Metadata-Driven Near-Exclusive Caching for NVMe-oF SANs with eBPF

Daegyu Han*
Sungkyunkwan University
Suwon, Korea
hdg9400@skku.edu

Sangeun Chae
Sungkyunkwan University
Suwon, Korea
andrewchae48@gmail.com

Jaehyun Hwang
Sungkyunkwan University
Suwon, Korea
jh.hwang@skku.edu

Beomseok Nam
Sungkyunkwan University
Suwon, Korea
bnam@skku.edu

ABSTRACT

In SAN systems with a multilevel caching hierarchy, the storage cache lacks visibility into the page cache of the compute node, resulting in redundant caching of the same blocks and inefficient use of limited cache space. Prior efforts have explored exclusive caching, but they often incur network overhead by transferring evicted pages for cache admission and require intrusive modifications to components on compute nodes. In this paper, we present BNEX, a metadata-driven exclusive caching system for NVMe-oF SANs that significantly reduces network overhead by avoiding full-page transfers. BNEX non-intrusively traces page cache evictions using eBPF and transmits only lightweight block-level metadata. Leveraging metadata of evicted pages, the storage cache prefetches nearby blocks in fixed-size block groups and selectively retains them without compromising exclusivity. Our evaluation shows that BNEX improves throughput by up to 1.15× and cache hit ratio by up to 8× compared to inclusive SAN caching systems.

CCS CONCEPTS

• **Information systems** → **Storage network architectures**.

KEYWORDS

NVMe-oF, Storage Area Network, Multilevel Caching, Exclusive Caching, eBPF

*Now at Samsung Electronics

This work is licensed under a Creative Commons Attribution 4.0 International License.

APSys '25, October 12–13, 2025, Seoul, Republic of Korea
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1572-3/25/10...$15.00
https://doi.org/10.1145/3725783.3764404

ACM Reference Format:
Daegyu Han, Sangeun Chae, Jaehyun Hwang, and Beomseok Nam. 2025. Metadata-Driven Near-Exclusive Caching for NVMe-oF SANs with eBPF. In *16th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '25), October 12–13, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3725783.3764404

1 INTRODUCTION

To meet the needs of latency-sensitive and performance-critical applications, modern Storage Area Network (SAN) systems are increasingly adopting NVMe over Fabrics (NVMe-oF), which offers high I/O performance [20, 24]. These systems typically use a storage controller to aggregate multiple NVMe SSDs into a single array, enabling parallel access and high bandwidth. However, striping data across multiple devices can lead to unpredictable performance, particularly under random or small-sized I/O workloads [8, 35]. To mitigate this and ensure predictable response times, storage caching remains a crucial component even in NVMe-oF SANs [2].

In SAN systems, the page cache on compute nodes serves as the first-level cache, and the storage cache functions as the second-level cache. In this multilevel caching hierarchy, overall cache efficiency is influenced not only by the replacement policy but also by the admission policy that governs which data should be cached in each level. *Inclusive caching*, in particular, can cause the same data to be stored at both levels of the cache, which reduces the effective capacity of the storage cache. In contrast, *exclusive caching* prevents duplication and improves space efficiency by ensuring that the data exists in only one place in the hierarchy.

Previous studies [6, 9, 17, 42, 43] have explored exclusive caching in SAN systems. Since the second-level storage cache lacks visibility into the state of the page cache, it requires inter-layer communication or non-trivial coordination to avoid duplication, which results in two key limitations: (1) non-trivial network overhead due to page transfers triggered by eviction, (2) intrusive system modifications, which hinder real-world validation. First, to preserve exclusivity, DEMOTE [42] and Eviction-based Placement [9] transfer evicted pages from compute nodes to storage over the network, incurring significant traffic overhead. Second, many exclusive caching proposals, such as DEMOTE [42]

and Karma [43], require intrusive modifications to compute node components. Some modify device drivers to introduce new commands [9, 17, 42], while others rely on application-level hints [29, 40, 43], limiting their practicality. Due to these intrusive changes, many of these techniques have only been evaluated in simulation [6, 29, 42], raising concerns about their deployability in real-world environments. Although X-Ray [6] adopts a non-intrusive approach for RAID caching, it depends on specific file system layouts and infers cache decisions based on file access times.

In this study, we present BNEX (e**B**PF-based **N**ear-**EX**clusive Caching), a SAN caching system that achieves near-exclusive caching by transmitting only lightweight block-level metadata to the storage node. Instead of sending evicted pages over the network, the metadata enables the storage node to fetch the corresponding pages directly from its local block device and cache them. This design eliminates the need for compute nodes to transfer evicted pages to the storage cache, significantly reducing network traffic. BNEX consists of two main components: (i) *KTracer* leverages eBPF to non-intrusively monitor kernel events on compute nodes and transmits only metadata of evicted pages to the storage node. (ii) On the storage side, *PrefCache* uses these metadata to prefetch adjacent blocks in fixed-size groups, improving cache efficiency while preserving near-exclusivity. Together, these components reduce network traffic and prevent redundant caching between compute and storage nodes, while requiring only minimal coordination.

Key contributions of this study are as follows:

- We propose BNEX, a SAN caching system that improves cache coordination efficiency by replacing costly page transfers with compact metadata communication.
- BNEX uses eBPF to monitor page cache activity on compute nodes in a non-intrusive manner and supports metadata-based prefetching on storage nodes to improve cache effectiveness.
- We implement KTracer as a user-space tool and integrate PrefCache into an NVMe-oF-based software-defined storage system [36]. Our evaluations with real-world workloads show that BNEX improves throughput and cache hit ratio by up to 1.15× and 8×, respectively, compared to existing caching systems.

2 BACKGROUND AND MOTIVATION

2.1 NVMe-oF SAN

A SAN is a storage architecture in which multiple hosts access a shared storage pool over a high-performance network. It is widely deployed in enterprise systems and data centers that require large-scale data processing [4, 12, 32]. Figure 1 illustrates the architecture of a SAN system built with NVMe SSDs as a storage backend. Compute nodes are connected to storage nodes via NVMe-oF [34], a modern

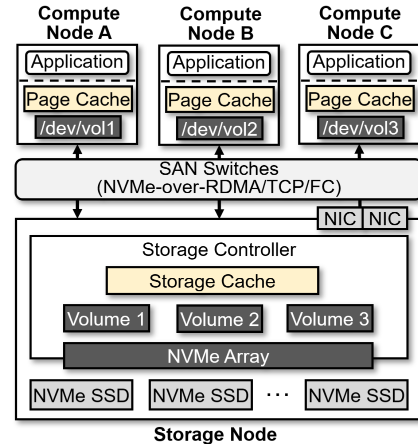


Figure 1: NVMe-oF SAN in Modern Datacenters

protocol that leverages RDMA to provide remote access with performance comparable to local block devices. Since remote storage devices connected through NVMe-oF appear as local block devices to compute nodes, applications can access them through the standard file system interface without any modification.

To manage multiple NVMe devices, the SAN employs a storage controller that aggregates them into a single RAID-based array, providing both high performance and fault tolerance. By striping data across devices, the storage controller exposes a unified block address space to compute nodes, enabling scalable access to storage.

Although NVMe arrays provide high bandwidth, their performance can degrade for workloads with small or non-sequential access patterns. This degradation stems from striping overhead (e.g., access amplification and reduced I/O locality) [8, 35] and inefficient internal parallelism within devices [25]. To address these issues and reduce overall latency, DRAM-based storage caches are commonly deployed on storage nodes [1, 2, 12, 32]. While such storage caches can function as write buffers, workloads in disaggregated storage systems, where SANs are typically deployed, tend to be read-intensive [27]. Accordingly, this work focuses on leveraging the storage cache as a *read cache* to improve read performance.

To evaluate the benefits of storage caching, we compare the read performance of an NVMe array and a RAMDisk using the FIO microbenchmark, using the experimental setup described in Section 4. Figure 2 shows the results for random reads with varying block sizes.

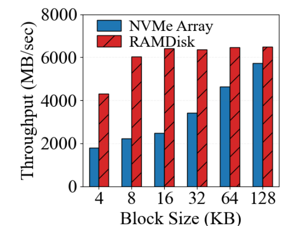


Figure 2: Read Throughput of NVMe-oF SAN

The NVMe array shows degraded performance at smaller block sizes and fails to reach peak bandwidth. In contrast,

the RAMDisk, which emulates a storage cache with memory-level access latency, delivers consistently high throughput across all sizes. These results highlight that effective use of storage caching can significantly improve application-level performance.

2.2 Hierarchical Caching in SAN

SANs exhibit a hierarchical caching architecture composed of a page cache on compute nodes and a storage cache on storage nodes. Effective multilevel caching requires careful design of both replacement and admission policies to avoid redundancy across layers. Ideally, the two cache layers should coordinate to maintain exclusivity in their contents. However, enabling such coordination adds significant complexity to both design and implementation. As a result, most existing systems [2, 3, 10, 22, 31] adopt inclusive caching, applying sophisticated replacement policies to the storage-side cache. In multilevel cache hierarchies, however, replacement policies that perform well in single-level often fail to maintain their effectiveness when layered [30]. This mismatch reduces effective cache capacity (i.e., the usable portion of the cache), and causes cache thrashing due to repeated admission and eviction of the same data blocks.

To address these challenges, several studies have proposed exclusive caching techniques [6, 9, 42]. By maintaining disjoint contents between cache layers, exclusive caching can maximize the effectiveness of multi-level cache. However, prior exclusive caching approaches often require modifications to the operating system or application on compute nodes, such as kernel patching [9, 17, 42] or application-level caching hints [29, 40, 43]. Such intrusive modifications inevitably increase system complexity and hinder maintainability. Moreover, deploying such changes across a cluster typically requires service downtime and restarts, which negatively affects availability. Exclusive caching can also incur network overhead, as evicted pages are transferred from compute nodes to the storage cache [9, 42].

3 DESIGN OF BNEX

3.1 Design Goals and Overview

We design BNEX to achieve two goals:

- **Exclusive Caching with Minimal Overhead.** BNEX implements a near-exclusive caching strategy to reduce redundancy between the compute-side and storage-side caches, with minimal network traffic during eviction-based cache admission.
- **Non-Intrusive Integration.** For seamless integration into existing SAN environments, BNEX avoids any modifications to user-space applications or kernel components on compute nodes.

Figure 3 shows the architecture of BNEX implemented in an NVMe-oF SAN. KTracer, running on the compute node,

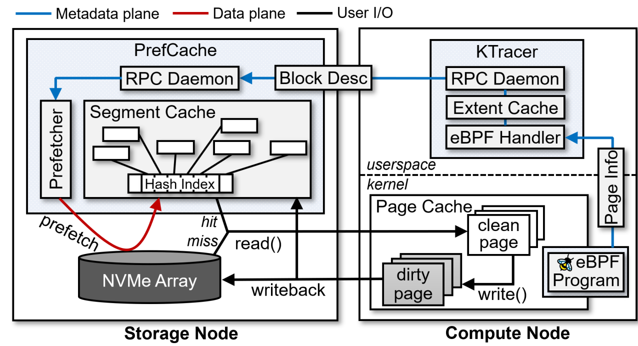


Figure 3: Architecture of BNEX

consists of an eBPF [14] program in the kernel space and a user-space daemon. The eBPF program traces file system events, including page cache replacements, and forwards the collected information to the user-space daemon for further processing. Then KTracer converts the evicted page information into block-level metadata, including block address and NVMe array information, and sends it to the storage node. On the storage node, PrefCache prefetches adjacent blocks in fixed-size units, called *Segments*, based on the received block addresses and performs cache admission accordingly.

By non-intrusively tracing the page cache on compute nodes and transmitting only lightweight metadata, BNEX decouples cache admission into metadata and data planes. Instead of transferring evicted pages over the network, it sends metadata to trigger storage-side prefetching, enabling effective near-exclusive caching while minimizing network traffic.

3.2 eBPF-based Kernel Tracer

Non-intrusive File System Tracing. We leverage eBPF to non-intrusively trace file system events related to page cache evictions for storage caching. Since eBPF programs are verified by the kernel before execution, they ensure system stability and security with minimal performance overhead. In addition, their ability to trace events at the VFS layer and compatibility across kernel versions allow them to operate regardless of the underlying file system.

Specifically, our eBPF program hooks into file system operations, such as `page_cache_delete()`, which are triggered when file-backed pages are evicted from the page cache. It selects and transmits only clean pages that have been already written to disks to prevent outdated or inconsistent pages from being fetched and used by the storage cache. The metadata about selected pages, i.e., the inode number and page index (i.e., file offset), are transferred to user space in batches to reduce communication overhead between the kernel and user space.

Mitigating Address Translation Overhead. Since prefetching on the storage node operates based on block address, the file offset of each evicted page obtained through tracing

needs to be translated into a logical block address (LBA). This is accomplished using the `FIEMAP` `ioctl()` system call [16], which retrieves the file's extent information to translate file offsets into block addresses.

However, this address translation requires a system call and can be costly, potentially becoming a bottleneck even when handled by a background daemon, as it may interfere with foreground applications on the compute node. To mitigate this overhead, KTracer employs a small LRU-based metadata cache, called the *Extent Cache*, which stores recently translated file offsets to block address mappings. The Extent Cache is indexed by inode number and maintained on a per-file basis. Each entry contains lightweight extent metadata (e.g., File Offset: 8B, LBA: 8B, Length: 4B), which enables efficient address translation with minimal memory usage.

When a file is modified and its extent mapping is updated, KTracer refreshes the corresponding entries in the Extent Cache to maintain consistency and prevent PrefCache from using stale block addresses.

Reducing Metadata Transfer. Once an evicted page is translated into a block address, a corresponding *block descriptor*, which contains the block address and NVMe array information, is sent to the storage node via RPC for prefetching. Although lightweight compared to actual data, these descriptors still incur nontrivial overhead, as RPCs involve remote communication and consume both CPU resources and network bandwidth [21, 33].

To minimize this overhead, KTracer aggregates individual block descriptors and transmits them as a batch to the storage node. In addition to simple batching, KTracer employs *Spatial Locality-aware Request Batching*, which buffers recently evicted pages and analyzes their spatial locality. When a sufficient number of adjacent block addresses are detected, the corresponding region is coalesced into a single Segment and represented by its starting block address, computed as $b - (b \bmod s)$, where b denotes the block address and s represents the Segment size. Contiguous blocks are grouped into a Segment and represented by a single block descriptor using the aligned starting address, thereby reducing the number of remote requests. If the evicted blocks are not contiguous, KTracer falls back to conventional batching. This strategy reduces RPC overhead by consolidating metadata of sequential pages into a single transmission.

As an additional optimization, KTracer monitors file access hints provided by the application via the `fadvise()` system call (e.g., `FADV_DONTNEED`, `FADV_NOREUSE`) [15]. When such hints are associated with evicted pages, KTracer suppresses the transmission of the corresponding block descriptors, further reducing unnecessary network traffic.

3.3 Prefetch-based Storage Cache

Segment-based Prefetching. Upon receiving block descriptors for evicted pages, PrefCache initiates prefetching. However, prefetching individual blocks can incur significant overhead due to random access patterns and frequent background I/O activity [5, 28]. To mitigate this, the *Prefetcher* fetches data in fixed-size, coarse-grained *Segments* (e.g., 32 KB) that contain contiguous blocks. This design assumes, optimistically, that that prefetched blocks within a Segment will exhibit spatial locality and be accessed shortly. Based on this assumption, PrefCache implements a simple, *sequential prefetching* that captures short-range locality without requiring complex history tracking. We leave history-based and workload-aware prefetching optimizations [5, 18, 26] to future work.

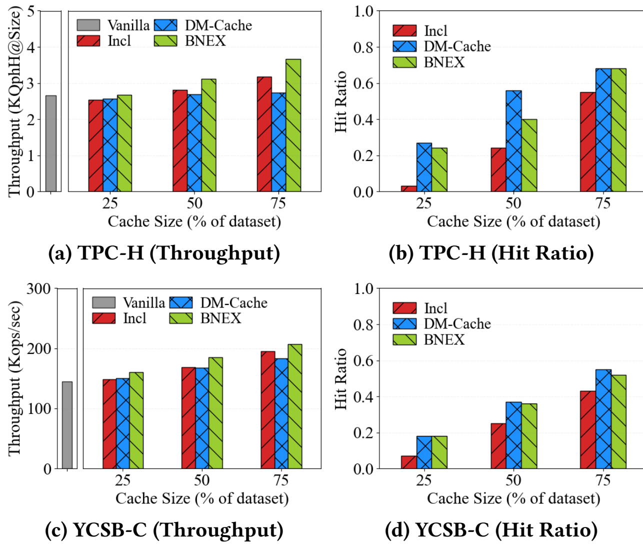
Each Segment is indexed in the *Segment Cache* using a hash table to enable fast lookups during read operations. To maintain consistency, the Segment Cache uses range locks on block address ranges to manage contention between prefetch and foreground I/O operations (i.e., user I/O requests). Additionally, write requests update the corresponding cache entries to ensure that subsequent reads return the most recent data.

To avoid interfering with foreground I/O initiated by compute nodes, the Prefetcher monitors system load conditions, such as I/O queue depth, and performs prefetching only when the NVMe array is under low load. This load-aware mechanism helps mitigate performance interference with foreground I/O.

Balancing Exclusivity and Prefetching Efficiency. PrefCache performs *segment-level prefetching* for cache admission, which introduces a trade-off between maintaining exclusivity and maximizing prefetch efficiency. To preserve exclusivity, prefetched blocks must be evicted from the storage cache once they are accessed by compute nodes. However, if the same block, or a nearby one, is later evicted again from a compute node, the system must reissue the prefetch, incurring unnecessary overhead and performance penalties.

To alleviate this inefficiency while maintaining near exclusivity, we adopt *Selective Segment Retention*, which permits limited redundancy in the cache [23, 44]. When a block within a Segment is requested by a compute node, it is served and marked as invalid. If the proportion of invalid blocks within a Segment exceeds a predefined threshold (e.g., 75%), the Segment is considered largely redundant and is proactively evicted to reclaim cache space. This eviction decision is implemented using a lightweight policy that combines toss-immediately [39] with FIFO eviction, thereby minimizing the management overhead.

While this approach does not enforce strict exclusivity, it achieves a practical balance between avoiding redundant



(a) TPC-H (Throughput)
(b) TPC-H (Hit Ratio)
(c) YCSB-C (Throughput)
(d) YCSB-C (Hit Ratio)

Figure 4: Performance Results Varying Cache Size

prefetches and maintaining reasonable exclusivity with minimal overhead.

4 EVALUATION

4.1 Experimental Setup

We evaluate the performance of BNEX using compute and storage nodes interconnected via a Mellanox FDR InfiniBand 40/56GbE switch. Each node is equipped with a single-port HCA. The compute node features two Intel Xeon Gold 5215 processors (2.5 GHz, 20 vCPUs with hyper-threading enabled, and 13.75 MB L3 cache) and 64 GB of DRAM. The storage node has the same processor, 128 GB of DRAM, and an NVMe array composed of four Samsung 970 EVO Plus M.2 250 GB SSDs. To emulate a large-scale system environment in a reduced-scale testbed, we provision the compute node with 16 GB of memory and format the NVMe volume with the EXT4 file system.

Implementation. BNEX [1] consists of two components. The compute side tracer, KTracer, is implemented in C++ using the BCC toolkit (v0.24) [7], and its eBPF programs are written in C based on the Linux kernel v5.4. We use gRPC (v1.4.1) [19] for communication between compute and storage nodes, with optional RDMA-based RPC for faster metadata transmission. On the storage node, PrefCache is integrated into PoseidonOS (v0.12.0) [36], an SPDK-based NVMe-oF storage controller, and is designed to be portable across other NVMe-oF SAN environments.

4.2 Application Performance

In the first set of experiments, we evaluate BNEX's performance with varying storage cache sizes using real-world applications to measure throughput and cache hit ratio. Here, the cache size is given as a percentage of the dataset that exceeds the compute node's memory capacity. Since BNEX is designed as a near-exclusive read cache and does not target the write path, our evaluation focuses on read-only workloads.

For comparison, we include `Incl`, an inclusive caching baseline with LRU that admits blocks to the storage cache upon read. To minimize the impact of replacement policies, we configure both `Incl` and BNEX to use similar eviction strategies. We omit prior exclusive caching techniques [6, 9, 42] in our evaluation because most of them are not publicly available. Instead, we compare against block-layer caching systems such as DM-Cache [13], which run on the compute node and use a remote RAMDisk on the storage node via NVMe-over-RDMA as a caching device. While they follow an inclusive caching model, unlike `Incl`, each block is first fetched to the compute node before being admitted to the remote caching device. Although this is not an eviction-based cache admission mechanism, we include it to evaluate the performance impact of full-page transfers over the network.

TPC-H on PostgreSQL. We deploy PostgreSQL [37] to run TPC-H [41], a widely used benchmark for OLAP workloads. In our setup, we load a 64 GB dataset, which is approximately four times larger than the memory available to the application, and execute all 22 read-only queries.

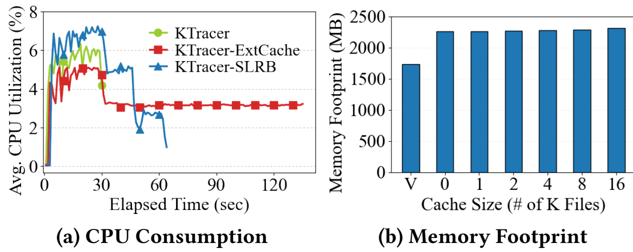
Figure 4a shows that BNEX consistently delivers the highest throughput across all cache sizes by making exclusive caching effective, outperforming `Incl` by 5–15% and `DM-Cache` by 4–33%.

As shown in Figure 4b, when the cache size is limited to 25% of the dataset size, `Incl` suffers from severe thrashing: cache hit ratio drops to 0.03, and throughput falls below that of the Vanilla (no-cache) baseline. In contrast, BNEX maintains a hit ratio that is 8× higher under the same condition. Due to the already high performance of the NVMe array, these gains do not fully translate into throughput improvements.

As the cache size exceeds 50% of the dataset, `DM-Cache` often achieves a higher hit ratio than BNEX. This is because `DM-Cache` employs the sophisticated `SMQ` (Stochastic Multi-Queue) replacement policy to prevent cache pollution. Nevertheless, its throughput falls below that of BNEX due to increased network traffic incurred by frequent cache admissions.

YCSB on RocksDB. To evaluate performance under workloads representative of cloud-based key-value systems, we deploy RocksDB [38] and run the YCSB benchmark [11]. We first load 5 million key-value pairs (approximately 64 GB) with 256-byte records, and then execute the YCSB-C workload (100% reads) using 32 client threads with a uniform access pattern.

[1]The source code of BNEX is available at https://github.com/DICL/bnex



(a) CPU Consumption (b) Memory Footprint

Figure 5: Compute Node Resource Usage

Figure 4c shows that BNEX outperforms both `Incl` and `DM-Cache` in terms of throughput, with gains ranging from 6–9% and 6–12%, respectively. As shown in Figure 4d, BNEX achieves 1.2–2.5× higher cache hit ratio than the `Incl`, similar to the trend observed in the TPC-H workload.

Since each 4 KB page contains multiple key-value records, pages are often accessed repeatedly during the workload. However, the re-reference rate is lower than in TPC-H, which limits the performance gains from caching.

4.3 Resource Overhead Analysis

BNEX tracks evicted pages from the compute node's page cache and forwards their metadata to the storage node to trigger prefetching. This design improves cache efficiency by eliminating redundancy across cache layers, but it introduces additional CPU overhead on the compute node and increases network traffic between the compute and storage nodes.

Compute Node. Figure 5a shows the average CPU utilization of KTracer during a 30 second random read workload using FIO, where page evictions occur continuously. We compare KTracer with two variants: `KTracer-ExtCache`, which disables the *Extent Cache* and performs a `FIEMAP ioctl()` system call for each block translation, and `KTracer-SLRB`, which enables the Extent Cache but disables *Spatial Locality-aware Request Batching*, transmitting all evicted block addresses to the storage node without considering page contiguity.

KTracer completes tracing almost immediately after the FIO job finishes. In contrast, `KTracer-ExtCache` and `KTracer-SLRB` continue processing after the workload ends, due to expensive address translation and RPC transmissions, respectively. `KTracer-ExtCache` incurs the highest overall CPU usage due to inefficient block address translation. Although KTracer shows a slightly higher peak CPU usage than `KTracer-ExtCache`, this is because it buffers evicted page metadata and performs sorting and grouping to identify sequential pages. By reducing unnecessary metadata transfers, KTracer lowers its overall CPU usage by 47% compared to `KTracer-SLRB`. Running as a background daemon, KTracer maintains an average CPU utilization of only 5.1%, indicating negligible interference with foreground applications.

Figure 5b shows the memory footprint of KTracer as the size of the Extent Cache varies. The dataset consists of 20,000 files, each 16 MB in size, stored in one or more extents based on the file system's allocation policy. Since KTracer uses eBPF for kernel tracing, it consumes approximately 11% more memory than the Vanilla baseline (`V`), due to the overhead of maintaining eBPF program state and user-kernel communication buffers. Although memory usage increases with the number of cached files, the extent-based metadata structure enables efficient representation. Even with 16 K cached entries, KTracer uses only 2% more memory compared to the configuration in which the Extent Cache is disabled (i.e., cache size = 0).

Inter-Node. Figure 6 shows the normalized network traffic overhead incurred when evicted page metadata, including block addresses, is transmitted via RPC from the compute node to the storage node during microbenchmark execution. `PageTransfer` models a conventional exclusive caching scheme in which entire evicted pages are directly transferred to the storage node. To estimate its network usage, we derive total traffic as the number of evicted pages multiplied by the page size. BNEX transmits only lightweight metadata containing block addresses, resulting in 1.52× higher traffic than the Vanilla baseline. However, `PageTransfer` incurs 233× more network traffic than BNEX, as it transfers the full content of each evicted page. These results indicate that BNEX substantially reduces network overhead while preserving cache exclusivity. By avoiding full-page transfers and eliminating redundancy across cache layers, BNEX achieves more efficient coordination between compute and storage nodes.

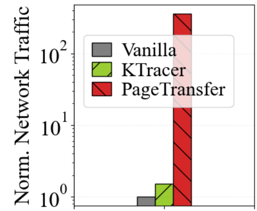


Figure 6: Inter-Node Network Traffic

5 CONCLUSION

In this work, we present BNEX, a non-intrusive near-exclusive caching system for NVMe-oF SANs that eliminates redundancy between cache layers and reduces network traffic through metadata-driven design during cache admission. To achieve this, BNEX employs KTracer, an eBPF-based kernel tracer that tracks page cache evictions and sends only lightweight block-level metadata to the storage node. On the storage side, PrefCache performs prefetch-based caching by leveraging spatial locality and grouping blocks near evicted pages, improving prefetching efficiency while preserving cache exclusivity. Our performance study shows that BNEX improves throughput and cache hit ratio by up to 1.15× and 8×, respectively, compared to traditional inclusive caching systems.

6 ACKNOWLEDGMENT

This work was supported by Samsung Electronics. The corresponding author is Beomseok Nam.

REFERENCES

[1] Optimizing VMware vSAN with Supermicro All-Flash Intel NVMe Systems. https://www.supermicro.org.cn/white_paper/white_paper_Vitual_SAN.pdf.

[2] Mohammadamin Ajdari, Pouria Peykani Sani, Amirhossein Moradi, Masoud Khanalizadeh Imani, Amir Hossein Bazkhanei, and Hossein Asadi. Re-architecting I/O Caches for Emerging Fast Storage Devices. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 542–555, 2023.

[3] I. Ari, M. Gottwals, and D. Henze. Sanboost: automated san-level caching in storage area network. In *Proceedings of International Conference on Autonomic Computing*, pages 164–171, 2004.

[4] Azure Elastic SAN. https://azure.microsoft.com/en-us/products/storage/elastic-san.

[5] Sung Hoon Baek and Kyu Ho Park. Prefetching with Adaptive Cache Culling for Striped Disk Arrays. In *Proceedings of the 2008 USENIX Annual Technical Conference (USENIX ATC)*, 2008.

[6] Lakshmi N Bairavasundaram, Muthian Sivathanu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs. *ACM SIGARCH Computer Architecture News*, 32(2):176, 2004.

[7] BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more. https://github.com/iovisor/bcc.

[8] Peter M Chen and Edward K Lee. Striping in a RAID level 5 disk array. In *Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 136–145, 1995.

[9] Zhifeng Chen, Yuanyuan Zhou, and Kai Li. Eviction-based Cache Placement for Storage Caches. In *USENIX Annual Technical Conference, General Track*, pages 269–281, 2003.

[10] Citrix Hypervisor - Storage read caching. https://docs.xenserver.com/en-us/citrix-hypervisor/storage/read-cache.html.

[11] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*, pages 143–154, 2010.

[12] Dell: Next-Generation PowerMax Family Overview. https://www.delltechnologies.com/asset/ru-ru/products/storage/industry-market/h19312-next-generation-powermax-wp.pdf.

[13] DM-Cache. https://www.kernel.org/doc/Documentation/device-mapper/cache.txt.

[14] eBPF - Introduction, Tutorials & Community Resource. https://ebpf.io.

[15] fadvise64,fadvise64_64 - Give advice about file access. https://linux.die.net/man/2/fadvise.

[16] Fiemap Ioctl. https://docs.kernel.org/filesystems/fiemap.html, 2025.

[17] Binny S Gill. On Multi-level Exclusive Caching: Offline Optimality and Why Promotions Are Better Than Demotions. In *FAST*, volume 8, pages 1–17, 2008.

[18] Binny S Gill and Dharmendra S Modha. SARC: Sequential Prefetching in Adaptive Replacement Cache. In *USENIX Annual Technical Conference, General Track*, pages 293–308, 2005.

[19] The C based gRPC (C++, Python, Ruby, Objective-C, PHP, C). https://github.com/grpc/grpc.

[20] Gabriel Haas, Michael Haubenschild, and Viktor Leis. Exploiting Directly-Attached NVMe Arrays in DBMS. In *CIDR*, volume 20, page 2, 2020.

[21] Daegyu Han, Sungho Moon, Kyeungpyo Kim, Sung-Soon Park, and Beomseok Nam. Improving Remote File Access in Distributed Object Stores by Decoupling Metadata and Data Paths using NVMe-oF. *IEEE Micro*, 2025.

[22] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Zhenlin Wang, Chen Ding, and Chencheng Ye. Fast miss ratio curve modeling for storage cache. *ACM Transactions on Storage (TOS)*, 14(2):1–34, 2018.

[23] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C Steely Jr, and Joel Emer. Achieving Non-Inclusive Cache Performance with Inclusive Caches: Temporal Locality Aware (TLA) Cache Management Policies. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 151–162. IEEE, 2010.

[24] Young Tack Jin, Sungjoon Ahn, and Sungjin Lee. Performance Analysis of NVMe SSD-based All-flash Array Systems. In *2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 12–21. IEEE, 2018.

[25] Yuhun Jun, Shinhyun Park, Jeong-Uk Kang, Sang-Hoon Kim, and Euiseong Seo. We Ain't Afraid of No File Fragmentation: Causes and Prevention of Its Performance Impact on Modern Flash SSDs. In *Proceedings of the 22nd USENIX Conference on File and Storage Technologies (FAST)*, pages 193–208, 2024.

[26] Tom M. Kroeger and Darrell D. E. Long. Design and Implementation of a Predictive File Prefetching Algorithm. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, page 105–118, USA, 2001. USENIX Association.

[27] Changji Li, Hongzhi Chen, Shuai Zhang, Yingqian Hu, Chao Chen, Zhenjie Zhang, Meng Li, Xiangchen Li, Dongqing Han, Xiaohui Chen, et al. ByteGraph: a high-performance distributed graph database in ByteDance. *Proceedings of the VLDB Endowment*, 15(12):3306–3318, 2022.

[28] Chun-Yi Liu, Jagadish B Kotra, Myoungsoo Jung, Mahmut T Kandemir, and Chita R Das. SOML read: Rethinking the read operation granularity of 3D NAND SSDs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 955–969, 2019.

[29] Xin Liu, Ashraf Aboulnaga, Kenneth Salem, and Xuhui Li. CLIC: CLient-Informed Caching for Storage Servers. In *FAST*, pages 297–310, 2009.

[30] Daniel Muntz and Peter Honeyman. Multi-level caching in distributed file systems. Technical report, Center for Information Technology Integration, 1991.

[31] Flash Cache Best Practice Guide. https://www.netapp.com/pdf.html?item=/media/19754-tr-3832.pdf&v=202010152013.

[32] NetApp: Best practices for modern SAN ONTAP 9. https://www.netapp.com/media/10680-tr4080.pdf.

[33] Tuan Anh Nguyen, Hyeongjun Jeon, Daegyu Han, Duck-Ho Bae, Young Jin Yu, Kyeungpyo Kim, Sungsoon Park, Jinkyu Jeong, and Beomseok Nam. Nvme-driven lazy cache coherence for immutable data with nvme over fabrics. In *Proceedings of the 16th International Conference on Cloud Computing (IEEE CLOUD)*, pages 394–400. IEEE, 2023.

[34] NVM Express over Fabrics. https://nvmexpress.org/wp-content/uploads/NVMe-over-Fabrics-1.1a-2021.07.12-Ratified.pdf.

[35] Chanhyun Park, Seongjin Lee, and Youjip Won. An Analysis on Empirical Performance of SSD-Based RAID. In *Information Sciences and Systems 2014: Proceedings of the 29th International Symposium on Computer and Information Sciences*, pages 395–405. Springer, 2014.

[36] PoseidonOS. https://poseidonos.io.

[37] Mirror of the official PostgreSQL GIT repository. https://github.com/postgres/postgres.

[38] A library that provides an embeddable, persistent key-value store for fast storage. https://github.com/facebook/rocksdb.

[39] Abraham Silberschatz, Henry F. Korth, and S. Sudershan. *Database System Concepts*. McGraw-Hill, Inc., USA, 3rd edition, 1998.

[40] Gokul Soundararajan, Madalin Mihailescu, and Cristiana Amza. Context-Aware Prefetching at the Storage Server. In *2008 USENIX*

Annual Technical Conference (USENIX ATC 08), 2008.

[41] Swarm64 DA Benchmark Toolkit. https://github.com/swarm64/s64da-benchmark-toolkit.

[42] Theodore M Wong and John Wilkes. My cache or yours?: Making storage more exclusive. In *USENIX Annual Technical Conference, General Track*, pages 161–175, 2002.

[43] Gala Yadgar, Michael Factor, and Assaf Schuster. Karma: Know-It-All Replacement for a Multilevel Cache. In *Fast*, volume 7, pages 169–184, 2007.

[44] Mohamed Zahran, Kursad Albayraktaroglu, and Manoj Franklin. Non-inclusion property in multi-level caches revisited. *International Journal of Computers and Their Applications*, 14(2):99, 2007.