Beyond System Calls: Uncovering Hidden I/O Behavior of mmap-Based On-Device LLMs

Heejin Kim
Department of Computer Engineering
Ewha Womans University
Seoul, Republic of Korea
hjhj4542@ewha.ac.kr

Jeongha Lee
Department of Computer Engineering
Ewha Womans University
Seoul, Republic of Korea
jeongha.lee@ewha.ac.kr

Hyokyung Bahn*
Department of Computer Engineering
Ewha Womans University
Seoul, Republic of Koera
bahn@ewha.ac.kr

***Abstract* — The growing deployment of on-device large language models (LLMs) on smartphones has accelerated the adoption of memory-mapped I/O (mmap) as the default mechanism for loading model weights. Unlike traditional system call based access, mmap defers data loading to page faults, making storage activity invisible at the system-call level and requiring kernel-level tracing to understand real execution behavior. This paper uses ftrace to collect page-fault-driven page-cache loading events for mmap-based model access and uses this approach to analyze four representative on-device LLM workloads: text interaction, vision-language reasoning, audio transcription, and text-to-image generation. Our measurements show that mmap-based execution continues to perform large, predominantly sequential scans of model-weight files at each configuration or inference phase, with effective working-set sizes ranging from approximately 500 MB to over 3 GB. These findings indicate that the primary bottleneck for on-device LLM performance is the amount of DRAM available to maintain model pages in the page cache, rather than the choice of I/O interface. Ensuring sustained inference performance thus requires keeping a substantial portion of model weights resident in memory. Otherwise, page eviction leads to repeated major faults and degraded latency. The results highlight the need for memory-aware model deployment strategies, OS-level LLM-informed caching policies, and footprint-reduction techniques to support increasingly large models on resource-constrained mobile devices.**

***Keywords* — on-device LLM, memory-mapped I/O, page-fault, model weight, access pattern, mobile AI systems**

I. Introduction

Recent advances in mobile hardware and model compression technologies have accelerated the deployment of on-device large language models (LLMs) [1–3]. Modern smartphones are now capable of running multi-billion parameter models locally, enabling interactive AI features such as conversational assistants [4], vision-language reasoning [5], and audio transcription [6] without relying on remote servers [7, 8]. As these models typically consist of hundreds of megabytes to several gigabytes of weight files stored on local flash storage, the manner in which LLM applications access model files has become a primary determinant of end-to-end inference latency. Prior work has shown that storage bottlenecks can significantly degrade on-device AI responsiveness, especially during initial model loading and repeated inference cycles [9, 10].

Research on AI inference performance has traditionally centered on cloud-based execution, where mobile devices function as thin clients and the primary challenge lies in scheduling multi-tenant workloads [11, 12]. In contrast, on-device inference presents a fundamentally different set of constraints. Because execution occurs on resource-limited personal devices and serves a single user at a time, efficient memory utilization and the mitigation of storage bottlenecks become essential for stable inference latency [13, 14].

A growing body of system-level studies has analyzed storage behaviors of AI workloads using traditional file I/O system calls such as read() and write() [15]. These analyses model I/O volume, sequentiality, and cache hit ratios in terms of explicit read/write requests observable at the system-call layer. However, modern commercial LLM applications rarely rely on system call operations. Instead, they increasingly adopt memory-mapped I/O (mmap) to load model weights [16]. With mmap, the model file is mapped into the application's virtual address space, and physical data loading is deferred until the corresponding virtual pages are actually accessed. This demand-paging mechanism eliminates explicit read calls and performs data transfers implicitly through major page faults, thereby reducing redundant copying between kernel and user space and improving temporal locality for large, read-intensive model parameters.

While mmap offers substantial benefits for inference workloads, it fundamentally changes the observability and structure of storage accesses. Unlike explicit file I/O, mmap-based accesses do not emit system calls that reflect data-transfer events. The actual I/O occurs inside the kernel's page-fault handler path, where page faults trigger filesystem operations such as readpage. As a result, existing read/write-based analyses fail to capture when and how LLM applications truly access their model files. In particular, the latency-critical major page faults during initial model loading are invisible at the user level and cannot be accurately measured through tools such as strace [17]. Thus, understanding the performance characteristics of modern on-device LLMs requires page-fault-level tracing and analysis of memory-mapped execution.

Accordingly, this work examines the file access behavior of on-device LLM applications through page-fault-level tracing. Because the demand-paging behavior of mmap-based loading is not visible from file-I/O system calls, we use ftrace [18] to

capture kernel executions of fault-handling routines and relevant address_space_operations, and post-process these events to observe how model file is actually retrieved during execution. Our event-level analysis shows that mmap-based LLM applications still exhibit large, long-range sequential scans of their model-weight files—mirroring the sequential patterns previously reported under read/write-based analyses and standing in sharp contrast to the largely non-sequential behavior of non-AI workloads. By revealing these distinct access behaviors, this study provides a clearer view of how on-device LLM inference interacts with the underlying storage system. The contributions of this work are as follows:

- **mmap-aware trace extraction tool.**
  In contrast to prior studies that rely on system-call-level file-I/O logs, we trace page-fault events to observe demand-paging behavior that system-call traces cannot reveal.
- **Characterization of page-fault-level LLM access patterns.**
  Using real-world LLM applications, we analyze how model weights are accessed under demand paging and show that mmap-driven loading produces highly selective and non-sequential access flows, fundamentally different from the repetitive and largely sequential scans reported in prior file-I/O-based AI workload studies.
- **Comparison with traditional file I/O behaviors.**
  Based on the measured working-set sizes and reuse characteristics of mmap-based LLM execution, we show that memory capacity—not the choice of I/O interface—dominates inference stability on mobile hardware. These findings highlight practical optimization directions, including working-set reduction, eviction-aware page management, and transition-aware prefetching.

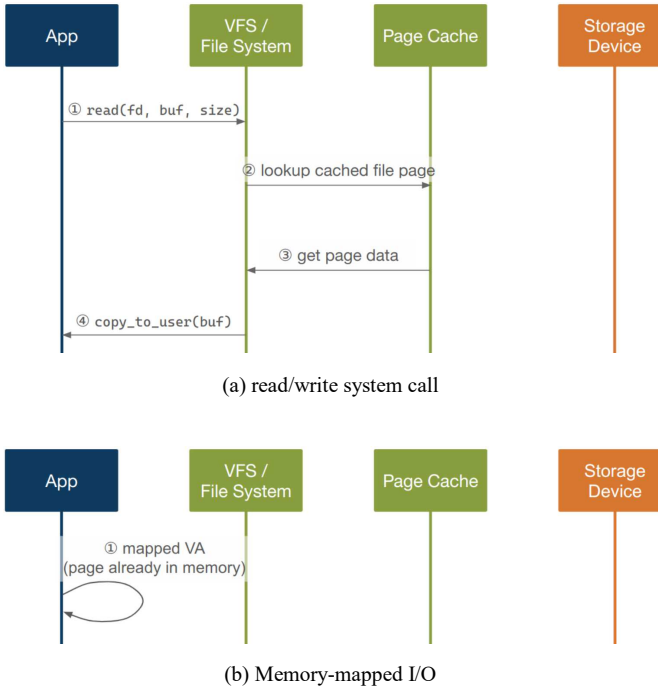
The remainder of this paper is organized as follows. Section II provides preliminaries on traditional file I/O and memory-mapped I/O, focusing on the aspects relevant to interpreting mmap-induced storage behavior. Section III describes our mmap-aware tracing methodology and experimental environment. Section IV presents detailed analyses of page-fault-level model access patterns. Section V discusses system-level implications. Finally, Section VI concludes the paper.

II. Preliminaries

Modern on-device LLM applications increasingly adopt memory-mapped I/O as the default mechanism for loading large model files. Unlike traditional read-based file access, mmap exposes file contents directly through the virtual address space and shifts actual data loading to the moment a page is touched. Because the primary objective of this work is to analyze such demand-paging behavior through page-fault traces, this section provides a brief overview of the mechanisms underpinning traditional file I/O and mmap. The goal is not to give a full OS-level description, but to clarify the aspects that affect how storage activity appears in the traces analyzed in later sections.

A. Cold Access

In traditional file I/O, an application explicitly issues a system call (e.g., read) to request data at a specific file offset. The kernel then checks the page cache and, upon a miss, loads the corresponding file page from storage before copying it into the user's buffer. Thus, the initiation of a cold access is tied directly to system-call execution.



(a) read/write system call

(b) Memory-mapped I/O

Fig. 1. Comparison of read/write and mmap paths during warm access.

Under mmap, a file region is first mapped into the process's address space without reading any data. The file contents are brought into memory only when the application first dereferences a mapped virtual address. This produces a page fault, which the kernel handles by fetching the corresponding file page from storage and inserting it into the page cache. After the page table is updated, the original instruction completes normally.

Although the initial trigger differs—explicit system call versus implicit page fault—the internal path for obtaining a cold page (page-cache lookup, offset translation, block I/O, and page-cache population) is essentially identical for both mechanisms. The difference lies in what is observable externally: storage activity initiated by mmap does not appear in system-call traces, because the cause is a CPU page fault rather than a user-visible read request.

B. Warm Access

Once file pages are resident in the page cache, the reuse paths of the two mechanisms diverge significantly. In read-based I/O, each re-access still requires invoking a system call, entering the kernel, checking the page cache again, and copying data into a user buffer as shown in Figure 1(a). Even though no disk I/O occurs, the application repeatedly incurs syscall and copying overheads.

In contrast, mmap enables warm accesses to bypass the kernel entirely. If the page is cached and its page-table entry remains valid, the CPU resolves the virtual address in user space, and the access completes as a normal memory operation. No

page fault, system call, or copying is involved. This behavior makes mmap particularly well suited for LLM inference, where certain parts of the model may be repeatedly referenced across multiple tokens or prompts. Figure 1(b) illustrates the warm-access path of mmap, in contrast to the read/write system-call path shown in Figure 1(a).

C. Relevance to On-Device LLMs

For on-device LLM workloads, model-weight files are large, read-mostly, and frequently revisited. mmap therefore provides substantial benefits by eliminating the overhead of repeated system calls and user–kernel buffer copies during inference. However, because mmap defers data loading to page faults, system-call-based methods cannot reveal when or where actual file I/O occurs. Key performance characteristics—such as which weight-file regions are accessed, how much of the model must remain resident to avoid major faults, and when eviction forces reloading—are observable only through kernel-level tracing.

This study adopts a page-fault-centric approach for this reason. Using ftrace, we capture demand-paging and filesystem events generated during inference, reconstruct the storage access stream at page granularity, and analyze real LLM access patterns in Sections III and IV.

III. Trace Collection

In mmap-based file access, disk I/O does not manifest through explicit system calls. Instead, it is performed internally by the kernel only when a mapped page is not present in the page cache and a page fault occurs. As a result, storage activity leaves no trace in user-level system-call logs, and tools such as strace cannot observe when or how much I/O is issued during execution.

To analyze the actual I/O behavior of mmap-based workloads, we use ftrace, which enables tracing of internal kernel events. In particular, we collect filesystem-level address_space_operations (e.g., readpage and related functions) that are invoked when the filesystem reads a page from disk. By capturing these events, we identify storage accesses triggered by memory-mapped I/O during application execution and obtain visibility into the file pages loaded as a result of page faults. This allows us to reconstruct the true I/O behavior of mmap-based LLM workloads.

The raw trace consists of kernel events containing timestamps, filenames, file offsets, page-cache operations, and other relevant metadata. To enable consistent analysis of access locality and reuse patterns, we normalize all events into 4 KB, file-offset-aligned block IDs. Each block represents a fixed region of a file. When a page fault loads data for a particular offset, we map it to the corresponding block ID. Repeated faults on the same offset map to the same block. Newly observed offsets result in new block IDs. This yields a time-ordered block-level trace that clearly exposes sequential runs, discontinuities, and selective access patterns in the model's weight files.

All experiments were conducted on a Samsung Galaxy S22 smartphone running a vendor-custom Android kernel [19]. The device specification is summarized in Table I. The S22 represents a typical mid-to-high-end mobile environment for today's on-device LLM applications.

TABLE 1. Experimental device specifications

Component	Specification
CPU	1× Cortex-X2 @ 3.0 GHz + 3× Cortex-A710 @ 2.5 GHz + 4× Cortex-A510 @ 1.8 GHz
GPU	Adreno 730
RAM	8 GB
Storage	256 GB UFS 3.x
OS	Android (Linux kernel)

To analyze mmap-based access patterns across diverse modalities and model architectures, we evaluate four LLM applications. These workloads originate from Google AI Edge Gallery [20] and MediaPipe Image Generator [21]. Google AI edge gallery includes three user-facing applications—AI Chat, Ask Image, and Audio Scribe—which provide text interaction, image-based queries, and audio transcription, respectively. Although they are bundled within a single host application, these components represent distinct usage scenarios. For clarity and to avoid mixing semantically different operations, we extract traces for each component separately, treating them as independent workloads in our analysis. The workloads used in this study are summarized as follows.

- **AI Chat [20] (Gemma-1B-IT-q4)**
  Multi-turn conversational LLM. The workload includes two configuration updates followed by several text queries. It generates 570 MB of page-fault-driven reads.
- **Ask Image [20] (Gemma-3n-E3-2B-it)**
  A vision-language model supporting image-based queries. During the experiment, the user provides gallery or camera images and asks related questions. This workload induces 2458.35 MB of file accesses.
- **Audio Scribe [20] (Gemma-3n-E3-2B-it)**
  Speech-to-text transcription and translation. The user records or uploads audio clips and poses follow-up questions. It generates 3883.25 MB of model-file loads.
- **Image Generator [21] (Stable Diffusion v1.5)**
  Text-to-image generation. Two prompts with varying seed values are used to produce multiple images, resulting in 1818.95 MB of storage access.

IV. mmap Access Characteristics of On-Device LLM

Using the ftrace-based tracing method described in Section III, we analyze the page-fault-driven memory-mapped I/O behavior of four on-device LLM applications. For each application, we reconstruct the time-ordered sequence of block offsets accessed during model loading and inference. This section presents our findings, focusing on whether modern LLM applications still exhibit sequential model-weight scans—an access pattern frequently observed in earlier read/write-based studies—or whether mmap changes the underlying structure of file access. Across all workloads, we find that sequential weight-file traversal remains a dominant characteristic, even though the access stream includes minor non-sequential segments corresponding to metadata reads and application-level caches.

A. AI Edge Gallery – AI Chat

As shown in Figure 2(a), three distinct model-configuration and inference phases were observed. During the first phase, the application issues a long sequential scan of what appears to be its primary model-weight file, starting from block ID 0 and extending to approximately block 90,000. This initial scan is followed by intermittent activity concentrated around the 90,000 block region, which corresponds to small, irregular accesses likely associated with internal application cache files.

The second phase also begins with a sequential traversal starting from block 0, but it extends much further—reaching block IDs near 140,000. This indicates that the second configuration loads a larger or differently structured model file compared to the first phase.

During the third phase, the same weight-file range used in the second phase is accessed again. However, block regions between 100,000 and 140,000 do not trigger page faults and thus do not appear in the trace. This absence suggests either that these segments were not required by the inference path or, more likely, that the corresponding pages remained in memory and were accessed without causing faults.

Overall, AI Chat consistently performs large sequential scans at the beginning of each phase, demonstrating that mmap-based inference preserves traditional sequential weight loading behavior, even though the triggering mechanism is page-fault based rather than read-based.

B. AI Edge Gallery – Ask Image

The Ask Image workload consists of four image-query phases. Each phase shows a combination of sequential weight-file scans and accesses to application-level cache files.

Upon the initial launch, the application first touches metadata-related files, scanning block IDs beginning at 0 as shown in Figure 2(b). Subsequent phases repeatedly issue long sequential scans of the main model-weight file, typically up to around block 250,000. When new prompts are issued under the same configuration, some of these accesses generate no page faults and consequently appear only faintly in the trace, reflecting memory hits rather than storage activity.

In the final phase, an entirely new region of the weight file is activated. This results in a new sequential scan between block IDs 400,000 and 600,000. The emergence of this previously untouched region indicates that additional model functionality or larger parameter segments are engaged for certain question types.

These patterns confirm that Ask Image repeatedly loads large, contiguous sections of its weight file during inference and occasionally activates new regions depending on query type.

C. AI Edge Gallery – Audio Scribe

The Audio Scribe workload involves three valid configuration-and-inference cycles. Although the trace visually suggests additional phases, those segments reflect failed inference attempts and can be disregarded.

During the first phase, the application performs a long sequential scan that extends up to approximately block ID 750,000 as shown in Figure 2(c). This primarily reflects sequential loading of a very large model-weight file, potentially in conjunction with additional segments stored in the application's cache directory. In the second phase, accesses appear in the same general region but generate few major page faults, leaving only partial traces. This indicates that many pages remained in memory and were serviced without faulting.

The third phase exhibits two distinct sequential regions. The first spans block IDs around 0–300,000, similar to the earlier phases. The second, however, begins near block 800,000 and continues beyond this point. This new high-offset region corresponds to the activation of an additional weight-file component when the application performs translation, which is a separate feature within Audio Scribe. Thus, the third phase loads both the earlier model regions and a new segment required for translation functionality.

In total, Audio Scribe demonstrates the largest working set among the evaluated workloads, activating more than 800,000 blocks throughout its execution.

D. MediaPipe – Image Generator

The Image Generator application also shows clear sequential scans of its model-weight files. During the first two phases, the application accesses contiguous block regions in a manner similar to the previous workloads. However, in later phases, the trace contains no new page-fault entries for weight-file offsets. This behavior suggests that the application did not alter its model configuration for the new prompts, and therefore the relevant pages were already resident in memory. As a result, subsequent accesses were served entirely from DRAM without generating additional faults.

E. Summary of Findings

Across all four applications, mmap-based LLM inference exhibits large sequential scans of model-weight files at phase boundaries. These scans may differ in size depending on the model component activated by each phase, but the dominant pattern remains consistent: initialization- or feature-driven sequential traversal of substantial portions of the model file. Variation between phases reflects either the use of additional model modules or changes in functionality (e.g., translation vs. transcription). Despite the implicit nature of mmap-induced I/O, the underlying access behavior closely mirrors that of traditional read-based systems: long sequential reads followed by localized reuse.

V. System-Level Implications

Our analysis shows that mmap-based on-device LLM inference continues to rely on large sequential scans of model-weight files. Although mmap eliminates explicit read() system calls and permits efficient reuse of cached pages, the underlying requirement—that the device must maintain a sufficiently large fraction of the weight-file footprint in memory—remains unchanged. The maximum block IDs observed across the four workloads provide a direct indication of the effective working-set size, because each block corresponds to a 4 KB page in the page cache. Using this mapping, AI Chat accesses approximately 140,000 blocks (about 570 MB), Ask Image

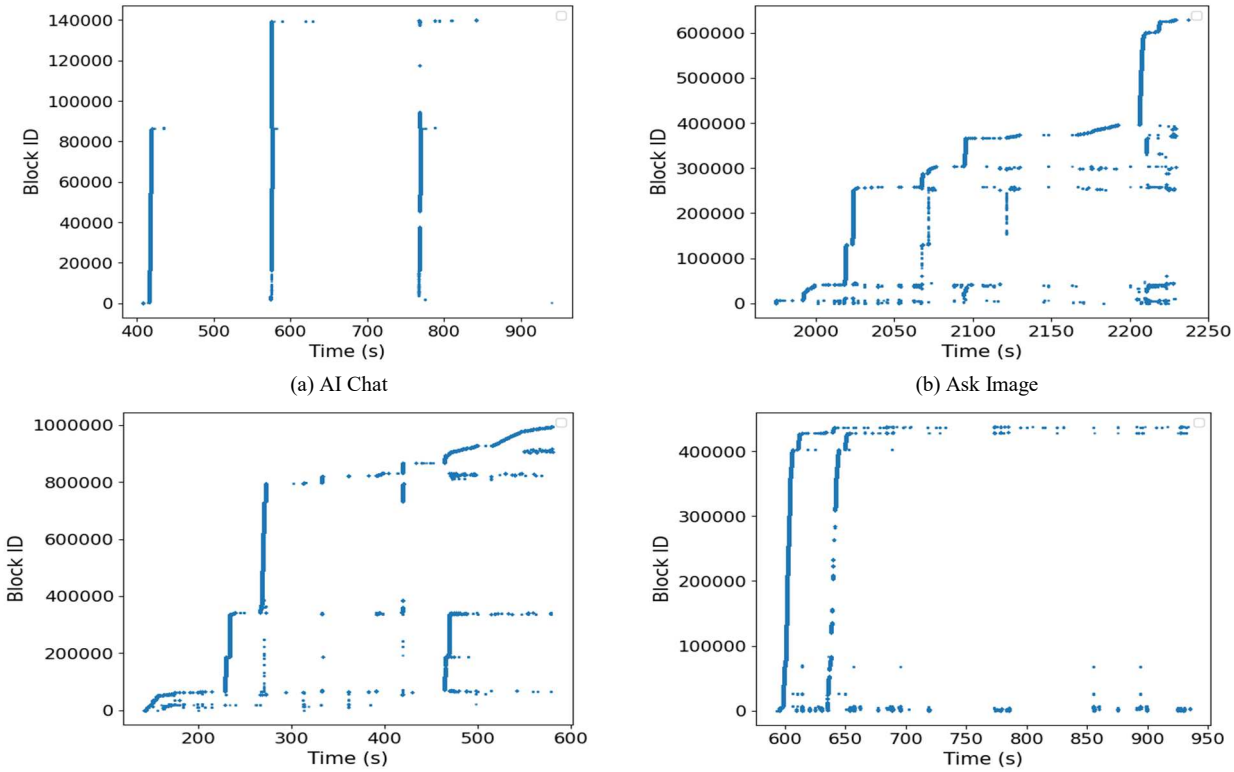


Fig. 2. Page-fault-level file-access patterns of on-device LLM inference under mmap.

reaches 620,000 blocks (about 2.5 GB), Audio Scribe touches roughly 950,000 blocks (about 3.9 GB), and the Image Generator workload uses around 450,000 blocks (about 1.8 GB). These values demonstrate that even with mmap, models of modest scale still require hundreds of megabytes to several gigabytes of DRAM residency in order to avoid major page faults during inference.

Prior study on system-call-based LLM I/O examined how much data must reside in the buffer cache to maintain acceptable inference latency [22]. A similar principle emerges in the mmap setting: the amount of memory required to prevent repeated page faults is essentially the working-set size, now observable as the maximum contiguous span of model offsets referenced during execution. In effect, the mmap working set is the maximum block ID multiplied by the 4 KB page size. This value directly reflects the memory footprint required for stable inference, regardless of how the data is accessed.

Whether a device can retain such a working set depends on the actual DRAM available to the page cache. Although a device such as the Galaxy S22 nominally provides 8 GB of memory, a substantial portion is consumed by the operating system kernel, system services, GPU and neural-accelerator reservations, and background Android processes. What remains available to the page cache is typically far smaller than the physical memory capacity. As a result, workloads with 2–3 GB working sets, or especially those exceeding 3 GB such as Audio Scribe, inevitably suffer page evictions under pressure. This explains why repeated major faults appear even during later inference phases, despite the model having previously accessed those regions.

These observations lead to broader implications for the design of on-device LLM systems. First, mmap improves performance not by reducing the model's memory footprint but by lowering the overhead of re-accessing cached data. Once the initial faults have populated the page cache, subsequent accesses bypass the kernel entirely and are resolved at memory speed. However, this benefit materializes only if the relevant pages remain resident. Thus, the feasibility of on-device LLM deployment is fundamentally tied to whether the device can allocate enough DRAM to preserve the working set. For transformers such as the Gemma family, the working set often manifests as a large, nearly contiguous region of several hundred thousand blocks, whereas diffusion models such as Stable Diffusion tend to access several distinct contiguous regions during different submodules, resulting in multiple medium-sized working sets rather than a single monolithic one.

A second implication is that the interaction between the OS page-cache replacement policy and LLM access patterns becomes a major determinant of performance. When memory pressure forces eviction of parameter pages, inference latency degrades sharply due to renewed page faults. Devices with aggressive background memory reclamation or limited free memory are likely to experience such interruptions regardless of mmap's theoretical efficiency.

Finally, these findings suggest that future on-device LLM optimization efforts should focus more on memory-aware

strategies. Reducing the working-set size through model partitioning or quantization, improving OS-level heuristics to protect high-reuse parameter pages from eviction, and designing smarter prefetch strategies that anticipate module transitions may substantially improve inference stability. Similarly, understanding how different model architectures distribute their parameter access across the file can help guide fine-grained layout optimizations or runtime planning.

In summary, while mmap-based execution changes the mechanics of how model weights are loaded, it does not relieve the fundamental memory constraints associated with on-device LLM inference. The effective working-set sizes observed—ranging from roughly half a gigabyte to more than three gigabytes—demonstrate that memory capacity, rather than the choice of I/O interface, remains the primary bottleneck for sustaining predictable latency on resource-constrained mobile hardware.

VI. Conclusion

This paper analyzed the file-access characteristics of modern on-device LLM applications that rely on memory-mapped I/O rather than traditional read-based loading. By tracing page-fault-driven filesystem activity through ftrace, we showed that mmap changes the visibility of I/O but not the underlying structure of model-weight access. All evaluated workloads still perform large sequential scans of their weight files during model initialization and inference phases, with effective working-set sizes ranging from several hundred megabytes to multiple gigabytes. These results demonstrate that the primary performance determinant for on-device LLM inference is the amount of DRAM available to retain model pages in the page cache. While mmap provides a low-overhead reuse path once data is resident, any eviction caused by memory pressure induces renewed page faults and degrades responsiveness. Thus, sustaining stable inference on mobile hardware ultimately depends on memory capacity rather than the choice of I/O interface. Future work should explore techniques for reducing or restructuring the working set—such as model partitioning, better quantization layouts, and LLM-aware OS memory management—to improve the feasibility of running increasingly large models on resource-constrained devices.

Acknowledgment

This work was supported in part by the National Research Foundation of Korea (NRF) under Grant RS-2024-00461678 and RS-2025-02214322 funded by Korean Government (MSIT). H. Bahn is the corresponding author of this paper.

References

[1] D. Xu, W. Yin, H. Zhang, X. Jin, Y. Zhang, and S. Wei, "EdgeLLM: Fast On-Device LLM Inference with Speculative Decoding," IEEE Transactions on Mobile Computing, vol. 24, no. 4, pp. 3256-3273, 2025, doi: 10.1109/TMC.2024.3513457.

[2] Z. Yu, S. Liang, T. Ma, Y. Cai, Z. Nan, and D. Huang, "Cambricon-LLM: A Chiplet-Based Hybrid Architecture for On-Device Inference of 70B LLM," Proc. 57th Int'l Symp. on Microarchitecture (MICRO), pp. 1474-1488, 2024, doi: 10.1109/MICRO61859.2024.00108.

[3] T. Wang, R. Fan, M. Huang, Z. Hao, K. Li, T. Cao, Y. Lu, Y. Zhang, and J. Ren, "Neuralink: Fast On-Device LLM Inference with Neuron Co-Activation Linking," Proc. 30th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 147-162, 2025, doi: 10.1145/3676642.3736114.

[4] Google, "Meet Gemini, your helpful built-in AI assistant," available at https://store.google.com/intl/en/ideas/gemini-ai-assistant/

[5] Meta, "Llama 3.2: Revolutionizing edge AI and vision with open, customizable models," available at https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/

[6] B. Durmus, A. Okan, E. Pacheco, Z. Nagengast, and A. Orhon, "WhisperKit: On-device Real-time ASR with Billion-Scale Transformers," Proc. ICML Workshop on the Next Wave of On-Device Learning for Foundational Models, pp. 1-7, 2025.

[7] L. Chen, D. Feng, E. Feng, Y. Wang, R. Zhao, Y. Xia, P. Xu, and H. Chen, "Characterizing Mobile SoC for Accelerating Heterogeneous LLM Inference," Proc. 31st ACM Symp. on Operating Systems Principles (SOSP), pp. 359-374, 2025, doi: 10.1145/3731569.3764808.

[8] Z. Wang, J. Yang, X. Qian, S. Xing, X. Jiang, C. Lv, and S. Zhang, "MNN-LLM: A Generic Inference Engine for Fast Large Language Model Deployment on Mobile Devices," Proc. 6th Int'l Conf. on Multimedia in Asia Workshops, 2024, doi: 10.1145/3700410.3702126.

[9] X. Wang, Z. Tang, J. Guo, T. Meng, C. Wang, T. Wang, and W. Jia, "Empowering Edge Intelligence: A Comprehensive Survey on On-Device AI Models," ACM Computing Surveys, vol. 57, no. 9, pp. 1-39, 2025, doi: 10.1145/3724420.

[10] J. Lee, S. Lim, and H. Bahn, "Analyzing Data Access Characteristics of AIoT Workloads for Efficient Write Buffer Management," IEEE Internet of Things Journal, vol. 12, no. 15, pp. 31601-31614, 2025, doi: 10.1109/JIOT.2025.3573759.

[11] S. Kwon and H. Bahn, "Evolutionary Computation-Based Scheduling of Machine Learning Workloads for GPU Clusters," Proc. 5th IEEE Int'l Conf. on Advances in Electrical Engineering and Computer Applications (AEECA), pp. 697-701, 2024, doi: 10.1109/AEECA62331.2024.00123.

[12] Y. Jin and Z. Yang, "Scalability Optimization in Cloud-Based AI Inference Services: Strategies for Real-Time Load Balancing and Automated Scaling," Proc. 4th Int'l Conf. on Big Data, Information and Computer Network, pp. 266-270, 2025, doi: 10.1145/3727353.3727398.

[13] S. Kwon and H. Bahn, "Memory Reference Analysis and Implications for Executing AI Workloads in Mobile Systems," Proc. IEEE Int'l Conf. on Electrical and Information Technology, pp. 281-285, 2023, doi: 10.1109/IEIT59852.2023.10335577.

[14] Y. Zhang, J. Zhang, S. Yue, W. Lu, J. Ren, X. Shen, "Mobile Generative AI: Opportunities and Challenges," IEEE Wireless Communications, vol. 31, no. 4, pp. 58-64, 2024, doi: 10.1109/MWC.006.2300576.

[15] J. Lee, S. Lim, and H. Bahn, "Analyzing File Access Characteristics for Deep Learning Workloads on Mobile Devices," Proc. 5th IEEE Int'l Conf. on Advances in Electrical Engineering and Computer Applications, pp. 417-422, 2024, doi: 10.1109/AEECA62331.2024.00079.

[16] PyTorch, "Using ExecuTorch on iOS," available at http://docs.pytorch.org/executorch/stable/using-executorch-ios.html#id3

[17] Strace, available at https://strace.io/

[18] Ftrace, available at https://docs.kernel.org/trace/ftrace.html

[19] Galaxy S22, available at https://www.samsungmobilepress.com/media-assets/galaxy-s22

[20] Google AI Edge Gallery, available at http://play.google.com/store/apps/details?id=com.google.ai.edge.gallery

[21] Google-AI-Edge Mediapipe-samples, available at https://github.com/google-ai-edge/mediapipe-samples/tree/main/examples/image_generation

[22] H. Kim, J. Lee, and H. Bahn, "Rethinking I/O Caching for Large Language Model Inference on Resource-Constrained Mobile Platforms," Mathematics, vol. 13, no. 22, article 3689, pp. 1-17, 2025, doi: 10.3390/math13223689.