Bridging Storage and Execution: A Semantic Virtual Bus for On-Demand Application Streaming

Jun Lu
Central South University

Jialin Li
National University of Singapore

Yaoxue Zhang
Tsinghua University

Ju Ren*
Tsinghua University

Abstract

Traditional application delivery requires full local installation, incurring persistent security risks from outdated versions, significant download delays. Despite advances in network throughput and latency, existing dynamic loading solutions such as Web applications and network filesystems like NFS suffer from performance degradation, functionality limitations, and intrusive application modifications. We introduce StreamBus, a transparent application streaming system that redefines the network as a semantic-aware virtual storage bus beneath the file system layer. Supporting deployment across diverse environments, including WiFi-dependent mobile devices, it addresses two key challenges: maintaining microsecond-level latency comparable to local storage and bridging the semantic gap between stateless remote storage and stateful execution. To achieve this, StreamBus combines a dual-mode transmission mechanism that synchronously serves requested blocks and asynchronously prefetches predicted blocks, with a thread-aware Markov-chain model that captures fine-grained access patterns. Evaluation shows StreamBus delivers near-native performance across diverse networks. On desktops, it achieves 15–40% better per-page access latency than local NVMe in common cases. On mobile devices, it typically sustains startup overheads below 40% relative to local storage, even over variable Wi-Fi connectivity. Robustness experiments demonstrate stable performance under emulated network conditions with realistic delay patterns, supporting intra-city deployments.

1 Introduction

Traditional application delivery typically involves downloading the entire application package from an app store or third-party source to local storage, followed by installation, which introduces critical challenges in both security and usability. First, users often postpone or avoid upgrading applications [24, 26], leading to persistent security vulnerabilities [6, 7, 35], slower feature rollouts, and wasted development effort. Second, growing application sizes lead to longer download and installation times, which can be especially burdensome in short-term scenarios—such as travelers needing a transit app or users trying a newly released application.

Fundamentally, this model treats applications as fixed-function packages that must be fully installed onto local storage before use. Such a static deployment approach reduces flexibility in application provisioning and limits adaptability to dynamic usage scenarios. Meanwhile, advances in wired and wireless network technologies, such as 5G NR [39] and Wi-Fi 6 (802.11ax) [21], now provide multi-gigabit throughput with latencies that can closely match those of local storage buses, under ideal conditions. These trends pave the way for a new application delivery paradigm, in which the network acts as a functional extension of the storage bus [43, 44]. With sufficient bandwidth and low latency, it becomes feasible to stream applications on demand from remote storage, rather than pre-installing them locally.

Despite these promising technological developments, both industry and academia have proposed numerous solutions for dynamic application loading—each facing significant practical limitations. Framework-bound approaches typically require extensive modifications to existing applications, hindering widespread adoption [3, 4, 11, 22]. Alternative solutions based on Web or mini-program technologies simplify delivery but often sacrifice performance and native functionality due to their reliance on browser-based runtimes [17, 40]. Even network filesystems such as NFS [30], while supporting remote execution through OS-level page faults, fall short in efficiency. Their design favors throughput and sequential access, limiting performance for latency-sensitive, random execution patterns.

Motivated by these limitations, we present StreamBus, a system for transparent, on-demand application streaming that introduces a semantic-aware virtual storage bus abstraction, effectively bridging remote storage and local execution. Unlike traditional networked file systems (e.g., NFS) or virtual

*Ju Ren is the corresponding author.

disk solutions, which merely provide file-level or block-level remote access without semantic awareness, StreamBus operates as an intelligent extension of local storage. Specifically, it intercepts application-level page faults and file read operations beneath the file system layer, dynamically streaming data from remote storage based on execution-driven block prediction. By leveraging semantic awareness—embedding thread-specific runtime behaviors into predictive prefetching—StreamBus ensures that network latency is masked, closely matching the performance characteristics of local storage buses. Thus, our proposed semantic virtual bus abstraction allows unmodified applications to transparently execute directly from networked storage, eliminating installation delays, reducing local storage dependency, and adapting flexibly to dynamic usage scenarios.

Implementing StreamBus faces two fundamental technical challenges arising from the semantic gap between network storage and application execution. First, streaming executables requires microsecond-level latency: code blocks need to arrive on the order of microseconds to prevent execution stalls. This stringent timing requirement conflicts with network protocols on end devices that are primarily optimized for bulk throughput, a challenge intensified by mobile devices operating over fluctuating Wi-Fi connections. Second, storage systems are inherently stateless, treating each access independently, whereas program execution is stateful, with access patterns that evolve over time based on runtime control flow, input, and thread behavior. While prior systems [23] have employed Markov chains to model these patterns, their coarse-grained, per-executable modeling fails to capture the fine-grained, interleaved, and non-deterministic sequences introduced by concurrent thread execution.

To address these challenges, StreamBus introduces two key technical innovations. First, to reconcile microsecond-level latency demands with fluctuating network conditions, StreamBus employs a dual-mode transmission mechanism. The fast path synchronously delivers explicitly requested file blocks, while the slow path asynchronously prefetches predicted blocks into local memory. This parallel mechanism effectively transforms network I/O into memory-level operations, masking transfer latency and ensuring uninterrupted block availability even over variable connections. Second, StreamBus introduces a novel thread-aware prefetching paradigm that embeds per-thread block access sequences into the storage-layer prediction mechanism. Using server-side Markov chain models to track the access behavior of individual threads, StreamBus preserves execution semantics across network boundaries. These models track per-thread access patterns while generalizing across devices, improving prediction accuracy for recurring behaviors such as application startup sequences.

StreamBus adopts a distributed architecture comprising complementary client- and server-side components. On the client side, StreamBus is implemented using two types of dedicated kernel threads—fast path and slow path—which bridge the upper layer of the custom, lightweight, read-only filesystem and the remote server. The fast and slow path threads operate as a cooperative pair and are instantiated from a dedicated thread pool. Both StreamBus and the custom filesystem are implemented within a single kernel module. On the server side, StreamBus is implemented as a Rust userspace daemon that hosts a prefetcher built from a set of Markov chains, with one chain per executable. StreamBus incorporates thread-level semantics into each Markov chain. This allows the system to capture and predict fine-grained, thread-specific block access patterns. The prefetcher partitions the predicted sequence into two subsets: immediate successor blocks, which are sent via the fast path to minimize latency, and longer-range predicted blocks, which are delivered asynchronously via the slow path. Both subsets are transmitted efficiently using the zero-copy sendfile syscall.

This paper makes the following contributions:

1) StreamBus, a system introducing a *semantic-aware virtual storage bus* abstraction, transparently bridging remote storage and local execution, enabling unmodified applications to execute directly over the network without installation and achieving near-native performance.

2) A dual-mode transmission mechanism that masks network latency by combining synchronous delivery of explicitly requested blocks with asynchronous prefetching of predicted blocks.

3) A thread-aware, per-executable Markov prefetching model that captures fine-grained block access patterns and preserves execution semantics, substantially improving prediction accuracy and prefetch timeliness.

4) An evaluation on both mobile and desktop systems demonstrating that StreamBus achieves significantly lower startup overheads compared to traditional methods. On desktop systems, StreamBus outperforms NFS by up to 3× in per-page access latency and even surpasses local NVMe storage by 15%–40% in most scenarios. On mobile devices, StreamBus typically sustains startup overheads below 40% relative to local storage, even under variable Wi-Fi conditions.

2 Background: Bridging the Execution-Storage Divide

This section analyzes the key challenges in bridging application execution with remote storage. We first examine why network latency remains fundamentally different from storage latency, despite advances in throughput and access speed. Next, we discuss how stateless storage interfaces fail to capture the semantics of execution. Finally, we contrast the limitations of existing solutions with the design requirements of StreamBus.

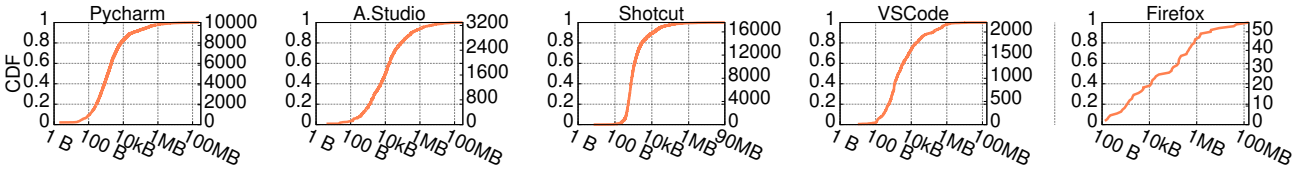

Figure 1: CDF of file sizes across application binaries (file counts on right y-axis). A.Studio = Android Studio.

2.1 Why Network Latency ≠ Storage Latency

The fundamental challenge in streaming application execution stems from the persistent disparity between network and storage latency characteristics. While modern storage devices achieve microsecond-scale access latencies [15, 19], remote access over commodity networks suffers from orders-of-magnitude higher and less predictable delays. In particular, state-of-the-art wireless networks (e.g., Wi-Fi 6, 5G) exhibit long-tail latency spikes reaching into the tens of milliseconds in real-world conditions [21, 39]. This discrepancy becomes critical for code execution, where microsecond-level latency consistency is essential to avoid processor pipeline stalls.

This problem is amplified by two key architectural mismatches. First, conventional network protocol stacks introduce substantial overhead. TCP congestion control algorithms (e.g., Cubic, BBR) introduce latency variability due to queuing, retransmissions, and window adjustments, while connection setup and security handshakes further inflate end-to-end delays [5, 13]. Even DCTCP, a state-of-the-art congestion control algorithm designed for datacenter efficiency, can incur millisecond-scale delays when handling small requests [2]. While far-memory systems also support remote access [31, 46], they rely on low-latency interconnects for volatile memory extension, making them unsuitable for high-latency mobile and edge environments. Second, network filesystems such as NFS were originally designed for throughput-oriented workloads, as their protocols and caching strategies are optimized to maximize sustained data flow rather than minimize individual access latency. This design favors sequential, megabyte-scale transfers typical of bulk data or media streaming. In contrast, as shown in Figure 1, application executables range from kilobytes to hundreds of megabytes, demanding versatile systems that efficiently support variable code loading patterns.

StreamBus addresses these limitations by rethinking the network not as a communication channel, but as an extension of the storage bus. Instead of relying on traditional remote file protocols like NFS, StreamBus introduces a semantic-aware virtual bus that directly exposes fine-grained block access operations over existing TCP transport. This design decouples application execution from bulk file transfer models, enabling precise, on-demand fetching of code and data blocks with microsecond-scale latency. To fully realize this capability in dynamic network environments, StreamBus further employs a dual-mode transmission mechanism, which will be described in Section 3.2.

2.2 The Myth of "Generic" Prefetching

Traditional prefetching mechanisms operate under a critical misconception: that storage access patterns exhibit predictable spatial and temporal locality. While this assumption holds for workloads such as media streaming and database scans, it breaks down in the context of application code execution, where access patterns are governed by control flow and thread interactions rather than file offset contiguity.

Existing techniques fall short along three key dimensions. First, there is a granularity mismatch: prefetchers like Linux ReadAhead [38] and Leap [1] optimize for sequential file offsets, whereas code execution follows virtual address jumps that are often unrelated to the physical block layout of the executable on disk. Second, StreamSys, a representative Markov-based model, is concurrency-blind [23]. Based on Markov chain principles, such models learn probabilistic transitions between block accesses from application execution patterns, enabling prediction and prefetching of future blocks to convert random I/O into sequential access. However, StreamSys tracks only process-level access sequences and ignores thread-level interactions. In practice, thread A may block on thread B's I/O, producing complex, interleaved access patterns that render process-wide prediction ineffective. Third, there is a statefulness gap: storage systems treat each access as an independent event, whereas program execution maintains evolving control flow. This often leads prefetchers to aggressively fetch cold code paths while missing hot, frequently executed branches.

StreamBus breaks from this tradition by embedding thread execution semantics directly into the storage layer. Our thread-aware prefetching model (Section 3.3) tracks storage access sequences at thread granularity, modeling block transitions as per-thread Markov chains. This design enables it to distinguish interleaved access patterns across concurrent threads and capture common execution paths without requiring program instrumentation or source-level analysis.

2.3 The Illusion of Transparency

Prior attempts at transparent application streaming generally fall into three categories, each introducing trade-offs in compatibility, performance, or complexity. Framework-bound approaches, such as Android Instant Apps [11], require explicit modularization and disallow native code via the JNI interface, limiting compatibility and increasing developer burden. Web-based solutions, including WebAssembly [34]

and mini-program [14] platforms, simplify distribution but operate in constrained sandboxed environments, leading to performance overheads—up to 2× slower than native binaries [40]—and limited system API access. Other systems, such as AppSlicer [4] and LegoDroid [22], dramatically modify core application lifecycle behavior (e.g., Activity or class loading) to enable partial loading. While effective, these approaches involve deep changes to the runtime, increasing system complexity and reducing portability. Most prior solutions rely on non-native frameworks or runtime modifications, compromising system-level transparency and hindering adoption across diverse platforms.

In contrast, StreamBus achieves full binary transparency, supporting unmodified ELF and APK binaries with complete system API and native code access. No source changes, modularization, or special build steps are required. Our Android implementation requires only minimal AOSP framework modification to bypass APK installation checks, without altering application behavior.

3 Design

This section presents the overall architecture of StreamBus, followed by two key mechanisms that enable low-latency execution and accurate prediction: a dual-mode transmission mechanism and a thread-aware prefetching model. We conclude with a case study demonstrating integration into the Android software stack.

3.1 System Architecture

StreamBus adopts a client–server architecture where the client runs a custom filesystem backed by a virtual storage bus, and the server provides application content on demand using predictive prefetching, as illustrated in Figure 2. This subsection describes the key components on both sides of the system and how they interact to support low-latency, installation-free execution. Throughout this section, we refer to transmitted data units as blocks, which are mapped to memory as pages (typically 4 KB each) on the client side. We sometimes use "block" and "page" interchangeably, depending on context.

StreamBus is designed to support these three goals:

- **Transparency**: Applications require no installation and execute seamlessly over the network using remote content transparently.
- **Efficiency**: A dual-path mechanism decouples latency-sensitive execution from predictive background transfers.
- **Compatibility**: StreamBus integrates with existing OS infrastructure, supporting standard file APIs and Linux's page cache without requiring modifications to applications.

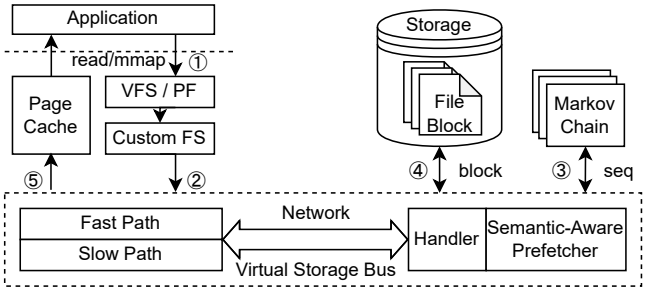

Figure 2: StreamBus architecture illustrating client-side stack, virtual bus, and predictive access over fast and slow paths.

The client includes a kernel module that provides a read-only filesystem supporting traditional file operations (e.g., `open`, `read`, `mmap`), and maps virtual file blocks to remote network requests ①. When a block is requested, whether through a file read or a Page Fault, the custom filesystem forwards the request to the StreamBus runtime ②. This runtime manages paired types of dedicated kernel threads where the fast-path thread synchronously handles demand-driven requests and the corresponding slow-path thread asynchronously receives prefetched blocks predicted by the server. Both threads maintain persistent TCP connections to the server and store received blocks in the Linux page cache ⑤, enabling seamless integration with the virtual memory subsystem.

The server consists of two main components: a semantic-aware prefetcher and a block handler. Since all executable content is read-only and shared across users, the prefetcher is placed on the server to avoid per-user duplication and to leverage its stronger computational resources for low-latency prediction. The prefetcher maintains a per-executable Markov model to predict future block accesses based on observed execution patterns. As a semantic-aware component, the prefetcher leverages two key sources of context. First, it uses high-level filesystem information, such as file paths, to form the foundation for building accurate, file-specific prediction models. Second, it incorporates OS-level thread context to capture per-thread access behaviors and temporal locality. When serving a request from the client, the prefetcher uses both the block index and this rich contextual information to generate a predicted sequence of future accesses ③. Unlike traditional block-based storage systems that operate solely on numerical sector addresses, this semantic-aware interface enables more accurate and timely prediction by incorporating execution-level semantics. The handler, in turn, manages network communication and block dispatch. It retrieves the required blocks from storage ④, forwarding the explicitly requested block to the fast path while sending predicted blocks to the slow path. This division ensures that critical execution proceeds promptly, while background predictions are delivered opportunistically.

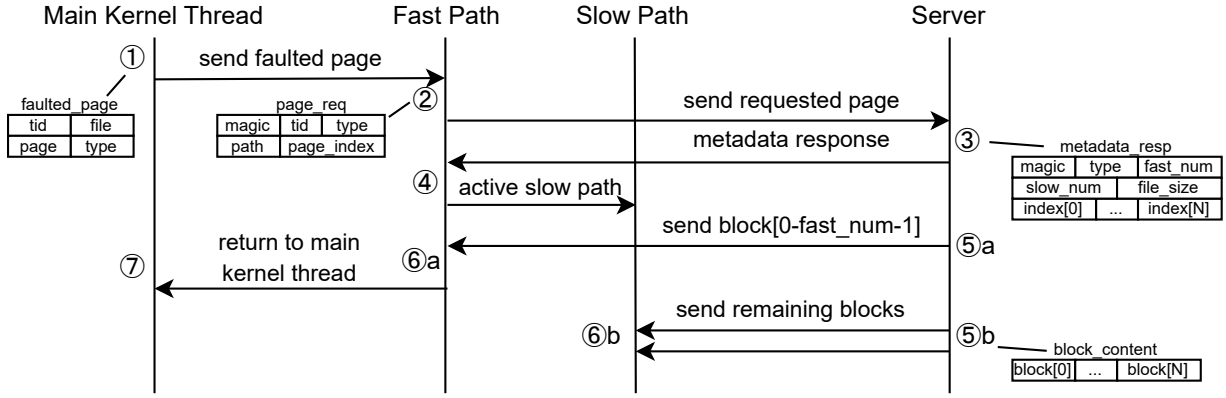

Figure 3: Dual-mode block delivery workflow (Fast & Slow Path).

3.2 Dual-Mode Transmission Mechanism

Unlike bulk data transfers or media streaming, application execution requires microsecond-scale responsiveness for fetching individual code blocks. Even modest delays in block availability can block execution progress, particularly when a thread is suspended during page fault handling. While modern wireless and mobile networks provide high throughput, they still suffer from variable latency and long-tail jitter. STREAMBUS addresses this challenge by decoupling latency-sensitive execution from background prefetching, using a dual-mode transmission mechanism that routes predicted blocks across two parallel paths.

The dual-mode mechanism is implemented using multiple pairs of persistent TCP connections between the client and server, where each pair consists of a fast path and a slow path thread. The fast path serves synchronous, demand-driven block requests; the slow path handles asynchronous prefetch transfers. This separation allows latency-critical blocks to be served quickly while enabling background prediction to proceed in parallel without interfering with execution. The overall workflow is illustrated in Figure 3.

Fast path: demand-driven block retrieval. When an application triggers a page fault or issues a file read, the kernel module constructs a page request structure that includes the file pointer (file), thread ID (tid), object type (file or directory), and the faulted page (page). The file is subsequently used to resolve the file path and to access the page cache via its associated address mapping. The main kernel thread selects an available fast/slow path pair from a round-robin queue and forwards the fault context to the designated fast path thread ①. This thread constructs and sends a `page_req` message, which includes the file path (resolved from the file pointer), thread ID, object type, and page index ②. These fields together capture both file-level and thread-level context, enabling the server-side predictor to make accurate prefetching decisions. Upon receiving the page request, the server invokes its semantic-aware predictor to generate a predicted sequence of future block accesses. It then constructs and returns a metadata response, which includes the predicted block indices, `fast_num`,`slow_num`, and total file size ③. After receiving this response, the fast path parses the total number of predicted blocks and their indices, prepares memory pages for them, and activates the corresponding slow path, passing along the metadata ④. The server then transmits the explicitly requested block along with the first few predicted blocks (up to `fast_num`) to the fast path ⑤a. These blocks are inserted into the page cache ⑥a, allowing the main kernel thread to resume execution ⑦.

Slow path: asynchronous prediction delivery. After responding to the fast path, the server sends the remaining predicted blocks (i.e., from fast_num onward) to the corresponding slow path connection ⑤b. The client's slow path thread receives these blocks asynchronously, inserts them into the page cache, and populates them with file content ⑥b. Since many execution paths exhibit spatial or temporal reuse, this speculative delivery improves future hit rates and reduces blocking I/O. Pages fetched via the slow path are maintained in reclaimable page cache, avoiding memory pressure and allowing safe coexistence with other memory workloads.

Concurrency and flow control. Each fast/slow path pair operates independently, supporting multiple outstanding requests across CPUs. Once a pair is selected, it is marked busy and excluded from further scheduling until both its fast and slow path transfers complete. This explicit pairing preserves prediction alignment between metadata and data streams, simplifying both server-side dispatch and client-side state management. In practice, the number of concurrent page faults remains low due to the serialized nature of fault handling in the kernel, which reduces contention. Nonetheless, the design could be extended in future work to support NUMA-local prefetch threads or adaptive prediction reuse based on runtime feedback.

The dual-mode transmission mechanism bridges OS-level demand paging with predictive content delivery by proactively staging blocks into the page cache, effectively masking network latency.

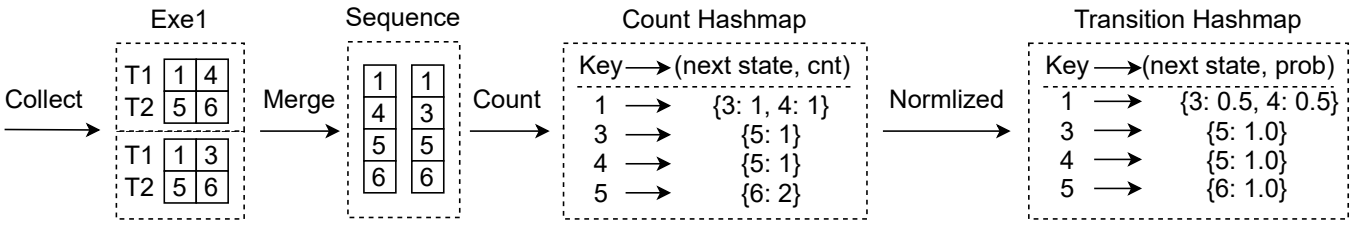

Figure 4: Markov model training in StreamBus.

3.3 Thread-Aware Prefetching

As discussed in Section 2.2, existing approaches fail to capture the interleaved, thread-specific code paths seen in real-world applications. To address this limitation, StreamBus employs thread-aware prefetching based on Markov chains, enabling accurate prediction of complex execution patterns.

Training and model construction. The training pipeline consists of four stages, as summarized in Figure 4. To build predictive models, StreamBus first collects block-level execution traces by running each target application multiple times (e.g., 10 runs) in a controlled environment, where each run involves a complete cold-start execution followed by clearing the page cache. This limited collection suffices because startup behavior exhibits highly stable block access patterns across users. During application execution in the profiling phase, the client records per-thread block access sequences, tagging each block request with the responsible thread ID to accurately capture interleaved execution patterns. In the second stage, per-execution thread traces belonging to the same executable are merged into a unified sequence. This process discards thread identifiers while preserving the common access transitions observed across threads. Since precise cross-thread timing information is not recorded, StreamBus merges thread-local sequences in randomized order. Although this can cause overlaps between independent thread paths when the prediction window exceeds individual sequence lengths, the affected blocks typically belong to critical execution paths and remain beneficial for prefetching. Potential over-prediction issues arising from this merging strategy are addressed during the inference stage. By aggregating at the executable level, StreamBus balances model generality with training simplicity, while preserving thread-level execution semantics without incurring the overhead of modeling each thread independently.

Next, StreamBus constructs a transition count hashmap that records the frequency with which each block is followed by another. For example, if block 1 is followed by blocks 3 and 4 in separate traces, both transitions are counted. This structure preserves transition frequencies across runs, thereby forming the basis for probabilistic prediction in later stages. Finally, the transition probabilities are computed by normalizing the count map, producing a Markov chain that links each block to its potential successors with associated probabilities. This chain forms the basis of runtime prediction, enabling the server to speculate future accesses from any given starting block. The entire training process is lightweight, application-specific, and performed offline. This approach is particularly effective for application execution workloads, which, as shown in Figure 1, often involve large numbers of files with highly diverse size distributions. In such cases, Markov chains provide efficient prediction by modeling transitions between code blocks. We use hash-based storage instead of dense matrices for fast lookup and reduced memory overhead.

Runtime inference. Upon receiving a block request from the client, which includes the application name, ELF file name, and starting block index, the server performs a hierarchical lookup to locate the appropriate prediction model. It first queries the application-level map using the application name to retrieve the set of associated ELF binaries, then locates the corresponding ELF entry based on the provided file name and subsequently retrieves the pre-trained Markov chain associated with the executable. This Markov chain encodes transition probabilities between code blocks, learned from previously observed execution sequences. At each prediction step, a given state may have multiple candidate successors, as illustrated in Figure 4 (e.g., block 1 has successors 3 and 4). The server samples a random probability and traverses the successor list according to their transition probabilities. For instance, if the random probability is less than 0.5, it selects block 3; otherwise, it selects block 4. This stochastic selection process ensures that frequent execution paths are favored while still allowing for variability across runs. Starting from the requested block, the server performs iterative prediction by selecting the most probable successor at each step and using it as the new input state. Prediction continues until either a fixed number of blocks has been generated or no valid successor exists. Due to the use of optimized hash-based lookup structures, each transition incurs sub-microsecond latency in practice, enabling near-instantaneous generation of the full prediction sequence. Once the sequence is produced, it is partitioned into two parts. A short prefix, typically consisting of 1 to 4 blocks, is immediately dispatched via the fast path to satisfy critical execution demands. The remaining blocks are

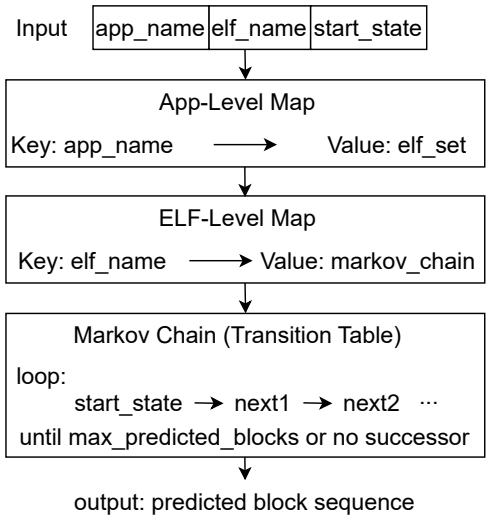

Figure 5: StreamBus inference procedure.

transmitted asynchronously over the slow path to opportunistically prepopulate the page cache. This division prioritizes immediate responsiveness while proactively masking network delays for subsequent accesses.

To address potential over-prediction introduced during training sequence merging, StreamBus maintains an accessed sequence list for each executable during inference. After the server completes a Markov chain traversal for a given prediction, the corresponding sequence indices are recorded in this list. When a future request targets a block already present in the accessed sequence list, the server returns only the requested block without initiating additional prediction. This mechanism suppresses redundant speculation and prevents unnecessary network transfers. Given the typical stability of client-side page caches, blocks once prefetched are unlikely to be immediately evicted. Therefore, it is reasonable to assume that recent prefetches remain valid for a window of time. The accessed sequence list is periodically refreshed every few minutes to account for cache turnover and maintain prediction freshness.

4 Implementation

We implement StreamBus with a combination of kernel-space and user-space components, deployed across Linux (desktop) and Android (mobile) platforms. The following sections detail the key components and integration efforts.

4.1 Client-Side Runtime

StreamBus's client-side runtime is implemented as a Linux kernel module, comprising approximately 3,200 lines of C code (excluding comments and blank lines). It consists of a custom read-only filesystem and a pair of dedicated kernel threads per CPU: one for the fast path (demand-driven requests) and one for the slow path (asynchronous prefetching). The module interacts with the page fault handler, page cache, and the remote server to fulfill block-level access requests from applications.

Filesystem and Request Dispatch. The filesystem is mounted by the user and supports standard file operations such as `read`, and `mmap`. Once mounted, applications can access remote executables transparently through the local filesystem interface, similar to distributed filesystems like NFS. When a page fault or `read` operation occurs, the filesystem constructs a `faulted_page` request containing high-level context, including the file pointer and thread ID (tid). The request is then dispatched to the fast path thread through a lock-free ring buffer (`kfifo`), with synchronization managed via a pair of semaphores (`sem_send`, `sem_recv`). To avoid inter-core contention, each CPU maintains its own independent set of `kfifo` queues and semaphores. After enqueuing the request, the filesystem signals the fast path thread using `sem_send` and blocks on `sem_recv` until the request is fulfilled.

Fast Path Thread. Each fast path thread is bound to a CPU core and maintains a persistent TCP connection to the server on port 9999. Upon receiving a filesystem request, the fast path sends it over the socket and waits for a `metadata_resp` containing the requested page and predicted successors. The fast path thread performs the following steps: ① It first receives the page indices (requested plus predicted) from the server, allocates empty kernel pages via `__page_cache_alloc`, and notifies the slow path to receive the remaining pages. ② It inserts each allocated page into the file's page cache mapping using `add_to_page_cache_lru`. ③ It then receives the actual page content over the TCP connection using `kernel_recvmsg`. ④ Finally, it marks the pages as uptodate and unlocks them for access by the main thread.

Slow Path Thread. Each slow path thread (also one per CPU) handles prefetching of predicted blocks and communicates with the server over a second persistent TCP connection (port 9998). After receiving a batch of page indices and associated metadata forwarded from the fast path, the slow path inserts the pages into the page cache and retrieves their contents from the server using the same TCP-based data transfer mechanism as the fast path.

4.2 Server-Side Runtime

The server is implemented in Rust for safety and performance, with two main components: a 1,340-line block handler for I/O and dispatch, and a 210-line semantic-aware prefetcher for prediction.

The block handler manages persistent TCP connections to the client, operating in a multithreaded, event-driven model. For each newly established fast path connection (on port 9999), the server spawns a dedicated handler thread. Upon

receiving a request, the handler consults the prefetcher to generate a prediction sequence and orchestrates block dispatch between the fast and slow paths. The prefetcher maintains per-application Markov models trained offline using startup traces. To locate the appropriate model, it first identifies the application by name and then the specific ELF file by its hashed path. Given a requested page index, the prefetcher samples a sequence of likely successors. The prediction results are split into two groups: a short prefix (1–4 blocks) for the fast path and the remainder for the slow path. The server sends a structured response containing metadata and page indices, and transfers file data using Linux's zero-copy `sendfile()` syscall, coalescing adjacent pages to reduce syscall overhead and disk I/O. The server also maintains a background listener on port 9998 for slow path connections. Each slow path thread, paired with a fast path thread, asynchronously fetches deferred predictions, enabling concurrent computation and data transfer for higher throughput and lower latency.

4.3 Integration into the Android Platform

To demonstrate the practical viability of StreamBus in production environments, we integrate it into the Android Open Source Project (AOSP) to enable direct execution of APK applications from remote storage without installation. This integration is non-trivial because Android's tightly coupled application lifecycle enforces a strict installation pipeline: the system expects every application to be formally installed via the package manager, which involves copying the APK to local storage, parsing and verifying metadata, performing permission and signature checks, and compiling bytecode into native code. These stages assume that application artifacts are locally resident and registered with the system, posing fundamental challenges for on-demand execution.

To enable remote execution without installation, we introduce minimal modifications to the Android framework. Specifically, this integration adds approximately 130 lines for a dedicated interception class and only a few dozen lines of changes to the package manager and launcher components. We skip the APK copy step (since the APK remains remote), bypass permission checks (assuming trusted delivery from a verified server), and omit dex2oat compilation (as precompiled artifacts can also reside remotely). All remaining parts of the launch process—including application sandboxing, process spawning, and system service access—remain unchanged.

This modification is narrow in scope, affecting only a few routines in the package manager. It does not impact other applications, does not alter kernel behavior, and requires no changes to the APK itself. From the application's perspective, the launch experience is identical to a locally installed app: the APK runs in the standard runtime environment with full system API access, requiring no code changes or additional runtime support.

Table 1: Evaluated applications and corresponding installation sizes

Desktop		Mobile	
Pycharm	3.0GB	Uber	447MB
A.Studio	2.9GB	WPS	377MB
Shotcut	685MB	Spotify	193MB
VSCode	409MB	Whatsapp	126MB
Firefox	260MB	Telegram	122MB

5 Evaluation

Evaluation Goals. We evaluate StreamBus to answer the following key questions:

- **Q1: Microsecond-Scale Responsiveness.** Can the system serve remote content with low latency, even under unpredictable network delays?
- **Q2: End-to-End Performance.** How much speedup does StreamBus offer for application launch, compared to traditional installation-based workflows?
- **Q3: Robustness.** How does system performance degrade under adverse conditions, such as latency injection and packet loss?

Scope. All experiments in this section primarily focus on the *application startup* phase. Startup is chosen because it represents a critical, measurable, and user-facing performance window, where latency and responsiveness are most impactful. This focus ensures consistent and meaningful comparisons across different baselines.

Testbed. We evaluate StreamBus on a Google Pixel 6 Pro and two Dell OptiPlex 5070 desktops. The Pixel 6 Pro features an octa-core CPU (2×2.80 GHz Cortex-X1, 2×2.25 GHz Cortex-A76, 4×1.80 GHz Cortex-A55), 8 GB LPDDR5 RAM, and 128 GB UFS 3.1 storage. Each desktop is equipped with an 8-core Intel Core i7-9700 @ 3.00 GHz, 16 GB LPDDR4 RAM, and a 1 TB NVMe SSD. One desktop serves as the backend for both the Pixel 6 Pro and the other desktop client. In the wired setup, the desktops connect via 1 Gbps Ethernet through a *NETGEAR* router. In the wireless setup, the Pixel connects over 802.11ac (5GHz) to the same router. Kernel versions are 5.10.198 (Pixel) and 5.10.157 (desktops).

Baseline Configuration. In the Wi-Fi environment, we compare StreamBus running on the Pixel 6 Pro against the native Android file system backed by local UFS 3.1 storage. In the wired desktop environment, we evaluate StreamBus against two baselines: local execution from an NVMe SSD and remote execution via NFSv4. To ensure fair and consistent comparisons, we clear the client-side page cache before each measurement unless otherwise specified, as page cache effects can significantly skew performance results. Table 1 summarizes the installation directory sizes of the evaluated applications. Note that, due to lack of NFS support on Android, comparisons involving NFS are only conducted on the desktop platform.

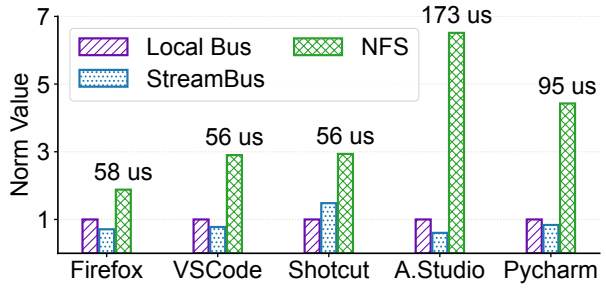

Figure 6: Per-page average latency across applications in the wired desktop environment.

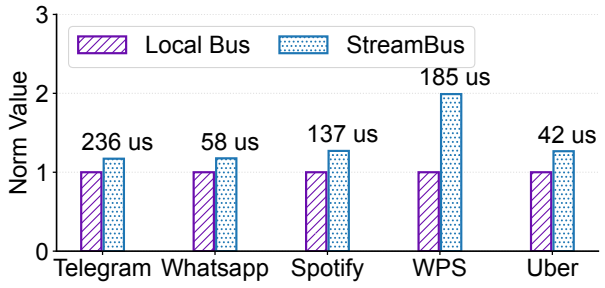

Figure 7: Per-page average latency across applications in the mobile Wi-Fi environment.

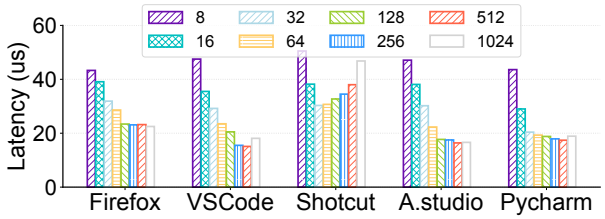

Figure 8: Effect of total prefetch size (in pages).

5.1 Microbenchmark

Per-Page Latency. To understand StreamBus's responsiveness during execution, we evaluated the average per-page latency for application startup. Specifically, we first identify the number of code blocks accessed during the launch phase, and then record the cumulative time spent serving these pages under different I/O methods. Latency is measured at two points within the kernel: the `vm_operations_struct.fault()` callback, which is triggered by page faults, and the `file_operations.read_iter()` function, invoked either by the kernel or from userspace during explicit file reads. The average latency is obtained by dividing the total serving time over the number of accessed pages.

Based on Figure 6, in the wired environment, even when equipped with a high-performance NVMe SSD, StreamBus consistently outperforms the local bus in most cases under cold-start conditions. Specifically, StreamBus achieves 16–40% lower per-page latency than the local NVMe SSD across most applications in the wired setting, demonstrating its ability to outperform even high-performance local storage. The only exception is Shotcut, where StreamBus is 48.1% slower than local access. This performance drop can be attributed to Shotcut's directory structure, which contains a large number of small files (as shown in Figure 1). This leads to extensive metadata traversal, which diminishes the effectiveness of predictor-guided block prefetching and reduces the benefits of execution-aware streaming. As shown in Figure 7, in the Wi-Fi environment, StreamBus maintains competitive latency despite operating over wireless links. In most cases, the per-page latency increases only modestly, ranging from 17.2% to 27.0% compared to UFS 3.1 on the Pixel 6 Pro. Overall, the moderate overhead is a direct result of StreamBus's dual-path transmission mechanism, which shifts the critical path from network I/O to local memory access. By prefetching predicted blocks into memory ahead of demand, StreamBus effectively hides wireless latency and sustains sub-millisecond responsiveness for most applications.

Insights into Dual-Path Transmission. To evaluate the effectiveness of StreamBus's dual-path design, we conduct a microbenchmark to explore how total prefetch size affects latency while keeping the fast path size small (1 to 4 pages).

As shown in Figure 8, increasing the total number of prefetched pages generally reduces latency across all applications. This improvement arises because a larger slow path buffer increases the likelihood of future accesses being served from memory, thereby reducing blocking I/O. However, the benefit plateaus when the total prefetch size reaches 128 pages (512KB). This is due to two main reasons. First, a 128-page window provides a sufficiently large memory-resident page pool at the client side, allowing most faulted accesses to be served directly from memory without triggering additional network fetches. Second, many executable files in typical applications are smaller than 512 KB (as shown in Figure 1). As a result, the predictor often captures and returns most or all of the accessed blocks within the executable, effectively covering the file's working set. This explains why larger prefetch sizes yield diminishing returns. In apps like Shotcut, which contain many small files and a few large ones, excessive prefetching pulls in cold blocks from large files, polluting memory and delaying critical pages, thereby increasing latency.

Prediction Accuracy. Figure 9 presents the page cache hit rate observed during both page fault handling (`filemap_fault()`) and buffered read operations (`generic_file_buffered_read()`), calculated as the number of cache hits divided by the total number of `find_get_page()` invocations within these two functions. StreamBus consistently outperforms both the local system and NFS across all applications, with particularly large gains for Shotcut, Android Studio, and PyCharm. These improvements stem from StreamBus's execution-aware

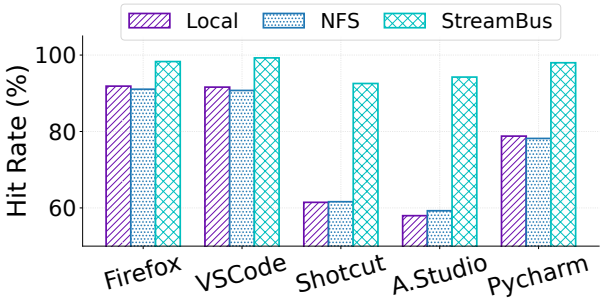

Figure 9: Page cache hit rate during application startup.

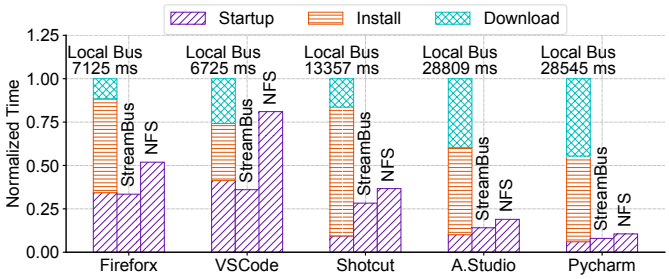

Figure 10: End-to-end launch time comparison on desktop.

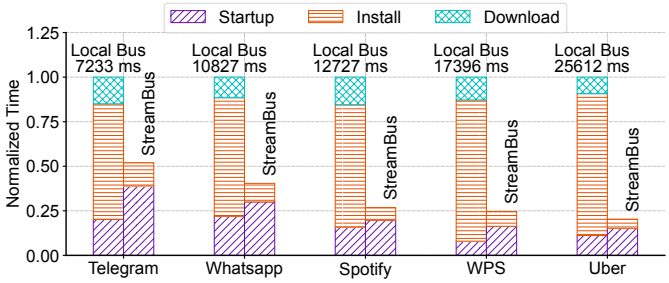

Figure 11: End-to-end launch time comparison on mobile.

predictor, which anticipates future accesses based on runtime semantics rather than relying solely on past sequential access patterns. In contrast, Firefox and VSCode show high hit rates even under the local and NFS setups. This is largely due to their relatively small working sets and the presence of large data files within their installation directories. These files are often accessed sequentially and in bulk, which naturally boosts the hit rate of prefetchers even without semantic insight. Additionally, such workloads tend to benefit from Linux's default readahead heuristics, masking the performance difference.

5.2 End-to-End Performance

We evaluate the end-to-end performance of StreamBus to assess its impact on user experience across both desktop and mobile environments. A typical application usage flow involves three stages: downloading the package (e.g., APK), installing or extracting it, and launching the application. On both desktop and mobile platforms, traditional workflows involve downloading the application via FTP from a local server, followed by installation and launch. In contrast, StreamBus and NFS enable direct execution from remote storage on desktop, bypassing the download and installation stages. While StreamBus eliminates installation entirely on desktop, it still incurs installation overhead on Android due to system restrictions. For desktop evaluations, we measure only the startup time, defined as the interval from the launch command to the last stable block request. For mobile, startup time is measured using the `am start` command, while download and installation times are recorded separately for comparison.

As shown in Figure 10, compared to local execution from a high-speed NVMe SSD, StreamBus reduces startup latency by 2.6% for Firefox and 12.9% for VSCode, highlighting the effectiveness of its execution-aware streaming design. However, for applications like Pycharm and Android Studio, StreamBus incurs modest overheads, while Shotcut performs significantly worse due to its highly fragmented file layout. Against NFS, StreamBus consistently outperforms, with startup time reductions ranging from 22.9% to 55.5%. These improvements stem from its dual-path transmission and semantic prediction design, which together reduce blocking I/O and improve cache efficiency. Figure 11 presents results in the mobile setting. Compared to UFS 3.1 local storage on the Pixel 6 Pro, StreamBus incurs moderate startup overheads ranging from 23.7% to 36.9% for most applications. Despite operating over Wi-Fi, these latencies remain acceptable for cold starts. The only notable outlier is WPS Office, which shows a 105.0% overhead, due to its larger working set and less predictable access patterns.

Overall, StreamBus yields substantial reductions in end-to-end application launch time. On desktop systems, it reduces the total launch time, which includes download, installation, and startup, by between 45.2% and 78.7% compared to the conventional installation-based approach. On mobile devices, while StreamBus introduces slightly higher startup latency, it compensates by eliminating the download phase and significantly accelerating installation. Consequently, the total user-perceived delay is reduced by 36.9% to 80.4%. These findings demonstrate that StreamBus enables efficient and low-latency app streaming on both desktop and mobile platforms.

5.3 Robustness Under Network Variability

To evaluate the robustness of StreamBus under real-world network fluctuations, we conduct experiments by injecting round-trip latencies of 2 ms, 10 ms, 30 ms, and 100 ms, corresponding approximately to communication over distances of 10 km, 100 km, 1,000 km, and 10,000 km [29, 37], respectively. To emulate realistic wide-area network (WAN) conditions, all configurations are tested with an added packet loss rate of 0.05% and jitter of 10%, reflecting transoceanic

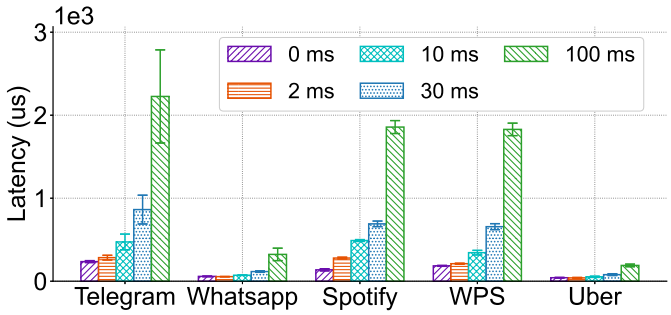

Figure 12: Average per-page latency under varying network conditions on mobile.

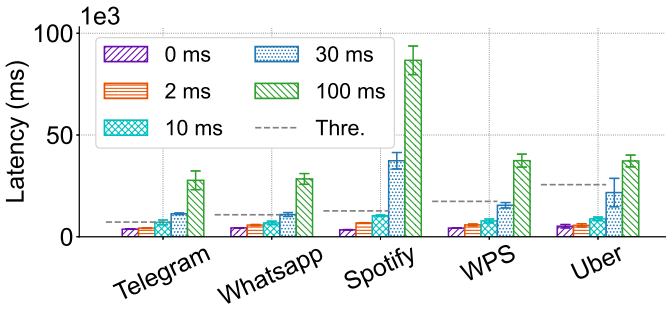

Figure 13: End-to-end application startup time under varying network latency on mobile.

transport scenarios [33]. We evaluate both mobile and desktop applications. For reference, the dashed line (Thre.) in Figure 13 and Figure 15 indicates the total delay (download + install + launch) observed under local storage, as shown in Figure 11 and Figure 10, respectively.

Robustness on Mobile. Figure 12 presents the average per-page latency for mobile applications under varying network latencies. We observe that WhatsApp and Uber maintain the most stable latency profiles. This is largely because their startup workflows involve intensive APK parsing operations, which result in predominantly sequential access patterns. Consequently, these applications benefit from higher page cache hit rates and reduced I/O blocking. Notably, despite the 100ms round-trip latency injection, all applications achieve low average per-page latency (<2ms) through persistent TCP connections that enable large payloads per round trip and high prediction hit rates (>93%), where most accesses are served from prefetched cache at memory speed. Figure 13 shows the corresponding startup times across latency conditions. In most cases, applications remain below the local-installation threshold even under 1,000 km-equivalent delay. Under low-latency intra-city conditions (e.g., 100 km or less), startup performance improves further, demonstrating StreamBus's resilience to moderate network delays.

Robustness on Desktop. Figure 14 shows the average per-page latency for each application under varying network delays on desktop. StreamBus consistently outperforms NFS across all latency conditions, achieving 2× to 7× lower latency at typical RTTs such as 10 ms and 30 ms. For example, at 30 ms, VSCode sees a drop from 995 μs (NFS) to just 195 μs with StreamBus, while Android Studio drops from over 2 ms to under 300 μs. These improvements result from StreamBus's dual-path architecture and semantic-aware prefetching, which proactively caches necessary blocks and isolates execution from round-trip delays. Figure 15 presents the corresponding end-to-end launch times on desktop. StreamBus consistently delivers lower startup latency than NFS across all delay settings, with particularly notable improvements under practical latency conditions (10 to 30 ms RTT), where it achieves 1.5× to 2× speedup. These conditions correspond to typical intra-city and regional deployments. Most applications launch well below the local-installation threshold, confirming StreamBus's suitability for on-demand usage at scale, particularly under 10–30 ms RTTs that approximate intra-city ($\approx$ 100 km) and inter-province ($\approx$ 1,000 km) deployments. In contrast, NFS incurs longer delays and higher variance due to its synchronous, throughput-oriented design. Its reliance on Remote Procedure Calls (RPC), coarse-grained locking, and lack of semantic context makes it ill-suited for handling the fine-grained and unpredictable access patterns common during application startup. It is worth noting that both StreamBus and NFS preserve program correctness even under high-latency conditions, as they operate purely at the block level without modifying execution behavior.

6 Related Work

Remote Application Execution. Several prior efforts have explored remote application execution and streaming-based delivery in both industry and academia, most of which require application repackaging or refactoring. Commercial systems such as Microsoft App-V [25], Novell ZENworks Application Virtualization [27], and Numecent Cloudpaging [28] virtualize entire applications to reduce installation overhead, but rely on packaging tools and runtime layers that operate at the application level. Troy leverages VHD features to partially load Windows images from a server [42]. Academic proposals like AppFlux [3], AppSlicer [4], and LegoDroid [22] implement streaming mechanisms by modifying the Android framework at the activity granularity, requiring intrusive changes to both apps and system components. Google Play Instant avoids framework changes but imposes size limits, bans native code, and requires modular refactoring [11].

Network File Systems and Remote Storage. Various distributed file systems have been proposed to meet the unique demands of scalability, consistency, and performance under different hardware and workload assumptions. Ceph [36], for instance, separates metadata and data management using a pseudo-random distribution function to improve scalability

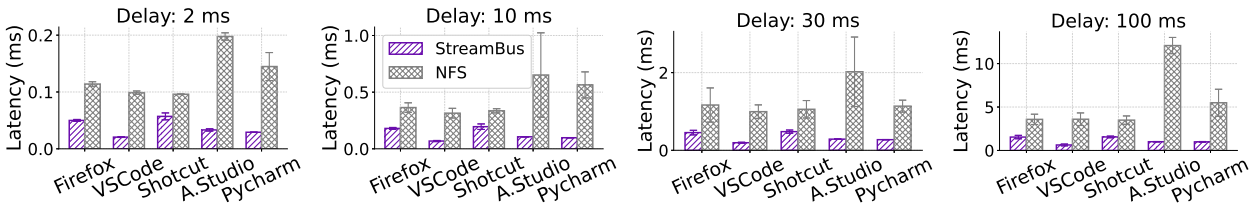

Figure 14: Average per-page access latency on desktop under varying network delays.

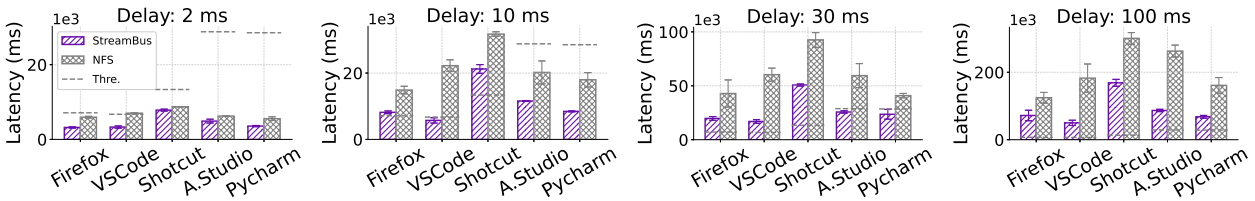

Figure 15: End-to-end application launch latency on desktop across different network delays.

and fault isolation. Orion [41] and Octopus [47] both leverage NVM and RDMA to achieve low-latency access, but with distinct optimizations for metadata and bulk data paths, respectively. GFS [9], in contrast, emphasizes fault tolerance and high throughput at scale, enabling efficient operation across thousands of nodes with commodity infrastructure. Other systems aim to provide high-performance remote storage. i10 [16] shows that TCP can match RDMA performance by minimizing CPU overhead through zero-copy and parallel I/O. RIO [8] reduces latency by overlapping computation and data transfer but still requires staging the entire dataset before execution. Tolvanen et al. [32] propose a mobile-optimized remote storage framework for Symbian OS that supports disconnected operation and enables file access via FTP.

Prefetcher. Zhang et al. [45] drew a novel connection between storage prefetching and supply chain management (SCM), applying inventory theory to dynamically manage memory allocation for concurrent access streams. By modeling per-stream demand deviation and access rates, their SCM-inspired techniques significantly improved prefetch efficiency and resource utilization. Griffioen and Appleton [12] proposed automatic file system prefetching based on past access patterns, significantly improving performance over traditional caching methods. Kaplan et al. [18] explored dynamic, workload-aware demand pre-paging, reducing page faults and improving virtual memory performance under constrained resources. MobiRA [20] addressed read-ahead inefficiencies on Android by dynamically adjusting prefetch size and stop conditions to better balance latency and cache efficiency. SARC [10] treats prefetching aggressiveness as a resource allocation problem, using supply chain management principles to dynamically balance memory across concurrent streams based on access patterns. Leap [1] is a lightweight, adaptive prefetching framework for RDMA-based remote memory systems that learns dominant access patterns to reduce page cache misses and remote access latency.

7 Conclusion

This paper presents StreamBus, a system that introduces a semantic-aware virtual storage bus abstraction to enable seamless, on-demand streaming of unmodified applications from remote storage. By bridging the semantic gap between application execution and network storage at the block layer, StreamBus eliminates installation overheads while achieving near-native responsiveness. Extensive evaluation shows that StreamBus consistently outperforms traditional approaches like NFS and, in some cases, even surpasses local NVMe storage, enabled by its dual-path transmission mechanism and thread-aware predictive prefetching. Robustness tests demonstrate stable performance under realistic network delays, confirming StreamBus's practicality for deployments within urban networks. By shifting focus from throughput-oriented remote storage to execution-aware streaming, StreamBus redefines the boundary between execution and storage, opening new opportunities for distributed systems, mobile computing, and edge-cloud integration—making seamless access a practical direction for application delivery.

Acknowledgement

This work was supported in part by the National Key R&D Program of China under Grants No. 2022YFF0604504 and 2022YFF0604502, and by the National Natural Science Foundation of China under Grant No. 62432004. Additional support was provided by the China Scholarship Council.

References

[1] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 843–857, 2020.

[2] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[3] Ketan Bhardwaj, Pragya Agrawal, Ada Gavrilowska, Karsten Schwan, and Adam Allred. Appflux: Taming app delivery via streaming. *Proc. of the Usenix TRIOS*, 2015.

[4] Ketan Bhardwaj, Matt Saunders, Nikita Juneja, and Ada Gavrilovska. Serving mobile apps: A slice at a time. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–15, 2019.

[5] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. Bbr: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time. *Queue*, 14(5):20–53, 2016.

[6] Sen Chen, Lingling Fan, Guozhu Meng, Ting Su, Minhui Xue, Yinxing Xue, Yang Liu, and Lihua Xu. An empirical assessment of security risks of global android banking apps. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1310–1322, 2020.

[7] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2187–2200, 2017.

[8] Ian Foster, David Kohr Jr, Rakesh Krishnaiyer, and Jace Mogill. Remote i/o: Fast access to distant storage. In *Proceedings of the fifth workshop on I/O in parallel and distributed systems*, pages 14–25, 1997.

[9] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

[10] 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.

[11] Inc Google. Native android apps, without the installation, 2018. https://developer.android.com/topic/google-play-instant.

[12] Jim Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. In *USENIX summer*, pages 197–207, 1994.

[13] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5): 64–74, 2008.

[14] Lei Hao, Fucheng Wan, Ning Ma, and Yicheng Wang. Analysis of the development of wechat mini program. In *Journal of Physics: Conference Series*, volume 1087, page 062040. IOP Publishing, 2018.

[15] Bean Huo, Blair Pan, Peter Pan, and Zoltan Szubbocsev. Comparing ufs and nvme™ storage stack and system-level performance in embedded systems.

[16] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. {TCP} {RDMA}:{CPU-efficient} remote storage access with i10. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 127–140, 2020.

[17] Abhinav Jangda, Bobby Powers, Emery D Berger, and Arjun Guha. Not so fast: Analyzing the performance of {WebAssembly} vs. native code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 107–120, 2019.

[18] Scott F Kaplan, Lyle A McGeoch, and Megan F Cole. Adaptive caching for demand prepaging. *ACM SIGPLAN Notices*, 38(2 supplement):114–126, 2002.

[19] Shine Kim, Jonghyun Bae, Hakbeom Jang, Wenjing Jin, Jeonghun Gong, Seungyeon Lee, Tae Jun Ham, and Jae W Lee. Practical erase suspension for modern low-latency {SSDs}. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 813–820, 2019.

[20] Yu Liang, Riwei Pan, Yajuan Du, Chenchen Fu, Liang Shi, Tei-Wei Kuo, and Chun Jason Xue. Read-ahead efficiency on mobile devices: Observation, characterization, and optimization. *IEEE Transactions on Computers*, 70(1):99–110, 2020.

[21] Ruofeng Liu and Nakjung Choi. A first look at wi-fi 6 in action: Throughput, latency, energy efficiency, and security. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 7(1): 1–25, 2023.

[22] Yi Liu, Yun Ma, Xusheng Xiao, Tao Xie, and Xuanzhe Liu. Legodroid: flexible android app decomposition and instant installation. *Science China Information Sciences*, 66(4):142103, 2023.

[23] Jun Lu, Zhenya Ma, Yinggang Gao, Sheng Yue, Ju Ren, and Yaoxue Zhang. Streamsys: A lightweight executable delivery system for edge computing. *IEEE Transactions on Cloud Computing*, 2024.

[24] Stuart McIlroy, Nasir Ali, and Ahmed E Hassan. Fresh apps: an empirical study of frequently-updated mobile apps in the google play store. *Empirical Software Engineering*, 21:1346–1370, 2016.

[25] Microsoft. Microsoft app-v, 2015. https://learn.microsoft.com/en-us/microsoft-desktop-optimization-pack/app-v/appv-for-windows.

[26] Maleknaz Nayebi, Bram Adams, and Guenther Ruhe. Release practices for mobile apps–what do users and developers think? In *2016 ieee 23rd international conference on software analysis, evolution, and reengineering (saner)*, volume 1, pages 552–562. IEEE, 2016.

[27] Novell. Novell zenworks application virtualization, 2012. https://www.novell.com/documentation/zav90/esd/di_zav90.html.

[28] Numecent. Numecent cloudpaging. https://www.numecent.com/cloudpaging/.

[29] Selim Ozcan, Ioana Livadariu, Georgios Smaragdakis, and Carsten Griwodz. Longitudinal analysis of inter-city network delays. In *2023 7th Network Traffic Measurement and Analysis Conference (TMA)*, pages 1–9. IEEE, 2023.

[30] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and Dave Hitz. Nfs version 3: Design and implementation. In *USENIX Summer*, pages 137–152. Boston, MA, 1994.

[31] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. {AIFM}:{High-Performance},{Application-Integrated} far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 315–332, 2020.

[32] Jarkko Tolvanen, Tapio Suihko, Jaakko Lipasti, and N Asokan. Remote storage for mobile devices. In *2006 1st International Conference on Communication Systems Software & Middleware*, pages 1–9. IEEE, 2006.

[33] Verizon. Ip latency statistics. https://www.verizon.com/business/terms/latency/.

[34] W3C. Webassembly, 2017. https://webassembly.org/.

[35] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. An empirical study of usages, updates and risks of third-party libraries in java projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–45. IEEE, 2020.

[36] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, 2006.

[37] WonderNetwork. Global ping statistics. https://wondernetwork.com/pings.

[38] Fengguang Wu, Hongsheng Xi, Jun Li, and Nanhai Zou. Linux readahead: less tricks for more. In *Proceedings of the Linux Symposium*, volume 2, pages 273–284. Citeseer, 2007.

[39] Dongzhu Xu, Anfu Zhou, Xinyu Zhang, Guixian Wang, Xi Liu, Congkai An, Yiming Shi, Liang Liu, and Huadong Ma. Understanding operational 5g: A first measurement study on its coverage, performance and energy consumption. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 479–494, 2020.

[40] Yutian Yan, Tengfei Tu, Lijian Zhao, Yuchen Zhou, and Weihang Wang. Understanding the performance of webassembly applications. In *Proceedings of the 21st ACM Internet Measurement Conference*, pages 533–549, 2021.

[41] Jian Yang, Joseph Izraelevitz, and Steven Swanson. Orion: A distributed file system for {Non-Volatile} main memory and {RDMA-Capable} networks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 221–234, 2019.

[42] Deyu Zhang, Yu Xie, Mucong Xu, En Cheng, Xiaoyan Kui, Bangwen He, and Yunhao Li. Troy: Efficient service deployment for windows systems. *Chinese Journal of Electronics*, 33(1):313–322, 2024.

[43] Yaoxue Zhang and Yuezhi Zhou. Transparent computing: A new paradigm for pervasive computing. In *International Conference on Ubiquitous Intelligence and Computing*, pages 1–11. Springer, 2006.

[44] Yaoxue Zhang, Sijing Duan, Deyu Zhang, and Ju Ren. Transparent computing: Development and current status. *Chinese Journal of Electronics*, 29(5):793–811, 2020.

[45] Zhe Zhang, Amit Kulkarni, Xiaosong Ma, and Yuanyuan Zhou. Memory resource allocation for file system prefetching: from a supply chain management perspective. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 75–88, 2009.

[46] Yang Zhou, Hassan MG Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E Culler, Henry M Levy, et al. Carbink:{Fault-Tolerant} far memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 55–71, 2022.

[47] Bohong Zhu, Youmin Chen, Qing Wang, Youyou Lu, and Jiwu Shu. Octopus+: An rdma-enabled distributed persistent memory file system. *ACM Transactions on Storage (TOS)*, 17(3):1–25, 2021.