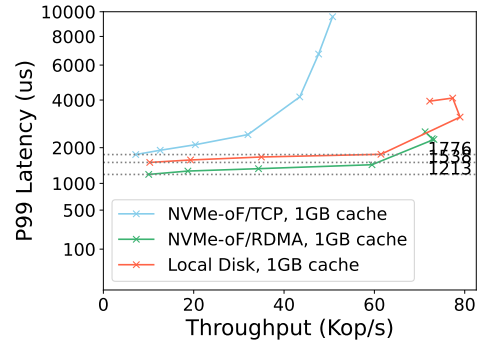


Figure 2.2: NVMe-oF overview.

fabric type (TCP, RDMA, Fiber Channel), and it is transmitted to the target. At the target (step 5), after the driver extracts the NVMe-oF command, it generates the block layer request and submits it to the block layer for I/O scheduling (step 6a). The target's NVMe driver, at last, receives the I/O request from the block layer (step 7) and reads the user's data from the local NVMe SSD (step 8), which is sent back to the host via the fabric-specific mechanism (not shown). Major NIC models (e.g., NVIDIA ConnectX, Broadcom Stingray, Intel IPU) support offloading the NVMe-oF target datapath to the NIC when using RDMA, and some NICs even support TCP offload [86]. This allows the NIC to directly read from or write to the NVMe device, bypassing the target's CPU (step 6b).



(a) Throughput.



(b) Throughput vs. latency.

Figure 2.3: RocksDB run locally compared to NVMe/TCP on a NAND SSD.

### 2.2.3 Improving NVMe-oF CPU Efficiency

A significant drawback of NVMe-oF is the CPU overhead of processing packets, both at the host and target [64]. This is especially acute with NVMe/TCP, due to the relatively high CPU cost of processing TCP packets [63, 84, 87]. High CPU consumption can adversely affect the performance of data-intensive applications, especially when they are CPU-bound.

To demonstrate NVMe-oF’s network processing cost, we run uniform and uniform read-write workloads on a 100 GB RocksDB instance with a local NAND SSD and NVMe/TCP (Figure 2.3). Each RocksDB instance uses 1 GB of cache and three cores locally at the host. The remote target is allocated three cores, which are only partially used by the NVMe/TCP setup. Our experimental setup is fully described in §2.6.1.

By running on a local NAND SSD, RocksDB achieves a 1.5–1.6× throughput improvement compared to running on the same NAND SSD accessed via NVMe/TCP. The primary cause of NVMe-oF’s slowdown is the CPU overhead of processing the packets at the host, as RocksDB is often bottlenecked by CPU. In addition, NVMe/TCP consumes additional CPU cycles at the target. Similarly, running RocksDB with NVMe/TCP incurs a significant latency overhead. While NVMe-oF provides significant operational advantages, it incurs a non-negligible performance overhead due to the extra network processing cost and communication time.

**Motivation for storage pushdown.** A promising way to reduce the resource consumption of NVMe-oF is to “push down” computation to the target storage server. This reduces potential roundtrip communication between the target and host. Storage function pushdown has been explored extensively in the database community as “predicate pushdown” [88, 89, 90], in the systems community as near-storage compute [5, 8, 10, 9, 13] and, more recently, as user-defined functions in the kernel [1, 2, 75, 4]. For more details, see §???. However, to the best of our knowledge, no prior work has designed storage function pushdown for a general-purpose networked storage protocol such as NVMe-oF.

Many storage applications, such as key-value stores, are amenable to such an approach for two reasons. First, a typical logical read operation is composed of a sequence of dependent I/O operations. For example, in the previous experiment, RocksDB issues on average 3.8 and 4 I/Os per read request, for the read-only and 50-50 read-write workload, respectively. Most read requests access multiple files, since files are spread across multiple levels of RocksDB’s LSM-tree data structure. Second, modern SSD-optimized storage systems typically write data to disk in large immutable files. This simplifies synchronization between potentially-conflicting simultaneous operations. We will discuss this point later in the paper. Since many SSD-optimized storage systems exhibit these properties [30, 91, 92, 26, 93, 72, 94, 95], pushing computation closer to the device can alleviate the overheads of accessing storage devices over the network for a wide variety of deployments.

#### 2.2.4 eBPF and XRP

Recently, eBPF has been used for storage pushdown within one server [1, 4, 5]. BPF-oF uses eBPF [96] to run custom storage pushdown functions on a remote server. The primary advantages of eBPF for our setting are: (a) the eBPF verifier and sandbox ensure functions are isolated from one another and from sensitive kernel state, and (b) eBPF can be run on a variety of operating systems and architectures (CPUs, FPGAs, smartNICs, smartSSDs), thereby supporting a wide range of networked-storage appliances and configurations.

## 2.3 Challenges

In this section, we describe the challenges in triggering storage functions over NVMe-oF on a modern storage system.

**Challenge 1: Safely access files in the target.** As described in §2.2.2, in NVMe-oF, once a request reaches the target, it only knows how to access blocks, and lacks any file system information, which resides at the host. Therefore, an eBPF function that submits storage I/O for a file on the target must determine the next block’s location without access to the file system file-to-block mapping. In addition, our system must enable resubmission across different file descriptors to support modern storage applications (e.g., traversing an LSM-tree). Thus, the target must be able to translate a file descriptor to its corresponding inode and find the block offset corresponding to the requested file offset. It must do so in a safe manner, without the risk of returning invalid data.

**Challenge 2: Integrate in-memory data structures.** Integrating a production storage system with BPF-oF presents a number of interesting challenges, primarily owing to the interactions between the system’s in-memory data structures, which sit locally at the host, with the remote function execution, which now occurs remotely at the target. First, when executing a storage operation, storage systems typically do not produce a “pure” chain of I/O accesses. Instead, some of the intermediate data structures in the operation path may be cached in memory. For example, in LSM-tree storage systems [91, 25, 97, 27, 92] each file has its own index, and these indices are often cached in memory, while the actual data mostly resides on disk. Therefore, a naive pushdown operation that traverses multiple files will either ignore these cached indices, issuing many more storage I/Os than necessary, or return back to the host for each file, negating the benefit of storage pushdown. In order to achieve optimal speedup with BPF-oF, we need to ideally chain the I/O storage accesses together in one network traversal.

Second, assuming we offload such a chain of I/O accesses, the host application’s in-memory cache may not be up-to-date. While the final result of the query will always be returned, the

“intermediate” I/O accesses (e.g., accessing a file in the middle of the LSM-tree lookup) would not. A strawman approach would be to return the results of *all* the I/O accesses back to the host and then appropriately update the application’s in-memory data structures. However, this would negate much of the benefit of storage pushdown, as these intermediate results are not actually needed by the application (only the final query result is needed), and would lead to approximately the same network bandwidth consumption as running the requests with no storage pushdown.

## 2.4 BPF-oF Design

We present BPF-oF’ design principles (§2.4.1), its architecture (§2.4.2), and how it addresses the first challenge from §3.3.

### 2.4.1 Principles

We design BPF-oF with the following principles in mind.

1. **Wide applicability.** Minimal changes to NVMe-oF and Linux. The system should be immediately deployable on existing infrastructure, without requiring specialized hardware support. BPF-oF’ API should be backwards-compatible with NVMe-oF.
2. **CPU efficiency.** High throughput per core.
3. **Concurrent access.** Must not return invalid data due to concurrent requests that modify file system metadata.
4. **Flexible resubmissions.** Support variable-sized resubmissions across different files, to enable data structure traversal in modern storage systems.
5. **Target applications.** SSD-optimized storage applications [30, 91, 92, 26, 93, 72, 94], wherein incoming updates are buffered in memory and periodically flushed to disk as immutable files. BPF-oF focuses on read operations, since writes are handled in the background.

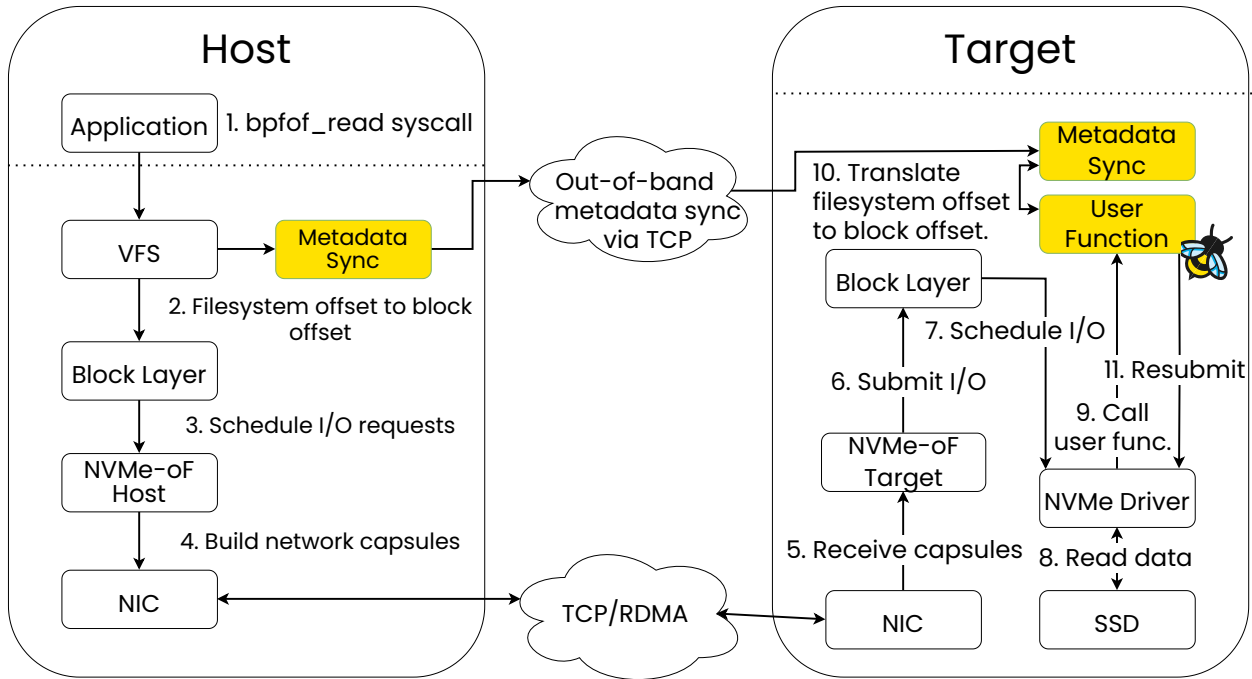


Figure 2.4: BPF-oF architecture.

## 2.4.2 Architecture

BPF-oF supports storage pushdown with eBPF (Figure 2.4) over NVMe-oF by adding three mechanisms. First, it adds a mechanism to asynchronously update the target’s file-to-block mapping. Second, it adds a versioning scheme that allows eBPF functions to safely access storage blocks at the target concurrently with file metadata updates. These two mechanisms jointly enable the third mechanism: safely running user-defined functions at the target server’s NVMe driver. Applications interact with BPF-oF using two interfaces: a new NVMe command that sends a “push-down” NVMe-oF request and a userspace API that allows applications to install remote eBPF functions at the target. Before diving into these mechanisms and interfaces, we walk through an example of a BPF-oF request that traverses a B-Tree index by pushing down the traversal to the target. We match the steps of the example to the steps shown in Figure 2.4.

**End-to-end example of a BPF-oF request.** At initialization, the application installs the remote eBPF index traversal function using BPF-oF’ API. For each query, the application first stores the

query key in a buffer, which is called the *scratch buffer* [1]. The scratch buffer is allocated by the application and stores the query key and the result from the remote eBPF function. The application also specifies a file descriptor (or list of file descriptors) to read from, the number of bytes and offset to read from this file descriptor, and the eBPF function ID to call on the target.

Then, the application calls our `read_bpfof` system call (step 1). The system call obtains a reference for each relevant file descriptor, ensuring that a file's data blocks are not overwritten if a file is deleted while there are outstanding requests. In addition, it creates a file-descriptor-to-inode mapping, which lets us use inode versions for synchronization. As described in §2.4.2, an inode's version changes whenever the file-to-block mapping (e.g., extent tree) for that inode is updated. Then, the application's file system request is translated to a block request (step 2) and reaches the NVMe-oF host driver (steps 3, 4). There, the host constructs a new `read_bpfof` NVMe command. The command transfers the scratch buffer and its size to the target, along with the file-descriptor-to-inode mapping, the offset to read from, the number of bytes to read, and the eBPF function ID to call. Before submitting the request, the host driver checks that the inode versions have been synchronized, as described in Algorithm 1.

Next, the NVMe-oF target driver receives the request (step 5), and prepares a block request and submits it (step 6). The block request is translated to an NVMe read request and submitted to the target's NVMe device (steps 7, 8). The target's NVMe driver receives the request result and calls the user's eBPF function (step 9), passing in the result and the scratch buffer. The eBPF function processes the request result and decides what happens next. In this example, the returned data represents an internal node of the B-tree; hence, the eBPF function parses the node to find the file offset of the child node that it should read to continue the point query. The eBPF function indicates to BPF-oF' *resubmission mechanism* that it should read this additional block, so BPF-oF resubmits the read request to the target's NVMe driver with the new offset (steps 10, 11). The resubmitted read's file descriptor is translated with the previously sent file-descriptor-to-inode mapping, and the offset is translated using the inode's file-to-block mapping, which have been synchronized.

When the eBPF function finds the correct key, it writes the value to the scratch buffer and

indicates that the result should be sent to the host. On the host, BPF-oF will again verify that the inode versions remain in sync, as shown in Algorithm 2, and return the result to the application.

In this way, BPF-oF implements index traversal at the target without repeated communication with the host. This workflow contains a versioning scheme, which allows BPF-oF to work correctly with concurrent operations, which can lead to stale remote metadata. We now provide more details on these mechanisms. For simplicity, in the description below, we assume an integration with the ext4 file system.

## **Versioning**

BPF-oF guarantees safety by versioning file system metadata. To do so, we add a version to each file's extent tree and increment it when the tree changes.

The versioning algorithm runs on the host, before and after a request (Algorithms 1 and 2). Before submission, the host gets the latest inode ID and version for the request's file descriptors. For all of them, it checks whether that version of the inode's extent tree has been synchronized. If not, it aborts the request. This ensures the target always has the latest version of the file system metadata. This check is sufficient to guarantee that the file-to-block mapping is correct on the target, except for one case: when the extent tree is remapped, i.e. an existing file offset points to a new disk offset. This can happen during truncation and defragmentation. We handle this case by adding a final version check when the response is received from the target: if the version has changed, the host aborts the request. It also wipes the scratch buffer to ensure that the target will not return stale data, even if the extent tree is remapped during the request.

## **Metadata synchronizer**

The next challenge is synchronizing the file system metadata (i.e. file-to-block mappings) from the host to the target. The metadata synchronizer consists of two kernel modules running on the host and the target. The host module watches for file system changes by installing *fsnotify* kernel hooks. For each change, it schedules a metadata synchronization for the specified inode.

---

**Algorithm 1: Host - Check Before Submission**

---

```
1: for  $idx = 0$  to  $num\_fds - 1$  do
2:    $fd \leftarrow bpf\_request.fds[idx]$ 
3:    $curr\_version \leftarrow get\_fd\_version(fd)$ 
4:    $sent\_version \leftarrow$  last version sent to target
5:   if  $curr\_version \neq sent\_version$  then
6:     abort request
7:   end if
8:    $request.fd\_versions[idx] \leftarrow curr\_version$ 
9: end for
```

---

---

**Algorithm 2: Host - Check After Completion**

---

```
1: for  $idx = 0$  to  $num\_fds - 1$  do
2:    $fd \leftarrow response.fds[idx]$ 
3:    $curr\_version \leftarrow get\_fd\_version(fd)$ 
4:    $used\_version \leftarrow get\_request\_version(response)$ 
5:   if  $curr\_version \neq used\_version$  then
6:     abort request
7:   end if
8: end for
```

---

A kernel workqueue processes these requests and checks if the file-to-block mapping has changed from the last version it synchronized, using the versioning scheme described above. Then, the synchronizer copies the inode’s extent tree and transfers it to the target module, where it is used for resubmissions. Metadata synchronization is done asynchronously over TCP and does not block the data path. For simplicity, we implemented the synchronizer as an out-of-band TCP client and server, instead of creating a new NVMe command.

**eBPF resubmission.** The eBPF execution and resubmission mechanism at the target reuses the XRP [1] API. In addition, we add support for (a) resubmitting eBPF functions across multiple files instead of a single file and (b) specifying a different request size for each resubmission. In short, the application developer can write an eBPF function that can issue I/O resubmissions with different sizes and across different files. The BPF-oF API for these eBPF functions is shown in Figure 2.5. We provide more details on these changes in §2.5.

```

struct bpf_of_ebpf_context {
    // Set by kernel.
    // Current address, file descriptor,
    // and corresponding disk data.
    __u64 cur_addr;
    __s32 cur_fd;
    char *data;

    // Set by resubmission program.
    // Scratch buffer initially contains
    // query key and in the end, the result.
    char *scratch;
    // Next resubmission address, size, fd.
    uint64_t next_addr;
    uint64_t next_size;
    __s32 next_fd;
    // When set, the scratch buffer is returned.
    int32_t done;
};

```

Figure 2.5: BPF-oF eBPF functions control resubmission via this struct.

## Interfaces

To integrate with NVMe-oF, BPF-oF implements a new NVMe command following the specifications for NVMe and NVMe-oF [98, 99]. The command transfers the scratch buffer, which contains the query key and result (on completion), from the host to the target and back. The command’s arguments are the same as a read request. We transfer the additional arguments needed by BPF-oF, as described above, with NVMe-oF’s data transfer mechanism (scatter-gather lists). BPF-oF also implements a new `read_bpf_of` system call (Figure 2.6) with the following arguments: a list of file descriptors it might read from, the number of bytes and offset to read from the first file descriptor, the scratch buffer and its size, and the eBPF function ID to call on the target.

## 2.5 Supporting In-Memory Data Structures

A significant challenge in implementing remote-storage pushdown is utilizing a storage system’s in-memory data structures (stored at the client) with the on-disk lookup (occurs remotely).

```

int bpf_of_read(int fd,
               size_t count,
               off_t offset,
               void *scratch_buf,
               size_t scratch_buf_count,
               unsigned int bpf_id)

```

Figure 2.6: BPF-oF read system call.

Our solutions to this challenge apply to any key-value storage system that needs to use storage pushdown, but in this section, we focus on how we tackled this problem in the RocksDB integration, as it utilizes a wide variety of in-memory caches (e.g., index and data block caches, Bloom filters) common to many systems [27, 92, 91, 93].

### 2.5.1 RocksDB Architecture

**LSM-tree.** RocksDB is an LSM-tree based storage engine [97]. To avoid SSD write amplification and wear out, incoming writes to RocksDB are buffered in memory, in a data structure called the MemTable. After it fills up, the MemTable is asynchronously flushed to disk as a sorted-string table (SST) file, which is composed of sorted key-value pairs. SST files are divided into data and metadata blocks. Metadata blocks include index blocks, which map keys to data blocks, and filter blocks, which contain Bloom filters (these are disabled by default in RocksDB, but are supported by BPF-oF).

These SST files are organized into hierarchical levels ( $L_0, L_1, \dots, L_N$ ), where the “upper” levels store the latest versions of each key-value pair (e.g.,  $L_0$  is “higher” than  $L_1$ ). The immutable MemTables are flushed into  $L_0$ , which stores files with overlapping key ranges, whereas files in the lower levels ( $L_1, \dots, L_N$ ) hold non-overlapping ranges. RocksDB uses a background thread that periodically scans SST files from adjacent levels (e.g.,  $L_2$  and  $L_3$ ), and combines them into a single file, which is written to the lower level (e.g.,  $L_3$ ). In the process, RocksDB removes deleted and stale versions of keys, freeing up space. It has been shown that to obtain good performance in LSM-trees [97], each level contains a multiple of (e.g., 10×) the capacity of the previous level, whereby the lowest level ( $L_N$ ) makes up most of the LSM-tree’s capacity.

**Workflow of a read.** Since more updated versions of key-value pairs sit at the “upper” levels of the tree, when a read request arrives, RocksDB searches for the object in the levels of the tree from top to bottom. It first checks if the key-value pair is in RocksDB’s MemTable. If not, it checks in  $L_0$ ’s files, followed by  $L_1$ , etc. For each file, RocksDB first reads the filter blocks to check if the key might be in that file (with a high probability). If so, it checks the index block to find the relevant data block and searches for the key in the data block. These blocks may or may not be cached in memory by RocksDB: typically many (or all) of the index and filter blocks are cached. Therefore, even when searching within a single file, BPF-oF may consult an in-memory data structure (e.g., a cached index block), followed by disk I/O (e.g., a data block). This can create an interleaved pattern of memory and disk lookups, which is challenging for BPF-oF, since it only provides a significant speedup for a *chain* of storage I/Os.

To facilitate lookups, LSM-trees cache the key ranges each file contains. We make the following observation, which we take advantage of in our design as a performance optimization: this cache can be used to determine all files that may contain a particular key, and therefore *might* be accessed in order to satisfy a read operation. BPF-oF is able to support systems which do not exhibit this property by maintaining a mapping between the application’s on-disk file “pointer” and the corresponding inode. However, we have yet to observe a modern storage system where this would be required.

### 2.5.2 Integration with BPF-oF

We focus on accelerating RocksDB’s read path, since writes are buffered in memory and written sequentially to disk. Initially, our prototype only accelerates point queries. We highlight the following two key contributions: (a) a new technique, *query splitting*, that splits in-memory accesses from storage accesses to maximize the effectiveness of storage pushdown, and (b) periodic cache sampling to keep the storage engine’s cache fresh. Overall, integrating RocksDB with BPF-oF required  $\sim 700$  lines of changes to RocksDB and  $\sim 1,200$  LoC to reimplement RocksDB’s read logic in eBPF.

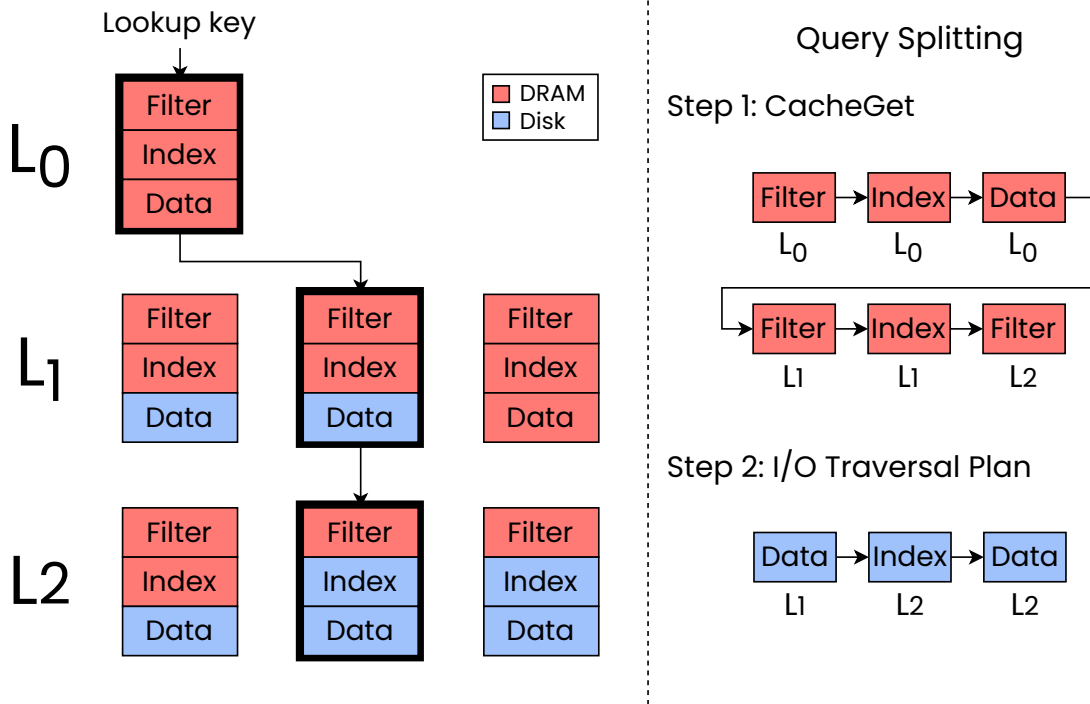


Figure 2.7: RocksDB LSM-tree traversal and query splitting.

## Query Splitting

In order to maximize the efficiency of storage lookups, the remotely-executed function should involve a chain of disk I/O requests, with no access to in-memory (userspace) data structures. To this end, we split the query into two phases: first, the query accesses all in-memory data structures that *may* be involved in a key lookup. Then, it triggers all the required I/O as a series of dependent I/O functions at the remote node. Recall that due to the design of LSM-trees, the host can determine all files that may be needed to complete the operation. We are guided by the observation that accessing RocksDB’s in-memory data structures is orders of magnitude cheaper than storage I/O (especially when that I/O is over the network). Therefore, there is little cost in potentially accessing more in-memory data structures than would have been necessary otherwise.

To this end, we implement a new function on top of RocksDB, `CACHEGET`. `CACHEGET` is based on RocksDB’s existing per-SST file `GET` function, but it only reads data from a file’s cached blocks without triggering any storage I/O. Our prototype runs `CACHEGET` on all files that may

contain the key, across all levels. Based on the results of this function (e.g., whether a key is found in a particular cached data block), our prototype creates an *I/O traversal plan*, which only fetches potentially required *uncached* blocks. The traversal plan skips files that have both index and data blocks cached, those that are filtered by a per-file Bloom filter, and it starts I/O at the data block if the index block is cached.

We illustrate this process in Figure 2.7. In this example, three files across  $L_0$ ,  $L_1$ , and  $L_2$  have key ranges that contain the requested key. All of these files may be required to find the key. As the figure shows, these files' blocks are partially or fully cached. CACHEGET retrieves the relevant cached blocks. If the key is not found in the cached blocks, our RocksDB prototype issues a `read_bpfof` system call. The remaining uncached blocks form the on-disk I/O traversal plan. In the case that a higher level (e.g.,  $L_2$ ) block is cached, while a lower level (e.g.,  $L_1$ ) block is uncached, BPF-oF will still issue a `read_bpfof` call to ensure that the correct value of the key is found. Note that `read_bpfof` does not blindly follow the I/O traversal plan to completion, and it terminates once the key is found. For example, if a key exists in  $L_1$ , the parser will not read data from  $L_2$ , since the value in  $L_1$  will be more up-to-date than other versions lower in the tree. As such, CACHEGET will often be called on more files than necessary, since RocksDB does not know the *minimal* set of files that will be involved in the traversal in advance.

## Cache Sampling

In RocksDB, all data blocks that were accessed as part of a query are cached by default. However, BPF-oF returns only the result (i.e. the final data block accessed), not the intermediate blocks. With BPF-oF, RocksDB's cache wouldn't contain these intermediate blocks, despite them having been accessed. This problem would exist in any system with storage pushdown that caches intermediate results. A naive solution would be to return all of the data blocks accessed in the query, but this would negate much of the benefit of storage pushdown, by increasing network bandwidth (and CPU costs) significantly. As such, we seek another way to keep the cache fresh. We are inspired by prior work on cache sampling [100], where a low percentage of the randomly-sampled

requests update the cache with recently-accessed intermediate data blocks (and the final block). We implement this idea by sending a small percentage of requests through the slower, non-offloaded “normal” read path, which caches all the intermediate results. Empirically, we find this technique is quite effective with sampling rates as low as 0.1-1% (see §2.6.4).

## Converting File Parsers to eBPF

We converted RocksDB’s SST file parsers, which parse SST index and data blocks, to eBPF. Our eBPF parser first uses the generated I/O traversal plan, which is stored as an array in the scratch buffer, to parse each file at the correct file offset and parsing stage (i.e. index or data block). If the requested key is not found, the eBPF parser will resubmit a new I/O request for the next file in the plan. This process continues until either the key is found or all relevant files have been traversed. If the eBPF parser fails, we fall back to the regular RocksDB read path.

## 2.6 Evaluation

This section answers the following questions. Q1-Q4 focus on RocksDB, and Q5 focuses on WiredTiger and BPF-KV:

- Q1:** How does BPF-oF perform under different network protocols, storage devices, and cache sizes compared to the baseline NVMe-oF environments?
- Q2:** How does BPF-oF affect CPU and network utilization?
- Q3:** How do different sampling rates affect performance?
- Q4:** What is the performance impact of version mismatches?
- Q5:** How does BPF-oF generalize to other storage systems?

### 2.6.1 Experimental Setup

We conduct our experiments in three different CloudLab storage configurations [101]. We measure an average roundtrip latency of 30 $\mu$ s and 18 $\mu$ s for TCP and RDMA, respectively.

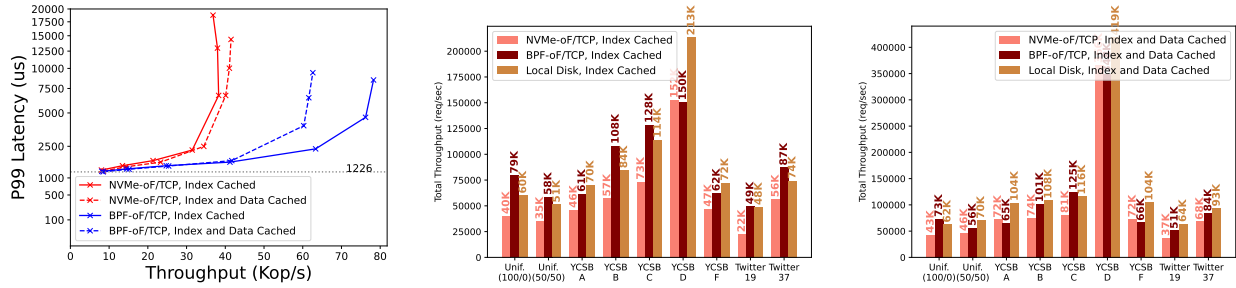
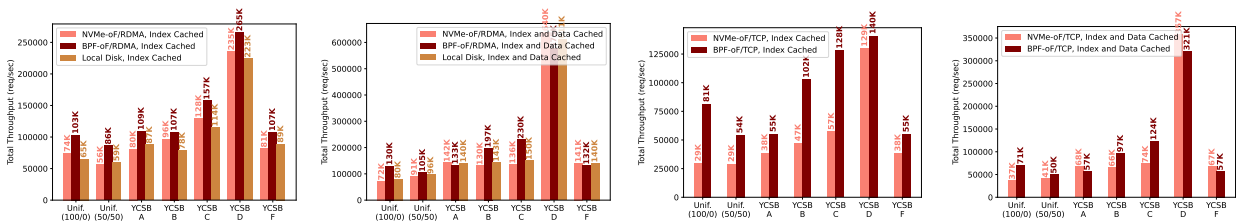


Figure 2.8: RocksDB (a) uniform read throughput-latency and throughput (b) without and (c) with data blocks cached. BPF-oF vs. NVMe/TCP on NAND SSD.



(a) NVMe/RDMA without data blocks cached. (b) NVMe/RDMA with data blocks cached. (c) NVMe/TCP without data blocks cached. (d) NVMe/TCP with data blocks cached.

Figure 2.9: RocksDB throughput under different configurations.

**NAND configuration.** Both host and target are c625-100g machines in CloudLab. Each machine has a 24-core AMD EPYC 7402P CPU, 128 GB of RAM, two 1.6 TB NAND SSDs (Dell Ent NVMe AGN MU U.2), and a Dual-port Mellanox ConnectX-5 NIC with a 100 Gbps network. The NAND SSD’s read latency is 90µs and can sustain 700K read IOPS.

**Optane configuration.** This setup uses a low-latency storage device. The host is a c625-100g machine, while the target is d750 with Intel Optane SSD. The target has two 16-core Intel Xeon Gold 6326 CPUs, 128 GB of RAM, a 400 GB Optane 5800X SSD, and a Quad-port BCM57504 NetXtreme-E on a 25 Gbps network. The Optane SSD has a 3µs 512B read latency and can sustain 5M read IOPS.

**RDMA offloading configuration.** This setup uses RDMA NIC offloading. Both host and target are r6525 machines, with 32-core AMD EPYC 7543 CPUs, 256 GB of RAM, one 1.6 TB NAND

SSD (Dell Ent NVMe AGN MU U.2), and a Dual-port Mellanox ConnectX-6 100 GB NIC on a 100 Gbps link.

**Software configuration.** We increase the threads per core up until saturation, and use CPU pinning to bind each thread to a specific core, and flow-steering to associate each flow with a specific host and target core. To make our results reproducible, we disable hyperthreading and address space randomization. We run Ubuntu 20.04 and Linux 5.12.0.

**Evaluated systems.** We integrate BPF-oF with three systems: RocksDB, WiredTiger and BPF-KV. RocksDB is a complex and popular key-value store, and we test it with BPF-oF to show whether a pushdown approach would benefit state-of-the-art storage applications. WiredTiger is a simple production-grade key-value store used primarily by MongoDB. BPF-KV is a bare-bones academic key-value store custom-designed for running eBPF functions [1]. All systems bypass the OS page cache with direct I/O.

**RocksDB setup.** We build upon RocksDB 7.7.3 with a 100 GB database in 5 layers ( $L_0 - L_4$ ) and 3 cores (which saturate our enterprise SSD); we use 24 and 20 threads-per-core for NAND and Optane SSD, respectively. Unless noted, all index blocks are cached and Bloom filters are disabled (RocksDB’s default). We run RocksDB with both TCP and RDMA; our results focus on TCP, as it is more prevalent. By default, we use a cache sampling rate of 1% (i.e. a random 1% of reads go through the regular non-BPF-oF read path).

**Workloads.** To test BPF-oF, we run the YCSB [102] benchmark suite under Zipfian (0.99) key distributions, along with uniform read and read-write workloads. We do not run YCSB E, because our RocksDB integration does not yet accelerate scans. We also test BPF-oF with a production trace taken from clusters 19 and 37 of the Twitter cache workloads [103]. These traces are read-dominant (75% and 63%, respectively) and have low skew ( $\alpha = 0.7$  and 0.4, respectively), ensuring that their working set does not fit entirely in memory.

## 2.6.2 BPF-oF With Different Setups (Q1)

**TCP with NAND SSD.** Figure 2.8 shows RocksDB’s throughput with BPF-oF on NAND and TCP, compared to NVMe/TCP.<sup>1</sup> Without a data cache, BPF-oF can achieve 1.4-2.2× higher throughput than NVMe/TCP, including on the Twitter cache traces, which are real-world workloads. This trend continues when adding a data cache, where BPF-oF provides 1.2-1.7× higher throughput than NVMe/TCP. In addition, we see a surprising result: BPF-oF without a cache yields higher throughput than BPF-oF with a cache. We discuss this behavior below.

The relative speedup of BPF-oF with a data cache is generally lower, because some lookups may be cached, reducing the average number of roundtrips saved by BPF-oF. The only cases where BPF-oF falls slightly short are YCSB A, D, and F, with data blocks cached. Since these workloads are highly-cacheable, they issue very few I/Os per request, and BPF-oF’s slightly higher overhead for in-memory requests (e.g., its preemptive access of all possible in-memory caches in each query) leads to a slight slowdown. However, with just index blocks cached, BPF-oF provides a speedup in all workloads except YCSB D, and it even beats the baseline with cache for a majority of the workloads. Figure 2.8 also shows that BPF-oF reduces tail latency by up to 2×.

**RDMA with NAND SSD.** Figures 2.9a and 2.9b show the throughput of RocksDB with BPF-oF on NAND and RDMA, compared to *NIC-offloaded* NVMe/RDMA. Even though BPF-oF does not use offloaded RDMA, BPF-oF is up to 1.8× faster than offloaded NVMe/RDMA. In our experiments, there was no noticeable difference in the baseline’s throughput between offloaded and non-offloaded RDMA, as while offloading RDMA to the NIC saves CPU at the target, the target’s CPU is usually not a bottleneck with RocksDB.

**TCP with Optane SSD.** We also test BPF-oF with Optane, providing a glimpse of BPF-oF’s benefits as SSDs become faster. BPF-oF provides even greater speedups with Optane (Figures 2.9c and 2.9d), providing 2-2.8× higher throughput. This is because Optane can support a high I/O

---

<sup>1</sup>The TCP results in Figure 2.3 were obtained on the offloading machine (to compare with RDMA), yielding different results than in Figures 2.8.

bandwidth, while in the NAND SSD experiments I/O bandwidth may become a bottleneck. BPF-oF also provides a consistent improvement (up to 2.6×) in tail latency in all workloads.

**Caching and Storage Pushdown.** Caching has an interesting and unexpected interaction with BPF-oF. In certain cases, running RocksDB with a cache can actually slow down the system, as seen in Figures 2.8 and 2.9d. In the baseline NVME-oF case, RocksDB transfers all the intermediate blocks over the network until it finds the final result. Thus, maintaining a cache for these blocks incurs little additional cost. However, when using storage pushdown with BPF-oF, RocksDB only transfers the final result over the network, and we employ cache sampling to keep the cache fresh with intermediate results. In some workloads, the cost of keeping the cache fresh outweighs its benefits. This balance depends on the workload type, the cache miss rate, and the I/O cost.

**Non-Default Configurations.** By default, we evaluated RocksDB with indices pinned and without Bloom filters, as these are the default options in RocksDB. For completeness, we also tested (1) with Bloom filters enabled and (2) with indices not pinned. The results were similar to our default configuration: BPF-oF yields significant speedups, with up to 2× improvement with Bloom filters and 1.8× improvement with unpinned indices.

### 2.6.3 CPU and Network Savings (Q2)

BPF-oF provides significant CPU and network savings, which are directly correlated with performance increases. To measure these savings, we run a benchmark while monitoring the CPU time spent on the host *and* target, and network bytes exchanged. We present these results (normalized per-request) for uniform reads with a 1 GB data cache, in Table 2.1. For TCP, BPF-oF uses 37% fewer CPU cycles (measured in mCPUs, i.e. 1/1000th of a CPU) and generates 23% less network traffic, yielding a 70% throughput improvement. For offloaded RDMA, BPF-oF uses 29% and 19% less CPU and network, respectively, yielding a 65% throughput increase.

Setup	Throughput (req/s)	Host (mCPU / req)	Target (mCPU / req)	Total (mCPU / req)	Network (kB / req)
NVMe-oF/TCP	48279	0.07	0.03	0.10	12.07
BPF-oF/TCP	69935	0.04	0.02	0.06	9.27
NVMe-oF/RDMA (offload)	77553	0.04	0.00	0.04	12.3
BPF-oF/RDMA	120991	0.02	0.01	0.03	10.02

Table 2.1: Resource consumption with uniform read and 1 GB data cache.

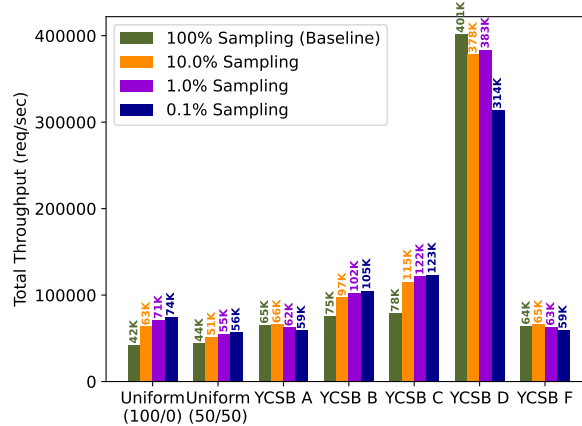


Figure 2.10: RocksDB throughput with different sampling rates with cache under BPF-oF using TCP and NAND SSD.

## 2.6.4 Sampling Rate (Q3)

As discussed in §2.5.2, our RocksDB integration sends a small portion of reads through the normal read path in order to keep the cache warm. We evaluate the impact of this sampling rate on RocksDB. Figure 2.10 shows that throughput peaks at a sample rate between 0.1% and 10%, except for YCSB D, which is highly cacheable and thus benefits from a higher sampling rate. Conversely, we see a slight decrease in YCSB A and F at higher sampling rates. These workloads are write-heavy and highly skewed, which means that as the popular keys are written repeatedly, they reappear in the higher levels of the LSM tree and shorten the resubmission chains, leading to a smaller improvement. Based on these experiments, we run BPF-oF with a 1% sampling rate.

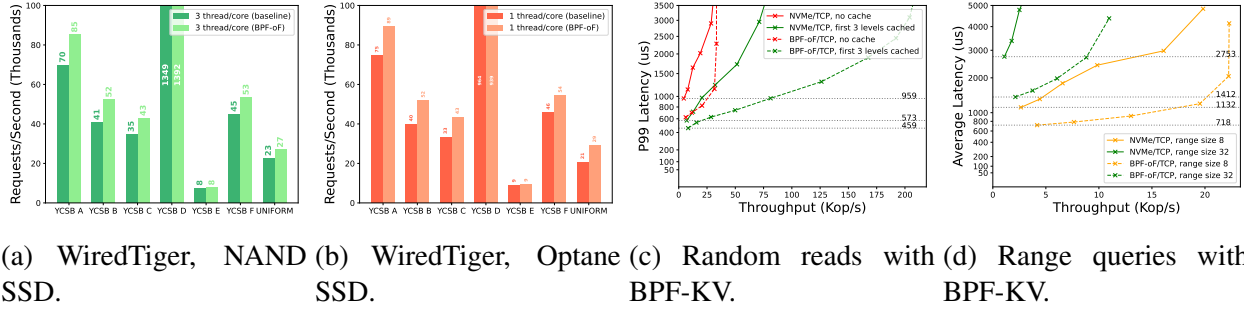


Figure 2.11: WiredTiger and BPF-KV evaluations with TCP.

### 2.6.5 Version Mismatches (Q4)

As discussed in §2.4.2, BPF-oF versions file metadata in order to guarantee safe concurrent access to files for offloaded applications. A version mismatch occurs when the synchronized metadata in the target becomes stale. This leads to a failed request, which is retried through the normal read path. Thus, version mismatches are very costly: they approximately decrease throughput by one request and more than double the request’s latency. Fortunately, they are very rare. We observed only 20 mismatches per benchmark, out of millions of requests. Only YCSB D had a slightly higher number of mismatches, due to its write-heaviness, but they occurred in only 0.03% of requests. Thus, the performance impact of version mismatches is negligible in our RocksDB integration.

### 2.6.6 WiredTiger (Q5)

We run WiredTiger in its LSM-tree version, with data split into levels and each level stored in a single file. Each file uses a B-tree index with the data stored as leaf nodes and a 512B B-tree page size. The database contains a billion items with a 16B object size. We test WiredTiger with BPF-oF using the YCSB benchmark on four cores. Figure 2.11a shows that for read-write workloads such as YCSB A, BPF-oF has 20% higher throughput. Figure 2.11b shows that for read-heavy workloads such as YCSB B, YCSB C, and uniform, the improvement reaches 30% when using an Optane SSD. The performance gain of WiredTiger is more modest than that of RocksDB (and BPF-KV), since WiredTiger issues far fewer I/Os, so speeding up I/O yields diminishing returns.

### 2.6.7 BPF-KV (Q5)

BPF-KV is a high-performance prototype key-value store optimized for eBPF functions [1]. It stores data in a  $B^+$ -tree, caches the top-most levels of the tree, and uses fixed-sized 512B key-value pairs. The tree's leaves contain a pointer to the values, which are stored in an unsorted log. We use a 67 GB database file with 6 index levels and 1 log level.

**Point queries.** We run a BPF-KV microbenchmark with 8B keys and 64B values stored on the target. Figure 2.11c compares running BPF-oF vs. NVMe-oF on NAND SSDs on three cores. Beyond three cores BPF-KV bottlenecks on the NAND disk's I/O bandwidth. BPF-oF consistently provides  $2.6\times$  higher throughput than NVMe/TCP when the upper levels of BPF-KV are cached. When no cache is used, the access pattern limits the SSD to 120K IOPS (30K req/s), because the upper level nodes become read hot-spots in the NVMe queues. This speedup is due to BPF-oF saving CPU cycles in processing TCP packets, which are instead used to handle additional I/O requests.

The results also show that BPF-oF provides a more modest 20-25% improvement in unloaded 99<sup>th</sup>-percentile latency. The tail latency improvement is more modest than the throughput improvement because NAND SSD latency (90 $\mu$ s) is significantly higher than TCP latency for our CloudLab nodes (30 $\mu$ s). Therefore, since BPF-oF reduces the network round-trips, but does not reduce the number of storage I/O requests, the unloaded latency improvement is less significant. Still, as is evident from the graphs, BPF-oF keeps the tail latency under control as the load gets higher, in contrast to the NVMe-oF baseline.

Figure 2.12a shows results with RDMA and NAND. As expected, BPF-oF can accelerate NVMe-oF with RDMA, although the throughput gains are generally less significant than with TCP, since RDMA's CPU processing cost is less expensive than TCP's. The throughput peaks around 220K because we saturate the SSD's bandwidth.

Figures 2.12b and 2.12c similarly compare BPF-oF to NVMe-oF using an Optane SSD. For TCP on Optane, BPF-oF provides more than  $8\times$  higher throughput for BPF-KV without a cache. In

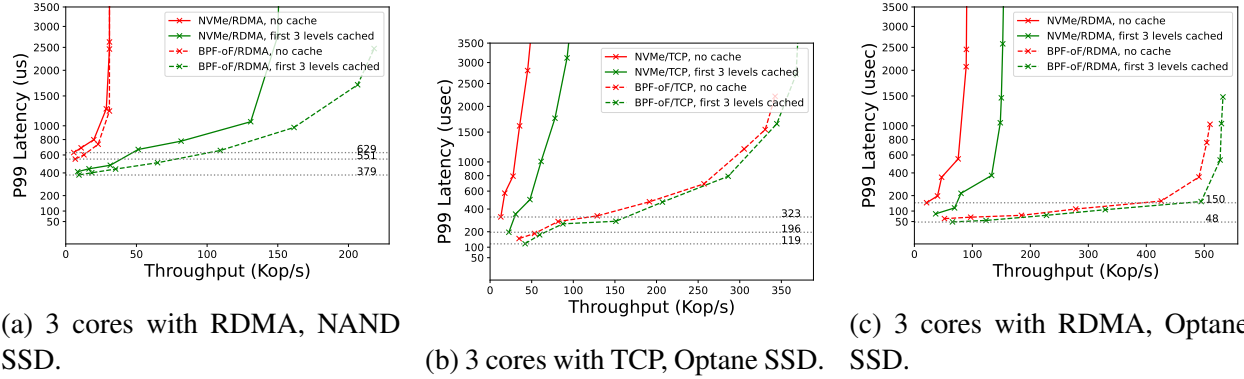


Figure 2.12: Tail latency and throughput of BPF-oF vs. NVMe-oF on BPF-KV with random 512B reads.

this case, the throughput improvement of BPF-oF is even more significant with TCP (consistently higher than 5 $\times$ ) since the CPU overhead of TCP is even more dominant than with NAND due to Optane’s much higher IOPS (5M reads per second). Similarly, the tail latency improvement is also more significant, and is reduced by up to 3.1 $\times$ . Similar to our experiments with RocksDB, in the case of BPF-oF with Optane SSD, there is almost no difference between running BPF-KV with a cache and without a cache for a uniform workload. The reason for this is that since Optane SSD is so fast (device latency of 3 $\mu$ s), the cached version of BPF-KV reduces the number of I/O resubmissions by 3, saving a total of about 9 $\mu$ s in I/O latency and negligible CPU overhead. This is significantly lower than the latency (and CPU overhead) of the network, even with RDMA.

**Range queries.** Figure 2.11d shows the results for TCP on NAND with range queries (a sequential range of keys is read from BPF-KV). BPF-oF achieves over 5 $\times$  higher throughput.

## 2.6.8 Takeaways

BPF-oF provides performance benefits for networked-storage systems, of up to 2.8 $\times$  higher throughput and 2.6 $\times$  lower tail latency with RocksDB, and significantly improves BPF-KV and WiredTiger workloads. In particular, BPF-oF helps workloads that are read-heavy, and whose working set does not fit in the cache, as these yield longer resubmission chains, which BPF-oF accelerates. BPF-oF also provides significant CPU and network savings. Notably, we find that

caching with storage pushdown is a mixed bag. When workloads are highly skewed, a cache avoids issuing unnecessary network traffic, but in some workloads a cache can be detrimental due to the high CPU and network costs of maintaining it.

## **2.7 Conclusion**

This work explores the design of a networked-storage protocol allowing clients to push user-defined storage functions to a server over the network, thereby significantly improving both client and server CPU efficiency. We believe there are significant future research challenges in this area, such as adding support for commonly-needed storage capabilities (e.g., disk arrays, RAID, encrypted storage), or running the functions on a smartNIC. Furthermore, our insights into the interaction of memory and remote disks indicate the potential for nearly memory-less networked storage systems.

## Chapter 3: `cache_ext`: Customize Page Eviction With eBPF

### 3.1 Introduction

In his 1981 paper on OS support for databases, Michael Stonebraker lamented that OS buffer cache mechanisms were ill-suited for the needs of databases at the time [104]. He argued that the buffer cache’s one-size-fits-all eviction policy – approximate least-recently used (LRU) – cannot possibly accommodate the heterogeneity of database workloads.

In the intervening decades, there have been many efforts to allow applications to customize the page cache. In the ’80s and ’90s, several designs were proposed for extensible operating systems, which allow applications to customize the page cache eviction policies [21, 24, 20, 105, 106, 107]. However, none of these operating systems achieved widespread adoption, and they were overtaken by monolithic kernels such as Linux. Some recent studies have tried to introduce page cache customization into Linux. P2Cache [17] is preliminary work that proposed allowing applications to customize the Linux page cache. However, it has significant limitations. Applications can only implement their policies using a single LRU (or MRU) queue, which means P2Cache cannot support many classes of eviction policies that either do not rely on an LRU queue, or require multiple queues (e.g., LFU, LHD [108], MGLRU [109], ARC [110], S3-FIFO [111]). It also globally takes over the page cache, requiring all processes to use it, and restricts applications from accessing pages “owned” by other applications. Another recent effort, FetchBPF [18], allows users to customize Linux prefetching policies, but not page cache eviction, which historically has required much more complex data structures. Finally, Linux itself has tried to contend with this problem, by adding some customization options to its LRU policy (e.g., using `fadvise()`), and by introducing a new MGLRU policy [109] to replace the old one. However, as we will show in §4.5, these options often do not result in any meaningful improvement to application performance,

and they do not allow applications to take full advantage of page cache customization.

At the same time, the diversity of applications and workloads running on Linux has only increased, from enterprise file systems and large-scale distributed datacenter ML training, to multi-media rich applications running on an Android phone. All of these applications must use Linux’s fixed LRU-based policies, despite the fact that they are widely known to be inadequate for many workloads and scenarios (e.g., large scans [112, 113, 114], multi-core applications [115]).

Applications are “stuck” with Linux’s rigid eviction policy for two reasons. First, modifying the Linux page cache is a hard task, requiring extensive kernel knowledge. Second, upstreaming changes to the page cache is difficult, because the changes must work well for the wide range of applications that run on Linux. For instance, it took Google years to upstream its proposed Multi-Generational LRU (MGLRU) policy, and even after several years, it is not enabled by default in all Linux distributions or upstream [109, 19].

The goal of our work is to allow applications to run a very wide range of custom page cache policies within Linux. To this end, we design a novel framework, `cache_ext`, which provides visibility and control of the OS page cache, without requiring the application to make kernel changes. `cache_ext` takes advantage of eBPF [96], a Linux (and Windows) supported runtime that allows safely running application code inside the kernel. We take a cue from `sched_ext`, an eBPF-based framework that allows applications to customize the OS scheduler [116, 117] and that has been adopted by Linux [118].

`cache_ext`’s design is motivated by four main insights. First, modern storage devices support millions of IOPS with low latency, so custom page cache policies must run with low overhead. Therefore, we design `cache_ext` so that its eBPF-based policies run in the kernel, avoiding expensive and frequent synchronization between the kernel and userspace. Second, caching algorithms are very diverse and may use complex data structures. To address this challenge, `cache_ext` exposes a simple yet flexible interface that allows applications to define one or more variable-sized lists of pages, and a set of policy functions (e.g., admission, eviction) that operate on these lists, which can be used to express a wide range of eviction policies. Third, in order for

`cache_ext` to be useful in multi-tenant scenarios, it should allow each application to use its own policy without interfering with others. We identify cgroups as a natural isolation boundary. Thus, `cache_ext` allows each cgroup to implement its own eviction policy without interfering with other cgroups. Finally, custom policies determine which pages to evict and return page references to the kernel. However, these references may be invalid, which could lead to kernel crashes or security breaches. To solve this, `cache_ext` maintains a registry of valid page references, which is used to validate the page references returned by the user-defined policies.

We demonstrate `cache_ext`'s utility and flexibility by implementing several custom eviction policies, from “state-of-the-art” sophisticated policies to “classic” ones: least hit density (LHD) [108], S3-FIFO [111], Multi-Generational LRU (MGLRU) [109], least-frequently used (LFU) and most-recently used (MRU). We also show how `cache_ext` enables *application-informed* policies with only minor changes, allowing applications to use policies that utilize application-level insights. For example, a database can use a custom policy that prioritizes point queries over scans, yielding higher throughput for point queries, or an admission filter based on the thread accessing the page. We compare these `cache_ext` policies with the kernel's default eviction policy and various options (e.g., `fadvise()`), and with the recently-upstreamed MGLRU algorithm. We show that with `cache_ext`, developers can significantly improve their applications' performance far beyond the existing algorithms provided by the Linux page cache. In general, we find that there is no one-size-fits-all policy that improves all workloads – customization is necessary to maximize performance. In particular, `cache_ext` can improve application throughput by up to 38% using “generic” policies, and achieve up to 1.70× throughput and 58% lower P99 latency with application-informed policies.

We will open source `cache_ext` and all implemented policies upon publication. A key benefit of `cache_ext` is that any publicly available policy can be used by anyone, lowering the barrier to using `cache_ext` and experimenting with eviction policies on different workloads.

Our primary contributions are:

- `cache_ext`, a flexible, scalable, and safe eBPF framework for custom eviction policies in the

Linux page cache.

- A suite of custom eviction policies and userspace libraries allowing developers to easily create new policies.
- An evaluation of `cache_ext` across various applications, demonstrating the benefits of custom policies.

## 3.2 Background and Motivation

By default, the page cache buffers write and read operations to and from storage devices. In Linux, the page cache tracks pages and stores them in lists (see §3.2.1), on which it approximates the LRU algorithm. While this scheme works reasonably well for some workloads, it is inadequate for many others. For example, scan-heavy workloads perform poorly with LRU or its approximations [108, 104, 110]. While Linux provides interfaces (e.g., `fadvise()` or `sysctl`) through which the page cache behavior can be tweaked on a global or per-application basis, these interfaces are opaque and may not perform as intended, as we describe in §3.2.1 and §3.6.1.

Therefore, to avoid compromising performance, some applications implement their own userspace-based caches [119, 30, 120, 121]. However, userspace caches are not a panacea. First, they require significant effort to implement. Second, they typically require the application to specify the size of the cache in advance. However, the amount of memory available to an application may change over time (e.g., when multiple applications run on the same physical server). Third, application-specific caches are hard to share across processes, due to security and compatibility issues. Ultimately, even applications that implement their own userspace-based cache often still rely on the page cache by default as a “second-tier” cache [119, 30, 120], allowing operators to fully utilize the server’s memory and share memory across processes. As such, despite the page cache’s limitations, it is still used extensively by storage-optimized workloads, such as key-value stores [25, 91, 30], databases [29, 121], and ML inference and training systems [122, 123].

Customizing the page cache is not an easy task – it is deeply intertwined with other performance- and correctness-critical memory management and filesystem code paths. While work to modernize

the page cache is ongoing, it does not yet seem to have achieved this goal. In particular, MGLRU, an alternative page cache eviction policy, has still not been enabled by default in upstream Linux several years after it was introduced, and it does not provide customization interfaces [109, 19]. Indeed, in §4.5 we show that MGLRU sometimes underperforms and sometimes outperforms the default LRU algorithm, and that in general there is no single eviction policy that performs best across a wide range of workloads.

We now provide a primer on the Linux page cache. We also describe the eBPF framework, which `cache_ext` uses to allow applications to write custom page cache policies.

### 3.2.1 Linux Page Cache

The page cache is a core component of the Linux kernel, responsible for accelerating access to storage. While anonymous memory pages are stored similarly to file-backed memory, in this paper we focus specifically on file-backed memory. The kernel’s default eviction policy is an LRU approximation algorithm which uses two FIFO lists.<sup>1</sup> As shown in Figure 3.1, when a page is first fetched from storage, it is added to the tail of the inactive list. If that page is accessed again, it will eventually be promoted to the active list. This policy uses the inactive list as a preliminary filter and keeps frequently accessed pages in the active list. On eviction, pages are removed from the head of the inactive list. If necessary, the page cache will balance the lists by demoting pages from the head of the active list to the tail of the inactive list. Notably, during balancing or shrinking, pages in the active list that have been referenced are typically demoted to the inactive list, rather than being given another chance in the active list, as is typical for LRU or CLOCK algorithms.

Importantly, active and inactive lists are segmented by cgroup. cgroups are a Linux feature which isolate resource usage for groups of processes [124]. Each cgroup has its own set of page cache lists which count toward its memory allocation, allowing for cgroup-specific eviction when its memory threshold is reached. Processes in cgroup A can access a page “owned” by cgroup B – such an access will update the page’s metadata (affecting its placement in cgroup B’s lists), but

---

<sup>1</sup>The Linux page cache algorithm description is based on Linux v6.6.8.

will not count against cgroup A’s memory limit. The combination of these per-cgroup lists make up the page cache as a whole.<sup>2</sup>

The page cache also keeps track of “shadow entries” in order to mitigate thrashing. These entries keep track of metadata enabling calculation of a page’s refault distance (i.e. the time elapsed between eviction and the new request). If a page has been evicted and then fetched again recently enough, the kernel may decide to insert it directly into the active list instead of the inactive list. There are several additional edge cases and heuristics in the kernel’s implementation, but these are the broad strokes of the existing policy.

**Folios.** Linux developers are converting various usages of `struct page` to *folios*, which represent either zero-order pages (a single page) or the head page of a compound page (a set of contiguous physical pages that can be treated as a single larger page) [125]. While the page cache now largely uses folios, we use the terms “folio” and “page” interchangeably, as in our workloads all folios represent a single page.

**Userspace interfaces.** While LRU is a commonly-used eviction policy that works well across many workloads, there are many applications that would benefit from a different policy for their I/O requests. For example, LRU is notoriously bad for scan-like access patterns. This gap between applications and the kernel can be partially mitigated by the `madvise()` and `fdadvise()` system calls. These interfaces allow userspace applications to give *hints* to the kernel about how to handle certain ranges of memory or files.

While these hints may help in simple cases, we show in our evaluation that they don’t function as expected for some workloads. Additionally, while the hints may have a semantic meaning, their actual behavior is highly dependent on the kernel implementation, which is opaque, may change across versions, and can yield unexpected results [126, 127]. Advice values may also be ignored by the kernel for a range of reasons, or may have restrictions on what memory they can be applied to. Most importantly, these hints are still subject to the basic inflexible structure of the kernel’s

---

<sup>2</sup>Technically, each NUMA node has its own set of per-cgroup lists, but this does not affect our design.

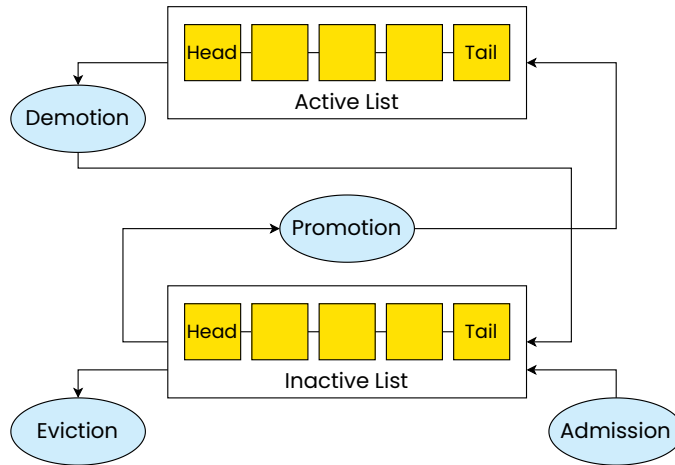


Figure 3.1: Overview of the current Linux page cache eviction policy.

LRU-like policy.

### 3.2.2 eBPF

eBPF [96] is a sandboxing technology that enables userspace functions to run in the Linux kernel in a safe and controlled manner. eBPF has found many use cases, including observability [128], security [129, 130], scheduling [118, 116, 117], and I/O acceleration [1, 131, 6, 7, 132]. Recent work has also proposed using eBPF for customizing page cache behavior [17] and prefetching [18], but these works do not provide a general and flexible interface that allows multiple processes to customize their page cache policies. eBPF programs are *verified* by the kernel before they can be run, ensuring, for example, that the programs don't contain illegal memory accesses, and that they will terminate within a fixed number of instructions.

## 3.3 Challenges

There are several challenges in allowing applications to customize the page cache using eBPF. We describe them below.

1. **Scalability.** Modern SSDs support millions of IOPS [133, 134], requiring the page cache to efficiently handle millions of events a second. Any changes to the page cache in order to enable custom policies must incur a low overhead, and the policies themselves must also be

efficient.

2. **Flexibility.** Researchers have proposed many different caching algorithms for different use cases. These algorithms often require custom data structures. Any interface for custom policies must be flexible enough to accommodate the diversity of existing caching algorithms.
3. **Isolation and sharing.** The page cache is shared by many applications. Therefore, we must avoid a situation where one application's policy interferes with those of other applications, while still allowing applications to benefit from the shared nature of the page cache.
4. **Security.** Custom eviction policies return page references to the kernel to indicate which pages to evict. This must not lead to unsafe memory references.

### 3.4 Design and Implementation

In this section, we present `cache_ext`'s architecture and discuss how it addresses the challenges described in §3.3. Figure 3.2 shows a diagram of the system. At a high level, `cache_ext` allows users to run custom eviction *policy functions*, which are implemented as eBPF functions in the kernel. The policy functions are triggered by particular events (e.g., folio eviction, access, admission), and they operate on a user-specified number of variable-sized *eviction lists*, which store *pointers* to folios managed by the policy. The policy functions determine which folios to admit or evict to and from the lists based on metadata (e.g., access frequency, recency, which thread accessed the folio), which is stored in eBPF maps. On eviction, `cache_ext` runs a user-defined eviction function to propose a set of *eviction candidates* for the kernel to evict. While this interface is relatively simple, it is quite flexible, and can support a wide set of eviction policies from the literature either exactly or approximately.

We now describe `cache_ext`'s design in detail, starting with our design choice to implement `cache_ext`'s eviction policies within the kernel, rather than in userspace, to ensure scalability (challenge 1 from §3.3).

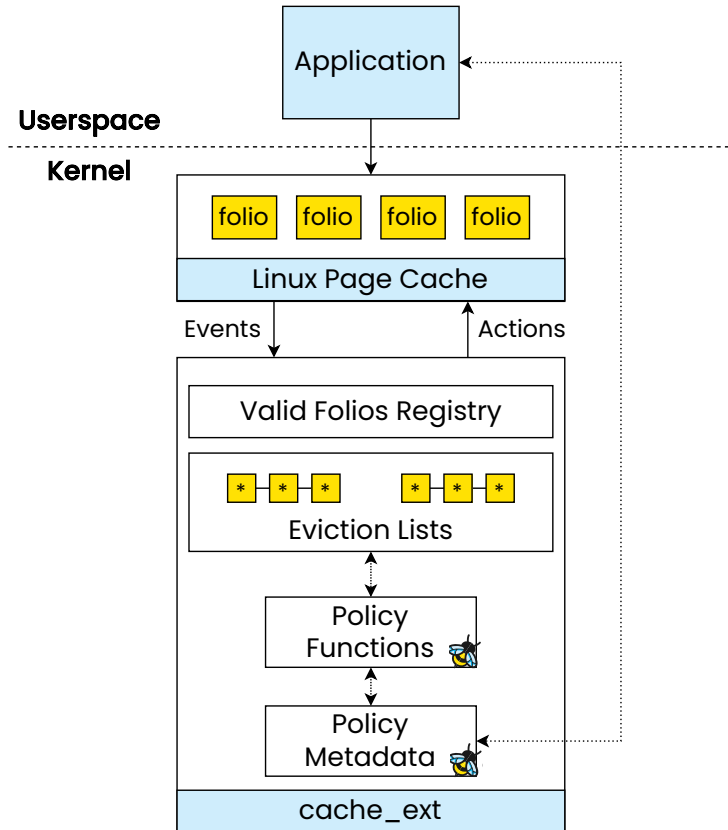


Figure 3.2: Overview of `cache_ext`. Eviction lists hold pointers to folios.

### 3.4.1 Policies in Kernel or Userspace?

Our first key design decision is whether to run `cache_ext`'s policies in the kernel or in userspace. While from a development standpoint it might be simpler to run the eviction policies in userspace, doing so would require notifying userspace about all page cache events. However, modern SSDs can service millions of IOPS, each of which may trigger a page cache event (e.g., folio access, insertion).

**Userspace-offload overhead.** We estimate the “best-case” overhead of such a userspace-offload architecture. We attach eBPF programs to existing kernel tracepoints (folio inserted, accessed, and evicted). The eBPF programs use a lockless ring buffer to notify userspace on each event [135]. Since no userspace logic actually processes these events, this provides an optimistic measure of this architecture’s overhead.

Workload	Baseline	Benchmark	% Degradation
YCSB A	82,808 op/s	69,089 op/s	-16.6%
YCSB C	76,166 op/s	62,578 op/s	-17.8%
Uniform	44,618 op/s	35,443 op/s	-20.6%
Search	42.3s	44.4s	-4.7%

Table 3.1: Performance of workloads without and with userspace-dispatch.

We run two applications on an enterprise SSD to evaluate this architecture: YCSB workloads with RocksDB [25], a key-value store, and a file search workload using ripgrep [136], a parallelized grep-like tool, where we search the Linux kernel sources 10 times. The workloads are allocated 8 GiB and 1 GiB of memory, respectively. We run these applications on both the baseline system and with the eBPF benchmark programs. The results are presented in Table 3.1, with the benchmark yielding up to a 20.6% performance decrease, *without even implementing a custom eviction policy*.

As such, we rule out implementing page cache policies in userspace, and instead opt to run the policies within the kernel as eBPF functions. We decide to use eBPF as it has already proven to match the kernel’s performance, even in performance-critical domains such as networking [131] and storage [1]. While eBPF programs face many restrictions due to the verifier, we find that `cache_ext` can provide sufficient flexibility for custom policies, as we describe below.

### 3.4.2 Interface

Caching is an active research area, with many recently-proposed eviction and admission algorithms [137, 138, 108, 139, 140, 141, 111, 142] that take advantage of different workload features (e.g., recency, frequency, size), using various techniques (e.g., conditional probability models [108], Markov chains [137], machine learning [139]). To ensure flexibility, `cache_ext` should allow experimenting with a wide range of policies, including relatively sophisticated ones. We now describe `cache_ext`’s API and demonstrate how it can be used to implement a wide range of policies, addressing challenge 2 in §3.3.

## Policy Functions

`cache_ext` allows applications to define custom eviction policies as *policy functions*, a set of eBPF programs that trace caching events and determine which folios to evict from the page cache. Policy functions are triggered by five events: policy initialization, request for eviction, folio admission, folio access, and folio removal. Policy functions are implemented using eBPF's `struct_ops` kernel interface [143], as shown in Figure 3.3.

**Eviction vs. removal.** These five events are central to page cache decisions. Notably, requests for eviction and folio removal are different: in the former, the kernel asks the policy to propose folios to evict, while in the latter the kernel informs the policy that a folio was actually evicted. This distinction exists for the following reasons. A folio can be evicted in circumvention of the “normal” eviction path if, for example, the file containing it is deleted. Conversely, in rare cases, proposing a folio for eviction does not guarantee that it will be evicted (e.g., the folio is in use by the kernel).

**struct\_ops.** We use eBPF's `struct_ops` feature to minimize verifier changes to add new eBPF hooks. `struct_ops` is designed to allow kernel subsystems to expose modular interfaces to eBPF components. `struct_ops` programs are loaded into the kernel like any other eBPF program. Using `struct_ops` also makes it much easier to extend `cache_ext` and add new hooks. For example, we implemented an extension to `cache_ext` that added a page cache admission filter with only 15 additional lines of verifier-related code, which we evaluate in §3.6.1. Our `struct_ops` hooks are located within the core page cache code, requiring ~200 lines of modifications. Once a set of policy functions is loaded and verified, the kernel will call the appropriate eBPF program when the corresponding event occurs. The eBPF programs can then manipulate the eviction data structures and metadata, as we now describe.

```

// Policy function hooks
struct cache_ext_ops {
    s32 (*policy_init)(struct mem_cgroup *memcg);
    // Propose folios to evict
    void (*evict_folios)(struct eviction_ctx *ctx, struct mem_cgroup *
        memcg);
    void (*folio_added)(struct folio *folio);
    void (*folio_accessed)(struct folio *folio);
    // Folio was removed: clean up metadata
    void (*folio_removed)(struct folio *folio);
    char name[CACHE_EXT_OPS_NAME_LEN];
};

struct eviction_ctx {
    u64 nr_candidates_requested; /* Input */
    u64 nr_candidates_proposed; /* Output */
    struct folio *candidates[32];
};

```

Figure 3.3: `struct_ops` for `cache_ext` and eviction context.

## Eviction Lists

Eviction algorithms are implemented on a wide range of data structures. Nevertheless, we observe that many of these policies can be implemented either exactly or approximately using linked lists, where the policy iterates over one or more lists and evicts items based on a calculated per-item score. For example, the “classic” eviction policies, (e.g., LRU, MRU) are all based on lists, with items inserted or evicted from the head or tail of a list. Similarly, families of policies like ARC [110], segmented LRU [144] or MGLRU [109], can be implemented using multiple variable-sized lists, where items are inserted into any list or moved between lists. Even recent “state-of-the-art” policies, such as LHD, S3-FIFO, or LRB either store data directly in a list [111, 141], or sample objects and evict the ones with the lowest *score* [108, 139].

In order to facilitate an interface flexible enough for all these policies, `cache_ext` is built around an *eviction list API*, a simple interface for policies to construct and manipulate a set of variable-sized linked lists. Each node in the list corresponds to a single folio, and stores a pointer to that folio, rather than the folio itself. Importantly, the actual folios are still stored and maintained

Eviction list API
<code>u64 list_create(struct mem_cgroup *memcg)</code>
<code>int list_add(u64 list, struct folio *f, bool tail)</code>
<code>int list_move(u64 list, struct folio *f, bool tail)</code>
<code>int list_del(struct folio *f)</code>
<code>int list_iterate(struct mem_cgroup *memcg, u64 list, s64(*iter_fn)(int id, struct folio *f), struct iter_opts *opts, struct eviction_ctx *ctx)</code>

Table 3.2: `cache_ext` eviction list API.

by the default kernel page cache implementation, in order to minimize changes to the kernel.

**kfuncs.** This API is implemented as a set of eBPF `kfuncs` (kernel functions that are exposed to eBPF) and is shown in Table 3.2.<sup>3</sup> For example, `init()` can call `list_create()` to create a new eviction list, and `folio_added()` can call `list_add()` to add the folio to a list. Newly-created lists are added to a “registry”, an internal per-policy hash table which maps from list IDs (exposed to eBPF) to the lists themselves. Notably, these lists are *indexed* – that is, given a folio pointer, the APIs can directly obtain that folio’s list node. This is necessary for operations such as deletion from the list, and is facilitated using a per-policy hash table which maps from folios to list nodes. We discuss this hash table further in §3.4.4.

### Eviction Candidate Interface

Policy functions iterate over their eviction lists in order to determine which folios to evict. Note that policies do not directly evict folios – rather, they propose *eviction candidates* to the kernel, which checks if the folios are indeed valid eviction targets (i.e. not pinned or in use by the kernel) and evicts them if they are. Eviction candidates are proposed to the kernel in batches of up to 32 folios. Most of `cache_ext`’s changes to the core page cache code are to facilitate this batched eviction process.

**List iteration.** eBPF currently does not provide a straightforward way to iterate over the eviction lists, so `cache_ext` provides a new iteration `kfunc` which allows policy functions to iterate

<sup>3</sup>The actual functions have a “`cache_ext`” prefix to prevent name collisions, but we omit it for brevity.

over an eviction list and make decisions for each node. Specifically, `list_iterate()` takes a list to iterate over, an `iter_opts` struct, an eviction context, and a callback function. The callback function, which is also an eBPF program, is called on each node, and the policy decides whether to keep or evict that folio. Folios chosen for eviction are added to the `candidates` array in the `eviction_ctx` struct. The `iter_opts` struct specifies how the interface should treat evaluated folios. For example, they can be left in place, moved to the tail of the list, or moved to a different list. This enables implementing policies that make use of multiple lists and require balancing the lists, such as S3-FIFO or ARC. We also provide a “batch scoring” mode for this interface, where the callback function is used to compute *scores* for  $N$  folios, with the  $C$  folios with the lowest score chosen for eviction. This mode can be used for policies such as LFU.

## **eBPF limitations**

We ran into a number of challenges when implementing `cache_ext`'s eviction lists. eBPF maps, the standard way to maintain state in eBPF programs, do not provide interfaces that both store items in a specified order while also providing random access, both of which are necessary to implement eviction properly. Specifically, eBPF provides maps such as `BPF_MAP_TYPE_QUEUE` and `BPF_MAP_TYPE_STACK`, which provide `pop()` and `push()` operations, but do not allow deleting or accessing elements from the “middle” of the map. Conversely, `BPF_MAP_TYPE_HASH` provides random access, but no method to easily maintain an ordering of elements (e.g., MRU order). A notable exception is `BPF_MAP_TYPE_LRU_HASH`, which provides both an LRU structure and random-access, but is too deeply tied to its specific algorithm for our purposes [145]. This necessitated the development of a custom data structure for `cache_ext`.

While eBPF has introduced experimental support for custom data structures and more complex locking in eBPF, this support is not yet mature enough for our use case [146, 147, 148]. As such, we designed our list API to be managed by the kernel and exposed to eBPF via `kfuncs`. Additionally, in order to avoid concurrency issues and verifier limitations around locking, the provided API is concurrency-safe and makes use of locks under the hood, in the kernel implementations. As eBPF

```

u64 lfu_list;
int lfu_policy_init(struct mem_cgroup *cg) {
    lfu_list = list_create(cg);
    return 0;
}
void lfu_folio_added(struct folio *folio) {
    u64 freq = 1;
    list_add(lfu_list, folio, true); // Add to tail
    bpf_map_update_elem(&freq_map, &folio, &freq);
}
void lfu_folio_accessed(struct folio *folio) {
    u64 *freq = bpf_map_lookup_elem(&freq_map, &folio);
    __sync_fetch_and_add(freq, 1); // Increment freq
}
long score_lfu(int id, struct folio *folio) {
    return bpf_map_lookup_elem(&freq_map, &folio);
}
void lfu_evict_folios(struct eviction_ctx *ctx, struct mem_cgroup *cg)
{
    struct iter_opts opts = { /* Set scoring mode */ };
    list_iterate(cg, lfu_list, score_lfu, &opts, ctx);
}
void lfu_folio_removed(struct folio *folio) {
    bpf_map_delete_elem(&freq_map, &folio);
}

```

Figure 3.4: Simplified LFU implementation with `cache_ext`.

matures, new features could further reduce overhead and provide even more flexibility for eBPF policies.

### Example: LFU Policy

To illustrate how `cache_ext`'s policy functions can be used to implement custom policies, we walk through implementing a simple eviction policy, LFU, using `cache_ext`. LFU evicts the least-frequently accessed item in the list, which requires storing access frequency metadata. Our LFU implementation uses a single list and an eBPF map for frequencies. It approximates LFU with `cache_ext`'s batch scoring mode to select the  $C$  (e.g., 32) least-frequently accessed folios out of the first  $N$  (e.g., 512) folios.

A simplified version of the policy is shown in Figure 3.4. When the policy is loaded, `lfu_policy_init()`

is called, creating a new eviction list. On insertion, `lfu_folio_added()` adds the folio to the tail of the list using `list_add()` and sets its frequency to 1 in the `freq_map` eBPF map (not shown). When a folio is accessed, we increment its frequency. On eviction, `lfu_evict_folios()` calls `list_iterate()`, which calls the `score_lfu()` callback function on  $N$  nodes in the list, until  $C$  eviction candidates are proposed. The score function returns the frequency of each folio as its score. `cache_ext` then selects the  $C$  folios with the lowest scores, which are added to `ctx->candidates` as eviction candidates. Folios not selected as eviction candidates are then moved to the end of the list by `list_iterate()`. The kernel will then attempt to evict the eviction candidates. When a folio is evicted, `lfu_folio_removed()` is called, and the folio's metadata is removed from the map. Note that it is not necessary to remove the folio from the list on eviction, as this is done by `cache_ext`. We discuss this point further in §3.4.4.

### 3.4.3 Isolation

We now tackle the third challenge from §3.3: allowing applications to deploy their own policy functions without interfering with other applications' policies, while preserving the sharing property of the page cache, whereby applications can avoid having to load duplicate pages into memory.

**Cgroups.** We observe that implementing policies within a cgroup can address this challenge, due to the fact that within a cgroup, processes have the same custom eviction policy, and different cgroups can each use their own eviction policy. In addition, deploying per-cgroup policies fits the common pattern of deploying modern applications via containers, which isolate each application in its own memory cgroup. Note that processes from cgroup A can still access page cache memory managed by cgroup B, and benefit from accessing shared data. However, both `cache_ext` and the baseline page cache are not fully isolated. If a page critical for cgroup A is managed by cgroup B, which evicts it, that could lead to a performance degradation. In that case, cgroup A will likely bring the page back into memory and manage it in the future. However, we believe that such

workloads are rare in practice, as it is unlikely that two applications with entirely different access patterns will operate on the exact same files frequently enough to cause a problem.

We extend eBPF’s `struct_ops` functionality to support cgroup-specific `struct_ops` for per-cgroup policies (it currently only supports system-wide policies). This involved adding a cgroup identifier (in the form of a file descriptor) to the kernel’s `struct_ops` loading interface, along with corresponding `libbpf` interfaces in userspace.

### 3.4.4 Security

**Memory Safety.** We must ensure that `cache_ext` prevents unsafe memory accesses (challenge 4 from §3.3). Specifically, `cache_ext` must ensure that eBPF programs only return valid pointers (i.e. in the eviction candidate interface). Otherwise, a malicious eBPF program could return invalid values, leading to memory corruption or a kernel crash. We note that `sched_ext` solves this in part by using PIDs as identifiers for processes. However, folios do not have analogous easily-obtainable unique identifiers, so we resort to using pointers.

In order to validate these pointers, we implement a “valid folios” registry in the kernel. When a folio is inserted, it is added to the registry. When a folio is evicted, it is removed from the registry. When `cache_ext` proposes folio eviction candidates, the kernel uses the registry to verify that each candidate is indeed a valid folio before proceeding with eviction. This registry is implemented as a hash table with a per-bucket lock, which also stores a folio’s list node (as described in §3.4.2), which maps from folio pointer to list node. We find that this design incurs minimal overhead, which we evaluate in §3.6.3. Future developments in eBPF may make it easier to keep track of “trusted” pointers, potentially allowing us to remove this check and further reduce overhead.

**Eviction fallback.** We protect against adversarial behavior by providing a fallback for eviction. For example, if the kernel asks a faulty policy to evict 10 folios, but it only proposes 5 candidates, the kernel will fall back to its default policy and evict additional folios. Similarly, when a folio is evicted, the kernel ensures that it is removed from any eviction lists, in order to release memory

resources and minimize stale references lying around. Similar fallbacks are present in other frameworks, such as `sched_ext`, which implements a watchdog that forcibly removes misbehaving policies.

**kfuncs.** We ensure that `cache_ext`'s list `kfuncs` verify inputs, perform bounds-checking, and enforce loop termination, in order to minimize potential security risks. We note that `kfuncs` are the current best practice for eBPF to interface with complex data structures, and have been used in multiple upstream kernel components, such as `sched_ext` [118].

**Root privileges.** Loading `cache_ext` policies requires root privileges, like other eBPF frameworks. This impacts both security and usability. Custom scheduling frameworks like `sched_ext` and `ghOSt` [118, 117] mitigate this with a privileged policy loader, allowing policies to be managed through `systemd`. We envision a similar solution for `cache_ext`.

### 3.4.5 Kernel Implementation Complexity

Implementing `cache_ext` required adding ~2000 lines to the kernel. Only a fraction of these lines modified the core kernel: 210 lines in the page cache (mostly the eBPF hooks and eviction candidate batching), 80 lines in the verifier (registering our callback functions), and 80 lines in cgroup code. Implementing per-cgroup `struct_ops` required 220 lines in the kernel and 75 lines in `libbpf`. The remaining lines implemented pure `cache_ext` functionality: 750 lines for `cache_ext`'s eviction list `kfuncs` and 580 lines for registry operations.

## 3.5 Policies

In this section, we describe our experience implementing several custom page cache policies on `cache_ext`: from simple “classic” policies (MRU and LFU) to state-of-the-art policies such as LHD [108], which uses conditional probabilities to model different page features (e.g., age, frequency), and S3-FIFO [111]. We also re-implemented the newly-introduced kernel policy MGLRU [109] on `cache_ext`, and in §3.6.3 compare the `cache_ext` version with the

native-kernel version. We also explore how an application can make its eviction policy *aware* of application-specific information, such as assigning different priorities to specific types of requests.

### 3.5.1 S3-FIFO

S3-FIFO [111] is a recent caching policy designed for key-value caches, which uses three FIFO queues to quickly remove “one-hit wonders” (keys that are accessed only once). It has been shown to yield high throughput for in-memory key-value caches. S3-FIFO uses a main FIFO and a small FIFO to hold ~90% and 10% of the objects, respectively. New objects are added to the small FIFO. The small FIFO filters out short-lived objects, while objects that are accessed more often are promoted to the main FIFO. It uses a third ghost FIFO to track recently-evicted objects, in order to promote them to the main FIFO on readmission.

**Implementation.** We create eviction lists for the main and small FIFOs, and a `BPF_MAP_TYPE_LRU_HASH` map for the ghost FIFO. The map then automatically removes entries from the ghost FIFO in LRU order when it hits capacity. When a folio is evicted, we create a ghost entry using a pointer to its `struct address_space` (which represents a file’s contents), along with the folio’s offset in the file, as the key. Note that we cannot use folio pointers as the key, as they are not persistent across evictions. We maintain folio access frequencies in an eBPF map. We use eviction candidate requests to evict folios, but also to maintain the 90-10 ratio between the main and small lists. We use `cache_ext`’s eviction iteration interface: if a folio’s access frequency is greater than 1, we move it to the tail of the main list, balancing the lists. Otherwise, we propose the folio for eviction, and move it to the tail of the small list so that it isn’t considered again before it is evicted. When evicting from the main list, we use the iteration interface to find folios with access frequency of 0.

### 3.5.2 Least Hit Density (LHD)

LHD is a sophisticated eviction policy that uses conditional probabilities to predict future object accesses [108]. LHD uses a *hit density* metric to determine which objects should be evicted, along

with a *dynamic ranking* approach which allows it to automatically tune its eviction policy over time.

We use one eviction list, with folios divided into *classes* based on their last access and their age at that time. Each class stores statistics (e.g., hits, evictions, hit densities) for different folio ages. Folios use metadata from classes based on which class they most closely correspond to at a given time. We maintain folio metadata, such as last access time, in an eBPF map. LHD iterates over the list and selects the folios with the lowest hit density as eviction candidates. To maintain accurate hit densities, LHD requires periodically “reconfiguring” its statistics in order to ensure that its probability distributions are accurate and aged appropriately over time using an exponentially weighted moving average (EWMA).

**Reconfiguration.** Reconfiguration runs every  $N$  folio insertions or accesses (where  $N$  is a relatively large number – e.g.,  $2^{20}$ ). Reconfiguration is a relatively expensive process, as it updates all metadata. In order to avoid the page cache’s insertion or access hot paths, we use an eBPF ring buffer to notify userspace that reconfiguration needs to take place. Userspace then calls a `BPF_PROG_TYPE_SYSCALL` program, which runs an eBPF program without attaching it to a specific hook. This program then performs the required reconfiguration, including computing updated hit densities, and scaling or compressing distributions as necessary. We use atomic operations to ensure that the page cache can continue using these values, albeit with some potential inaccuracy, which we permit for the sake of performance. While we could implement this reconfiguration step in userspace, doing so would have required numerous syscalls to interact with eBPF maps, and atomic updates would not have been possible. Additionally, we note that in a standard LHD policy, hit densities and other parameters are stored as floating-point values. However, eBPF does not support floating-point operations, so we resort to scaling values by a large constant in order to approximate such calculations.

### 3.5.3 Multi-Generational LRU (MGLRU)

MGLRU is a complex page eviction policy that was recently added to the Linux kernel [109]. MGLRU groups folios into *generations*, each capturing folios with similar access recency. Each generation is a list and is further divided into *tiers*, with tiers acting as logarithmic buckets based on access frequency. The kernel maintains up to four generations and four tiers per generation. Eviction starts from the oldest generation, using a *tier threshold* computed by a PID controller based on eviction and refault statistics. Pages above the threshold are promoted to the next generation; others are evicted.

Concretely, generations are implemented as eviction lists, stored in a circular buffer. We track the minimum and maximum *generation numbers* – `min_seq` and `max_seq`, respectively. A generation number is mapped to its eviction list by computing the generation number modulo `max_nr_gens` to find the list index. A new generation is created (aging) by incrementing `max_seq`. When the oldest generation is empty, it is retired by incrementing `min_seq`. We use a map to store the generation and access frequency of each folio. We port the PID controller logic from the kernel, and maintain the global statistics it needs using atomic operations. Refault detection, needed for computing refault statistics, is implemented using ghost entries, similar to the LHD policy in §3.5.2. Finally, we serialize the generation aging operations using an eBPF spinlock. On folio insertion, we calculate the folio’s generation and add the folio to the generation’s head. On eviction, the policy iterates the minimum generation list, obtains a tier threshold from the PID controller, and selects folios below the threshold as eviction candidates, while promoting folios above the threshold.

### 3.5.4 Classic Policies: LFU, MRU, FIFO

We implemented three additional “classic” eviction policies: LFU, MRU and FIFO. §3.4.2 described our LFU implementation. For MRU, we add folios to the head of the list upon insertion, and move them to the head on access. However, if we evict folios right after they are added to the page cache, they may still be in use by the kernel to service the I/O request. This would lead to

the kernel refusing to evict the folios and resorting to the fallback path to evict folios. Therefore, we skip a small fixed number of folios when iterating the eviction list before proposing eviction candidates. For FIFO, folios are removed from the head of the eviction list.

### 3.5.5 Application-Informed Eviction (GET-SCAN)

In addition to enabling the implementation of a variety of general eviction algorithms, `cache_ext` enables applications to use eviction algorithms tailored to their design. In other words, the eviction algorithm can be made *aware of application-level abstractions* and uses this information to make better decisions. To illustrate the value of application-informed policies, consider the case of heterogeneous queries in databases. For example, a database serving financial transactions could see many small queries for individual payments, while also performing slower scan-like queries in the background to conduct fraud detection, reconciliation, and other business processes. While these scan-like queries are important, they typically have more relaxed service-level objectives. However, these large scan-like requests can “pollute” the page cache and degrade the performance of the smaller requests, as generic eviction algorithms struggle to isolate the folios used by these requests. Ideally, the page cache should prioritize the small requests over the large ones in the presence of memory pressure. We note that such workloads have frequently been examined in the literature [116, 149]. Using `cache_ext`, we can build an application-informed policy that is aware of the different request types.

A folio accessed by a SCAN should not be worth the same as a folio accessed by a GET. To implement prioritization, the policy uses two eviction lists: one for folios inserted by GETs, and the other for those inserted by SCANS. When loading the policy, the application initializes an eBPF map with the PIDs of the SCAN threads. When a folio is inserted, the policy checks whether the PID of the current task is in the map to determine which eviction list to add the folio to. Each eviction list independently maintains an approximate LFU policy, as described in §3.4.2. When the kernel requests eviction candidates, the policy prioritizes evicting folios from the SCAN list. Figure 3.5 illustrates this policy.

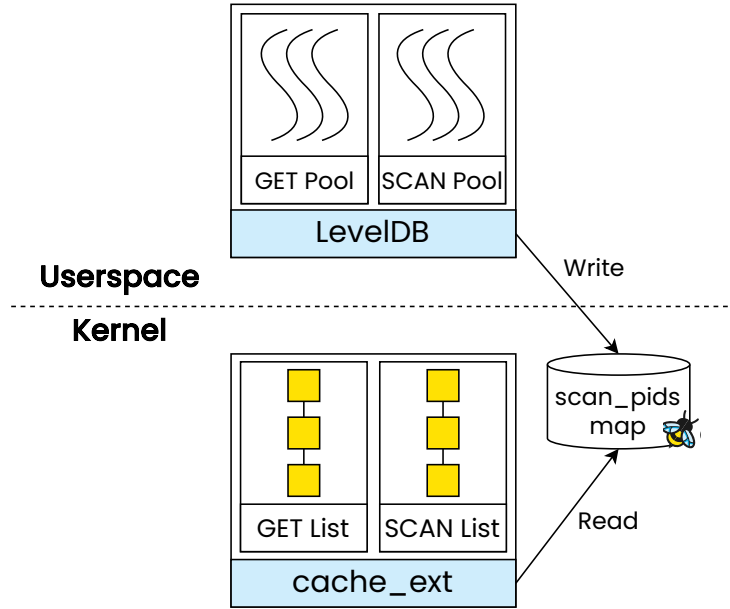
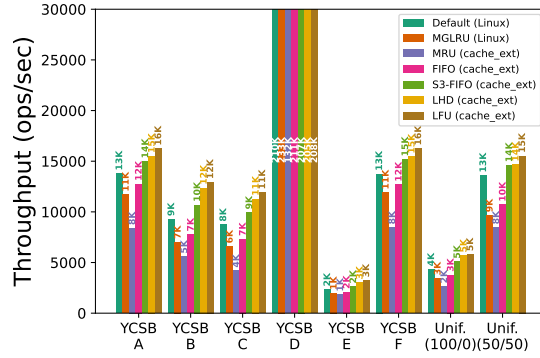


Figure 3.5: Overview of GET-SCAN policy implementation.

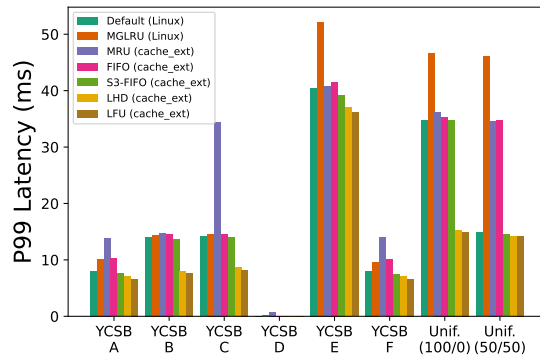
### 3.5.6 Application-Informed Admission Filter

As mentioned in 3.4.2, we extend `cache_ext` to also support an admission filter hook. When a folio is requested and *is not yet present* in the page cache, the admission filter determines whether to allow the folio into the page cache, or to service it analogously to a direct I/O request. Such a hook is useful for preventing thrashing in workloads that may need to read many pages once, but cannot use direct I/O for performance or coherency reasons. For example, LSM-tree key-value stores use a compaction process which periodically reads large files potentially unrelated to servicing requests, resulting in thrashing of “relevant” files. However, direct I/O cannot easily be used for these large reads, as these files (using the same file descriptor) may still be read from the page cache by other threads to process requests.

To this end, we create an admission policy that prevents this thrashing behavior, by preventing background compaction from evicting folios that will be needed for read requests in the critical path, on an LSM-tree key-value store (RocksDB). When RocksDB spawns compaction threads, we place the thread TIDs in an eBPF map. When a folio is proposed to be added to the page cache, the policy checks whether the thread adding it is a compaction thread in the TID map. If so, our



(a) Throughput.



(b) P99 latency.

Figure 3.6: YCSB workload results.

policy does not add it to the page cache. Otherwise, it is added to the page cache in the “normal” path.

### 3.6 Evaluation

We aim to answer the following questions:

- Q1:** Can `cache_ext` policies improve application performance with low developer effort? (§3.6.1)
- Q2:** Can different applications use different policies without interfering with each other? (§3.6.2)
- Q3:** What is the overhead of `cache_ext`? (§3.6.3)

**System configuration.** We conduct our experiments on Cloudlab [101] c6525-25g machines, with a 16-core AMD Rome CPU, 128GB of memory and a 480GB SSD drive. We use CPU-pinning and disable SMT, swap, and address space randomization to make our results more repro-

ducible. We also drop the page cache before each test. We run Ubuntu 22.04 with Linux v6.6.8 as the kernel.

### 3.6.1 Custom Policies (Q1)

In this section, we evaluate the policies described in §3.5 on a number of different workloads and configurations.

#### YCSB

We evaluate our policies by running LevelDB [91], a popular key-value store, on the YCSB (Zipfian) workloads, as well as against uniform and uniform-read-write workloads. We compare the custom policies against both the default and MGLRU Linux policies, using a 100GiB database with a 10GiB cgroup. Our results in Figure 3.6 show that `cache_ext`'s LFU policy performs best, outperforming both the default and MGLRU, for all the YCSB Zipfian workloads and the uniform workloads, except for YCSB D, which only uses the latest key-value pairs and as such is cached entirely in-memory. This is not surprising: LFU should be an ideal policy for YCSB, since YCSB by default assigns a fixed Zipfian distributed for the access distribution of each key. Therefore, a frequency-based policy should perform best. `cache_ext`'s LFU achieves up to 37% better throughput than the default Linux policy, and interestingly, it outperforms MGLRU by an even greater margin. We note that LHD and S3-FIFO also outperform the Linux policies; in particular, LHD provides throughput very similar to LFU. As expected, the MRU policy performs worse than the baseline, due to its mismatch with the workload's access pattern. However, our simple FIFO policy slightly outperforms MGLRU in most cases, but not the default policy, likely due to its low overhead compared to MGLRU. We also measure the P99 read latency, for which `cache_ext` beats the default policy by up to 55%. Note that YCSB D's tail latency barely registers in the figure due to its lack of disk accesses.

<b>Takeaway 1:</b> <code>cache_ext</code> can significantly improve application performance even with simple policies (e.g., LFU) that match the application's access patterns.
---

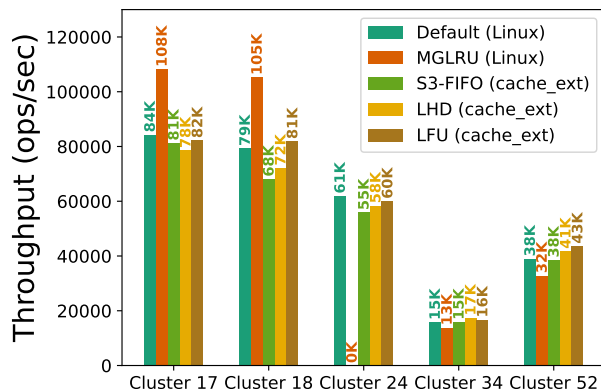


Figure 3.7: Twitter workload results (LHD, S3-FIFO, and LFU policies) using LevelDB. No one policy performs best across the different clusters.

## Twitter Traces

For many real-world workloads, it may not be obvious in advance which policy works best for a given workload. `cache_ext` makes experimentation easy, allowing developers to implement a set of policies and *empirically* choose the best one for each workload.

We evaluate our LHD, LFU, and S3-FIFO policies on production traces taken from the Twitter cache workloads [103]. The workloads divide the traces by cluster ID. We compare these policies to Linux’s default and MGLRU policies. Each cluster was evaluated with a cgroup size set to 10% of the cluster’s data size using LevelDB. As shown in Figure 3.7, we find that, in general, there is no single policy that is best for all workloads. While LHD beats the default and MGLRU policies by 13% and 30%, respectively, on cluster 34, and LFU beats them by 13% and 34% on cluster 52, MGLRU dominates on clusters 17 and 18. In cluster 24, the default policy is best, while MGLRU consistently resulted in out-of-memory errors, hence its 0 throughput result. Meanwhile, S3-FIFO beats or matches the baseline on clusters 34 and 52, but does not outperform the other `cache_ext` policies.

**Takeaway 2:** There is no one-size-fits-all policy that performs best for all workloads. Customization and experimentation are necessary to maximize performance.

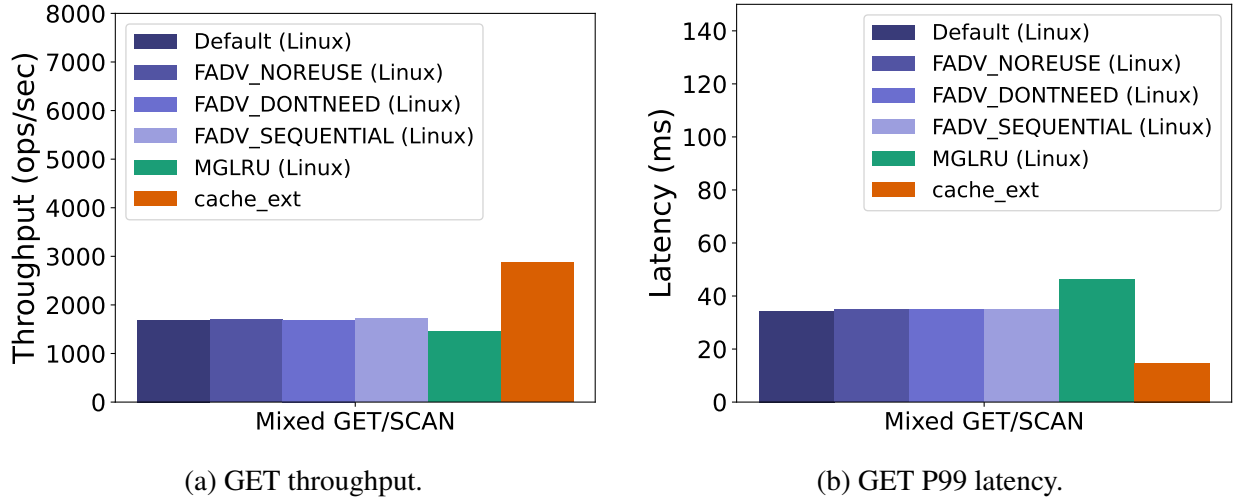


Figure 3.8: Mixed GET-SCAN workload results.

### Application-Informed Policy (GET-SCAN)

To simulate the application we describe in §3.5.5, we run LevelDB with a mixed SCAN/GET workload. This workload is highly skewed and is composed of 99.95% GET requests with a few SCANS (0.05%). We use a separate thread-pool for SCAN requests to avoid head-of-line blocking at the scheduling level, as per prior work [116, 149], with a disjoint set of threads handling GET requests. While the workload exhibits good cache locality for GETs, it has poor locality for SCANS, which span many folios and exhibit high reuse distance. The default kernel eviction policy does not handle this scenario well, leading to cache pollution due to the SCAN folios.

To evaluate the policy, we compare against Linux’s default and MGLRU policies, and various `fsync()` options: `FADV_DONTNEED`, `FADV_NOREUSE` and `FADV_SEQUENTIAL` (on top of the default policy). We apply these options to files used by SCAN requests, in order to inform the kernel that we plan to read the files sequentially or only once (`SEQUENTIAL` and `NOREUSE`) or that we no longer need the folios after their use (`DONTNEED`). As shown in Figure 3.8, `cache_ext`’s application-informed policy achieves 70% higher throughput and 57% lower P99 latency for GETs, while SCANS experience an 18% throughput decrease. In addition, the `fsync()` options do not help much, demonstrating the inadequacy of existing kernel page cache interfaces compared to `cache_ext`. MGLRU performs even worse than the default LRU.

**Takeaway 3:** Even very simple application-aware eviction policies can significantly improve performance.

**Takeaway 4:** Existing Linux page cache customization interfaces are ineffective.

### Application-Informed Admission Filter

We run our admission policy that filters folios fetched by background compaction (§3.5.6) on RocksDB [25] with a uniform R/W workload. This simple admission filter improves P99 latency by 17% (from 2.61ms to 2.16ms), because folios required for read requests are not evicted by folios accessed by the compaction threads. We do not see a meaningful difference in throughput, which is expected, as compaction is infrequent.

### File Search

We construct a file search workload that searches the Linux kernel codebase (v6.6), using the multi-threaded *ripgrep* CLI tool [136]. More specifically, we perform 10 searches within a 1GiB cgroup, which is roughly 70% the size of the codebase (excluding Git history). File search would be a particularly challenging workload for Linux’s LRU-based policies, since it involves a large amount of scans. We compare the `cache_ext` MRU policy with the default Linux kernel policy as well as the kernel’s experimental MGLRU policy. The results in Figure 3.9 show that `cache_ext` is almost 2× faster than both baseline and MGLRU, since both policies suffer from the scan “pathology” of LRU.

### Implementation Complexity

Table 3.3 shows the lines of eBPF and userspace loader code necessary to implement each of the aforementioned policies. The policies are all implemented in at most a few hundred lines of code, a much smaller amount than would be necessary to implement them within the kernel (or in userspace). We find that `cache_ext` reduces the complexity of developing new policies by using its list and policy function abstractions. In addition, developer experience and velocity are

Policy	eBPF LoC	Userspace LoC
Admission filter	35	262
FIFO	56	118
MRU	101	87
LFU	221	107
S3-FIFO	287	139
GET-SCAN	324	107
LHD	366	152
MGLRU	570	90

Table 3.3: Lines of eBPF and userspace loader code in `cache_ext` policies.



Figure 3.9: File search workload results (MRU policy).

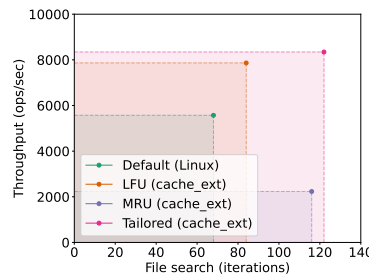


Figure 3.10: Using `cache_ext`, multiple applications can run different eviction policies, yielding better performance for all.

greatly improved, since eBPF prevents kernel crashes and many types of bugs, enabling developers to focus on the policy logic. Thus, `cache_ext` allows developers to accelerate their applications with a relatively modest amount of effort.

Additionally, we plan to open source all of our policies, allowing developers to easily try them with their applications, lowering the barrier to entry for using `cache_ext`.

### 3.6.2 Isolation (Q2)

The Linux page cache already provides a measure of isolation by establishing per-cgroup LRU lists. `cache_ext` utilizes this to enable each cgroup to have its own custom policy. We demonstrate the utility of this by simulating and comparing against “global” policies, as opposed to

`cache_ext`'s per-cgroup policies. We create two cgroups, one running a YCSB C workload with LevelDB, and the other running a file search workload with `ripgrep`. The YCSB cgroup is allocated 10GiB and the file search cgroup is allocated 1GiB. We run four configurations: both cgroups using the default policy, both using LFU, both using MRU, and a “tailored” setup: YCSB with LFU and file search with MRU.

**Evaluation.** Figure 3.10 shows that the tailored setup beats the other configurations, yielding 49.8% and 79.4% improvements for YCSB and file search, respectively, over the baseline. While the other two `cache_ext` configurations provide performance improvements for the workloads corresponding to their policy, they can significantly degrade the performance of the other workload, demonstrating that global policies are not a viable solution. Note that YCSB improves further in the tailored setup compared to the LFU configuration (and vice versa for file search compared to the MRU configuration). This is due to improved caching of the workloads yielding reduced disk contention. The file search workload improves in the LFU configuration for the same reason.

**Takeaway 5:** Using `cache_ext` with per-cgroup policies allows for fine-grained control and improved performance.

### 3.6.3 Memory and CPU Overhead (Q3)

The advent of faster and larger storage devices means that the page cache (and `cache_ext`) must be able to handle millions of events per second. We run a number of micro-benchmarks to investigate `cache_ext`'s memory and CPU overhead.

#### Memory Overhead

`cache_ext`'s primary memory usage is the valid folios registry hash table (§3.4.4). In the worst case, we set up the hash table with as many buckets as there are 4KiB pages in the cgroup (based on its configured size). Each bucket requires 16 bytes to store the hash table's internal list pointers. Thus, the memory overhead for an empty registry is:  $\frac{16}{4096} = 0.4\%$ . Following the same logic, each filled entry in the hash table uses 32 more bytes for the `cache_ext` list node, so the

Cgroup Size	Linux default	cache_ext no-op	Overhead (%)
5 GiB	234.80	236.51	0.72%
10 GiB	217.48	221.14	1.66%
30 GiB	197.67	198.01	0.17%

Table 3.4: cache\_ext  $\mu$ CPU per I/O operation using fio.

full registry memory overhead is 1.2%. Therefore, the memory overhead for cache\_ext’s registry is between 0.4%-1.2% of a policy’s cgroup’s memory. This overhead can be further reduced with recent improvements to eBPF’s handling of kernel objects, allowing eBPF to directly ensure that some pointers are trusted.

### CPU Overhead

To measure the baseline CPU overhead of our framework, we use a no-op cache\_ext policy and run the fio microbenchmark [150] with 8 threads on a *randread* workload. A no-op policy defers to the default kernel eviction policy while still maintaining cache\_ext data structures, allowing us to measure the baseline CPU overhead imposed by cache\_ext. We compare to the default Linux policy, using a metric of CPU usage per I/O operation (measured in  $\mu$ CPUs, i.e. one-millionth of a CPU). Table 3.4 shows that the CPU overhead of no-op cache\_ext is at most 1.7%.

### Overall Application Impact

To estimate the overall impact of cache\_ext to application performance, we re-implemented the kernel’s MGLRU policy, as described in §3.5.3. We compare the throughput achieved by the Linux MGLRU implementation with the cache\_ext implementation. Table 3.5 shows the relative performance for each YCSB benchmark (shown in §3.6.1), calculated as cache\_ext throughput over baseline MGLRU throughput. The two implementations perform very similarly, with an average 1% throughput decrease.

<b>Takeaway 6:</b> cache_ext incurs relatively low overhead.
--

YCSB A	B	C	D	E	F	Uniform	Uniform R/W
0.97	0.99	0.96	0.98	0.98	0.99	1.06	1.05

Table 3.5: Relative performance of `cache_ext` vs. baseline MGLRU with YCSB. The harmonic mean is 0.99, indicating a 1% slowdown on average.

### 3.7 Conclusions

This work explores the design of a new eBPF framework to implement custom eviction policies in the kernel, enabling applications to choose a policy according to their needs and making the latest caching research accessible to the kernel. We believe our work opens the door to explore new dynamic page cache policies, such as exploring ML-based or more sophisticated application-informed policies. Furthermore, recent efforts in the Linux community to support more complex eBPF data structures could benefit `cache_ext`.

## **Chapter 4: xPU-Shark: A new way to analyze uarch-scale performance bottlenecks for ML accelerators**

### **4.1 Introduction**

As generative artificial intelligence (AI), is projected to become a trillion dollar market by 2032 [151], an increasing number of companies invest in developing ML accelerators. In addition to the established chip designers, such as NVIDIA [152], AMD [153] and Intel [154], hyperscalers like Google [155, 156], Meta [157], and Amazon [158, 159], startups like Cerebras [160], and academic researchers [161, 162, 163] have also developed custom ML chips.

A complex ecosystem of tools have been built around these accelerators to support ML development. High-level frameworks like TensorFlow [164], JAX [165], and PyTorch [166] allow engineers to express machine learning models with simple APIs. These high-level models are then translated into ML intermediate representations like MLIR [167] or OpenXLA StableHLO [168]. These mid-level representations provide a layer of indirection between high-level ML frameworks and machine-level code, allowing compilers to target custom hardware from a reduced set of intermediate representations. These portable mid-level representations are then compiled into the byte-code which runs on the ML accelerator. The development of each of these levels of abstraction requires a huge engineering effort, and inefficiencies introduced at any level can cause performance degradation for the model. The companies that offer generative AI services are often doing so at a massive scale (for example, the infrastructure to provide inference for Microsoft's Bing AI chatbot is estimated to cost \$4 billion [169]), meaning that even a small degradation in performance can lead to large capital losses. Some companies such as DeepSeek are even implementing certain features directly in machine-code [170], showcasing the importance of low-level optimizations.

Because of the utmost importance of model performance, ML engineers need robust profiling and optimization tools in order to squeeze the maximum performance out of accelerator hardware. However, existing performance profiling tools fall short in analyzing low-level performance. Many tools (e.g., Nvidia NSight Systems [171], Tensorboard [37]) provide coarse-grained metrics on the high-level operations (HLOs) of the intermediate representation by leveraging the accelerator’s performance monitoring unit (PMU). This is useful for revealing inefficient HLOs, but offers little visibility into the interaction between the code and hardware. Moreover, the granularity of metrics provided by the PMU is constrained by its buffer size.

Other tools focus on finding places in the code that cause hardware stalls by sampling program counters (e.g., Nvidia CUPTI [35], Intel VTune [35]). However, the root cause for a stall can be hard to connect to a specific stalling instruction, and thus knowing where the stalls happen offers little insight on how to fix them. In addition, stall PC sampling requires hardware support which is not always available. Another category of tools (e.g., NVBit [42], CUDAAdvisor [48], ValueExpert [47]) are based on binary instrumentation, which records information about every low-level instruction executed in the accelerator. While this information enables fine-grained analysis of the software, it changes the hardware utilization characteristics at runtime. Finally, instrumentation typically requires re-compilation, which makes it hard to apply in a production setting where models and code are constantly changing.

To fill this gap, we present `xPU-Shark` (`xPU-Shark`), a novel methodology to analyze the microarchitectural efficiency of ML accelerators in the context of a hyperscalar datacenter. As shown in Figure 4.1, `xPU-Shark` enables a “record and replay” style of profiling by recognizing that a common artifact of the accelerator design process, a Golden Reference Model (GRM), can be repurposed as an Instruction Set Architecture (ISA) level simulator. This methodology allows the capture of fine-grained details for both the software (e.g., instruction dependencies, memory accesses) and the hardware (e.g., compute units utilization, memory stalls). `xPU-Shark` first uses a step debugger to capture traces from production deployments of our in-house ML models, then replays them in a modified ISA-level simulator. We then use the data produced from the

ISA-simulator to build a series of performance analyses, and automatically suggest performance optimizations.

In this paper, we describe how we used `xPU-Shark` to optimize our large language models (LLMs). First, we describe how `xPU-Shark` can help identify and visualize inefficient memory transfers (DMAs). Second, we use `xPU-Shark` to analyze the utilization and fragmentation of our accelerator’s data-cache and provide fine-grained instruction-by-instruction utilization information for various compute units of our accelerators, enabling our engineers to reason about how their code performs at the machine-code level. Third, we use `xPU-Shark` to analyze the dependencies of each instruction to automatically suggest optimizations via instruction reordering. Although our in-house LLMs are already highly optimized, `xPU-Shark` revealed several previously unknown inefficiencies that amounted to up to 4.1% of token generation time.

In summary, we make the following contributions:

- We introduce `xPU-Shark`, a new performance analysis framework that leverages microarchitectural simulation to provide performance insights. To the best of our knowledge, `xPU-Shark` is the first work that extends a microarchitecture simulator to tune performance on a *specific* accelerator architecture.
- We implement `xPU-Shark` for our in-house accelerator and construct several performance analyses based on the data we collect with it, which are very hard or impossible to implement with existing tools. `xPU-Shark` revealed several inefficiencies in our LLMs, even though they have been thoroughly optimized already.
- Using `xPU-Shark`’s suggestions, we optimize a common communication collective (All-Gather) by 15% and decrease the generation time of our LLMs by up to 4.1%. These LLMs are deployed at huge scale internally and each percent improvement in model serving leads to significant savings in total cost.

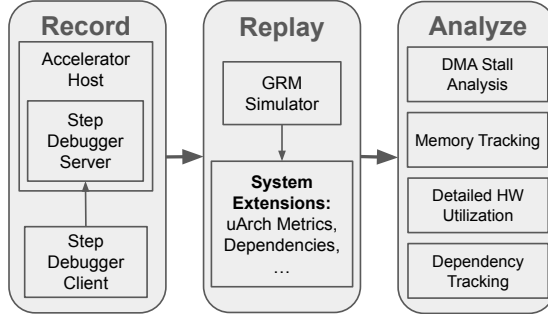


Figure 4.1: Overview of xPU-Shark.

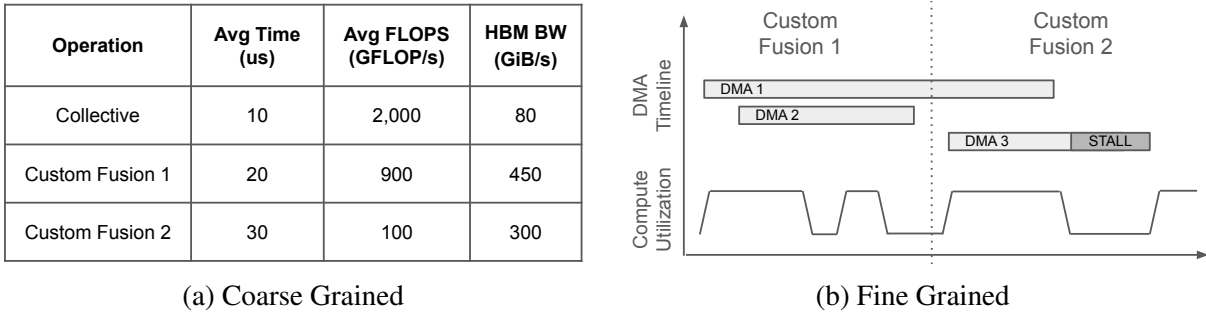


Figure 4.2: A toy example showing a coarse-grained vs fine-grained analysis. Many existing tools provide coarse-grained information at the kernel or HLO level, while deep optimization requires a fine-grained view.

## 4.2 Background and Related Work

In this section, we provide a brief overview of the ML software development landscape (§4.2.1). We then discuss the growing need for optimizing accelerator performance and how profilers are a crucial tool in doing so (§4.2.2).

### 4.2.1 ML Software Stack

High-level frameworks like TensorFlow [164], JAX [165] and PyTorch [166] provide simple APIs for constructing ML models. High-level models are then translated into intermediate representations like MLIR [167] and OpenXLA StableHLO [172]. The intermediate representation is usually a computation graph consisting of higher level operations (HLOs), like “matmul” and “transpose”. These mid-level representations provide a layer of indirection between the ML frameworks and ML compilers [168, 173, 174, 175]. The compiler then uses these representations to

generate accelerator code (e.g., CUDA kernels, PTX, SASS) in a process known as *lowering*.

Given the very high costs of deploying and operating AI infrastructure, there is a lot of work on squeezing more performance out of existing accelerators. Performance experts focus on constructing faster kernels through careful orchestration of the hardware and especially memory, via techniques like FlashAttention and PagedAttention [176, 177, 178, 179]. Compiler engineers invent new frameworks that expose greater control of the hardware, like Triton [180] and Pallas [181]. And at a cluster level, resource managers like Pathways [182] take ML workload characteristics into account when making global scheduling decisions.

#### 4.2.2 Profilers

To allow developers to debug and optimize their workloads, vendors and academics have developed a number of performance profiling tools.

**Motivating Example.** In Figure 4.2 we show a simplified example of an inefficiency which would be hard to detect with current profiling tools. Existing tools present performance information in a coarse-grained format, usually listing an operation (the HLO or kernel name, like “Custom Fusion 1”) and any aggregated statistics of interest (like runtime, FLOPS, or utilized memory bandwidth). These tools show which operations have low performance, but do not provide the reasons for low performance or clues on how to increase performance. In contrast, a low-level optimization tool like `xPU-Shark` can offer deeper insight into each microarchitectural component of the accelerator. In our example, the course grained analysis shows that the second custom fusion HLO has lower than expected average FLOPS. We may also be able to see that hardware bandwidth utilization can be improved which gives us some clue that the memory subsystem is responsible for the low performance. The fine grained analysis gives the information we need to fully analyze and fix the issue with our custom HLO. In Figure 4.2b we can see each memory transfer along with detailed compute utilization metrics. We can see that the compute utilization is low because the hardware is stalling when issuing DMA 3. We can also see that to fix the issue, we need to issue the DMA earlier, preferably during or before the first custom fusion HLO (cross-HLO opti-

mization). Finding opportunities for optimization across HLO's is especially difficult with coarse grained tools, as it is not clear how different operations interact from aggregated statistics. Here we showed a simple example of how a fine grained view into the microarchitectural utilization of the accelerator can make performance analysis much easier, and we now give an overview of existing profilers and their capabilities.

**PMU.** Vendors typically provide hardware support to record performance events, the performance monitoring unit (PMU), along with profilers that leverage it. The PMU has a number of registers (performance counters) which can record performance events such as cache hits, or collect statistics like cycles executed within a function. Profilers that use the PMU include Nsight Systems [33], nvprof [34], Intel VTune [35], AMD ROC-profiler [36] and Google Tensorboard [37]. More specifically, Nsight offers utilization metrics at the level of a CUDA kernel and Tensorboard at the level of a HLO. These tools are usually lightweight and rely on the PMU to provide performance metrics, which is constrained both in granularity, because of its limited buffer size, and in variety, as it typically allows recording a limited number of performance counters simultaneously [38].

**Program counter sampling.** Besides the PMU, some vendors (e.g., Nvidia) provide program counter (PC) sampling and stall attribution. PC sampling requires additional hardware support to sample the program counter during execution. If the code is stalling at the time of the sample, the hardware may also provide the reason for the stall (e.g., memory stall, synchronization stall). Tools like CUPTI [39], VTune [35], HPCToolkit [40] and DrGPU [41] use PC sampling to collect stack traces and their stall reasons, coalesce them and suggest optimization strategies. While this approach can be useful for finding opportunities to improve performance, it has a number of disadvantages. First, it requires additional hardware to sample running code and attribute stall reasons correctly, which is not available in all accelerators, including ours. Second, while PC sampling can pinpoint various inefficiencies like memory transfer stalls, it does not show the root cause instruction. For instance, a stalled instruction waiting for a memory transfer is the symptom, not the cause. Finally, it does not provide any information on how well the hardware is utilized, as it

focuses on stalls and not on hardware utilization.

**Instrumentation.** Another category of tools use binary instrumentation to gain performance insights on a more microscopic level, albeit with great overhead. Binary instrumentation engines such as Nvidia NVBit [42], SASSI [43], Sanitizer API [44], Intel GTPin [45] and LLVM [46] can change code at a very low level, so that every instruction can be recorded. This approach is used to analyze the behavior of the software at very high detail. For example, VALUEEXPECT [47] traces every load and store instruction to discover inefficient patterns in the data, like hidden sparsity or repeated computation. CUDAAdvisor [48] traces memory accesses to reveal memory metrics such as reuse distance. While these approaches are useful to explore the inner workings of a program running on an accelerator, they offer no insight on how well the program is using the underlying hardware, as instrumenting the program totally changes its execution characteristics which renders the PMU useless. In addition, instrumentation typically requires re-compilation, which makes it hard to use in the context of a hyperscalar where compilation is complex, while code and dependencies change frequently.

**Cross-cutting.** Last but not least, some tools combine multiple techniques to provide a more detailed performance analysis. GPA [52] combines PC sampling with instrumentation to detect inefficient parts of the code, analyze their dependencies and suggest root causes. It is a powerful optimization tool but does not provide specific optimization suggestions (i.e. move this instruction here). In addition, it relies on hardware support for PC sampling, which is not available in every accelerator, including ours. NVidia NSight Compute [33] is the most comprehensive tool we know, analyzing CUDA kernels down to the PTX level, detecting instruction dependencies, and warning of inefficiencies. However, it analyzes only one kernel at a time, misses Nvidia SASS-level insights, and remains opaque due to its proprietary nature.

### 4.3 Design Requirements

To get the best performance out of an accelerator, it is important to have good visibility of how the software interacts with the hardware, at a fine granularity. Based on these observations, we pose the design requirements for xPU-Shark as a series of questions that an ideal low-level profiler should answer.

**Q1 Are there optimization opportunities inside and across HLO boundaries?** Higher level performance analysis tools only present aggregate metrics for each HLO, even though data dependencies often span HLO boundaries. Can a detailed analysis below this abstraction unlock new opportunities?

**Q2 What is the instantaneous utilization of individual microarchitectural units?** ML workloads require heavy matrix and vector multiplications and manipulations which need to be carefully orchestrated to achieve optimal use of the accelerator. Aggregated statistics can hide opportunities to fully utilize the accelerator hardware.

**Q3 Is our data-cache properly utilized to alleviate memory bottlenecks?** Data-caches help alleviate the memory bottleneck, and bridge the gap between relatively slow memory and incredibly fast compute. Efficient use of the data-cache is perhaps the most crucial element of achieving maximum performance.

**Q4 Can the tool provide actionable insights?** Performance analyses and visualizations can help users better understand how the machine-code interacts with the hardware and spot inefficiencies. Existing tools can often point to general causes, but cannot suggest specific actions to fix them.

In addition to these questions, we face several challenges in the context of a hyperscaler:

**C1** Use **software, not hardware**. Techniques like PC sampling help in pinpointing inefficiencies, but they require hardware support that is not available in all accelerators, including ours. We want the solution to be implementable entirely in software, so that it can be applied immediately to our entire accelerator fleet.

**C2** **Avoid recompilation**. Many tools that analyze performance at the lowest level commonly need model code to be compiled with special flags that instrument the resulting program. However, large ML models often have complex compilation pipelines that are slow and hard to modify. In addition, code and dependencies are constantly changing, making it harder to accurately recompile production models.

#### 4.4 Design and Implementation

`xPU-Shark` consists of an *execution recorder*, a *replayer*, and an *analyzer* (Figure 4.1). The execution recorder (§4.4.1) uses the hardware step debugger to break in the middle of the model execution and record traces of the machine-instructions executing the ML model. The traces include the minimal architectural state and memory required to replay the execution from the midpoint of the model we started recording from. The replayer (§4.4.2) is a modified existing ISA-level simulator, which replays the execution trace and generates detailed raw microarchitectural metrics. Lastly, the analyzer (§4.4.3) takes the raw generated metrics and performs various analyses to pinpoint inefficiencies and provides a detailed view of the hardware utilization to the user.

`xPU-Shark` is entirely implementable in software, requiring only a step-debugger and a simulator, two pieces of software that are commonly co-developed with the accelerator (**C1**). Most importantly, `xPU-Shark` can be used to analyze any production model that already used in production without requiring special recompilation (**C2**). We have incorporated `xPU-Shark` in our performance analysis workflow, which we describe in §4.4.4.

The rest of this section will focus on the design and implementation of the `xPU-Shark` system, while the next section (§4.5) will dive deeper into the analyses developed with `xPU-Shark`

and how they answer the questions we posed.

#### 4.4.1 Execution Recorder

The first step for `xPU-Shark` is to capture the code running on the accelerator, along with any architectural state required to replay that code. It is necessary to record the execution trace directly from the accelerator, rather than taking the compiler output, as the code can contain conditional executions and loops, and the contents of memory and registers are unknown at compile time.

An alternative to instruction traces might be to capture targeted performance traces using an instrumentation engine. However, this would require recompilation ([C2]), which we want to avoid as it is both challenging in our environment, and we want to profile our models with their production compilation settings. Instead, we take advantage of a step debugger to capture these traces.

**Step debuggers.** Step debuggers are a common software tool that is typically developed along with any accelerator. A step debugger uses hardware support to set breakpoints in the machine-code, which will pause the execution once the breakpoint address is hit. Users can then explore the contents of the memory, registers, or execute the next instructions one at a time (*single stepping*). Because it is such a fundamental tool, all major vendors that we are aware of offer a step debugger for their ML accelerators (e.g., NVidia CUDA-gdb [183], Cerebras CSDB [184]).

We require the step debugger to provide three simple functions: a `step` function to execute instructions one-by-one, `read_memory` and `read_register` functions to read memory and registers respectively. To use the recorder, the user needs to specify a breakpoint at a location of interest and how many instructions to record. Once the breakpoint is hit, the recorder uses the `step` functionality of the debugger to execute and record instructions one-by-one. A sketch of the recorder’s logic is shown in Algorithm 3. We’ll now briefly describe the intuition behind it.

Each instruction takes zero or more inputs from input registers, and can write to zero or more output registers, or modify a memory region on the accelerator (e.g., in the case of a DMA). In addition, some instructions (like DMAs) read directly from device memory. So, to accurately replay an instruction, we need to know the contents of these input registers and memory regions.

To naively capture this information, we must first record the entire contents of all memory regions on the accelerator, as well as all registers that can be used as inputs to instructions. However, this can make the trace file very large. As an optimization, we recognize that we only need to store the contents of a register or memory region if those contents are *first* used as input (and not as output) by a traced instruction. This is important to avoid recording intermediate results, which will be reconstructed in the simulator and are unnecessary to capture. To avoid capturing this unnecessary information, the recorder maintains a set of each instruction’s output registers and memory region addresses, and then only saves each instruction’s input register contents or input memory region contents if those contents had not been modified by a previous instruction.

---

**Algorithm 3:** Execution Recorder Algorithm

---

```

1   $R \leftarrow \emptyset$  {Set of used registers}
2   $M \leftarrow \emptyset$  {Set of used memory regions}
3  for instruction_count = 1 to  $N$  do
4    Parse instruction input register IDs and memory region addresses.
5    Save READ_REGISTER( $r_i$ );  $\forall$  input register  $r_i \notin R$ 
6    Save READ_MEMORY( $m_i$ );  $\forall$  input memory region  $m_i \notin M$ 
7    Parse instruction output register IDs and memory region addresses.
8     $R \leftarrow R \cup r_o$ ;  $\forall r_o \in$  Output register IDs
9     $M \leftarrow M \cup m_o$ ;  $\forall m_o \in$  Output memory regions
10   Step instruction.
11 end for

```

---

#### 4.4.2 Replayer

The second step in the xPU-Shark methodology is to replay the captured trace of the model in a simulator and capture detailed metrics about the underlying microarchitecture of the accelerator. This is realized by the *replayer*, a component based on an existing ISA-level simulator for the accelerator.

xPU-Shark’s key insight is to reuse the already available hardware simulator of the accelerator for the purpose of performance analysis. Hardware simulators, often called Golden Reference Models (GRMs), are an artifact of the integrated circuit design process, where they help validate hardware design decisions before the final production [185, 186, 187]. These simulators imple-

ment the full instruction set in software, accurately modeling the architectural state inside the accelerator. Prior work has used simulators to try and evaluate Nvidia’s GPU performance [49, 50, 51, 188]. However, due to the proprietary nature of GPU architectures, these studies could only approximate the actual design and exhibit significant errors. In our case, as both the vendor and consumer of our in-house accelerator, we have access to a fully accurate simulator and can provide a new perspective in using it for performance analysis.

The `xPU-Shark` replayer repurposes this artifact of the design process for performance analysis, by extending its capabilities to capture metrics. We add a component to the GRM simulator, the *performance tracker* which tracks metrics of interest. The performance tracker is passed as a dependency of the simulation, and registers callback functions with the modelled architectural components. On performance events, such as a memory read, the GRM executes the callback to the performance tracker, which records the event. When loading a trace, we first modify the memory and register state in the simulator to match the starting condition in our trace, then resume the execution of the recorded instructions. The performance tracker collects performance event information as the trace runs, which is finally saved to a metrics file for further analysis.

#### 4.4.3 Analyzer

The third step in the `xPU-Shark` methodology is to analyze the captured events and metrics from the replayer. Our analysis framework is extensible and can examine any microarchitectural component which is modeled in our GRM Simulator. We briefly describe three sample analyses here, with full detail and results in our evaluation (§4.5).

The focus of our first analysis is the DMA subsystem. Memory bandwidth is one of the most precious accelerator resources and for this reason we want to understand how the model interacts with system memory at a fine-grained level (Q3). Using the replayer’s data, we construct a timeline view of all memory transfers along with their stalls. Users can leverage the information from this analysis to pinpoint inefficient DMAs. This analysis also reveals optimization opportunities across HLOs, as it is often hard to hide the DMA latency within a single HLO, but possible by

looking further (Q1).

Our second analysis focuses on analyzing the fine grained compute and memory utilization. Providing instruction-by-instruction utilization information can help developers reason about resource availability at every step of execution (Q2) and debug low utilization of the hardware.

Finally we add an extension to xPU-Shark to track instruction dependencies of the machine-code code in each trace. xPU-Shark can track all accesses to registers and memory locations in the simulator, capturing the data dependencies across instructions. We can then use this dependency information to automatically suggest which instructions can be issued earlier (Q4). We elaborate on how these analyses can be combined and provide valuable insights in §4.5.

#### 4.4.4 Workflow

Finally, we describe where xPU-Shark fits in the performance analysis workflow. This workflow is a result of our experience using xPU-Shark to analyze the performance of several ML models. It consists of 1) classic **profiling** to gather regions of interest, 2) **recording** a trace, 3) **replaying** the trace in the simulator, and lastly, 4) **analyzing and visualizing**. We first collect some initial profiling with our in house performance profiling tool, which has similar functionality to Tensorboard [37]. This gives us a coarse-grained overview of performance and where potential bottlenecks might be. We note the instruction addresses of any areas of interest to record with our execution recorder (§4.4.1). To record the execution trace, we launch the execution recorder on the accelerator host machine, which attaches to a local accelerator running the model and sets a breakpoint for the target instruction address. Once the breakpoint is hit, the recorder takes over and creates execution traces, which are then saved remotely. These traces are typically 100-600k instructions long, which corresponds to a couple of model layers. These traces then serve as the input to the replayer (§4.4.2), which replays the trace and outputs performance metrics. Finally, the raw metrics are fed into the analyzer to perform the various performance analyses.

Note that so far we have been recording models in a sandbox environment, as setting a breakpoint in a customer-facing model would introduce unacceptable latency. However there is no

<b>Model</b>	<b>#Params</b>	<b>#Accel</b>	<b>Speedup</b>
LLM-Small	<10b	N	1.0%
LLM-Medium	10b-100b	4N	4.1%
LLM-Big	>100b	16N	0.14%

Table 4.1: Model descriptions. We evaluate three important in-house LLMs.

technical restriction that keeps us from attaching the recorder to an ML model running in production.

## 4.5 Evaluating Models With xPU-Shark

Armed with xPU-Shark, we can now find out the answers to our research questions (§4.3). First, xPU-Shark’s DMA analysis (§4.5.2) reveals that our models frequently incur unnecessary stalls waiting for memory transfers to complete, because of ineffective DMA scheduling. Second, xPU-Shark’s microarchitectural utilization analysis helps users debug issues that aggregate metrics overlook, such as identifying the root cause of low compute utilization (§4.5.3). Finally, xPU-Shark enables the user to reason about possible optimizations, by providing detailed information about the data-cache utilization (§4.5.3) and the dependencies of each instruction (§4.5.4). By assembling a complete view of the system, xPU-Shark is even able to automatically suggest optimizations via instruction re-orderings.

### 4.5.1 Experimental Setup

We use xPU-Shark to investigate inefficient operations on three of our most important in-house LLMs, shown in Table 4.1. Each model is compiled with our in-house compiler using production configurations and deployed in the same topology as on production machines.

### 4.5.2 DMA Stall Analysis

We illustrate xPU-Shark’s effectiveness by describing a set of optimizations found using xPU-Shark within the DMA subsystem of our accelerator. Memory size and bandwidth are

critical for modern ML accelerators, especially as larger models become increasingly memory-bound. To maximize memory utilization, we use `xPU-Shark` to analyze DMA performance and uncover inefficiencies (Q1, Q3).

**Background on DMAs** DMAs are a mechanism used by ML accelerators to transfer data between high-bandwidth memory (HBM) and the various caches. In our accelerator, DMAs are performed by an asynchronous DMA engine. They are controlled with two simple machine-code instructions as shown in Figure 4.3. `ISSUE` will start a memory transfer and `WAIT` will block execution (sometimes stalling) until the transfer completes. Figure 4.4 shows the lifetime of a DMA. Once issued, a delay occurs before the command reaches the DMA engine for processing and queuing. This delay, called base latency ( $T_b$ ), is constant and non-accumulative: when multiple DMAs are issued in parallel, their base latencies are fulfilled simultaneously. Once the base latency is fulfilled, the DMA waits until it reaches the start of the engine queue and starts transferring data between the various memories. This delay is called the transfer latency, or  $T_t$ , and depends on the available bandwidth of the memories involved, the contention with other DMAs and the size of the transfer. Data transfer on a single link (source-destination pair) cannot be parallelized. Given the above, we can now examine the three scenarios that can happen for a DMA, as shown in Figure 4.4. The first case is when the `WAIT` instruction comes before the base latency is fulfilled. In that case, the DMA incurs stalls first because of the base latency, pictured as green, and then because of the transfer latency, pictured as purple. The second case is when the `WAIT` instruction comes after the base latency is fulfilled and the incurred stalls are only because of the transfer latency. The third case is when the `WAIT` comes after the DMA has completed. In that case, we say the DMA has "slack", as the `WAIT` can be moved earlier. Slack is denoted in gray with a cross pattern.

The DMA subsystem is a common source of stalls because `ISSUE` and `WAIT` instructions can be difficult to properly schedule. In order to avoid incurring stalls, the compiler must insert enough instructions between each `ISSUE` and `WAIT`, so that it can hide the DMA's latency. This could be very difficult or impossible, especially within the boundaries of a single HLO. Transfer stalls are

```
ML Model Machine Code  
ISSUE <dma settings>  
<other instructions>  
WAIT <dma_id>  
<instruction using transferred  
memory>
```

Figure 4.3: Example machine-code instructions for handling memory transfers in an ML model. The **ISSUE** command starts the DMA, while the **WAIT** command blocks until it is complete and is typically inserted close to the command that needs the memory. The compiler can hide the DMA latency by inserting instructions between **ISSUE** and **WAIT**.

also difficult to predict since transfers may be predicated, exist across multiple chips, and share limited bandwidth resources. We differentiate between these two types of stalls in our analysis to help end users understand and remedy each scenario.

To construct the DMA analysis, `xPU-Shark` captures DMA **ISSUE** and DMA **WAIT** in the replayer as it replays the trace. Then, given these events and the specifications about the transfer speeds of various memories, `xPU-Shark` can simulate these transfers and flag which ones will stall the program. `xPU-Shark` then plots the transfers in a timeline plot, highlighting the stalled parts. The goal of this analysis is to visualize DMAs in an intuitive way and clearly show areas of potential improvements.

### Collective Operations Optimization

Collective operations (like All-Gather, All-To-All, etc.) are fundamental operations for distributing a model across accelerators [173, 189, 190]. In this section, we describe our experience using `xPU-Shark` to profile and optimize an important communication collective, All-Gather, reducing its runtime by 15%.

We began our analysis by using our existing in-house start-of-the-art profiling tool to look at one of our most used models (LLM-Small). We observed that during generation for this model, the All-Gather operation was 13.3% of total runtime. Our existing tool also reported that roughly 40% of All-Gather execution time was spent stalling on DMAs. Beyond this information, we did

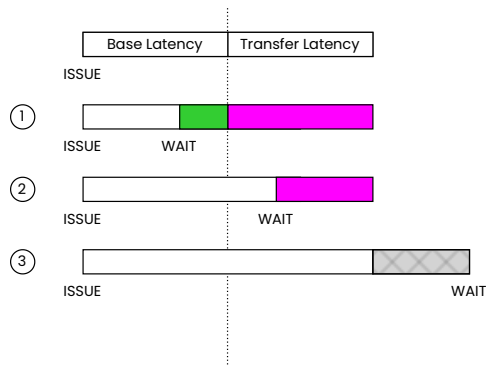


Figure 4.4: Lifetime of a DMA. A DMA is comprised of a base latency (constant) and a transfer latency (variable). DMAs begin with the ISSUE command. If the DMA has not completed by the time the WAIT command executes, the accelerator will stall until the DMA completes. We highlight three distinct scenarios. (1) WAIT comes before the base latency is fulfilled. The DMA incurs stalls first because of the base latency (green) and then because of the transfer latency (purple). (2) WAIT comes after the base latency but before the transfer latency is fulfilled. The DMA incurs stalls only because of the transfer latency (purple). (3) WAIT comes after the DMA finishes. The time between the DMA completion and the WAIT is slack (gray cross pattern).

not know what memory channels these stalls were coming from, what data was being transferred, or whether the stalls could be eliminated.

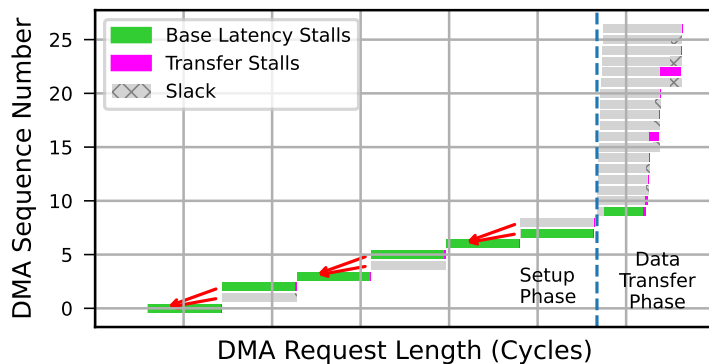


Figure 4.5: All-Gather DMA Pattern. Memory accesses are performed in two phases, the setup phase, and data transfer phase. Dependencies for the setup phase are shown with red arrows. These dependencies were manually discovered by reading the machine-code, a laborious process.

Using xPU-Shark, we capture a trace of 100,000 instructions for LLM-Small, and replay it on our modified ISA-simulator. The DMA stall analysis in Figure 4.5 shows most stalls happen in the first nine DMAs. These stalls are mainly due to base latency (shown in green), not transfer latency (shown in purple), as the data transferred is small. The transfers go from HBM to the data-cache.

We identify two optimization strategies. The first strategy is to issue these DMAs in parallel, if possible, since base latencies can be parallelized in our DMA engine. Manual dependency analysis of the machine-code shows the DMAs form three groups. Each group has an initial DMA, followed by two dependent DMAs. Issuing the three initial DMAs in parallel, then the remaining six in parallel, would reduce stalls by a factor of three. The second strategy requires more knowledge about how the All-Gather is implemented. The first nine DMAs are part of the operation setup phase, in which the addresses of other nodes in the topology are loaded. These addresses are then used as destinations for the DMAs in the subsequent data transfer phase. Since these addresses typically use a small amount of memory, especially with smaller models, our insight is that they could be permanently pinned in the data-cache. This would completely remove the need for the setup phase DMAs, eliminating the stalls they incur. However in larger models, where topology sizes are larger, pinning the node addresses in memory may consume too much data-cache memory.

After concluding our analysis, we worked with internal developers to optimize the collective. Our internal compiler team opted to implement the second approach (pinning the node addresses in memory), because of its lower implementation complexity. The optimized code reduced the runtime of the All-Gather operation by 15%, as shown in Figure 4.6. In terms of end-to-end runtime, it improved generation latency by 4% in LLM-Medium and 1% in LLM-Small, as shown in Table 4.1. LLM-Big's improvement was a smaller 0.14%, highlighting the trade-off of the second approach. We are in the process of implementing our first approach (parallelizing setup phase DMAs), which is expected to perform better on larger models.

In conclusion, we used `xPU-Shark` to diagnose and optimize several important collectives operations in our in-house LLMs. Our existing state-of-the-art internal tools could only tell us that these operations were stalling, but could not provide the actionable insights that were obvious with `xPU-Shark`. Because these collectives operations are so crucial to LLM performance, our optimization lead to a realized improvement in end-to-end performance for our models. Every percent improvement in model serving leads to significant savings in total cost to run our models.

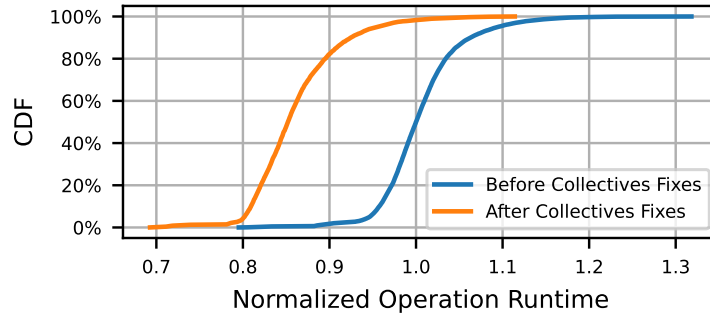


Figure 4.6: Reduction in runtime for the All-Gather collective operation after eliminating unnecessary DMAs. Runtime is normalized to the median value before optimization.

### 4.5.3 Detailed Microarchitectural Utilization

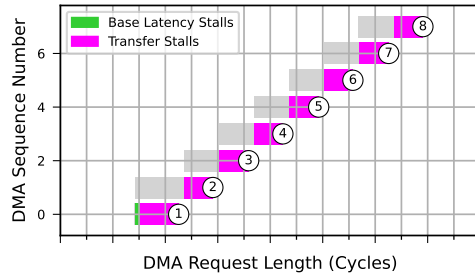
ML accelerators include an assortment of compute units (e.g., matrix unit, vector unit, scalar unit) and memories (e.g., HBM, data-cache). Underutilization in any of these units is an opportunity for further optimizations. Existing tools like Google’s Tensorboard provide coarse-grained metrics about higher level functions. While an aggregated utilization is useful as a measure of performance, it does not give developers a full picture of how the accelerator is used or, more importantly, what could be changed. In this section, we show how `xPU-Shark`’s detailed hardware utilization analysis enabled us to improve the runtime of a compute-heavy operation by 70%.

#### Compute Unit Utilization

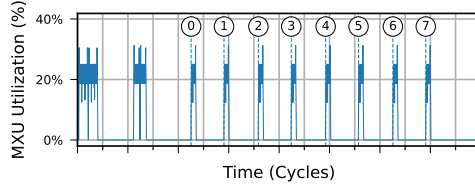
To aid users in maximizing utilization of the compute units in an accelerator, we would like to provide them with a view of how the model interacts with the hardware at very high detail. Thanks to its use of a hardware simulator, `xPU-Shark` can reveal how well the model utilizes the compute units at single cycle granularity. To understand how this can assist users, we provide an example of debugging low utilization of the matrix multiplication unit (MU) in a real model<sup>1</sup>.

Figure 4.7 shows `xPU-Shark`’s analysis for a model section that exhibits low compute utilization. Our high level tool simply reports a low compute unit utilization, making it hard to dig deeper into the root cause. In contrast, `xPU-Shark`’s analysis directly points out the problem.

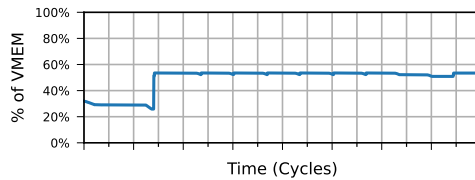
<sup>1</sup>Examples of matrix multiplication units include the MXU for Google TPUs and the TensorCore for Nvidia GPUs.



(a) DMA Sequence.



(b) Matrix Multiplication Utilization

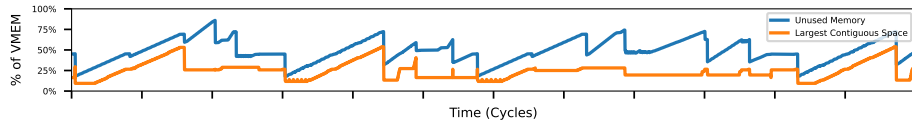


(c) Data-cache Utilization

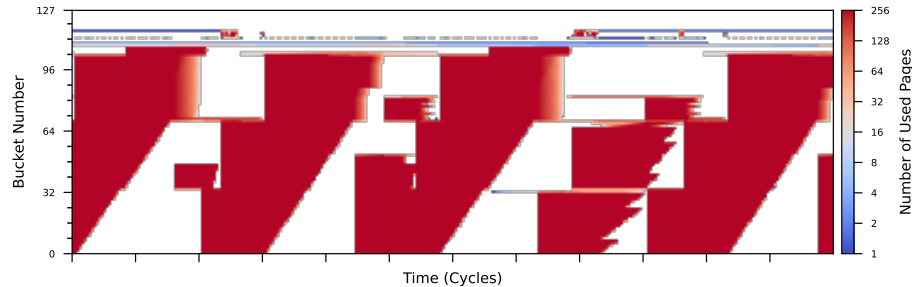
Figure 4.7: xPU-Shark can make it easy to diagnose low matrix multiplication utilization. We annotate the completion of each DMA, and the corresponding cycle on the MU utilization (Figure 4.7b). We see that the matrix multiplier is waiting on data transfers, resulting in low utilization. At the same time, the data-cache has enough is not fully utilized, so we might be able to prefetch more data.

First, Figure 4.7b shows the utilization of the matrix multiplication unit, the main workhorse of the accelerator. We see an intermittent pattern of computation and data transfer, indicating that the problem is caused by some other component bottlenecking the flow to the MU. We can immediately spot the problem by using xPU-Shark to look at the memory subsystem (Figure 4.7a). The high utilization coincide with DMA completions and the zero utilization spots coincide with the DMA stalls. The current DMA schedule is not feeding data to the matrix unit fast enough.

We see how an issue that was opaque before now becomes clear to the user, who can reason about optimizing it. A straightforward path from this finding would be to prefetch more aggressively before the chain of matrix operations begins. However, this requires reasoning about the



(a) Total Occupancy



(b) Fragmentation Heatmap

Figure 4.8: Figure 4.8a shows the percentage of data-cache space which is unused, and the percentage which is unused and contiguous. The largest contiguous region is consistently much smaller than the total unused portion. Figure 4.8b is the data-cache fragmentation analysis. The total memory space is divided into pages, where each read and write is tracked. Pages are then grouped into “buckets” of 256 pages. Each bucket is a horizontal line in the plot and its color denotes the number of used pages in the bucket for each cycle. We see considerable fragmentation of our data-cache, which is critical for performance.

availability of data-cache at that point in time. Because `xPU-Shark` also allows us to track this information, we can investigate data-cache utilization (plotted in Figure 4.7c), and we see that there is sufficient room in our data-cache to prefetch our data. Finally, by manually analyzing the machine-code code, we conclude that these DMA dependencies allow them to be issued earlier. With this information, we consulted with our internal compiler team to implement the prefetching, lowering operation runtime by 70% and increasing MU utilization.

## Memory and Data-Cache Utilization

In our previous example, we showed how examining the overall utilization of the data-cache can help users make decisions about data prefetching. However, `xPU-Shark` can analyze the data-cache at a much finer grained level, enabling developers to not only look at overall utilization but also cycle by cycle memory usage and fragmentation.

**Background on the data-cache** The data-cache is a relatively small, fast cache that stores vectors that can quickly be loaded into the matrix multiplication units. The data-cache bridges the slow HBM with the fast compute capability of the accelerator, making it crucial to model performance. The data-cache and memory allocations are static and managed by the compiler during the compilation process. However, because of predication and loops, it is hard to reason about the exact data-cache use at each point of the program. Moreover, a challenge with these memory allocation techniques is fragmentation. It is not enough to have a large on-chip memory, there also must be a contiguous region available as a target for memory transfer.

By using a hardware simulator, `xPU-Shark` can precisely track usage of the data-cache by tracking reads and writes on data-cache pages. A page is counted as “used” if it is read by an instruction after being written. If a page is brought in from HBM to the data-cache and the page is not read in subsequent instructions then that page is tracked as “unused”. We are only able to track these categories as long as our simulation runs, so theoretically if a page is read after the simulation ends, our analysis would miss that read. However, our simulations typically run for hundreds of thousands of cycles. A memory transfer should be read within that period, and if a page is transferred but not used for tens or hundreds of thousands of cycles, that is an inefficiency in the model.

The data-cache fragmentation analysis is shown for a production LLM in Figure 4.8. The total data-cache space is divided into 128 “blocks”, each of which contain 256 pages. The heatmap information shows how many pages were used within each block. In Figure 4.8a, we show both the percentage of unused memory, and the percentage of memory occupied by the largest contiguous region. The total portion of unused memory varies between approximately 20% and 80%, however the largest contiguous unused space is consistently less than this. In this trace, the median unused space is 47% of the total space, and the median contiguous block is only 25% of the total space. These results show that the data-cache space can be used more efficiently.

Beyond model developers, this analysis provides compiler engineers with a unique insight into data-cache utilization at the lowest level. Memory allocation is done by the compiler and relies

on finely-tuned heuristics. Compiler engineers often evaluate memory allocation techniques on models that quickly become outdated and don't have a way to gain insight into the way these heuristics are affecting the latest production models. `xPU-Shark` offers a new way of debugging these complex compiler subsystems that wasn't possible before.

#### 4.5.4 Dependency Analysis

In the previous examples, we demonstrated how `xPU-Shark`'s fine-grained analysis can identify inefficiencies. However, it's difficult to differentiate between poor performance which is actually unavoidable due to dependencies, and poor performance which can be eliminated through alternative schedules. So far, we tackled the issue by manually going through the machine-code and discovering dependencies, which is a labor intensive and error prone process. Automating the discovery of these dependencies would not only alleviate the need for users to manually analyze complex machine-code code but also enable `xPU-Shark` to autonomously propose optimizations through alternative instruction scheduling. In this section, we present how `xPU-Shark` implements dependency tracking and leverages this information to enhance the effectiveness of our existing analyses.

There are two ways to do dependency analysis: static and dynamic. While static analysis can be done by the compiler, a static implementation will be missing key information about runtime behavior, such as predication, which limits its usefulness. `xPU-Shark` implements dynamic dependency analysis, using the GRM simulator to discover instruction dependencies as it replays the trace. Instruction dependencies can be registers (e.g., for arithmetic instructions), data-cache locations (e.g., for load/store instructions) or even HBM locations (e.g., for DMA instructions). To discover instruction dependencies, `xPU-Shark` traces all accesses to registers, data-cache and HBM in the simulator. When an instruction reads from a register or piece of memory that was previously written by another instruction, then those instructions have a dependency. With this information, `xPU-Shark` knows the exact dependency graph between instructions.

One use case for this analysis is to reason about rescheduling DMAs. The way DMAs are

typically issued is to load some information about the DMA metadata from HBM. This metadata is put in registers and may be transformed before being used as input to the DMA instruction. These transformations are typically lightweight and they occur close to the DMA issue instruction. If we only track when a DMA instruction’s immediate input registers as dependencies, then in many cases the dependencies will be flagged as “fulfilled” immediately preceding the DMA. However, our insight is that as long as any transformations on the DMA metadata are lightweight, we can move the DMA ISSUE and any transformation instructions as early as when the data is ready in the data-cache. We call this method of dependency accounting “relaxed”, illustrated in Figure 4.9. In the conservative model, an instruction depends on its input registers. In the relaxed model, an instruction depends on the DMA that brought its inputs, propagated through dependencies, to memory. For DMA instructions, it is better to use the relaxed model to reason about their dependencies.

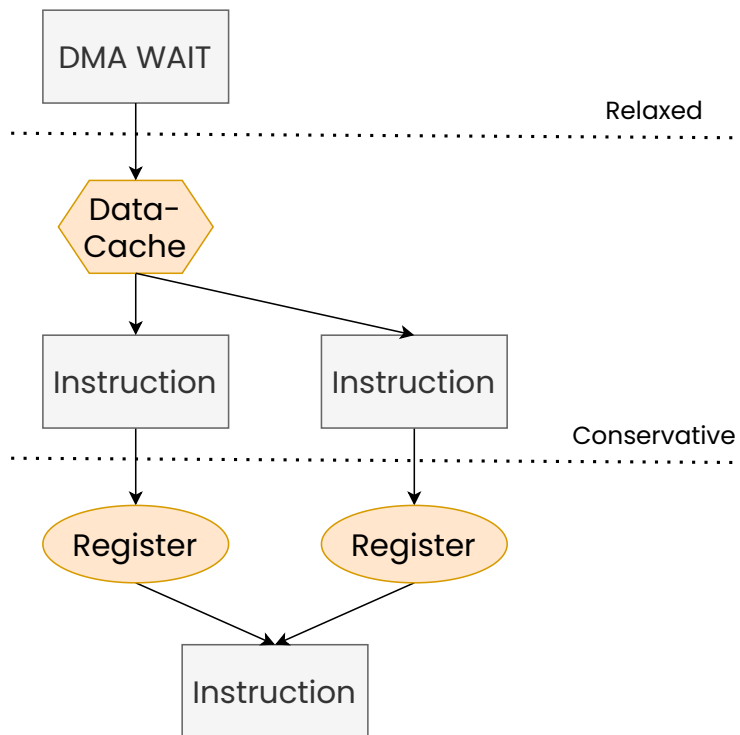


Figure 4.9: Different models of dependency tracking. In the conservative models, an instruction depends on the immediate instructions that shape its input. In the relaxed model, dependencies are propagated until the dependent memory is in the data-cache.

Using the generated dependency graph, xPU-Shark can display this information to the user,

allowing them to reason about alternative schedules that eliminate stalls by reordering instructions. We illustrate the power of `xPU-Shark`'s dependency tracking by applying it to the example shown above, where we had to manually discover dependencies by reading machine-code code. Figure 4.10 shows the DMA analysis with dependency information overlaid. Dependencies are visualized as “backtails” for each DMA, indicating how far back we can start it based on the relaxed model. The user can see that all DMAs can be issued earlier in time and thus prefetching is a viable optimization strategy.

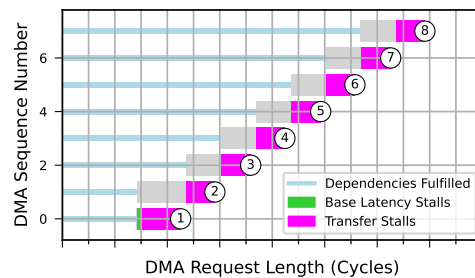


Figure 4.10: Example dependency analysis with `xPU-Shark`, showing the DMA analysis of section 4.5.3 with dependency information overlaid as “backtails”. Thanks to that, the user can immediately conclude the DMAs can be issued earlier, without needing to manually analyze the machine-code.

**Autonomous Optimization Suggestions** By combining the various analyses presented so far, `xPU-Shark` can reason about inefficient DMAs, check if alternative schedules are possible and flag them for further investigation. An outline of this logic is shown in Algorithm 4. We examine each DMA separately. If the DMA does not have stalls, we have nothing to suggest. Otherwise, we first check how far we can push the DMA based on its dependencies. Specifically, we would like to push it back at least as many instructions as stall cycles, to eliminate those stalls. Second, we check if the data-cache has enough contiguous memory available to accommodate the DMA, using the analysis from Section 4.5.3. If these two conditions are true, then we suggest a DMA reordering to eliminate stalls. This algorithm successfully discovers and suggests the optimizations from the two examples we showed.

In conclusion, dependency analysis enables `xPU-Shark` to both aid the user in reasoning

---

**Algorithm 4:** Automatic optimization suggestion algorithm

---

```
1: for each DMA operation do
2:   if not stalled then
3:     continue
4:   end if
5:   push_limit  $\leftarrow$  DMA dependency distance
6:   memory_available  $\leftarrow$  (contiguous data-cache  $\geq$  dma size)
7:   if push_limit > stall_duration and memory_available then
8:     Suggest reordering
9:   end if
10: end for
```

---

about what optimizations are possible, as well as automatically suggest them.

#### 4.5.5 xPU-Shark's Overhead

Since xPU-Shark traces every instruction in an ML model and replays it in a microarchitectural simulator, the overhead is high. Thus, a key concern is whether xPU-Shark captures enough of the model's execution or zooms in too narrowly, missing context.

Currently, xPU-Shark records 400 us of model execution, covering multiple layers. The snapshot size is about 100MB. The end-to-end process takes around 2 minutes, but can be easily optimized to under a minute. Given the model's self-similarity, capturing a few layers across different modes (i.e. prefill, decode) provides sufficient performance insights. Overall, xPU-Shark's overhead is manageable, with potential for further optimization.

## Chapter 5: Rosé: Flexible Replication With Strong Semantics For Partitioned Databases

### 5.1 Introduction

Replication is a ubiquitous technique used by database systems to improve durability and availability, as well as read latency by creating copies of the data geographically closer to the users. Asynchronous primary-backup replication is a widely used form of replication [191, 70, 192, 193, 194, 195] in which a designated primary replica-set executes and commits all transactions locally, and asynchronously sends the transaction writes to a set of backup replicas which then apply the writes to independently reconstruct the primary's state. Since writes need only be acknowledged by the primary (which is typically contained within a single zone or region), this technique achieves excellent write latency, but is prone to losing recent data if the primary replica fails (e.g. due to a datacenter or region-wide outage). The wide adoption of this technique in practice shows that many users and applications find that trade-off acceptable.

A desirable property in primary-backup replicated systems is *monotonic prefix consistency* [191], where each backup replica exposes a progressing sequence of the primary's recent states. This ensures that if failover has to happen, the backup replica that is promoted to become the new primary is going to be in a consistent state. In other words, while durability can be compromised during fail-over, all the other ACID properties would still be maintained, helping preserve application invariants. It also means the backups are able to serve consistent (albeit potentially stale) snapshots to read-only transactions.

Monotonic prefix consistency is relatively straightforward to provide in unpartitioned database systems with a single log. However, in partitioned systems each partition has its own log and is typically replicated independently. Thus, even if the replication of each partition preserves

monotonic prefix consistency, the overall state of the backup can be inconsistent and undefined. Consider the following example to illustrate the issue: Suppose a committed transaction  $T$  wrote a key  $K_1$  in partition  $P_1$  and a key  $K_2$  in partition  $P_2$ . It is possible, due to differences in replication pace, that the write for  $K_1$  is replicated to the backup partition  $P_1$  while the write for  $K_2$  is not yet replicated to the backup partition  $P_2$ . In this case, an operation executing at the backup that reads  $K_1$  and  $K_2$  would only observe a part of  $T$ , breaking atomicity.

We know from experience and from speaking with many developer teams that dealing with fail-over during disaster recovery in such cases often requires performing complicated consistency checks and repairs on the database. Since the occurrence of such disasters is rare, regularly performing *drills* to exercise and validate the code and recovery processes is usually required. Fortunately, many modern systems that support global consistency across partitions also offer the ability to restore a cluster to a consistent point-in-time snapshot [196, 58] which simplifies this process. However, the process remains time consuming, and the systems offer no guarantees on how far behind the consistent point-in-time snapshot on the asynchronously replicated backup can be. In addition, failover often causes degraded performance on the newly promoted the primary, due to the need to clean up data newer than the latest replicated snapshot.

Our work in this paper addresses the limitation of asynchronous primary backup replication in partitioned databases by designing a novel replication scheme that preserves the desirable properties of monotonic prefix consistency, while enabling a flexible, fast and efficient fail-over process, and bounds the replication lag to the backup as long as it is up. First, we observe that many distributed database systems already have support for globally consistent, serializable snapshot reads. Since these snapshot reads can span multiple partitions, the system must already have a notion of a global snapshot, which can be based on real time [59], hybrid logical clocks (HLC) [58, 196] or epochs [197, 198]. These snapshots can serve as a natural extension of monotonic prefix consistency to partitioned databases: the (externally visible) state of the backup should always be a globally consistent snapshot of the primary. Second, we propose a backpressure-based mechanism that effectively caps replication lag while maintaining high availability. Third, we tackle the core

problem of degradation after failover by separating the replication of the write-ahead log (WAL) entries from their application in the partition’s key-value store. WAL entries are replicated liberally but only applied in a coordinated fashion, up until the latest fully-replicated snapshot. Finally, we integrate Rosé with Chablis and evaluate several aspects of the protocol.

## 5.2 Background and Related Work

In this section, we first provide a brief overview of replication techniques (§5.2.1) and how Rosé fits in this landscape. Then, we provide some minimal background of Chablis [198], in order to later describe its integration with Rosé (§5.2.2).

### 5.2.1 Replication

Distributed databases seek to support high-availability and durability, such that when some servers fail, data is not lost and the rest of the servers can continue providing the db functionality. These objectives are commonly satisfied using replication. Replication can be broadly categorized as (1) synchronous and (2) asynchronous.

Synchronous replication ensures strong consistency by requiring that transactions do not commit until their writes have been replicated across all designated replicas. This approach is well-suited for single-region deployments where fast network connectivity minimizes overheads, but becomes problematic for cross-regional replication where network delays significantly inflate transaction latency. Two primary approaches dominate the landscape: state machine replication and primary-backup protocols. State machine replication, exemplified by consensus protocols such as Paxos [53] and Raft [54], integrates consensus and replication into a unified protocol, offering very high availability guarantees at the cost of resource efficiency. In contrast, approaches like simple primary-backup, chain replication [55], GFS, CRAQ [56], and Hermes [57], rely on fixed replica sets with specific role assignments coordinated by an external consistent service, trading some availability for improved resource efficiency. Modern distributed databases such as CockroachDB [58] and Spanner [59] often employ synchronous replication across regions to support

cross-region durability and availability. However, this design choice imposes a substantial latency penalty on all transactions, even those whose write-sets exhibit strong regional locality. This high latency compounds into increased lock contention and higher abort rates, particularly for transactions involving frequently accessed keys.

### 5.2.2 Chablis

Chablis is a scalable, geo-distributed, multi-versioned transactional key-value store that supports low-latency read-write transactions within a region and globally consistent, strictly-serializable, lock-free snapshot reads. Its architecture is shown in Figure 5.1 and consists of the following components:

- **RangeServer:** Each RangeServer is responsible for managing specific ranges or partitions of the key space and implements both 2-phase-locking and 2-phase-commit protocols while remaining mostly stateless. The RangeServer persists transaction data in a write-ahead log (offered by a separate WAL Service) through prepare, commit, and abort operations. In addition, it asynchronously applies these operations to a Key-Value Service, which fast read access, and occasionally trims the WAL. The WAL Service and Key-Value Service can either operate as separate services or be co-located on the same node depending on the deployment configuration.
- **Warden:** The Warden component is responsible for assigning ranges to individual RangeServers and continuously monitors the health of RangeServers through a heartbeat mechanism to ensure system reliability and proper load distribution.
- **Epochs:** Chablis achieves transaction serialization through the use of epochs, which provide a global ordering mechanism between transactions such that transactions in epoch  $e_i$  are guaranteed to have occurred before transactions in epoch  $e_j$  when  $e_i < e_j$ . While this epoch-based approach enables Chablis to deliver superior performance and strict serializability without requiring specialized hardware such as atomic clocks, it also means that global

ordering is only well-defined at epoch boundaries. Transactions read the current epoch from regional epoch publishers during the 2-phase commit process, while a global epoch service periodically advances the epoch across all regions, thereby enabling fast regional writes and global strictly-serializable lock-free snapshot reads.

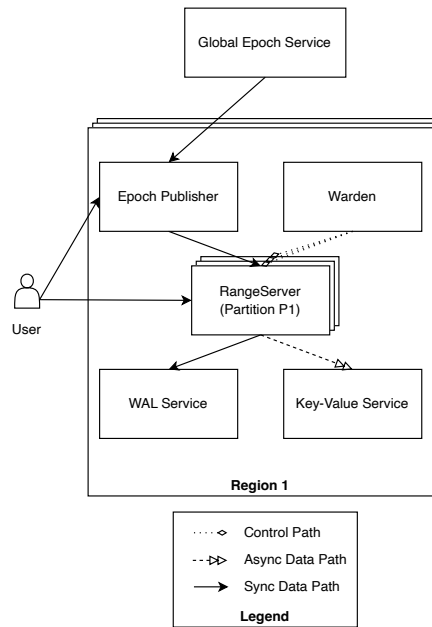


Figure 5.1: Chablis architecture.

### 5.3 Challenges

Rosé addresses the following challenges arising from the problem of asynchronous replication in partitioned databases:

- **Bounding Replication lag:** Asynchronous replication acknowledges writes before remote replicas persist them, so lag is unavoidable; when that lag grows too large it exposes users to stale data and undermines failover.
- **Minimizing Time to Recovery:** In synchronous schemes, every replica already has a complete, identical log, enabling near-instant failover. With asynchronous replication, partitions

may advance at different rates, which leads to accumulation of unusable data on failover. For example, if partitions  $p_1$  and  $p_2$  have replicated changes up to times  $t_1$  and  $t_2$  with  $t_1 < t_2$ , then a failover can rely only on state up to  $t_1$  and discard all updates with  $t > t_1$ . Thus, uneven progress can inflate downtime and/or decrease performance after failover.

## 5.4 Rosé Protocol

Rosé is an asynchronous primary-backup replication protocol for geo-distributed partitioned databases, shown in figure 5.2 It provides strong consistency guarantees (i.e. monotonic prefix consistency) at the secondary by integrating with a database’s existing notions of time (§5.4.1). As with any asynchronous protocol, Rosé trades durability for increased performance during normal operations. However, Rosé takes a principled approach (§5.4.2 to cap replication lag, through a queue-based backpressure mechanism. This way, Rosé limits the maximum amount of data that could be lost in the event of an regional outage. Finally, Rosé tackles the problem of quick and performant failover by applying replicated entries in a coordinated fashion across the backup cluster, ensuring that all backup replicas make equal progress and don’t accumulate potentially invalid data (§5.4.3).

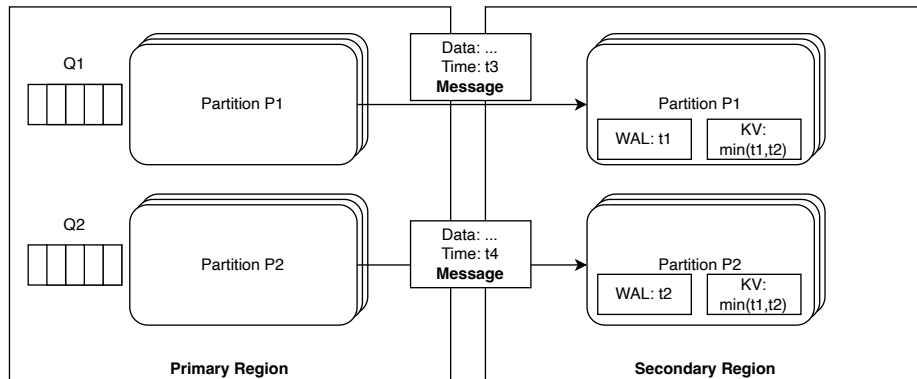


Figure 5.2: Rosé overview.

### 5.4.1 Maintaining Monotonic Prefix Consistency

Many distributed database systems already provide globally consistent, serializable snapshot reads as a core feature. To enable cross-partition snapshot queries, these systems necessarily maintain a coherent notion of global snapshots, implemented through various ordering mechanisms including real-time ordering [59], hybrid logical clocks (HLC) [58, 196], or epoch-based approaches [197, 198]. This existing global snapshot infrastructure presents a natural foundation for extending monotonic prefix consistency to partitioned database environments: backup replicas should maintain externally visible state that corresponds to globally consistent snapshots of the primary system at all times. Since we are integrating Rosé with Chablis, we will use epochs as our measure of time. However, any of the aforementioned methods would work as well.

Based on this observation, we can construct a minimal asynchronous replication protocol that provides monotonic prefix consistency. Each primary partition sends transaction write-sets, in increasing order of epochs (WAL order), to the corresponding backup partition. Correspondingly, each backup partition receives committed transactions and applies them to reconstruct the primary's state. In the meantime, the component responsible for cluster management at the backup region tracks the latest fully applied epoch  $e_i$  at each partition  $P_i$ , which can be piggybacked on existing health checks. The most recent snapshot that can serve reads at the backup corresponds to the minimum epoch of all partitions,  $e_{snapshot} = \min(e_i)$ . As the snapshot epoch  $e_{snapshot}$  advances monotonically, the backup provides monotonic prefix consistency for reads.

### 5.4.2 Bounding the Replication Lag

#### **Rosé's Mechanism**

Replication lag, defined as the number of transactions that have not yet been replicated from the primary to the backup, represents a fundamental challenge in asynchronous replication systems. For example, a partition  $P_i$  where the primary operates at epoch  $e_{primary,i}$  and the backup at epoch  $e_{backup,i}$ , the replication lag is  $replication\_lag_i = e_{primary,i} - e_{backup,i}$ . Excessive repli-

cation lag has proven to be a significant operational concern for asynchronous databases, causing both operational burden and system outages. In partitioned database environments, replication lag becomes particularly problematic because backup data remains useful only up to the snapshot epoch, as defined in §5.4.1. The effective replication lag across the system is thus  $effective\_replication\_lag = \min(replication\_lag_i) = \min(e_{primary,i} - e_{backup,i})$ . This formulation reveals a critical vulnerability: during failover scenarios, partitions that have progressed beyond the snapshot epoch contain data that becomes effectively useless and must be cleaned up before promotion to primary status. Consequently, the progress of all partitions becomes interdependent, as a single stalling partition can compromise the durability guarantees for the entire database.

Rosé proposes a backpressure-based mechanism that effectively caps replication lag in distributed partitioned databases, while maintaining availability. Rosé maintains a bounded queue of size  $L$  for each partition to track outstanding transactions, employing push-based replication where the primary actively propagates changes to backups while monitoring replication progress across all partitions. When any partition’s queue reaches capacity, writes are throttled for that specific partition. This maintains a very desirable property: straggler partitions only affect themselves but at the same time avoid accumulating an ever-increasing backlog that keeps the backup constantly behind. Furthermore, this gives Rosé an effective way to detect straggler partitions so it can take actions to mitigate them such as migrating them to faster or less loaded servers. We will show that Rosé provides enhanced availability during normal operation and at worst similar to synchronous availability under network outages.

### **Availability Proof**

We will show that Rosé’s availability is at worst as much as synchronous replication. Assume a distributed database with partitions  $P_1, \dots, P_n$ , queue sizes  $B_1, \dots, B_n$  and queue limit  $L$  under a primary-backup replication setup and transactions  $T_1, T_2, \dots, T_m$  issued in that order. First, we’ll show that in the case of a single partition, the accepted transactions of Rosé are a strict superset

of the accepted transactions for synchronous replication. Initially, any transaction accepted by synchronous replication must also be accepted by Rosé without accumulating backlog. Transactions that aren't accepted by the synchronous scheme are accepted by Rosé initially, while  $B < L$ . Once  $B = L$  and a new transaction appears, there are two possible outcomes. If the transaction is rejected by the synchronous scheme, it is also rejected by Rosé. If it is accepted, it means the link to the secondary is up and a slot can be cleared in the queue, so that there is space for the new transaction. We account for this case by waiting for some timeout  $\theta$  when the queue is full. Similarly, the same point holds for multiple partitions.

A key issue in this scheme is that replication lag can grow arbitrarily large if a partition is completely down. For example, a partition may accept a write at  $e_i$  and then go down for an arbitrarily large amount of time, preventing the snapshot epoch at the backup from advancing. Luckily, partitions are already replicated and highly available inside the region, so the odds of a partition being completely down for a long time is extremely unlikely. More specifically, assuming a standard replication factor  $R = 3$  and considering that we care about read availability for replication,

$$\begin{aligned} \Pr[\text{single partition unavailable}] &= \Pr[\text{all replicas down}] \\ &= \Pr[\text{replica node down}]^R \end{aligned}$$

Assuming a 99.9% availability for a single node, any single replica will be available 99.9999999% of the time. Thus, the probability of a persisting stalled partition is practically infeasible.

### **Important Details**

A robust and performant implementation is crucial to our backpressure design, so that writes aren't throttled just because the backup can't keep up. For this reason, we match the parallelism of the primary and backup using the C5 algorithm [191]. In addition,  $L$  is an important hyperparameter to tune, which will be different for each user. It should be high enough to utilize the full

network bandwidth and also absorb normal load and temporary spikes, only triggering in genuine edge cases. At the same time, it should match the user’s desired durability guarantees. Finally, in the event of a prolonged outage or loss of the backup region, we assume that an administrator or an external system monitoring uptime of the respective cloud will disengage the backpressure mechanism to restore availability if needed.

### 5.4.3 Minimizing Time to Recovery

As elaborate in §5.4.1, data at the secondary is only useful up to the snapshot epoch, which is the minimum replicated epoch across all partitions. This means that in the event of an outage, the backup cluster needs to restore every partition to that minimum epoch before it can be promoted to primary. So while a synchronously replicated backup would be immediately eligible for promotion, failover in asynchronously replicated partitioned databases presents this significant challenge. A straightforward way of restoring the backup would be to delete all records that were written after the desired snapshot time, albeit at the cost of inflating time to recovery. To work around this limitation, databases like Yugabyte have designed complex recovery schemes that can start serving requests immediately, while deferring clean up to background operations. However, these schemes still suffer from degraded performance after failover, as we will explain below.

#### **Yugabyte: An example from industry**

Yugabyte is a distributed partitioned database that uses a modified version of RocksDB to store its data on each node. It supports consistent asynchronous primary-backup replication through xCluster replication, which is pull-based unlike Rosé. It supports instant failover by deeply integrating with its underlying MVCC storage engine, RocksDB. RocksDB uses an LSM tree to organize key-timestamp-value data, as shown in figure 5.3. The LSM tree consists of many levels and each level contains several SST files, while each SST file contains a metadata block and several data blocks containing key-timestamp-value tuples. To support fast failover, Yugabyte extends RocksDB and stores an extra piece of information in each SST file’s metadata block: the maximum

timestamp contained in the file, shown as *max\_ts*. To rewind the partition to an earlier timestamp, Yugabyte simply reads each file’s metadata block to check if the maximum timestamp exceeds the desired snapshot timestamp. If it doesn’t, the SST file is left as is. Otherwise, it records the desired snapshot timestamp in the file’s metadata block, shown as *keep\_ts*. The partition is now ready to serve requests. Writes work as usual, reads need to do extra work and potentially parse invalid entries, while compaction slowly cleans up the unused data. As we will see in the evaluation, even this elaborate scheme can result in degraded performance after failover.

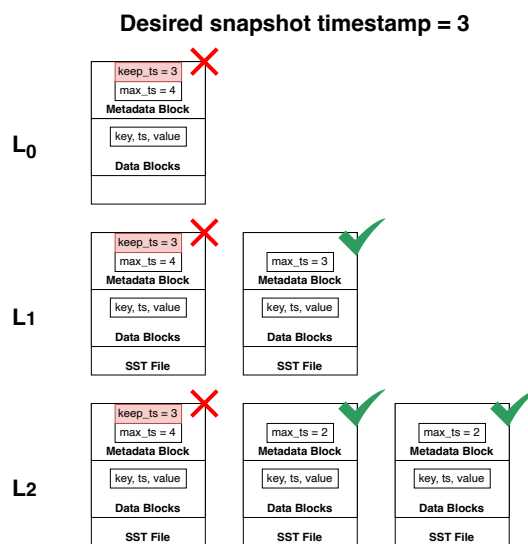


Figure 5.3: Yugabyte recovery of a single partition at  $t=3$ .

## Rosé Coordinated Apply

Rosé takes a different approach to address the fundamental issue of accumulating unusable data in asynchronous replication, by separating the *replication* of data from its *application* in the underlying MVCC store. We observe that most databases persist data in a similar fashion: by first making it durable to a fast write-ahead log (WAL) and periodically applying it to a structured key-value store for fast reads. Databases currently apply replicated records as soon as they arrive at the backup, incurring an expensive bulk-delete operation on failover or degrading their read performance. Instead, we notice that the WAL has a desirable property for this scenario. It keeps

data in insertion order, thus enabling fast trimming of the log to a past offset.

Based on this observation, we propose coordinating the application of the data from the WAL to the key-value store, instead of doing it blindly. More specifically, the backup constantly keeps track of the current minimum replicated epoch across all partitions. Then, it notifies partitions to apply their WAL up to that epoch and no more. This way, on failover Rosé only needs to trim the WAL in order to clean up data, which is a very fast operation. At the same time, it preserves the full performance of the backup cluster after a failover.

## 5.5 Evaluation

We evaluate Rosé’s fast and performance recovery and ability to cap the replication lag. To do so, we integrate Rosé into Chablis [198], a geo-replicated, transactional key-value store.

In our evaluation, we aim to answer the following questions:

- **Q1:** Does Rosé help cap replication lag in asynchronous replication?
- **Q2:** Does Rosé improve performance after failover over existing techniques?

### 5.5.1 Setup

We simulate a multi-node setup, along with network faults, on a single Cloudlab c6525-25g machine. For experiments with Yugabyte, we used version 2.25.2.0-b359 and their manual instructions for failover [199].

### 5.5.2 Q1: Capping the replication lag

To show how backpressure allows Rosé to effectively cap replication lag, we track replication lag over time for a partition with and without backpressure enabled. As shown in figure 5.4, backpressure effectively caps the replication lag and allows the partition to keep up, at the cost of reduced performance. However, only the overloaded partition is affected.

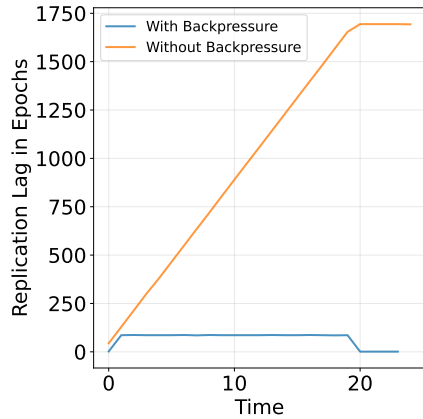
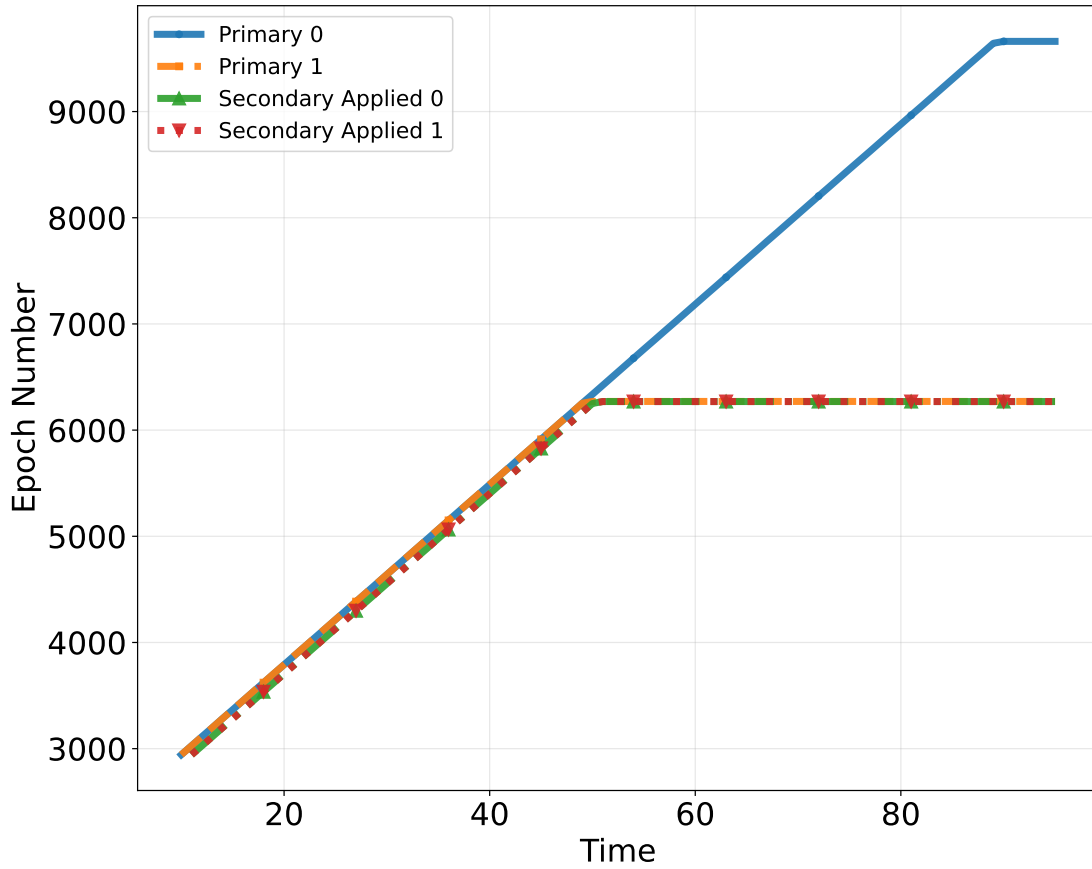


Figure 5.4: Rosé backpressure to cap replication lag.

### 5.5.3 Q2: Recovery with Rosé

To evaluate recovery, we run Chablis and Yugabyte in primary-backup configurations with two nodes in each region. We run uniform read-write transactions in the primary and then fail a link to one of the backup nodes. Yugabyte continues to apply writes in the reachable node, while in Chablis, coordinated apply prevents that. In figure 5.5(a), we see coordinated apply in action. Up until time step 50, both backup nodes are reachable. In time step 50, one of the backup nodes becomes unreachable. Thanks to coordinated apply, the remaining node will not apply any changes after the last replicated epoch of the unavailable node.

Both databases can failover instantly, in under two seconds, but only Rosé can provide the same performance after failover, since its key-value store is clean. On the contrary, Yugabyte’s performance for reads is degraded, with 22% less throughput and 15% higher P99 latency, as shown in figure 5.5(b).



(a) Coordinated apply in action.

Performance After Failover	Yugabyte	Rosé
Throughput Slowdown	22%	0%
P99 Latency Inflation	15%	0%

(b) Performance comparison after failover.

Figure 5.5: Coordinated apply and its impact on failover.

## References

- [1] Y. Zhong *et al.*, “XRP: In-Kernel storage functions with eBPF,” in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, Carlsbad, CA: USENIX Association, Jul. 2022, pp. 375–393, ISBN: 978-1-939133-28-1.
- [2] K. Kourtis, A. Trivedi, and N. Ioannou, “Safe and efficient remote application code execution on disaggregated NVM storage with eBPF,” *arXiv preprint arXiv:2002.11528*, 2020.
- [3] Y. Ghigoff, J. Sopena, K. Lazri, A. Blin, and G. 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)*, USENIX Association, Apr. 2021, pp. 487–501, ISBN: 978-1-939133-21-2.
- [4] A. Bijlani and U. Ramachandran, “Extension framework for file systems in user space,” in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA: USENIX Association, Jul. 2019, pp. 121–134, ISBN: 978-1-939133-03-8.
- [5] Z. Yang *et al.*, “lambda-IO: A unified IO stack for computational storage,” in *21st USENIX Conference on File and Storage Technologies (FAST 23)*, Santa Clara, CA: USENIX Association, Feb. 2023, pp. 347–362, ISBN: 978-1-939133-32-8.
- [6] Y. Zhou, Z. Wang, S. Dharanipragada, and M. Yu, “Electrode: Accelerating distributed protocols with eBPF,” in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, Boston, MA: USENIX Association, Apr. 2023, pp. 1391–1407, ISBN: 978-1-939133-33-5.
- [7] Y. Zhou, X. Xiang, M. Kiley, S. Dharanipragada, and M. Yu, “DINT: Fast In-Kernel distributed transactions with eBPF,” in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, Santa Clara, CA: USENIX Association, Apr. 2024, pp. 401–417, ISBN: 978-1-939133-39-7.
- [8] C. Kulkarni, S. Moore, M. Naqvi, T. Zhang, R. Ricci, and R. Stutsman, “Splinter: Bare-Metal extensions for Multi-Tenant Low-Latency storage,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA: USENIX Association, Oct. 2018, pp. 627–643, ISBN: 978-1-939133-08-3.
- [9] A. Bhardwaj, C. Kulkarni, and R. Stutsman, “Adaptive placement for in-memory storage functions,” in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, USENIX Association, Jul. 2020, pp. 127–141, ISBN: 978-1-939133-14-4.

- [10] J. You, J. Wu, X. Jin, and M. Chowdhury, “Ship compute or ship data? why not both?” In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, USENIX Association, Apr. 2021, pp. 633–651, ISBN: 978-1-939133-21-2.
- [11] *Redis functions*, <https://redis.io/docs/manual/programmability/functions-intro/>.
- [12] *Filtering and retrieving data using Amazon S3 Select*, <https://docs.aws.amazon.com/AmazonS3/latest/userguide/selecting-content-from-objects.html>.
- [13] J. LeFevre and C. Maltzahn, “SkyhookDM: Data processing in Ceph with programmable storage,” *USENIX login*, vol. 45, no. 2, 2020.
- [14] D. Skarlatos and K. Zhao, *Towards programmable memory management with eBPF*, 2024.
- [15] T. Zussman, T. Jiang, and A. Cidon, “Custom page fault handling with eBPF,” in *Proceedings of the ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions*, ser. eBPF ’24, Sydney, NSW, Australia: Association for Computing Machinery, 2024, 71–73, ISBN: 9798400707124.
- [16] K. Mores, S. Psomadakis, and G. Goumas, *Ebpf-mm: Userspace-guided memory management in linux with ebpf*, 2024. arXiv: 2409.11220 [cs.OS].
- [17] D. Lee, I. Choi, C. Lee, S. Lee, and J. Kim, “P2Cache: An application-directed page cache for improving performance of data-intensive applications,” in *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, ser. HotStorage ’23, Boston, MA, USA: Association for Computing Machinery, 2023, 31–36, ISBN: 9798400702242.
- [18] X. Cao, S. Patel, S. Y. Lim, X. Han, and T. Pasquier, “FetchBPF: Customizable prefetching policies in linux with eBPF,” in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, Santa Clara, CA: USENIX Association, Jul. 2024, pp. 369–378, ISBN: 978-1-939133-41-0.
- [19] J. Corbet, *Merging the multi-generational LRU*, <https://lwn.net/Articles/894859/>, 2022.
- [20] B. N. Bershad *et al.*, “Extensibility safety and performance in the SPIN operating system,” in *Proceedings of the fifteenth ACM symposium on Operating systems principles*, 1995, pp. 267–283.
- [21] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith, “Dealing with disaster: Surviving misbehaved kernel extensions,” in *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, ser. OSDI ’96, Seattle, Washington, USA: Association for Computing Machinery, 1996, 213–227, ISBN: 1880446820.

- [22] D. R. Engler, M. F. Kaashoek, and J. O’Toole Jr, “Exokernel: An operating system architecture for application-level resource management,” *ACM SIGOPS Operating Systems Review*, vol. 29, no. 5, pp. 251–266, 1995.
- [23] M. J. Accetta *et al.*, “Mach: A new kernel foundation for UNIX development,” in *Proceedings of the USENIX Summer Conference, Atlanta, GA, USA, June 1986*, USENIX Association, 1986, pp. 93–113.
- [24] C. A. Small and M. I. Seltzer, “Vino: An integrated platform for operating system and database research,” *Harvard Computer Science Group Technical Report*, 1994.
- [25] M. O. Source, *RocksDB*, <https://rocksdb.org/>, <https://rocksdb.org/>, 2022.
- [26] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseu, and R. H. Arpaci-Dusseu, “WiscKey: Separating keys from values in SSD-conscious storage,” in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, vol. 13, Santa Clara, CA: USENIX Association, Feb. 2016, pp. 133–148, ISBN: 978-1-931971-28-7.
- [27] A. Conway *et al.*, “SplinterDB: Closing the bandwidth gap for NVMe key-value stores,” in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, USENIX Association, Jul. 2020, pp. 49–63, ISBN: 978-1-939133-14-4.
- [28] G. Feng *et al.*, “TriCache: A User-Transparent block cache enabling High-Performance Out-of-Core processing with In-Memory programs,” in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, Carlsbad, CA: USENIX Association, Jul. 2022, pp. 395–411, ISBN: 978-1-939133-28-1.
- [29] T. P. G. D. Group, *PostgreSQL*, <https://www.postgresql.org/docs/12/index.html>, <https://www.postgresql.org/>, 2020.
- [30] MongoDB, *WiredTiger storage engine*, <https://docs.mongodb.com/manual/core/wiredtiger/>.
- [31] Y. Dai, J. Liu, A. Arpaci-Dusseu, and R. Arpaci-Dusseu, “Symbiosis: The art of application and kernel cache cooperation,” in *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, Santa Clara, CA: USENIX Association, Feb. 2024, pp. 51–69, ISBN: 978-1-939133-38-0.
- [32] Y. Qian *et al.*, “Combining buffered I/O and direct I/O in distributed file systems,” in *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, Santa Clara, CA: USENIX Association, Feb. 2024, pp. 17–33, ISBN: 978-1-939133-38-0.
- [33] *NVIDIA Nsight Compute*, Accessed on: October 15, 2024, NVIDIA, 2024.

- [34] *NVIDIA nvprof*, Accessed on: October 15, 2024, NVIDIA, 2020.
- [35] *Intel VTune Profiler*, Accessed on: October 15, 2024, Intel Corporation, 2024.
- [36] *AMD ROCm Profiler*, Accessed on: October 15, 2024, AMD, 2024.
- [37] *Google TensorBoard*, Accessed on: October 15, 2024, Google, 2024.
- [38] NVIDIA Corporation, *CUPTI Documentation - Multi-Pass Collection*, Accessed: 2025-02-17, 2025.
- [39] *NVIDIA CUPTI*, Accessed on: October 15, 2024, NVIDIA, 2020.
- [40] L. Adhianto *et al.*, “Hpctoolkit: Tools for performance analysis of optimized parallel programs <http://hpctoolkit.org>,” *Concurr. Comput. ≈: Pract. Exper.*, vol. 22, no. 6, 685–701, Apr. 2010.
- [41] Y. Hao, N. Jain, R. Van der Wijngaart, N. Saxena, Y. Fan, and X. Liu, “Drgpu: A top-down profiler for gpu applications,” in *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*, ser. ICPE ’23, Coimbra, Portugal: Association for Computing Machinery, 2023, 43–53, ISBN: 9798400700682.
- [42] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, “Nvbit: A dynamic binary instrumentation framework for nvidia gpus,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’52, Columbus, OH, USA: Association for Computing Machinery, 2019, 372–383, ISBN: 9781450369381.
- [43] *SASSI: A Low-Level GPU Instrumentation Framework*, Accessed on: October 15, 2024, NVIDIA Corporation, 2024.
- [44] *NVIDIA Compute Sanitizer API*, Accessed on: October 15, 2024, NVIDIA Corporation, 2024.
- [45] *Intel GTPin: Graphics Program Instrumentation*, Accessed on: October 15, 2024, Intel Corporation, 2024.
- [46] *LLVM Project*, Accessed on: October 15, 2024, LLVM Foundation, 2024.
- [47] K. Zhou, Y. Hao, J. Mellor-Crummey, X. Meng, and X. Liu, “Valueexpert: Exploring value patterns in gpu-accelerated applications,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’22, Lausanne, Switzerland: Association for Computing Machinery, 2022, 171–185, ISBN: 9781450392051.

- [48] D. Shen, S. L. Song, A. Li, and X. Liu, “Cudaadvisor: Llm-based runtime profiling for modern gpus,” in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, ser. CGO ’18, Vienna, Austria: Association for Computing Machinery, 2018, 214–227, ISBN: 9781450356176.
- [49] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, “Dynamic warp formation and scheduling for efficient gpu control flow,” in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, 2007, pp. 407–420.
- [50] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, “Analyzing cuda workloads using a detailed gpu simulator,” in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 163–174.
- [51] J. Lew *et al.*, “Analyzing machine learning workloads using a detailed gpu simulator,” in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019, pp. 151–152.
- [52] K. Zhou, X. Meng, R. Sai, and J. Mellor-Crummey, “Gpa: A gpu performance advisor based on instruction sampling,” in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 115–125.
- [53] L. Lamport, “The part-time parliament,” *ACM Trans. Comput. Syst.*, vol. 16, no. 2, 133–169, 1998.
- [54] D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm,” in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA: USENIX Association, Jun. 2014, pp. 305–319, ISBN: 978-1-931971-10-2.
- [55] R. V. Renesse and F. B. Schneider, “Chain replication for supporting high throughput and availability,” in *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, San Francisco, CA: USENIX Association, Dec. 2004.
- [56] J. Terrace and M. J. Freedman, “Object storage on CRAQ: High-Throughput chain replication for Read-Mostly workloads,” in *2009 USENIX Annual Technical Conference (USENIX ATC 09)*, San Diego, CA: USENIX Association, Jun. 2009.
- [57] A. Katsarakis *et al.*, “Hermes: A fast, fault-tolerant and linearizable replication protocol,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’20, Lausanne, Switzerland: Association for Computing Machinery, 2020, 201–217, ISBN: 9781450371025.
- [58] R. Taft *et al.*, “Cockroachdb: The resilient geo-distributed sql database,” in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’20, Portland, OR, USA: Association for Computing Machinery, 2020, 1493–1509, ISBN: 9781450367356.

- [59] J. C. Corbett *et al.*, “Spanner: Google’s globally distributed database,” *ACM Trans. Comput. Syst.*, vol. 31, no. 3, 2013.
- [60] *Netapp - san solutions*, <https://www.netapp.com/data-storage/san-storage-area-network>.
- [61] *Dell - san solutions*, <https://www.dell.com/en-us/dt/learn/data-storage/san-storage.htm>.
- [62] *Lightbits labs*, <https://www.lightbitlabs.com/>.
- [63] A. Klimovic, H. Litz, and C. Kozyrakis, “ReFlex: Remote flash = local flash,” in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’17, Xi’an, China: Association for Computing Machinery, 2017, 345–359, ISBN: 9781450344654.
- [64] Z. Guz, H. H. Li, A. Shayesteh, and V. Balakrishnan, “Performance characterization of NVMe-over-Fabrics storage disaggregation,” *ACM Trans. Storage*, vol. 14, no. 4, 2018.
- [65] J. Min *et al.*, “Gimbal: Enabling multi-tenant storage disaggregation on SmartNIC JBOFs,” in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ser. SIGCOMM ’21, Virtual Event, USA: Association for Computing Machinery, 2021, 106–122, ISBN: 9781450383837.
- [66] S. Legtchenko *et al.*, “Understanding Rack-Scale disaggregated storage,” in *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA: USENIX Association, Jul. 2017.
- [67] L. Bindschaedler, A. Goel, and W. Zwaenepoel, “Hailstorm: Disaggregated compute and storage for distributed LSM-based databases,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’20, Lausanne, Switzerland: Association for Computing Machinery, 2020, 301–316, ISBN: 9781450371025.
- [68] M. Nanavati, J. Wires, and A. Warfield, “Decibel: Isolation and sharing in disaggregated Rack-Scale storage,” in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA: USENIX Association, Mar. 2017, pp. 17–33, ISBN: 978-1-931971-37-9.
- [69] A. Klimovic, C. Kozyrakis, E. Thereska, B. John, and S. Kumar, “Flash storage disaggregation,” in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys ’16, London, United Kingdom: Association for Computing Machinery, 2016, ISBN: 9781450342407.
- [70] A. Verbitski *et al.*, “Amazon Aurora: Design considerations for high throughput cloud-native relational databases,” in *Proceedings of the 2017 ACM International Conference on*

*Management of Data*, ser. SIGMOD '17, Chicago, Illinois, USA: Association for Computing Machinery, 2017, 1041–1052, ISBN: 9781450341974.

- [71] Z. Guo *et al.*, “Cornus: Atomic commit for a cloud DBMS with storage disaggregation,” *Proc. VLDB Endow.*, vol. 16, no. 2, 379–392, 2022.
- [72] *Rockset: Real-time analytics at cloud scale*, <https://rockset.com/>.
- [73] S. Dong, A. Kryczka, Y. Jin, and M. Stumm, “Evolution of development priorities in key-value stores serving large-scale applications: The RocksDB experience,” in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, USENIX Association, Feb. 2021, pp. 33–49, ISBN: 978-1-939133-20-5.
- [74] M. Vuppalapati, J. Miron, R. Agarwal, D. Truong, A. Motivala, and T. Cruanes, “Building an elastic query engine on disaggregated storage,” in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA: USENIX Association, Feb. 2020, pp. 449–462, ISBN: 978-1-939133-13-7.
- [75] Y. Zhong *et al.*, “BPF for storage: An exokernel-inspired approach,” in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '21, Ann Arbor, Michigan: Association for Computing Machinery, 2021, 128–135, ISBN: 9781450384384.
- [76] Y. Matsunobu, S. Dong, and H. Lee, “MyRocks: LSM-tree database storage engine serving Facebook’s social graph,” *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3217–3230, 2020.
- [77] *Zippydb*, <https://engineering.fb.com/2021/08/06/core-data/zippydb/>, <https://engineering.fb.com/2021/08/06/core-infra/zippydb/>.
- [78] *Cockroachdb on rocksdb*, <https://www.cockroachlabs.com/blog/cockroachdb-on-rocksdb/>.
- [79] *MongoDB*, <https://www.mongodb.com/>.
- [80] *RocksDB users*, <https://github.com/facebook/rocksdb/blob/main/USERS.md>.
- [81] S. Dong *et al.*, “Disaggregating RocksDB: A production experience,” *Proc. ACM Manag. Data*, vol. 1, no. 2, 2023.
- [82] R. Miao *et al.*, “From Luna to Solar: The evolutions of the compute-to-storage networks in Alibaba Cloud,” in *Proceedings of the ACM SIGCOMM 2022 Conference*, ser. SIGCOMM '22, Amsterdam, Netherlands: Association for Computing Machinery, 2022, 753–766, ISBN: 9781450394208.

- [83] A Klotz, *SK Hynix's New SSD Boasts 1.4 Million IOPS*, <https://www.tomshardware.com/news/sk-hynix-pcie-4-ssd-record-breaking-random-speeds>.
- [84] J. Hwang, Q. Cai, A. Tang, and R. Agarwal, "TCP = RDMA: CPU-efficient remote storage access with i10," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA: USENIX Association, Feb. 2020, pp. 127–140, ISBN: 978-1-939133-13-7.
- [85] J Kim, *iSCSI - is it the future of cloud storage or doomed by NVMe-oF*, <https://www.snia.org/sites/default/files/news/iSCSI-Future-Cloud-Storage-Doomed-NVMe-oF.pdf>.
- [86] *100g kernel and user space NVMe/TCP using Chelsio offload*, <https://www.chelsio.com/wp-content/uploads/resources/t6-100g-nvmetcp-offload.pdf>.
- [87] I. Zhang *et al.*, "The demikernel datapath os architecture for microsecond-scale datacenter systems," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21, Virtual Event, Germany: Association for Computing Machinery, 2021, 195–211, ISBN: 9781450387095.
- [88] J. M. Hellerstein and M. Stonebraker, "Predicate migration: Optimizing queries with expensive predicates," in *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, 1993, pp. 267–276.
- [89] A. Y. Levy, I. S. Mumick, and Y. Sagiv, "Query optimization by predicate move-around," in *VLDB*, 1994, pp. 96–107.
- [90] Y. Yang *et al.*, "FlexPushdownDB: Hybrid pushdown and caching in a cloud DBMS," *Proc. VLDB Endow.*, vol. 14, no. 11, 2101–2113, 2021.
- [91] Google, *LevelDB*, <https://leveldb.org/>, <https://github.com/google/leveldb>, 2020.
- [92] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "PebblesDB: Building key-value stores using fragmented log-structured merge trees," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 497–514.
- [93] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS operating systems review*, vol. 44, no. 2, pp. 35–40, 2010.
- [94] R. Taft *et al.*, "CockroachDB: The resilient geo-distributed SQL database," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 1493–1509.

- [95] *SQLite pluggable storage engine*, <https://sqlite.org/src4/doc/trunk/www/storage.wiki>.
- [96] eBPF.io authors, *EBPF*, <https://ebpf.io/>.
- [97] P. O’Neil, E. Cheng, D. Gawlick, and E. O’Neil, “The log-structured merge-tree (LSM-tree),” *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.
- [98] *NVMe base specification*, [https://nvmexpress.org/wp-content/uploads/NVM-Express-1\\_4b-2020.09.21-Ratified.pdf](https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4b-2020.09.21-Ratified.pdf).
- [99] *NVMe over fabrics specification*, <https://nvmexpress.org/wp-content/uploads/NVMe-over-Fabrics-1.1a-2021.07.12-Ratified.pdf>.
- [100] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park, “Cache modeling and optimization using miniature simulations,” in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Santa Clara, CA: USENIX Association, Jul. 2017, pp. 487–498, ISBN: 978-1-931971-38-6.
- [101] D. Duplyakin *et al.*, “The design and operation of CloudLab,” in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, Renton, WA: USENIX Association, Jul. 2019, pp. 1–14, ISBN: 978-1-939133-03-8.
- [102] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with YCSB,” in *Proceedings of the 1st ACM symposium on Cloud computing*, ser. SoCC ’10, Indianapolis, Indiana, USA: Association for Computing Machinery, 2010, pp. 143–154, ISBN: 9781450300360.
- [103] J. Yang, Y. Yue, and K. V. Rashmi, “A large scale analysis of hundreds of in-memory cache clusters at Twitter,” in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association, Nov. 2020, pp. 191–208, ISBN: 978-1-939133-19-9.
- [104] M. Stonebraker, “Operating system support for database management,” *Commun. ACM*, vol. 24, no. 7, 412–418, 1981.
- [105] P. Cao, E. W. Felten, and K. Li, “Implementation and performance of application-controlled file caching,” in *First Symposium on Operating Systems Design and Implementation (OSDI 94)*, Monterey, CA: USENIX Association, Nov. 1994.
- [106] P. Cao, E. W. Felten, A. R. Karlin, and K. Li, “Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling,” *ACM Trans. Comput. Syst.*, vol. 14, no. 4, 311–343, 1996.

- [107] M. F. Kaashoek *et al.*, “Application Performance and Flexibility on Exokernel Systems,” in *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, ser. SOSP ’97, Saint Malo, France: Association for Computing Machinery, 1997, 52–65, ISBN: 0897919165.
- [108] N. Beckmann, H. Chen, and A. Cidon, “LHD: Improving cache hit rate by maximizing hit density,” in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, Renton, WA: USENIX Association, Apr. 2018, pp. 389–403, ISBN: 978-1-939133-01-4.
- [109] J. Corbet, *The multi-generational LRU*, <https://lwn.net/Articles/851184/>, 2021.
- [110] N. Megiddo and D. Modha, “Outperforming LRU with an adaptive replacement cache algorithm,” *Computer*, vol. 37, no. 4, pp. 58–65, 2004.
- [111] J. Yang, Y. Zhang, Z. Qiu, Y. Yue, and R. Vinayak, “FIFO queues are all you need for cache eviction,” in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP ’23, Koblenz, Germany: Association for Computing Machinery, 2023, 130–149, ISBN: 9798400702297.
- [112] S. Jiang and X. Zhang, “LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance,” in *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS ’02, Marina Del Rey, California: Association for Computing Machinery, 2002, 31–42, ISBN: 1581135319.
- [113] J. T. Robinson and M. V. Devarakonda, “Data cache management using frequency-based replacement,” *SIGMETRICS Perform. Eval. Rev.*, vol. 18, no. 1, 134–142, 1990.
- [114] E. J. O’Neil, P. E. O’Neil, and G. Weikum, “The LRU-K page replacement algorithm for database disk buffering,” in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’93, Washington, D.C., USA: Association for Computing Machinery, 1993, 297–306, ISBN: 0897915925.
- [115] D. Zheng, R. Burns, and A. S. Szalay, “A parallel page cache: IOPS and caching for multi-core systems,” in *4th USENIX Workshop on Hot Topics in Storage and File Systems (Hot-Storage 12)*, Boston, MA: USENIX Association, Jun. 2012.
- [116] K. Kaffes, J. T. Humphries, D. Mazières, and C. Kozyrakis, “Syrup: User-defined scheduling across the stack,” in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP ’21, Virtual Event, Germany: Association for Computing Machinery, 2021, 605–620, ISBN: 9781450387095.
- [117] J. T. Humphries *et al.*, “GhOSt: Fast & flexible user-space delegation of Linux scheduling,” in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Princi-*

- ples, ser. SOSP '21, Virtual Event, Germany: Association for Computing Machinery, 2021, 588–604, ISBN: 9781450387095.
- [118] J. Corbet, *The extensible scheduler class*, <https://lwn.net/Articles/922405/>.
- [119] Facebook, *Memory usage in RocksDB*, <https://github.com/facebook/rocksdb/wiki/memory-usage-in-rocksdb>, 2024.
- [120] N. Bansal, *An Overview of Caching for PostgreSQL*, <https://severalnines.com/blog/overview-caching-postgresql/>, 2020.
- [121] MySQL, *15.6.2.2 The Physical Structure of an InnoDB Index*, <https://dev.mysql.com/doc/refman/5.6/en/innodb-storage-engine.html>, <https://dev.mysql.com/doc/refman/8.0/en/innodb-physical-structure.html>, 2024.
- [122] P. contributors, *PyTorch torch.load*, <https://pytorch.org/docs/stable/generated/torch.load.html>.
- [123] Milvus, *Milvus chunk cache*, [https://milvus.io/docs/chunk\\_cache.md](https://milvus.io/docs/chunk_cache.md).
- [124] T. Heo, *Cgroup-v2*, <https://docs.kernel.org/admin-guide/cgroup-v2.html>.
- [125] J. Corbet, *Clarifying memory management with page folios*, <https://lwn.net/Articles/849538/>, 2021.
- [126] L. man pages project, *madvise – linux manual page*, <https://man7.org/linux/man-pages/man2/madvise.2.html>, 2024.
- [127] B. Cantrill, *A crime against common sense (MADV\_DONTNEED)*, <https://www.youtube.com/watch?v=bg6-LVCHmGM&t=3518s>, **OmniTi Surge 2015**, 2015.
- [128] C. Huang, S. Blackburn, and Z. Cai, “Improving garbage collection observability with performance tracing,” in *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, ser. MPLR 2023, Cascais, Portugal: Association for Computing Machinery, 2023, 85–99, ISBN: 9798400703805.
- [129] S. Miano, M. Bertrone, F. Risso, M. V. Bernal, Y. Lu, and J. Pi, “Securing linux with a faster and scalable iptables,” *SIGCOMM Comput. Commun. Rev.*, vol. 49, no. 3, 2–17, Nov. 2019.
- [130] J. Jia et al., *Programmable system call security with eBPF*, 2023. arXiv: 2302.10366 [cs.OS].

- [131] T. Høiland-Jørgensen *et al.*, *XDP*, <https://www.iovisor.org/technology/xdp>, Heraklion, Greece, 2018.
- [132] I. Zarkadas *et al.*, *BPF-oF: Storage function pushdown over the network*, 2023. arXiv: 2312.06808 [cs.OS].
- [133] Solidigm, *Solidigm d7-ps1030*, <https://www.solidigm.com/products/data-center/d7/ps1030.html#configurator>.
- [134] W. Digital, *Western digital PC SN8000S NVMe SSD*, [https://documents.westerndigital.com/content/dam/doc-library/en\\_us/assets/public/western-digital/product/internal-drives/pc-sn8000s-nvme-ssd/data-sheet-pc-sn8000s-nvme-ssd.pdf](https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/product/internal-drives/pc-sn8000s-nvme-ssd/data-sheet-pc-sn8000s-nvme-ssd.pdf), 2024.
- [135] T. kernel development community, *Bpf-ringbuf*, <https://www.kernel.org/doc/html/next/bpf/ringbuf.html>.
- [136] A. Gallant, *Ripgrep is faster than {grep, ag, git grep, ucg, pt, sift}*, <https://blog.burntsushi.net/ripgrep/>, 2016.
- [137] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter, “AdaptSize: Orchestrating the hot object memory cache in a content delivery network,” in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA: USENIX Association, Mar. 2017, pp. 483–498, ISBN: 978-1-931971-37-9.
- [138] D. S. Berger, B. Berg, T. Zhu, S. Sen, and M. Harchol-Balter, “RobinHood: Tail latency aware caching – dynamic reallocation from Cache-Rich to Cache-Poor,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA: USENIX Association, Oct. 2018, pp. 195–212, ISBN: 978-1-939133-08-3.
- [139] Z. Song, D. S. Berger, K. Li, and W. Lloyd, “Learning relaxed Belady for content distribution network caching,” in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA: USENIX Association, Feb. 2020, pp. 529–544, ISBN: 978-1-939133-13-7.
- [140] D. L.-K. Wong *et al.*, “Baleen: ML admission & prefetching for flash caches,” in *22nd USENIX Conference on File and Storage Technologies (FAST 24)*, Santa Clara, CA: USENIX Association, Feb. 2024, pp. 347–371, ISBN: 978-1-939133-38-0.
- [141] Y. Zhang, J. Yang, Y. Yue, Y. Vigfusson, and K. Rashmi, “SIEVE is simpler than LRU: An efficient turn-key eviction algorithm for web caches,” in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, Santa Clara, CA: USENIX Association, Apr. 2024, pp. 1229–1246, ISBN: 978-1-939133-39-7.

- [142] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li, “RIPQ: Advanced photo caching on flash for facebook,” in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, Santa Clara, CA: USENIX Association, Feb. 2015, pp. 373–386, ISBN: 978-1-931971-201.
- [143] M. K. Lau, *Struct-ops*, <https://lwn.net/Articles/809092/>, 2020.
- [144] R. Karedla, J. S. Love, and B. G. Wherry, “Caching strategies to improve disk system performance,” *Computer*, vol. 27, no. 3, pp. 38–46, 1994.
- [145] K. development community, *BPF\_MAP\_TYPE\_HASH, with PERCPU and LRU variants*, [https://docs.kernel.org/bpf/map\\_hash.html](https://docs.kernel.org/bpf/map_hash.html).
- [146] K. K. Dwivedi, R. Iyer, and S. Kashyap, “Fast, flexible, and practical kernel extensions,” in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, ser. SOSP ’24, Austin, TX, USA: Association for Computing Machinery, 2024, 249–264, ISBN: 9798400712517.
- [147] D. Alden, *A proposal for shared memory in BPF programs*, <https://lwn.net/Articles/961941/>, 2024.
- [148] J. Corbet, *Red-black trees for BPF programs*, <https://lwn.net/Articles/924128/>, 2023.
- [149] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, “Shinjuku: Preemptive scheduling for  $\mu$ second-scale tail latency,” in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, ser. NSDI’19, Boston, MA, USA: USENIX Association, 2019, pp. 345–360, ISBN: 9781931971492.
- [150] J. Axboe, *Fio: Flexible I/O tester*, <https://github.com/axboe/fio>.
- [151] Bloomberg Intelligence, *Generative AI Races Toward \$1.3 Trillion in Revenue by 2032*, <https://www.bloomberg.com/professional/insights/data/generative-ai-races-toward-1-3-trillion-in-revenue-by-2032/>, Accessed: 2024-11-22.
- [152] N. Corporation, *Nvidia official website*, Accessed: 2024-11-26, 2024.
- [153] I. Advanced Micro Devices, *Amd official website*, Accessed: 2024-11-26, 2024.
- [154] I. Corporation, *Intel official website*, Accessed: 2024-11-26, 2024.
- [155] N. P. Jouppi *et al.*, “In-datacenter performance analysis of a tensor processing unit,” 2017.

- [156] N. Jouppi *et al.*, “Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings,” in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA ’23, Orlando, FL, USA: Association for Computing Machinery, 2023, ISBN: 9798400700958.
- [157] A. Firoozshahian *et al.*, “Mtia: First generation silicon targeting meta’s recommendation systems,” in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA ’23, Orlando, FL, USA: Association for Computing Machinery, 2023, ISBN: 9798400700958.
- [158] Amazon Web Services, *AWS Inferentia*, <https://aws.amazon.com/ai/machine-learning/inferentia/>, Accessed: 2024-11-22.
- [159] Amazon Web Services, *AWS Trainium*, <https://aws.amazon.com/ai/machine-learning/trainium/>, Accessed: 2024-11-22.
- [160] Cerebras Systems, *Cerebras Wafer-Scale Engine 3*, <https://cerebras.ai/product-chip/>, Accessed: 2024-11-22.
- [161] Y.-H. Chen, J. Emer, and V. Sze, “Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks,” in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 367–379.
- [162] B. Reagen *et al.*, “Minerva: Enabling low-power, highly-accurate deep neural network accelerators,” in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA ’16, Seoul, Republic of Korea: IEEE Press, 2016, 267–278, ISBN: 9781467389471.
- [163] S. Han *et al.*, “Eie: Efficient inference engine on compressed deep neural network,” in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA ’16, Seoul, Republic of Korea: IEEE Press, 2016, 243–254, ISBN: 9781467389471.
- [164] M. Abadi *et al.*, “Tensorflow: A system for large-scale machine learning,” in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’16, Savannah, GA, USA: USENIX Association, 2016, 265–283, ISBN: 9781931971331.
- [165] R. Frostig, M. Johnson, and C. Leary, “Compiling machine learning programs via high-level tracing,” 2018.
- [166] A. Paszke *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” in *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.
- [167] C. Lattner and J. Pienaar, *Mlir primer: A compiler infrastructure for the end of moore’s law*, 2019.

- [168] A. Sabne, *Xla : Compiling machine learning for peak performance*, 2020.
- [169] J. Vanian and K. Leswing, “Chatgpt and generative ai are booming, but the costs can be extraordinary,” *CNBC*, 2023, Published: Mar 13, 2023, Updated: Apr 17, 2023, Accessed: 2025-02-10.
- [170] A. Shilov, *Deepseek’s ai breakthrough bypasses industry-standard cuda for some functions, uses nvidia’s assembly-like ptx programming instead*, Accessed: 2025-02-17, 2025.
- [171] *NVIDIA Nsight Systems*, Accessed on: October 15, 2024, NVIDIA, 2024.
- [172] OpenXLA Project, *StableHLO: A Portability Layer for ML Frameworks and Compilers*, <https://openxla.org/stablehlo>, Accessed: 2024-11-22.
- [173] OpenXLA Project, *OpenXLA: An Open Ecosystem for Machine Learning Infrastructure*, <https://openxla.org/>, Accessed: 2024-11-22.
- [174] H.-I. C. Liu, M. Brehler, M. Ravishankar, N. Vasilache, B. Vanik, and S. Laurenzo, “Tinyiree: An ml execution environment for embedded systems from compilation to deployment,” *IEEE Micro*, vol. 42, no. 5, 9–16, Sep. 2022, Accessed: 2024-10-30.
- [175] T. Chen *et al.*, “Tvm: An automated end-to-end optimizing compiler for deep learning,” in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’18, Carlsbad, CA, USA: USENIX Association, 2018, 579–594, ISBN: 9781931971478.
- [176] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, “Flashattention: Fast and memory-efficient exact attention with io-awareness,” in *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, Eds., vol. 35, Curran Associates, Inc., 2022, pp. 16 344–16 359.
- [177] T. Dao, “Flashattention-2: Faster attention with better parallelism and work partitioning,” in *The Twelfth International Conference on Learning Representations*, 2024.
- [178] J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, and T. Dao, “Flashattention-3: Fast and accurate attention with asynchrony and low-precision,” in *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.
- [179] W. Kwon *et al.*, “Efficient memory management for large language model serving with pagedattention,” in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP ’23, Koblenz, Germany: Association for Computing Machinery, 2023, 611–626, ISBN: 9798400702297.
- [180] P. Tillet, H. T. Kung, and D. Cox, “Triton: An intermediate language and compiler for tiled neural network computations,” in *Proceedings of the 3rd ACM SIGPLAN International*

*Workshop on Machine Learning and Programming Languages*, ser. MAPL 2019, Phoenix, AZ, USA: Association for Computing Machinery, 2019, 10–19, ISBN: 9781450367196.

- [181] JAX Developers, *Pallas: A Kernel Language for JAX*, <https://jax.readthedocs.io/en/latest/pallas/index.html>, Accessed: 2024-11-22.
- [182] P. Barham *et al.*, “Pathways: Asynchronous distributed dataflow for ml,” in *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu, Eds., vol. 4, 2022, pp. 430–449.
- [183] N. Corporation, *Cuda-gdb*, Accessed: 2025-02-14, 2025.
- [184] *Debugging with cerebras sdk*, <https://sdk.cerebras.net/debug/debugging#csdb-debugger>, Accessed: 2024-10-29, 2023.
- [185] H. Rotithor, “Postsilicon validation methodology for microprocessors,” *IEEE Design & Test of Computers*, vol. 17, no. 04, pp. 77–88, 2000.
- [186] Z. He and X. Chen, “Design and implementation of high-speed configurable ecc co-processor,” in *2017 IEEE 12th International Conference on ASIC (ASICON)*, 2017, pp. 734–737.
- [187] A. Tyagi *et al.*, *Thehuzz: Instruction fuzzing of processors using golden-reference models for finding software-exploitable vulnerabilities*, 2022. arXiv: 2201.09941 [cs.CR].
- [188] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, “Accel-sim: An extensible simulation framework for validated gpu modeling,” in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ser. ISCA ’20, Virtual Event: IEEE Press, 2020, 473–486, ISBN: 9781728146614.
- [189] N. Corporation, *Nvidia collective communications library (nccl)*, Accessed: 2025-01-21, 2025.
- [190] A. W. Services, *Collective communication*, Accessed: 2025-01-21, 2025.
- [191] J. Helt, A. Sharma, D. J. Abadi, W. Lloyd, and J. M. Faleiro, “C5: Cloned concurrency control that always keeps up,” *Proc. VLDB Endow.*, vol. 16, no. 1, 1–14, 2022.
- [192] B. Calder *et al.*, “Windows azure storage: A highly available cloud storage service with strong consistency,” in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP ’11, Cascais, Portugal: Association for Computing Machinery, 2011, 143–157, ISBN: 9781450309776.
- [193] L. Qiao *et al.*, “On brewing fresh espresso: LinkedIn’s distributed data serving platform,” ser. SIGMOD ’13, New York, New York, USA: Association for Computing Machinery, 2013, 1135–1146, ISBN: 9781450320375.

- [194] H. Lu *et al.*, “Existential consistency: Measuring and understanding consistency at facebook,” in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP ’15, Monterey, California: Association for Computing Machinery, 2015, 295–310, ISBN: 9781450338349.
- [195] P. Antonopoulos *et al.*, “Socrates: The new sql server in the cloud,” in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD ’19, Amsterdam, Netherlands: Association for Computing Machinery, 2019, 1743–1756, ISBN: 9781450356435.
- [196] *yugabyteDB*. <https://yugabyte.com/>.
- [197] T. Eldeeb, X. Xie, P. A. Bernstein, A. Cidon, and J. Yang, “Chardonnay: Fast and general datacenter transactions for On-Disk databases,” in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, Boston, MA: USENIX Association, Jul. 2023.
- [198] T. Eldeeb, P. A. Bernstein, A. Cidon, and J. Yang, “Chablis: Fast and general transactions in geo-distributed systems,” in *CIDR*, 2024.
- [199] Yugabyte, Inc., *Transactional xcluster replication setup (manual)*, “Set up transactional xCluster replication (Manual mode)”, 2025.