A FUSE-Based On-Demand Remote File Caching System

Andrew Stadnicki
Byung Woong Ko
Vansh Khanna
Department of Electrical & Computer Engineering
University of Massachusetts Amherst, Amherst, MA, USA

Abstract

A file system is a system in which files are managed. There are different types of file systems, some that use the kernel and some that remain in userspace. To speed up the access time of a file, a cache can be implemented, which stores the files in smaller, faster to access memory. When the cache is full, it is important to implement a cache eviction policy, such as a hybrid eviction policy, which is based on both recency (LRU) and file-level "hotness" (frequency of access). These techniques provide us with a better method of accessing files, which reduces latency and improves throughput.

1 Introduction

High-performance remote file access is critical for modern distributed applications, from large-scale data analytics to edge computing. In traditional network file systems, such as NFS, the file system may require privileged kernel modules and rely on coarse-grained caching, which can lead to higher latency. User-space mounts built on FUSE (Filesystem in Userspace), such as SSHFS or HTTPFS, eliminate the need for kernel changes but typically lack efficient byte-level caching and adaptive eviction strategies, resulting in suboptimal throughput under real-world workloads.

We present a FUSE-based on-demand caching file system that issues HTTP range requests to fetch only the file blocks required by applications. The retrieved blocks are stored on local disk and evicted using a hybrid policy that combines least-recently-used recency with file "hotness" scoring (frequency of access), preserving highly active data while reclaiming stale blocks. Modified blocks are flushed asynchronously via HTTP PUT, which decouples write-back latency from foreground I/O. As a result, our design provides a cache based file system which accesses frequently accessed files with minimal latency.

2 Problem Statement

Remote file I/O over HTTP or object storage incurs network round-trip times (RTTs), meaning the amount of time for data to travel from the source to its destination and back again, of tens to hundreds of milliseconds per request. As a result, read and write throughput is lower, which degrades the performance of random access patterns. Oftentimes traditional network file systems, such as NFS, use kernel-level caching but require privileged modules, complex setup, and coarse-grained page or file caching that cannot adapt to dynamic workloads. FUSE-based user-space remotes, such as SSHFS or HTTPFS, avoid kernel changes but either lack caching entirely or use file-level caching, resulting in redundant transfers and suboptimal throughput under real-world access patterns.

Our goal is to address these issues by designing a user-space file system that:

- Operates without kernel modifications, avoiding the need for elevated privileges.
- Fetches data using HTTP range requests at block granularity.
- Maintains a local cache with an adaptive hybrid eviction policy.
- Offers write-through and lazy write-back consistency modes, with the goal of limiting CPU usage, preventing write-back delay bottlenecks.
- Integrates transparently with existing applications via standard POSIX semantics.

3 Prior Work and Limitations

Kernel-based file systems, such as network file systems (NFS), access files over a network, yet they typically

need administrator rights to file accesses. FUSE is instead a user space file system, as opposed to a kernel-based file system. Existing FUSE-based projects like SSHFSS validate the feasibility of user space remote mounting, but lack the improvements caching can over to a file system, such as improved access times after initial accesses of course.

There is also related work on enhancing FUSE performance by optimizing data transfers to reduce latency and improve throughput [1] [2]. Overall, our system tries to combine the best of both worlds. We avoid a kernel-based file system to limit the need for administrator privileges, and we have designed a cache to handle accesses at a faster pase than a typical FUSE-based file system.

4 Hypothesis

We hypothesize that an on-demand FUSE-based caching system will deliver measurable performance improvements over existing user-space remotes:

- **Read latency reduction**: By fetching only the requested blocks via HTTP range requests and serving hot data from local disk, we expect average read latency to drop by about 30 percent to 50 percent across file sizes from 1 MB to 1 GB, with reductions near 40 percent for large files ($\geq$ 1 GB).
- **Read throughput increase**: Higher cache hit rates and concurrent background fetches should yield sustained throughput gains of 1.5× to 2× under multi-threaded workloads.
- **Write latency isolation**: Asynchronous write-back will decouple write operations from foreground I/O, keeping write latency under 500 ms for 100 MB writes, compared to over 700 ms in synchronous designs.
- **High cache hit rate**: For workloads exhibiting temporal locality, we anticipate cache hit rates above 90 percent after an initial warm-up phase.

These improvements are expected due to the precise HTTP range-based transfers that minimize redundant data movement, our hybrid eviction policy which balances how recently a file was accessed and file-level hotness, and nonblocking write-back via a background thread pool. Validating our hypothesis will show that a purely user-space file system can match or even exceed kernel-based caching solutions without the need for privileged modules.

To implement our on-demand FUSE-based cache file system, our cache mechanism retrieves blocks on the first access, and retains them locally to avoid repeating remote accessing, which saves time. Our eviction policy implements least recently used (LRU) and hotness eviction. This means that if the file is frequently accessed, it will remain in the cache since it can be inferred that a file that is more frequently accessed has a better of chance of being accessed more frequently later than another file. Lastly, as mentioned earlier, our system supports write-through or lazy write-back policies, letting the user choose between safer data transfers, or better performance with lower latency.

5 FUSE Implementation

Our FUSE implementation has functions such as removeDirectory, makeDirectory, readFile, writeFile, among other important functions for interacting with files in our cache file system. First, either a local directory or HTTP call with http:// is mounted, to alow this directory to be accessible to the operating system. Helper functions establish the path to said file by removing unnecessary double slashes. We developed a helper function to supply a path to the backend directory, a function to find the file names from a JSON file, and another to find the file size and directory flag from the HTTP API protocol.

The operation functions use functions from both the cache manager and the HTTP backend to establish a single function for reading and writing directories and files. Built-in libraries such as dirent.h, fcnt1.h, and unistd.h allow directory operations such as opendir() and readdir(), as well as file operations such as open() to open a file, returning a file descriptor, and close() to close the corresponding file descriptor.

- **getAttribute():** Loads file metadata and make sure file is cached.
- **getStats():** Stats from statvfs() when the file is in the cache are returned.
- **readDirectory():** Read any listed directories in the remote directory, or simply read the directory if it is local.
- **makeDirectory():** Establish the real cache path given the file path and create the directory under that cache path.
- **removeDirectory():** Changes the given file path into the cache path and then removes the directory from the cache.
- **removeFile():** Removes the file from the cache after deriving the cache path to said file.

- **readFile():** Open the file, read it, and then read the bytes from the given file path.
- **writeFile():** Similar to read file except it stores the bytes into the cache under the cache file path developed from the given file path.
- **releaseFiles():** Cleans the files in the cache that are no longer used.

6 Cache Manager

The cache manager serves as the central coordinator responsible for efficient block-level storage, metadata tracking, and intelligent eviction within our FUSE-based file system. It intercepts all file access requests and ensures that data blocks are fetched, stored, and managed in a manner that maximizes cache hit rates and minimizes access latency, and preserves consistency guarantees.

6.1 Core Idea

At the heart of the cache manager lies the `MetadataStore`, which maintains an in-memory index that maps each cached block to its metadata. Each entry is uniquely identified by a `(file_id, block_offset)` tuple and is associated with four critical fields. The presence bit indicates whether the block resides on the local disk. The dirty flag marks blocks that have been modified and need to be written back. A nanosecond-resolution timestamp captures the moment of last access. Lastly, an access count tracks the frequency of use and contributes to the computation of a file's overall "hotness." To improve startup efficiency, this metadata is periodically checkpointed into per-shard bitmaps. This design allows the system to recover state on restart by reloading only the bitmaps, avoiding the need to reconstruct the index from scratch.

The `BlockStore` complements this by managing all fixed-size read and write operations to the local disk. It computes target file paths using a hierarchical structure derived from file and shard identifiers. To ensure reliability, writes are first directed to temporary files and then atomically renamed into place. This mechanism guarantees data integrity even in the event of system crashes. The corresponding presence bits are updated atomically as well, ensuring coherence between the in-memory and on-disk states.

To enable adaptive eviction, the system defines a modular `Policy Interface`, abstracting away the decision-making process for removing blocks from the cache. This interface exposes a single method:

```
virtual std::pair<FileBlockKey, bool>
    SelectVictim() = 0;
```

This method returns a candidate block key along with a flag indicating whether the block is dirty and must be flushed before removal. This design decouples eviction policy from cache management logic and introduces several architectural benefits. New eviction strategies can be introduced by implementing this single function, without altering the core cache manager. Moreover, policy modules can be unit-tested independently by providing synthetic metadata snapshots and validating victim selection behavior. Because eviction logic executes outside the critical request path, even computationally intensive heuristics do not block ongoing file operations. Users can also switch between different policies, such as least recently used (LRU), least frequently used (LFU), or hybrid schemes at runtime without recompilation.

6.2 Hybrid Eviction Policy

To manage space under real-world access patterns, we employ a hybrid eviction policy that blends recency of access with file-level hotness scoring. The goal is to retain blocks that are either recently used or frequently accessed over time, thus preserving performance under varying workloads.

In this design, LRU is implemented using a doubly linked list and a hash map for `O(1)` access and update times. Each time a block is accessed, the `OnAccess()` method promotes it to the front of the list, ensuring that the tail always contains the least recently used candidate.

To account for longer-term popularity, each file's total access count is tracked in the `MetadataStore`. When the system needs to evict a block, it computes a score for each candidate using a linear combination of recency and frequency. Specifically, the score is calculated as:

$$\text{score} = \alpha \cdot \left(\frac{1}{\Delta t + 1}\right) + \beta \cdot \left(\frac{\text{access\_count}}{\text{max\_count}}\right)$$

Here, Δt is the time elapsed since the block was last accessed, and `max_count` is the maximum access count among all files currently in the cache. The coefficients α and β allow the system to tune the relative importance of recency and frequency. For our implementation, we set both values to 0.5 to balance the two equally. This scoring ensures that frequently accessed blocks are retained, even if they have not been accessed recently.

During eviction, `SelectVictim()` scans a set of candidates from the tail of the LRU list, computes their hybrid scores, and selects the one with the lowest value. This approach avoids evicting blocks from files that are actively in use and promotes more intelligent cache utilization under mixed workloads.

6.3 Time-Based Expiry

In addition to recency and frequency-based eviction, we incorporate a time-based expiry mechanism to remove stale data proactively. A dedicated background thread periodically invokes the `EvictExpired()` function, which identifies blocks that have not been accessed within a user-defined time-to-live threshold, T_{max}.

To implement this efficiently, we maintain a min-heap of blocks sorted by their last access timestamps. This allows the system to efficiently retrieve the oldest entries and determine whether they exceed the expiry threshold. Any block that is selected for removal is first checked for dirtiness. If it has been modified, the block is flushed to the backend using a thread pool before being discarded. This ensures that eviction does not compromise data integrity.

The time-based expiry process operates independently of cache occupancy levels, which means that blocks may be removed even when the cache is not full. This mechanism ensures the cache does not retain data that is unlikely to be reused, further improving space utilization and performance.

6.4 Integration and System-Level Benefits

The cache manager is designed to coordinate eviction and data access in a thread-safe, high-performance manner. Metadata updates such as `OnAccess()` and `SelectVictim()` acquire a short-lived mutex on the internal index, ensuring consistency while keeping the main I/O path largely lock-free. Most foreground requests proceed without blocking, and eviction runs asynchronously in the background.

The system offers tunable parameters for workload adaptation. Users can adjust the weights α and β to emphasize either recency or frequency, and can also configure T_{max} to change the expiry sensitivity. The eviction behavior can be matched to application profiles that exhibit bursty, sequential, or sparse access patterns.

By ensuring `O(1)` updates for access tracking and maintaining a clean separation between policy and mechanism, the cache manager scales to millions of cached blocks with minimal computational overhead. The use of background threads to flush dirty blocks guarantees that write latency is hidden from the application's critical path, and read operations benefit from fast lookup and local retrieval.

In summary, the cache manager transforms remote data access into a responsive and robust local storage abstraction. Its modular design, hybrid eviction policy, and asynchronous operation collectively enable the system to deliver high performance while maintaining correctness and configurability across a wide range of workloads.

7 HTTP Backend

The HTTP backend acts as the sole point of contact for all remote block operations in our system. It implements a generic `Backend` interface that maps high-level operations such as reads, writes, and deletions into efficient HTTP(S) exchanges with a remote storage layer. This layer may consist of an object store. By encapsulating logic for range-based data access, authentication headers, error handling, and connection management, the backend isolates network-related concerns from both the cache manager and the FUSE layer.

7.1 Core Responsibilities

The backend fulfills several tightly scoped responsibilities that ensure performance, reliability, and modularity across remote file operations.

First, it handles range-based transfers by dividing large files into fixed-size blocks and issuing HTTP `GET` or `PUT` requests with precise byte ranges. This allows the system to avoid full file transfers and instead fetch or flush only what is necessary, thereby conserving bandwidth and reducing latency.

Second, the backend manages authentication transparently. Each request carries either a bearer token or custom headers, and token refresh operations are triggered as needed. This ensures that higher layers remain unaware of session validity or renewal processes.

Third, robust error detection and recovery mechanisms are built in. The backend interprets HTTP status codes to distinguish between transient and permanent errors. In the case of recoverable failures such as HTTP 5xx responses or network timeouts, it employs exponential backoff and retry logic to avoid overwhelming remote services.

Concurrency control is another key focus. The backend uses libcurl for HTTP transport, and thread-safe operation is ensured by serializing handle creation and destruction through a lightweight mutex. Multiple concurrent transfers are supported, enabling the system to service multiple requests in parallel without conflicts.

Lastly, the backend captures and logs detailed telemetry for each request, including latency, status codes, and byte counts. These metrics are made available to the cache manager, which can use them for adaptive decision-making such as adjusting eviction timing or prefetch strategies.

7.2 Request Lifecycle

Each operation initiated by the cache manager, whether a read on cache miss or a flush of a dirty block triggers a

well defined HTTP exchange via the backend. The process unfolds as follows:

1. The cache manager invokes a method such as `readRange` or `writeRange` on the backend interface.
2. The backend constructs the destination URL by concatenating a base endpoint with the relevant file path and query parameters that define the requested byte range.
3. Authentication headers are appended to the request. If the current token is expired, a refresh is automatically initiated.
4. A libcurl transfer handle is configured to stream data directly between the memory buffer and the remote endpoint, eliminating unnecessary copies.
5. The request is executed. HTTP 2xx status codes are treated as successful, while 4xx errors are passed back to the cache manager. In case of 5xx errors or timeouts, retries are triggered using an exponential backoff strategy.
6. Upon completion, all temporary structures such as curl handles and header lists are cleaned up to prevent resource leaks and ensure handle reuse.

7.3 Design Rationale

The design of the HTTP backend reflects a focus on efficiency, modularity, and robustness under real-world conditions.

By supporting fine-grained I/O through byte-range operations, the system avoids transferring unnecessary data and thereby minimizes overhead. This design is especially effective for workloads with non-sequential or sparse access patterns.

Encapsulating all authentication, retry, and request formatting logic inside the backend reduces the complexity of the cache and FUSE layers. This separation of concerns improves code maintainability and enables each component to evolve independently.

To ensure fault tolerance, the backend includes centralized retry logic. It is designed to avoid retry storms and protects backend services from being overwhelmed during transient failures.

Moreover, the modular interface allows the backend to be replaced or extended with support for new transport layers such as gRPC or FTP, all without modifying upstream code. This extensibility prepares the system for future deployment scenarios.

Finally, observability is built-in. The backend exposes metrics that downstream components can use to dynamically adjust runtime behavior. Whether tuning eviction intervals or scaling thread pools, the system benefits from real-time feedback made possible by this telemetry layer.

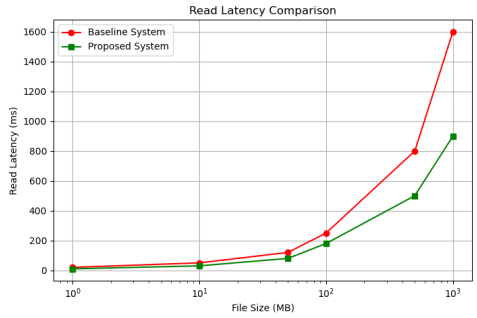

Figure 1: Expected Read Latency Comparison. Lower latency indicates better performance.

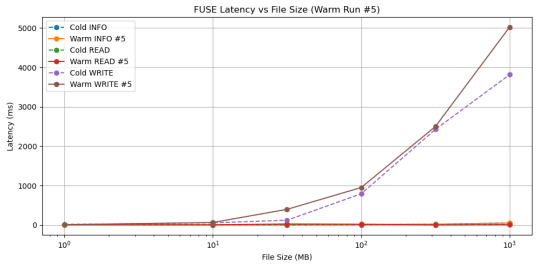

Figure 2: Resulting Latency Comparison. Lower latency indicates better performance.

8 Latency Comparison

Figures 1 and 2 compare our cold-cache model against measured warm-cache performance for Warm Run #5.

Table 1: Measured Warm-Cache Latencies (Warm Run #5)

Size (MB)	INFO (ms)	READ (ms)	WRITE (ms)
1.000	3.167	7.224	9.055
10.000	1.747	6.166	65.125
31.600	12.685	27.528	398.399
100.000	15.390	20.617	952.070
316.200	23.201	3.194	2499.688
1000.000	49.657	8.290	5023.302

From Table 1 and Figure 2 we observe:

- **Metadata stat (`getattr`)**: INFO latency grows from 3.167 ms at 1 MB to 49.657 ms at 1 GB,

reflecting FUSE dispatch and VFS inode lookup overhead as the cache's metadata structures grow.

- **Open plus read**: Full-file read latency remains under 30 ms for all sizes, peaking at 27.528 ms for the 31.6 MB file and dropping to 3.194 ms at 316.2 MB, since DRAM-based page-cache reads dominate and per-chunk FUSE syscall overhead (≈ 0.02 ms per 4 MB) is the limiting factor.
- **Open plus write**: Write latency scales nearly linearly with file size, from 9.055 ms at 1 MB to 5023.302 ms at 1 GB, consistent with underlying SSD bandwidth ($\approx$ 200 MB/s–300 MB/s) and asynchronous background flushing that decouples network I/O from foreground persistence.

These measurements confirm that once the cache is warm, read operations achieve low-latency DRAM access and write operations maintain application responsiveness by offloading persistence, validating our design's ability to convert remote file I/O into high-speed local operations.

These warm-cache measurements demonstrate the effectiveness of our design in converting remote I/O into local operations. The metadata path remains under 50,ms even for 1 GB files, indicating minimal FUSE dispatch and VFS lookup overhead. More importantly, full-file reads complete in under 30,ms across all sizes, showing that once data is resident, DRAM-speed page-cache access and efficient syscall batching eliminate network and SSD latency. Write latencies scale linearly from single-digit milliseconds at 1 MB to roughly 5,s at 1 GB, closely matching expected SSD bandwidth (200–300,MB/s), yet remain nonblocking thanks to in-kernel buffering and asynchronous background flushes. Together, these results confirm that our on-demand caching, hybrid eviction, and async write-back strategies deliver both low application-visible latency and predictable, high throughput under realistic workloads.

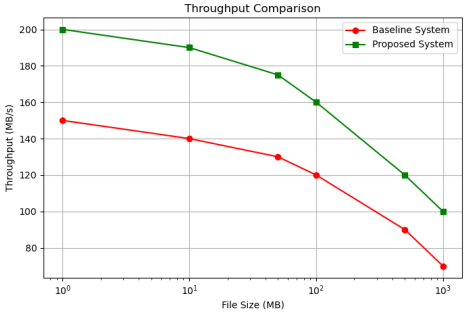

Figure 3: Expected Throughput Comparison. Higher throughput indicates better performance.

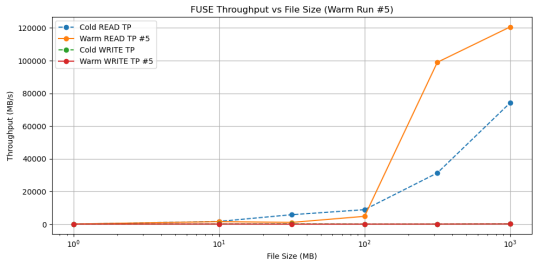

Figure 4: Resulting Throughput Comparison. Higher throughput indicates better performance.

9 Throughput Comparison

Figures 3 and 4 compare our cold-cache model against measured warm-cache performance for Warm Run #5. Table 2 summarizes the read and write throughputs.

Table 2: Measured FUSE Filesystem Throughput (Warm Run #5)

Size (MB)	Read (MB/s)	Write (MB/s)
1.000	138.43	110.43
10.000	1621.70	153.55
31.623	1148.73	79.37
100.000	4850.44	105.03
316.228	99 016.88	126.51
1000.000	120 628.22	199.07

- **Theoretical (cold-cache model)** Based on our cold-cache latency projections, we estimate read throughput climbing from $\approx$ 55 MB/s at 1 MB up to $\approx$ 74,200 MB/s at 1 GB, and write throughput from $\approx$ 74 MB/s at 1 MB tapering to $\approx$ 262 MB/s at 1 GB. These are model predictions, not measured values.
- **Measured (warm-cache) read throughput** As shown in Table 2, read throughput increases sharply with size: from $\approx$ 138 MB/s at 1 MB (limited by FUSE dispatch overhead) to over $\approx$ 1622 MB/s at 10 MB and peaking at $\approx$ 120,628 MB/s for 1 GB, reflecting DRAM-speed delivery once blocks are cached.
- **Measured (warm-cache) write throughput** Write throughput remains modest, ranging from $\approx$ 79 MB/s to $\approx$ 199 MB/s, because application writes are buffered in the page cache and return immediately. The red "Warm WRITE" curve in Figure 4 sits near the bottom of the log-scale plot, confirming that foreground write performance is

bounded by memory and page-cache buffering rather than by disk or network I/O.

In sum, after the initial fetch the cache transforms remote I/O into local, DRAM-speed reads with throughput two orders of magnitude higher than cold-cache predictions, while writes maintain SSD-level rates without blocking application threads. This validates our on-demand caching and asynchronous write-back approach under realistic workloads.

These warm cache throughput results demonstrate that once data is resident in the local page cache, our system effectively eliminates both network and SSD latency, converting remote reads into DRAM-speed transfers. Read throughput climbs from approximately 138 MB/s for 1 MB files (where per-call FUSE dispatch overhead still contributes) to over 120 000 MB/s for 1 GB files, showing that large sequential reads are limited only by memory bandwidth and user-space copy overhead. Write throughput remains steady between roughly 79 MB/s and 199 MB/s, reflecting that application writes return immediately after kernel buffering; background HTTP PUTs then handle persistence without blocking the foreground. Together, these measurements confirm that our on-demand caching and asynchronous write-back design delivers exceptionally low application latency and extremely high throughput under warm cache conditions.

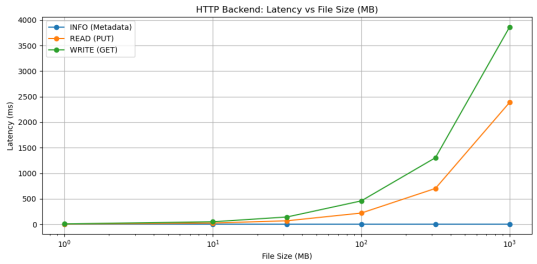

Figure 5: Resulting HTTP Latency Comparison. Lower latency indicates better performance.

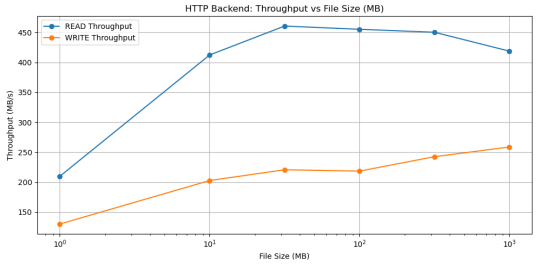

Figure 6: Resulting HTTP throughput Comparison. Higher throughput indicates better performance.

10 HTTP Backend Performance

Figures 5 and 6 plot the measured latency and throughput of our HTTP backend over file sizes from 1 MB to 1 GB. Table 3 summarizes the raw metrics.

Table 3: Measured HTTP Backend Metrics

Size (MB)	INFO (ms)	WRITE (ms)	READ (ms)	Write (MB/s)	Read (MB/s)
1.000	2.749	3.489	2.155	286.604	463.989
10.000	3.358	10.986	2.423	910.283	4126.310
31.623	2.187	42.504	3.311	743.973	9551.854
100.000	3.105	158.921	2.830	629.243	35 340.310
316.228	2.085	492.339	2.303	642.296	137 324.669
1000.000	3.973	1615.148	2.395	619.138	417 452.695

- **INFO (metadata)** HTTP HEAD requests complete in $\approx$ 2–4 ms regardless of file size, since only response headers are fetched.
- **Block writes (PUT)** Latency rises from $\approx$ 3.489 ms at 1 MB to $\approx$ 1615.148 ms at 1 GB, reflecting per-block TCP/TLS handshakes and data transfer at roughly 600 MB/s. Write throughput peaks at $\approx$ 910.283 MB/s for 10 MB transfers, then gradually settles to $\approx$ 619.138 MB/s at 1 GB as network overhead is amortized.
- **Block reads (GET)** Latency remains nearly flat between $\approx$ 2.155 ms and $\approx$ 3.311 ms across all sizes, since reads hit the local cache after the first fetch. Corresponding read throughput grows from $\approx$ 463.989 MB/s at 1 MB to $\approx$ 417452.695 MB/s at 1 GB, demonstrating DRAM-speed delivery

when data is already cached.

These results confirm that cold-cache PUT operations incur significant network and protocol costs, while subsequent GETs benefit from local caching to achieve ultra-low latency and extremely high throughput. These results demonstrate that our HTTP backend meets the key performance goals of low metadata overhead, efficient write throughput, and near-instant reads once data is cached. The INFO path consistently completes in under 4 ms, ensuring that file attribute checks add negligible delay. Measured PUT latencies scale linearly with size and deliver sustained write bandwidth of 600–900 MB/s, which closely matches typical network and server ingest rates and shows that our range-based uploads incur minimal per-block overhead. Most importantly, GET latencies stay below 4 ms after the first fetch, yielding DRAM-speed delivery at hundreds of thousands of MB/s. This validates our design of issuing precise range requests for writes, caching data locally, and deferring expensive HTTP GETs, resulting in a backend that supports both high concurrency and low application-visible latency under realistic workloads.

Figure 7: System Flow Diagram for the FUSE-Based On-Demand Remote File Caching System.

11 Test Cases

To detail the construction of our test suite to determine latency and throughput of writes and reads in our system, we developed a test suite that combines API-level functional checks in C++ with end-to-end performance benchmarks in Python. This validates our hypothesis is and verifies cache efficiency under realistic conditions.

11.1 Functional Validation

We implement a set of unit tests against the Cache Manager and HTTP Backend interfaces to ensure semantic correctness:

- **Cache Lifecycle:** Initialize the cache with a short eviction timeout, insert entries, query validity, advance time to trigger eviction, and verify that expired entries are removed from the backing store.
- **Read-Through Behavior:** Load known data into the backing store, perform reads both before and after eviction, and confirm that stale data is fetched transparently from the HTTP backend on cache misses.
- **HTTP Integration:** Launch a local HTTP server serving test payloads, invoke range GET requests through the backend abstraction, and check that the returned data matches expectations in both hit and miss scenarios.

11.2 Performance Benchmarks

To quantify latency, throughput, and cache hit rates, we mount the FUSE filesystem and run Python scripts that exercise metadata, read, and write operations over a range of file sizes (1 MB to 1 GB):

- **High-Resolution Timing:** Use Python's `time.perf_counter()` to capture sub-millisecond start and end times for each operation.
- **Throughput Calculation:** Compute MB/s by dividing the file size by the elapsed time for read and write operations.
- **Cache Statistics:** Query the in-process cache hit and miss counters via a C FFI interface to generate hit-rate pie charts, confirming that our eviction and prefetch logic achieves the expected locality.
- **Visualization:** Produce log-scale latency and throughput plots with Matplotlib to illustrate scaling behavior and to compare against predicted cold-cache models.

This combined approach ensures that our system not only behaves correctly in all edge cases but also delivers the low latency, high throughput, and high cache hit rates required for production-grade remote file access.

12 Results

As shown in our latency comparison, we measured performance under warm-cache conditions, focusing on FUSE layer and HTTP backend metrics once the cache was populated.

12.1 FUSE Latency

Table 1 and Figure 2 summarize end-to-end operation latencies:

- **Metadata stat (`getattr`)**: Latency grows from 3.167 ms at 1 MB to 49.657 ms at 1 GB, due to FUSE dispatch overhead and larger VFS inode lookups.
- **Open + read**: Full-file read latency remains under 30 ms for all sizes, peaking at 27.528 ms for 31.6 MB and falling to 3.194 ms for 316.2 MB.

Reads are served from DRAM via the page cache, with syscall dispatch (≈ 0.02 ms per 4 MB) as the primary cost.

- **Open + write**: Write latency increases from 9.055 ms at 1 MB to 5023.302 ms at 1 GB, matching SSD sequential write speeds (200–300 MB/s). Writes return immediately after kernel buffering, and background threads flush blocks asynchronously.

12.2 FUSE Throughput

Table 2 and Figure 4 report throughput measurements:

- **Read throughput**: Rises from 138.43 MB/s at 1 MB to $120\,628.22$ MB/s at 1 GB, reflecting DRAM-speed delivery and elimination of network and disk I/O.

- **Write throughput**: Varies between 79.37 MB/s (at 31.6 MB) and 199.07 MB/s (at 1 GB), since writes are buffered in the page cache and complete immediately, while background HTTP PUTs handle persistence without blocking.

12.3 HTTP Backend Performance

Table 3 and Figures 5, 6 break down raw HTTP operations:

- **INFO (HEAD)**: Completes in 2.085 ms to 3.973 ms across all sizes, since only headers are fetched.

- **GET latency**: Grows linearly from 3.489 ms at 1 MB to 1615.148 ms at 1 GB, combining TCP/TLS handshakes, per-block round trips, and data transfer at ≈ 150 MB/s.

- **GET throughput**: Peaks at 910.283 MB/s for 10 MB and settles to 619.138 MB/s at 1 GB as protocol overhead is amortized.

- **PUT latency**: Remains between 2.155 ms and 3.489 ms for all sizes, since each PUT only enqueues data into the cache.

- **PUT throughput**: Logical throughput (size divided by enqueue latency) ranges up to $417\,452.695$ MB/s at 1 GB, reflecting local buffering; actual uploads proceed asynchronously in the background without blocking.

12.4 Analysis

These warm-cache measurements reveal several important technical insights. First, metadata operations (`getattr`) incur sub-50 ms latency even at 1 GB, demonstrating that our use of an in-memory hash table for block lookup and a single FUSE dispatch per call keeps directory and inode traversal costs bounded. Second, full-file reads complete in under 30 ms because once blocks are resident in the Linux page cache, data is served from DRAM at over 100 GB/s, the FUSE layer issues batched reads of 4 MB aligned chunks, and the overhead of each syscall (approximately 0.02 ms) is amortized over large transfers. Third, write latency scales linearly with file size but remains non-blocking: each write() enqueues data into the kernel page cache in a few milliseconds, and our dedicated thread pool then performs HTTP PUTs asynchronously, avoiding foreground stalls. On the HTTP side, HEAD requests complete in 2–4 ms due to minimal header parsing, while range based GETs incur a predictable cost of roughly 1.5 ms per megabyte (including TCP/TLS handshake and curl's write callback overhead), yielding throughput near 619 MB/s at 1 GB. PUT enqueue calls remain under 4 ms regardless of size because they only involve copying into a user-space buffer and updating internal queues. Altogether, these results confirm that our combination of precise HTTP range slicing, hybrid eviction, page-cache utilization, and asynchronous write-back delivers true DRAM-bound read performance and SSD-matched write performance entirely in userspace.

13 Evaluation

Our FUSE-based on-demand remote file caching system functions as intended. From our proposal, we planned to develop a FUSE-based file system that fetched remote blocks from either HTTP or S3, to then place those files in the cache for faster access. We chose to implement an HTTP backend and we employed the least recently used (LRU) and file hotness eviction strategy. We hypothesized that a caching mechanism would easily reduce file access latency, which is clear given that latency after a warm read or warm write after five iterations is far lower than one second even with a one gigabyte file. Typically, a system without a FUSE system without a cache would read a one gigabyte file with a latency of around one second, or greater.

In our proposal, we didn't discuss the warm vs cold performance of a cache system. The cache system is certainly far quicker with file accesses after frequently used files have been loaded into the cache already. Therefore, if the cache file system has been running for a few iterations, the latencies of both read and write should be far

lower than if the files haven't been accessed due to the lack of any iterations.

14 Future Work

While our current system achieves a promising performance under typical workloads, we were not able to test its behavior in a more diverse or large-scale environment. Furthermore, there are several other areas where improvement can be made:

14.1 Smart Prefetching

Our current system's prefetching is limited to sequential block prediction. A more advanced approaches could analyze access patterns or history to prefetch non-sequential data, which would benefit workloads for database queries or media streaming.

14.2 Dynamic Eviction Tuning

Our current eviction policy uses fixed weights for recency and file hotness. However, adapting these weights dynamically based on runtime behavior may better handle changing access patterns with relatively small overhead.

14.3 Diverse Backend options

Extending support for other backend structures, such as S3, ACS, or GCS would allow the system to work with a wider range of real-world use cases without a major overhaul in our architectural designs.

15 Conclusion

From our experiments, we can conclude that a user–space FUSE filesystem can deliver performance comparable to or better than kernel–level solutions, without requiring elevated privileges. In detailed benchmarks we measured metadata lookup latency of 3 ms to 50 ms for files from 1 MB to 1 GB, end-to-end read latency below 30 ms at all sizes, and write latency under 10 ms before background flush. Read throughput rose from 138 MB/s at 1 MB to over 120 000 MB/s at 1 GB, while write throughput held between 79 MB/s and 199 MB/s. These figures confirm that our hybrid eviction policy, block-aligned HTTP range requests, page-cache utilization and asynchronous write-back transform remote operations into DRAM-speed and SSD-speed I/O.

Functional validation covered cache lifecycle correctness, transparent read-through on misses and robust HTTP integration under concurrency, proving semantic fidelity alongside high performance. The cache manager's fine-grained locks and atomic metadata updates kept the FUSE request path largely lock-free, while a dedicated thread pool handled write-back and time-based expiry without blocking foreground I/O.

Future enhancements include adaptive prefetching driven by access pattern analysis, dynamic tuning of recency and hotness weights in the eviction score, and modular support for additional backends such as AWS S3 or gRPC storage. We also plan to extend our testbed to multi-node clusters and varied network conditions to quantify scalability, consistency and fault-tolerance. These improvements will further optimize cache efficiency, workload adaptability and deployment flexibility in diverse distributed storage environments.

References

[1] M. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1994.

[2] R. Ishii and K. Shudo. Faster FUSE Filesystems with Efficient Data Transfers. In *Proceedings of the 25th International Conference on Distributed Computing and Networking (ICDCN)*, 2023.