

Article

Rethinking I/O Caching for Large Language Model Inference on Resource-Constrained Mobile Platforms

Heejin Kim [†], Jeongha Lee [†]  and Hyokyung Bahn ^{*} 

Department of Computer Engineering, Ewha University, Seoul 03760, Republic of Korea; hjhj4542@ewha.ac.kr (H.K.); jeongha.lee@ewha.ac.kr (J.L.)

^{*} Correspondence: bahn@ewha.ac.kr

[†] These authors contributed equally to this work.

Abstract

Large language models (LLMs) have traditionally relegated inference to remote servers, leaving mobile devices as thin clients. Recently, advances in mobile GPUs and NPUs have made on-device inference increasingly feasible, particularly for privacy-sensitive and personalized applications. However, executing LLMs directly on resource-constrained devices exposes severe I/O bottlenecks, as repeated accesses to large weight files can overwhelm limited memory and storage bandwidth. Prior studies have focused on internal mechanisms such as KV caching, while the role of the host OS buffer cache remains under-explored. This paper closes that gap with file-level trace analysis of real-world mobile LLM applications, and identifies three characteristic access patterns: (1) one-time sequential scans during initialization, (2) persistent hot sets (e.g., tokenizers, metadata, indices), and (3) recurring loop accesses to model weight files. Guided by these observations, we propose LLM-aware buffer cache strategies and derive cache-sizing guidelines that relate loop size, host-set coverage, and storage bandwidth. We further compare smartwatch-class and smartphone-class platforms to clarify feasible model sizes and practical hardware prerequisites for local inference. Our results provide system-level guidance for I/O subsystem design that enables practical on-device LLM inference in future mobile and IoT devices.

Keywords: large language model; I/O caching; mobile platform; buffer cache; inference

MSC: 68M20



Academic Editor: Radi Romansky

Received: 13 October 2025

Revised: 14 November 2025

Accepted: 14 November 2025

Published: 17 November 2025

Citation: Kim, H.; Lee, J.; Bahn, H. Rethinking I/O Caching for Large Language Model Inference on Resource-Constrained Mobile Platforms. *Mathematics* **2025**, *13*, 3689. <https://doi.org/10.3390/math13223689>

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Large language model (LLM) applications such as ChatGPT have traditionally treated mobile devices as dummy terminals, where actual inference is performed in the cloud and devices only display results [1,2]. With recent advances in mobile GPU and NPU technologies, however, partial or even full on-device inference is becoming increasingly feasible [3,4]. Modern chipsets (e.g., Apple A17 Pro [5], Snapdragon 8 Gen 3 [6], and Samsung Exynos [7]) integrate high-performance neural accelerators, and commercial platforms now demonstrate practical on-device inference deployments (e.g., CoreML-based Llama 3/3.1 on iOS [8] and Qualcomm-based demos on Android [9]). As LLMs become more personalized, privacy concerns further increase the appeal of on-device execution [10].

Unlike server-class environments, mobile and wearable devices are constrained by limited memory, storage bandwidth, and energy budgets [11,12]. In particular, repeated accesses to massive weight files create I/O bottlenecks that cannot be masked by compute

throughput alone. Such overheads can be alleviated by systematically analyzing data access and resource utilization patterns and mitigating bottlenecks accordingly [13,14]. While prior research has extensively examined model-internal mechanisms such as key-value (KV) caching [15,16], relatively little attention has been paid to the host operating system's buffer cache and its role in sustaining efficient I/O for LLM workloads. KV caching is sensitive to token order and cannot reduce accesses to large weight files in storage, limiting its overall effectiveness. Another line of research, semantic caching, has also been explored for LLM workloads but is limited by the computational cost of embedding-similarity matching. In contrast, we argue that system-level buffer caching on mobile platforms can alleviate the heavy I/O traffic to flash storage caused by repeated weight-file accesses, without semantic comparison or token-level dependency considerations. Although buffer caching has been studied extensively, LLM inference exhibits unique file-access patterns during model execution, which often lead to frequent cache thrashing and underutilized memory when generic LRU or traditional caching strategies are applied.

In this paper, we address this gap through fine-grained analysis of file-access traces collected from mobile LLM applications and discuss how the findings can be reflected in buffer cache design specialized for on-device LLMs. Using a Samsung Galaxy S22 testbed, we compare on-device inference apps with remote-inference apps (Copilot, Perplexity, Claude, and Gemini). We observe three distinctive access characteristics in on-device LLM workloads: (1) long, recurring loop references to model weight files, (2) persistent accesses to small hot sets such as tokenizers, metadata, and index files, and (3) one-time sequential scans for initial setup. These patterns are fundamentally different from the sequential, largely one-off access behaviors of remote-inference apps, underscoring the need for LLM-aware caching at the OS level.

Building on these observations, we present a buffer cache model tailored to LLM workloads. Rather than focusing solely on replacement policies, our model explicitly relates cache capacity, loop size, and storage bandwidth to determine when inference can proceed without I/O stalls. We also analyze how device class—smartwatches with eMMC storage versus smartphones with UFS storage—constrains viable model sizes and cache allocation. Our findings indicate that smartwatch-class devices realistically accommodate only heavily quantized micro-models, whereas smartphones can sustain on-device inference for small-to-medium models (e.g., 1 to 3 billion parameters). Larger models, however, still require hierarchical offloading to paired devices or edge/cloud resources. Just as the web browser became the dominant interface for smart devices in the past, we argue that LLM-driven interaction is poised to become the primary interface for future personalized IoT devices, making this study timely and of broad significance.

The main contributions of this work are summarized as follows:

- **Characterization of on-device LLM I/O behavior.** We analyze file-access traces of mobile on-device LLM workloads, identifying distinctive I/O patterns such as recurring loop and hot-block accesses, in contrast to remote inference LLMs.
- **Design of an LLM-aware buffer caching strategy.** We propose a lightweight caching mechanism that accounts for loop and hot-set accesses in LLM workloads, sustaining stable inference performance under limited memory and bandwidth conditions.
- **Cache-size modeling and hardware-specific validation.** We formulate a cache-sizing model that quantifies feasible cache capacities and validate the proposed strategy through empirical evaluation on smartphone and smartwatch platforms, providing design guidelines for LLM-aware buffer caching in practical device classes.

The remainder of this paper is organized as follows. Section 2 summarizes studies related to this paper. Section 3 analyzes the file-access characteristics of LLM workloads

precisely. Section 4 presents the buffer cache model for LLM workloads and its implications. Finally, Section 5 concludes the paper.

2. Related Work

Caching is a key technique for reducing user-perceived latency and improving throughput in LLM inference. Prior studies can be grouped into three technical directions: (1) inference-internal KV caching, which reuses intermediate key–value (KV) states inside the Transformer; (2) external or semantic caching, which reuses model inputs and outputs based on query similarity; and (3) system-level file-block caching, which underlies our work and operates at the operating-system buffer-cache layer.

2.1. Internal KV Caching

Research within the inference path focuses on managing KV states to eliminate redundant attention computation during decoding, primarily in large-scale server-side deployments. This line of work has established KV caching as the fundamental mechanism for improving inference efficiency, with many follow-up studies refining its reuse granularity and scalability.

2.1.1. Prefix-Aware KV Caching

In Transformer self-attention, computations with all prior tokens are repeated each time a new token is generated. KV caching stores previously computed keys and values, thereby avoiding redundant operations over all previous tokens during decoding and significantly reducing latency.

vLLM introduced PagedAttention to minimize GPU-memory fragmentation and enable efficient sharing of KV blocks across concurrent requests [17]. However, such reuse is highly sensitive to token order, so even small input variations may prevent cache hits. To remove prefix dependence, EPIC allows context blocks to be reused regardless of input position, greatly improving time-to-first-token (TTFT) [18].

2.1.2. Prompt- and RAG-Level Caching

Several extensions divide or merge prompts to enable reuse across similar contexts. Prompt Cache divides prompts into reusable modules and precomputes their attention states [19]. CacheBlend partitions prompts into chunks and fuses cached states to accelerate retrieval-augmented generation (RAG) workloads [20]. RAGCache introduces a hierarchical cache tailored for RAG scenarios to reduce the computational cost of incorporating external knowledge [21].

2.1.3. Workload-Aware Cache Eviction

Subsequent work analyzed cache-reuse distributions and optimized eviction policies for large-scale serving based on real-world usage data. A provider trace study quantified reuse behavior in KV cache and proposed workload-aware eviction strategies [16]. Attention-Gate evicts entries dynamically based on attention importance [22], while KeyDiff relies on key similarity for eviction under long-context settings [23]. MemServe manages a disaggregated memory pool to coordinate KV caches across distributed servers [24].

2.1.4. Memory Reduction and Compression

A complementary research line reduces the memory footprint of KV caches. SentenceKV groups KV states at the sentence level and loads only the relevant groups into GPU memory during decoding [25].

Research on compression has also been active. KIVI applies 2-bit quantization to KV caches, saving substantial memory usage without accuracy loss [15]. ShadowKV

combines low-rank key compression with CPU offloading of values [26]. KVzip identifies and discards less important entries by reconstructing context from cached states, reducing memory usage without degrading model performance [27]. KeepKV proposes a merging-based compression method that eliminates the attention distribution distortion typically introduced when redundant cache entries are merged [28].

2.1.5. Joint Optimization of Caching and Scheduling

Other studies jointly consider caching with scheduling or sensitivity analysis. Feng et al. identify critical cache entries via output-sensitivity analysis, retaining only those that materially affect model predictions [29]. Jaillet et al. integrate cache management with request scheduling as an online optimization problem under memory constraints [30].

2.2. External Semantic Caching

While the aforementioned techniques operate inside the inference pipeline and perform internal caching to reduce model computations, another major direction explores external or semantic caching around model invocation. Such techniques focus on reusing inputs and outputs around LLM calls. In particular, semantic caching interprets the meaning of a user's prompt and reuses cached results when new queries convey the same intent, even if phrased differently. Unlike KV caching, this approach is not sensitive to token order and is especially beneficial when handling many similar queries.

GPTCache stores query–response pairs in an embedding index and returns cached responses for semantically similar inputs [31]. MeanCache deploys personalized semantic caching at the client side to reduce network and server overhead [32]. SCALM analyzes large-scale chat logs, identifying limitations of existing caching and proposing clustering-based solutions for LLM chat services [33].

For multi-turn dialog, ContextCache employs two-stage retrieval pipelines: vector-based matching for candidate responses, followed by self-attention integration of dialog history to improve contextual matching [34]. Domain-specific studies fine-tune lightweight embeddings using synthetic data to better capture in-domain similarity [35]. Liu et al. propose a learning-based eviction framework that formulates semantic cache eviction as both offline optimization and online adaptation for LLM serving [36]. MinCache combines string-based and semantic caching hierarchically to balance coverage and efficiency [37], while Generative Caching synthesizes cached responses to serve new but related queries [38].

These approaches depend on embedding-similarity computation and are mainly suited to cloud or service-side environments. They tolerate paraphrasing and evolving context, but cannot ensure exact I/O-level reuse.

2.3. System-Level File-Block Caching

Now, let us discuss the limitations of prior work and the positioning of our approach. Both inference-internal KV caching and external semantic caching have clear merits but also inherent limitations. Inference-internal KV caching substantially reduces computation, but its effectiveness is constrained by token-order sensitivity and limited cache capacity, often requiring specialized eviction or compression mechanisms. Semantic caching improves reuse under paraphrase but incurs embedding-similarity computation and cannot guarantee exact I/O-level reuse as knowledge and context evolve. In addition, these two directions have been extensively explored in server environments, whereas the interaction between LLM workloads and the operating-system buffer cache has received little attention.

Distinct from prior approaches, our study focuses on system-level file-block caching, which targets I/O-level reuse outside the model runtime. At this layer, the operating-system buffer cache manages model weight files as regular I/O blocks, allowing exact-match reuse without semantic comparison or token-level dependency. Such a mechanism is crucial for

mobile and on-device LLM inference, where storage bandwidth and memory capacity are limited. To the best of our knowledge, little prior work has examined file-access behavior in mobile on-device LLM inference or analyzed how OS-level caching affects such behavior, leaving this layer largely unexplored until our study.

3. File-Access Characteristics of LLM Workloads

In this section, we analyze the file-block access characteristics of mobile LLM workloads. Specifically, we contrast remote inference, where the device acts as a thin client, with on-device inference, which reflects recent trends in mobile AI. Experiments were conducted on a Samsung Galaxy S22 (Samsung, Suwon, Republic of Korea) (octa-core 1.8–3 GHz CPU, Adreno 730 GPU (Qualcomm, San Diego, CA, USA), 8 GB RAM, and 256 GB UFS storage) [39]. Note that we also collected traces from other devices with different CPU/GPU architectures, memory capacities, and storage configurations, and observed similar results, indicating that our workload analysis reflects the intrinsic characteristics of the workloads rather than hardware-specific effects. For remote inference, we analyzed Copilot, Perplexity, Claude, and Gemini. For on-device inference, we used two Samsung LLM apps, AI Notes and AI Translate [40], and two Google Mediapipe apps, Text Embedder and Text Classification [41].

The experiments were conducted on Android 14 (One UI 6.1), and trace capturing was performed using the Linux strace utility together with a custom parser that recorded all file read and write system calls invoked by the target applications at 4 KB block granularity. In this setting, interactions issued from user space to kernel space were exhaustively traced to capture how each LLM application accessed its weight and configuration files through the operating system. For each workload, traces were collected while executing the application's typical usage scenarios, in which text, image, and voice inputs were provided and corresponding outputs (text, voice, or files) were obtained depending on each app's capability.

3.1. On-Device Inference

Figure 1 shows file-block references while executing LLM workloads that perform on-device inference. Figure 1a shows the results for AI Notes as time progresses. As shown in the figure, the workload reveals a consistent pattern: first, an initial one-time sequential read occurs, and then a small set of hot blocks are persistently referenced; additionally, long periodic read and write loops occur consistently. More specifically, at startup, a sequential read spanning block numbers 0–7500 occurs, corresponding to the accesses of configuration and setup files. These accesses are essentially one-shot and unlikely to be reused; caching such file blocks would provide no benefit and may even pollute the cache space. In contrast, near block number 7500, a region repeatedly referenced throughout the entire workload is observed. This region corresponds to core data structures of the internal engine—including tokenizer tables, LLM metadata, and index files—and thus represents essential caching targets. Subsequently, between block numbers 10,000–15,000, a periodic write loop appears, which we attribute to updates of the handwriting recognition engine. Finally, the range 15,000–25,000 shows a periodic read loop corresponding to model weight files used in neural machine translation.

Figure 1b shows the block access behavior for AI Translate workloads. Similar to AI Notes, configuration/setup files are sequentially read, while persistent references to a small hot set are also observed. The subsequent trace reveals two distinct periodic read loops: the low-numbered loop corresponds to dictation model files for source language recognition, while the higher-numbered loop corresponds to target language dictation model files. In

addition, a sequential write pattern was observed between block numbers 600 and 700, which was identified as temporary database copies generated during translation.

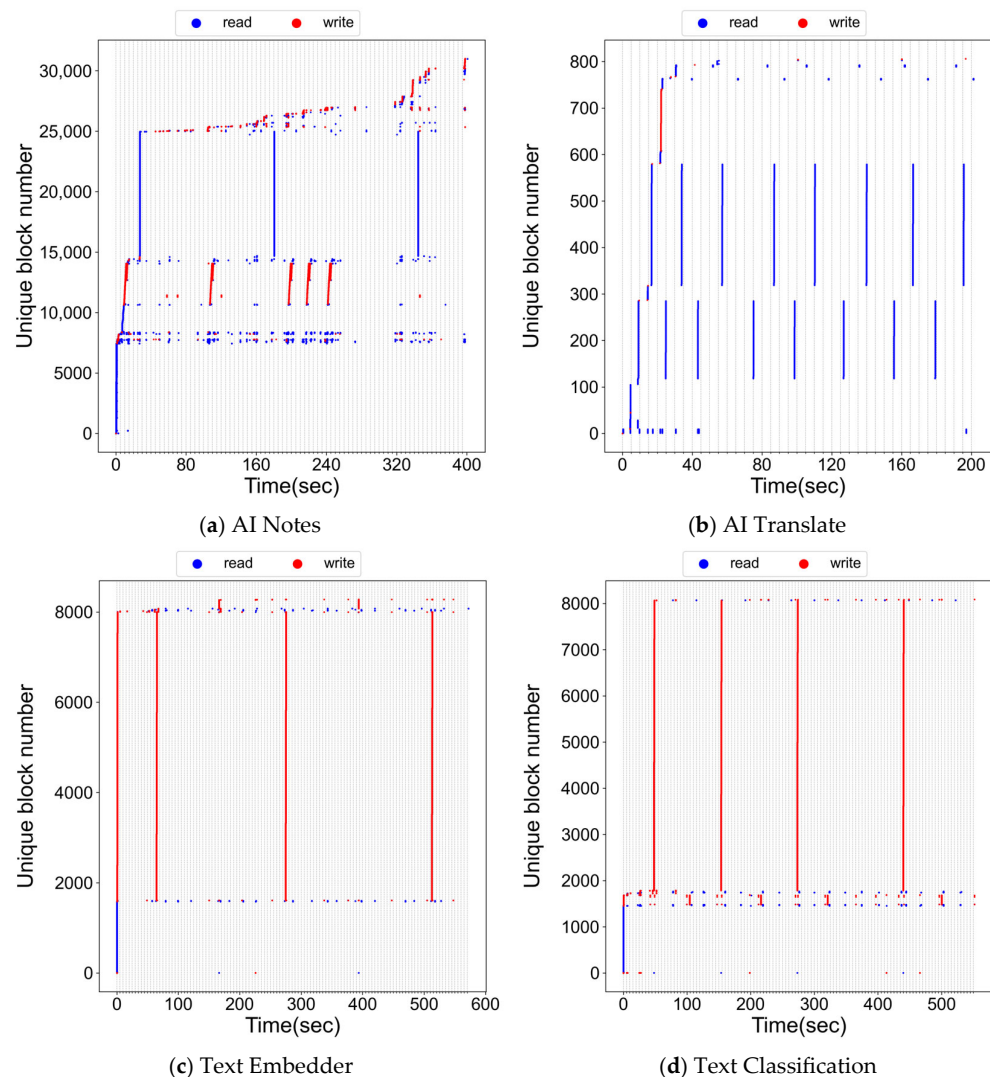


Figure 1. File block accesses of on-device inference LLM workloads.

Figure 1c presents the results for the Text Embedder workload. Similar to the previous workloads, a sequential read phase occurs at startup, corresponding to configuration and setup files, followed by persistent references to a small hot set of frequently accessed blocks. Between block numbers 1500 and 8000, a periodic write loop is observed, corresponding to temporary copies of the deep-learning model files used for inference. These writes occur because the model files are packaged inside the application's compressed, read-only installation bundle (.apk) and thus cannot be directly accessed by the native inference library. To enable direct access, the required model files are copied from the .apk package to a writable cache directory, where they become regular file-system objects accessible to the inference engine. The file-preparation process is intermittently triggered during execution as the application updates its runtime conditions, resulting in the periodic write-loop pattern observed in the trace.

Figure 1d shows the trace for the Text Classification workload. The workload also begins with a one-time sequential read corresponding to initialization files, after which write operations dominate. Similar to Text Embedder, a periodic write loop appears between block numbers 1800 and 8000, caused by the same model-file copying process. Although the two applications differ slightly in task structure, both exhibit the same system-

level looping behavior, originating from how model files are prepared and managed within Android’s application environment.

Despite differences in detail, all four apps share three common traits: (1) sequential access to configuration/setup files during initialization, (2) looped accesses to model weight files during inference, and (3) persistently accessed small hot sets that remain relevant over time.

3.2. Remote Inference

We now turn to applications where inference is executed remotely and the mobile device functions as a thin client. Figure 2a shows the file block references for Copilot. In contrast to the on-device case, there are no looped accesses to model weight files; instead, sequential references dominate. Persistent references to a small set of blocks near block 0 and at higher ranges are visible, but the majority of activity consists of bursts of sequential write–read–write patterns. These correspond to uploading a file and prompt, sending them to the network, and subsequently receiving and storing the results for display.

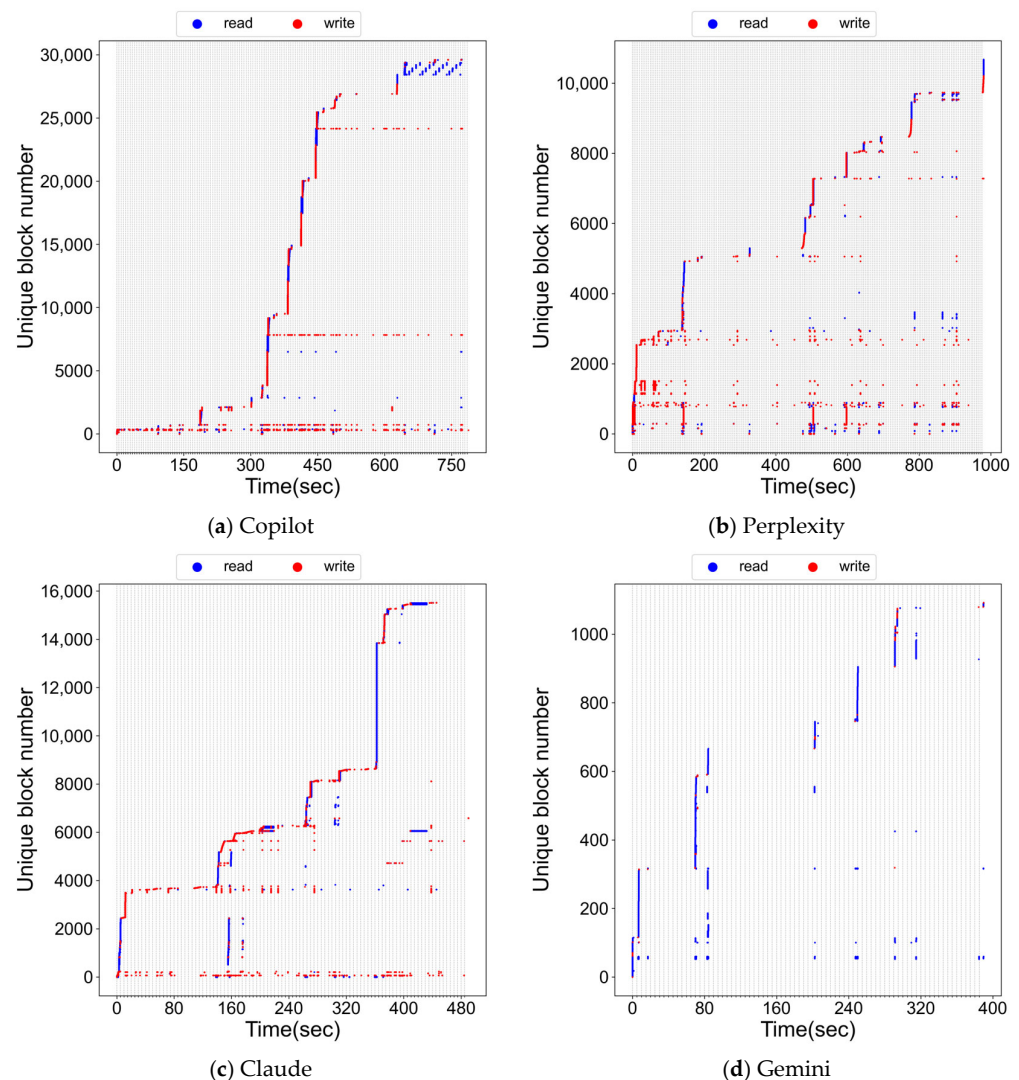


Figure 2. File block accesses of remote inference LLM workloads.

Figure 2b shows the trace for Perplexity. The overall pattern resembles Copilot, with sequential accesses dominating. Some low-numbered blocks appear to form loops, but closer inspection indicates that these are probabilistic re-references rather than structured loops. The low-numbered region corresponds to repeated small hot set accesses (e.g.,

metadata or indexing files), while higher block regions are associated with handling each user prompt—saving locally, reading for transmission to the remote inference server, and storing results when required. A subset of hot blocks at low numbers is also persistently referenced, interpreted as small internal data repeatedly needed during prompt handling.

Figure 2c presents the results for Claude. As in the previous apps, accesses corresponding to each prompt appear largely sequential. A persistent hot set, including metadata and indexing files, is observed near blocks 0 and 4000. Interestingly, references observed around 0 (s) reappeared near 160 (s), creating what looked like a loop; however, further analysis revealed these resulted from a failed prompt request being retried, leading to two consecutive accesses to the same file. They therefore do not represent structured loop patterns.

Figure 2d shows the trace for Gemini. Similar to the previous results, each prompt request exhibits sequential accesses. Small hot blocks, persistently accessed throughout the execution, are also present, though their access frequency is relatively lower than in other applications.

3.3. Summary

In summary, remote inference workloads are dominated by sequential file accesses and lack the repeated loop structures that characterize on-device inference. By contrast, on-device LLM workloads consistently exhibit three defining features: initialization-related sequential scans, persistent small hot sets, and looped accesses to large weight files during inference. These observations highlight the unique caching demands posed by on-device LLMs.

4. Buffer Cache Model for LLM and Implications

Personalized LLM devices—including not only smartphones but also smartwatches, wearables, and elder-care robots—are poised to become key platforms in the AI era. These devices, however, differ widely in memory capacity, power budget, and storage bandwidth, which constrains the feasible size of on-device models and often necessitates fallback to remote inference. Recent smartwatch SoCs have begun to adopt CPU-GPU-NPU architectures. For instance, Apple's S9 integrates a four-core Neural Engine to support on-device Siri [42]. Similarly, Samsung's Exynos W1000 employs a 3 nm Mali-G68 with a big.LITTLE CPU cluster [43] and Qualcomm's Snapdragon W5+ incorporates a dedicated ML core and low-power DSP [44].

Despite these advances, storage performance remains a critical bottleneck [45]. Most watches still rely on eMMC storage, which is considerably slower than the UFS storage commonly used in smartphones. Consequently, repeated reads of large weight files can easily saturate the I/O bandwidth. In practice, smartwatches are better suited to lightweight, specialized models, while larger models are more realistically deployed in a hierarchical fashion across paired smartphones or higher-tier personal AI devices. A pragmatic strategy is to embed aggressively quantized micro-LLMs for short-context, on-device processing, while adopting a resource cut-off rule to delegate larger contexts to remote servers. In contrast, smartphone-class devices with LPDDR memory, UFS storage, and capable NPUs are expected to support on-device inference for medium-scale models, although careful design must account for buffer cache size, hot set coverage, storage bandwidth, battery drain, and thermal constraints.

4.1. Cache Size Model

Figure 3 illustrates the overall structure of the buffer cache for model weight files, contrasted with the KV cache maintained in GPU memory. Unlike the KV cache, which

is highly sensitive to prompt length and session duration, the host buffer cache is mainly determined by the selected model and remains relatively stable against input variations. As shown in Section 3, file accesses to LLM workloads can be decomposed into three main components: (1) one-time sequential scans during initialization, (2) persistent accesses to a small hot set (e.g., tokenizer tables, metadata, and index files), and (3) looped accesses to model weight files. From a caching perspective, one-time sequential scans provide little benefit and should be evicted quickly to avoid polluting the cache. By contrast, hot sets must always be fully retained, and loop references to weight files must be sustained to prevent I/O stalls.

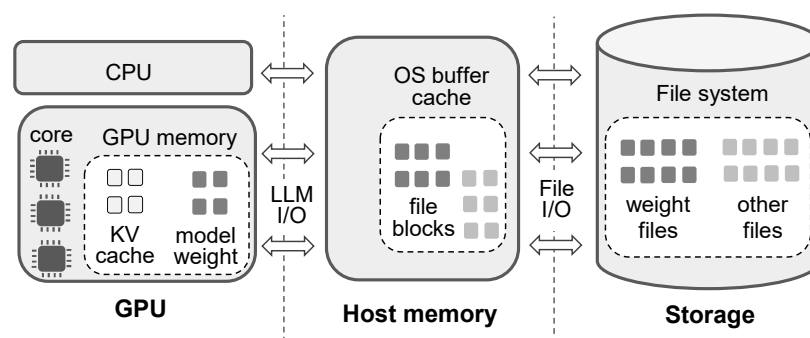


Figure 3. Overall structure of the buffer cache for model weight files.

Ideally, all looped weight-file blocks would fit into the cache. If not, the system must rely on scan-resistant policies. Standard LRU is problematic because loop blocks may be evicted before being reused, leading to repeated I/O [46]. Instead, loop blocks should be pinned in cache across iterations so that each cycle yields deterministic cache hits. If the loop consists of N blocks and the cache can accommodate only S of them, then each cycle yields S cache hits and $(N-S)$ blocks must be re-fetched from storage. To ensure that storage I/O does not become a bottleneck, the available bandwidth must be sufficient to serve the uncached portion within a loop cycle. Formally, the minimum cache size for loops must satisfy

$$S \geq \max\{0, \lfloor N - BW \times T \rfloor\} \quad (1)$$

where S denotes the cache portion available for looped weight blocks after fully retaining hot sets, N is the number of blocks in the loop, BW is the sustained storage bandwidth (in blocks per second), and T is the allowable fetch delay threshold within which any non-resident block must be fetched and staged so that inference proceeds as if served from cache (no stall). Each block corresponds to 4 KB, consistent with the page size used throughout our experiments; thus, BW in Formula (1) can be expressed in MB/s by multiplying by 4×10^{-3} (MB per block). If the equation is not met, the system will incur inference delays proportional to the deficit in cache coverage.

We derive the practical setting of T by profiling the token-generation latency of each on-device model. In our mobile setting, the per-token compute time ranges from tens to hundreds of milliseconds depending on model size and quantization level. Specifically, we measure the average token-generation speed on the Samsung Galaxy S22 under AI Notes and AI Translate workloads using the same experimental setup described in Section 3 and observe a throughput of approximately 8–12 tokens per second (corresponding to 80–125 ms per token). Based on these results, the default value of T is set to 0.1 s as a representative value within this range. We also examine the sensitivity of the model to T and observe that moderate variations (± 20 – 30%) cause negligible changes in predicted performance, since both computation and I/O maintain stable throughput under our tested workloads. In real-time interactive systems with strict QoS targets, T could be set to a lower

value (e.g., below 0.1 s), but since on-device LLM inference operates as a best-effort service, we believe that our chosen T remains practically valid.

Unlike server-class multi-tenant settings, mobile devices rarely run multiple latency-critical apps concurrently in the foreground. The OS prioritizes the foreground app and reclaims other apps' memory when needed, so the buffer cache is effectively dominated by the foreground app. Our LLM-aware policy therefore focuses on in-app arbitration and does not consider resource sharing among multi-tenants.

Designing cache replacement algorithms for LLM workloads is not inherently difficult, given the relative simplicity of their access patterns. In this paper, we adopt a lightweight yet effective strategy: sequential-only accesses are given the lowest priority and cached only opportunistically when space is available, whereas once such accesses evolve into loop patterns, their constituent blocks are pinned starting from the lowest block numbers up to the cache capacity. If multiple loops overlap or interleave, the policy prioritizes shorter-period loops, fully pinning their blocks if space permits, and then allocates any remaining cache space progressively to longer-period loops. For random accesses, we reserve a dedicated portion of the cache to guarantee persistence of hot blocks, and the space is managed using an LRU policy. The reserved cache space for hot blocks is configured according to the typical case analyzed in Section 3, but it can also expand or shrink adaptively as workload characteristics vary. Given the relative simplicity of LLM file accesses, we can even pre-classify the application's files into three categories—sequential, loop, and hot—so that cache management can be directly applied without online pattern detection. Algorithm 1 depicts the pseudocode of the proposed policy. Note that this version relies on online detection of access patterns rather than assuming pre-classification.

In practice, occasional noise or short interruptions may temporarily prevent a loop from being detected, but such cases are rare and automatically corrected in the next iteration. Because LLM weight-file accesses exhibit highly regular patterns, the proposed method consistently captured all recurring loops observed in our traces. Specifically, all loops present in the on-device LLM workloads were successfully detected by the proposed scheme.

Now, let us discuss the computational and memory overhead of the proposed caching policy. Each block access requires an adjacency check, set-membership lookup, and simple category updates, yielding an overall time complexity of $O(1)$ per access. Eviction is also performed in $O(1)$ time, since the oldest accessed block can be directly found from the LRU (least recently used) list, which moves a block to the newest position upon each access and selects the victim from the opposite end without any element comparison. Likewise, block numbers within a block sequence are appended sequentially, so the highest-numbered block can be identified without comparison. The memory usage for each cache block is also $O(1)$, as only minimal metadata are maintained per block sequence—specifically, the start and end block numbers and their category (sequential, loop, or hot)—requiring only a few bytes. Therefore, the proposed policy can be implemented efficiently in terms of both time and space overhead.

Figure 4 compares the cache hit ratio of the proposed LLM-aware policy with that of a generic LRU as cache capacity increases. The results demonstrate that LLM-aware policy consistently achieves higher hit ratios by discarding one-time sequential scans and ensuring persistent loop hits. Specifically, the proposed policy shows a gradual performance improvement as the cache size increases, whereas LRU exhibits regions where no performance gain is observed even with enlarged cache sizes. In particular, for Text Embedder and Text Classification, LRU remains at a near-baseline hit ratio even at the largest cache sizes tested. This occurs because file accesses in these workloads are dominated by a single long loop. Consequently, even when the cache capacity is large

enough for the other two policies to achieve high hit ratios, LRU cannot accommodate the entire loop and therefore gains no loop-level hits.

Algorithm 1. Pseudocode of the LLM-aware cache management policy

Block sequence BS is consecutive blocks $\{B, B + 1, \dots\}$;

Each BS belongs to one of SEQ, LOOP, or HOT;

BS(B) returns BS that contains block B (if any);

```

procedure Access(B)
  if previous access is  $B - 1$  then
    add B to BS( $B - 1$ );
    if BS(B)  $\in$  HOT then
      remove BS(B) from HOT;
      insert BS(B) into SEQ;
    end if
  else
    if BS(B)  $\in$  SEQ or LOOP then
       $BS' \leftarrow \{B\}$ ;
      insert  $BS'$  into LOOP;
    else if BS(B) is undefined then // B accessed first
       $BS' \leftarrow \{B\}$ ;
      insert  $BS'$  into HOT;
    end if
  end if
end procedure

```

```

procedure Evict( )
  if SEQ is not empty then
    evict highest-numbered block from oldest BS in SEQ;
  else if HOT is full then
    evict the least-recently-used block from HOT;
  else // LOOP is full
    evict highest-numbered block from longest-period BS;
  end if
end procedure

```

We also compare our policy with LeCaR (Learning Cache Replacement), a well-known machine-learning-based cache replacement policy [47]. Unlike deterministic algorithms, LeCaR makes use of expert-based learning where LRU and LFU serve as the two base algorithms, and determines their weights through online training using regret-minimization strategies, which reward the policy incurring fewer misses while penalizing the other. As shown in Figure 4, although there exist certain ranges where LeCaR performs slightly better in the AI Notes workload, the proposed policy significantly outperforms LeCaR in the AI Translate workload, confirming its competitiveness. Similar trends are also observed in the Text Embedder and Text Classification workloads. For Text Classification, LeCaR occasionally achieves slightly higher hit ratios in certain cache-size intervals due to its nondeterministic behavior. However, for both Text Embedder and Text Classification workloads, the proposed policy consistently delivers higher overall performance, further confirming its robustness across diverse application behaviors. Overall, while LeCaR shows noticeable performance fluctuations even with small cache-size variations, the proposed policy maintains stable and consistent improvement as cache capacity grows. Moreover, LeCaR incurs substantial overhead due to its online learning process, whereas the proposed policy achieves comparable performance with only lightweight adjacency

checks. Given the relatively straightforward access patterns of LLM workloads, designing a cache replacement algorithm is not particularly challenging. The more critical contribution of this work lies in developing a cache sizing model and analyzing the feasible weight file sizes that can be supported under practical cache capacities.

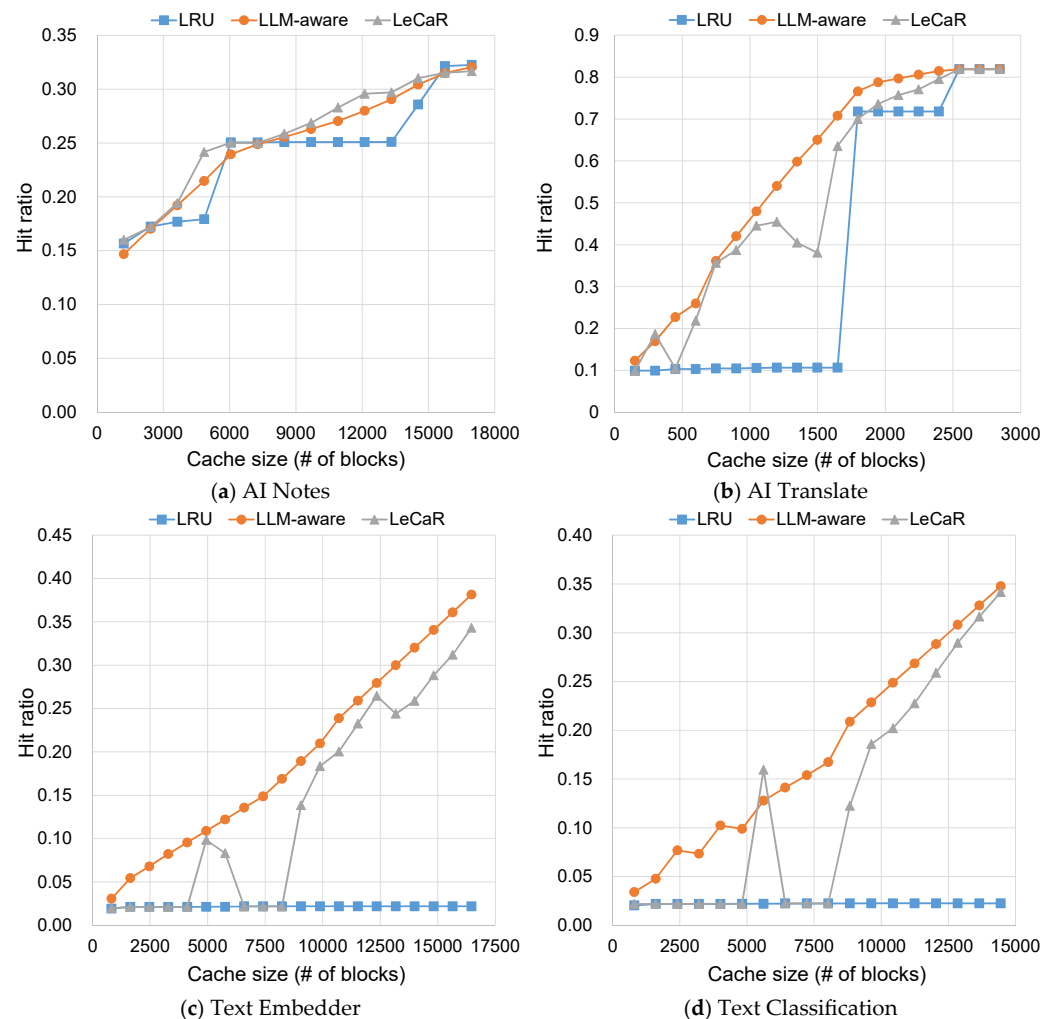


Figure 4. Hit ratio of LLM-aware caching in comparison with LRU and LeCaR as cache size varies.

We further evaluate the impact of storage bandwidth on end-to-end inference latency. In this experiment, the memory bandwidth for host-to-GPU transfers is set to 20 GB/s with a base latency of 2 μ s. Storage accesses assume a per-I/O setup latency of 50 μ s and sustained sequential read bandwidths of 100, 200, 500, and 1000 MB/s. Unless otherwise noted, request granularity is 1 KB while cache accounting and storage transfers follow the OS page size (4 KB). Figure 5 presents the total data-fetch latency as a function of storage bandwidth and cache size. As shown in the figure, the latency decreases as the cache size and storage bandwidth increase. However, the patterns differ across the four workloads. For AI Notes, Text Embedder, and Text Classification, the latency is strongly influenced by storage bandwidth, showing clear improvement until the bandwidth reaches about 500 MB/s, beyond which additional gains are negligible. Increasing the cache size also yields gradual improvement, but its impact is less pronounced than that of bandwidth. In contrast, AI Translate exhibits high sensitivity of latency to cache size. In particular, before the cache becomes sufficiently large, latency decreases sharply with cache enlargement, especially under lower bandwidth conditions. These results highlight that both cache

capacity and storage bandwidth must be jointly considered to guarantee QoS for mobile LLM inference, depending on model sizes and access characteristics of LLM workloads.

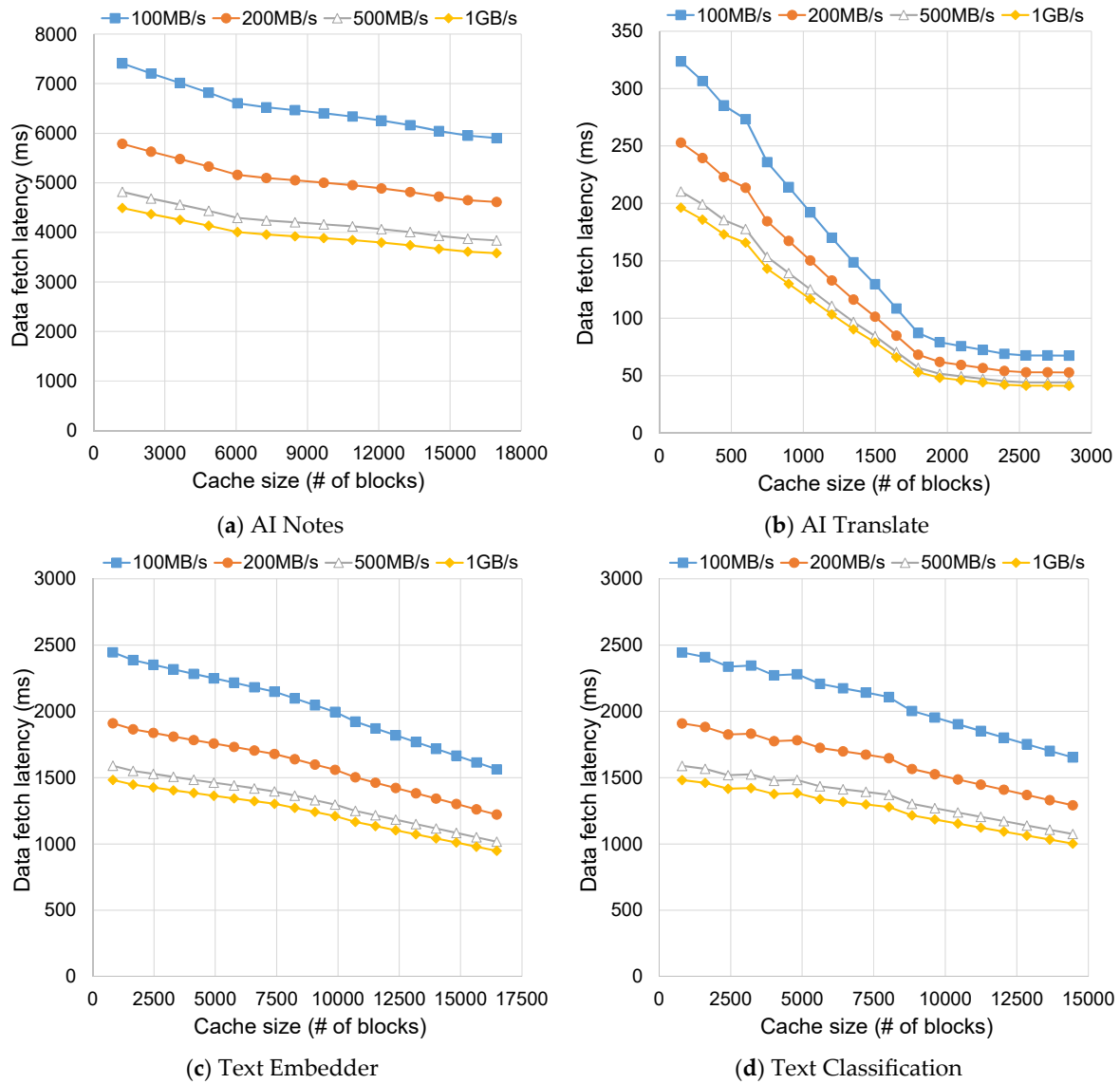


Figure 5. Data fetch latency of LLM-aware caching as cache size and storage bandwidth vary.

4.2. Discussion on Hardware Specifications

Smartwatches typically provide only 1–2 GB of RAM. After accounting for the operating system, platform services, and background applications (e.g., health monitoring, music, notifications, and sensor services), the buffer cache realistically available for LLM workloads is limited to about 100–200 MB. A conservative estimate for the portion exclusively dedicated to LLM inference is roughly 64 MB. Within this budget, tokenizer and index files (less than 4 MB as observed in Section 3) must be fully retained; otherwise, severe I/O slowdowns occur due to frequent reloading of hot data.

For looped weight-file accesses, only a subset of the loop can be cached in practice, but partial retention is still crucial to mitigate repeated reads from slow eMMC storage. While the nominal sequential read throughput of eMMC 5.x is 200–300 MB/s, its effective runtime bandwidth typically falls near 100 MB/s under thermal and power constraints. Assuming an allowable fetch delay of $T = 0.1$ s in Equation (1), local inference can sustain a loop size of $N = S + BW \times T \approx (64 - 4) + (100 \times 0.1) = 70$ MB, without causing I/O stalls. Here, the loop size denotes the total amount of model weight data accessed once per inference

iteration. Empirically, AI Notes (≈ 60 MB loop) and AI Translate (≈ 2 MB loop) fit within this bound, whereas state-of-the-art LLMs require over 500 MB of weight data, making direct on-device inference infeasible on watches. In such cases, paired smartphones or edge servers should be leveraged, with LTE-enabled watches falling back to cloud inference when disconnected.

In contrast, smartphone-class devices are equipped with 6–8 GB of host memory, of which 512 MB to 1 GB can typically be used for buffer caching. Among this, 256–512 MB can be reserved for LLM loop data, and UFS 4.0 storage provides sustained sequential read bandwidths of ≥ 1 GB/s. With $T = 0.1$ s and loop-dedicated cache size S ranging from 256 MB to 512 MB, Equation (1) yields sustainable loop sizes of ≈ 356 –612 MB without I/O stalls. This capacity corresponds to 1–3 billion-parameter models, depending on quantization and activation compression, and is adequate for small-to-mid-scale on-device LLMs. However, larger models still suffer from thrashing due to long reuse distances and extensive sequential scans, making remote or edge-assisted inference a more practical option.

In summary, the buffer cache in resource-constrained devices must (1) guarantee full coverage of hot sets, (2) retain looped weight-file blocks within storage-bandwidth limits, and (3) avoid cache pollution from one-time scans. At present, watches can only support highly compact LLMs under these constraints, while smartphones can accommodate small-to-mid-scale models up to 3 billion parameters. For larger LLMs, dedicated personal AI devices or edge/cloud offloading remain necessary to sustain real-time inference performance.

5. Conclusions

This paper analyzed the file-level I/O characteristics of LLM inference on mobile platforms, with a focus on distinguishing on-device inference workloads from conventional cloud-based ones. Through detailed trace analysis, we identified three defining patterns: one-time sequential scans during initialization, persistent small hot sets related to tokenizers and metadata, and recurrent loop references to model weight files. These insights highlight that, unlike cloud-based inference, on-device LLM workloads impose unique stresses on the operating system buffer cache, leading to potential I/O bottlenecks. Building on these findings, we proposed a buffer cache model tailored for LLM workloads in resource-constrained consumer devices. Our approach stresses the importance of allocating sufficient cache space to accommodate hot sets and loop segments, and it prevents cache pollution from one-time sequential accesses. We further analyzed the implications of hardware specifications, such as memory capacity and storage bandwidth, showing that smartwatch-class devices are limited to very small models or must rely on hierarchical offloading, whereas smartphone-class devices can sustain a broader range of on-device inference.

Our analysis, which focused on Transformer-based LLMs, indicates that the recurring loop pattern arises from the repeated reuse of weight files. Other architectures, such as State Space Models (SSMs), may exhibit less strict periodic access because hidden-state propagation replaces part of the attention mechanism. Even so, these models are also expected to involve segment- or layer-level weight reuse, implying that the proposed caching mechanism remains applicable when such recurring accesses are treated as generalized loop segments. As future work, we plan to extend our analysis and caching framework to these architectures.

Another open question is whether the loopiness we observed remains consistent under different quantization strategies. Although our analysis revealed stable recurring loop access patterns across various applications, model sizes, and quantization states, we have not yet quantified how specific quantization methods (e.g., INT8 vs. INT4 or sparsity-aware quantization) may affect loop size or structure. Models employing extreme sparsity or

novel quantization schemes could reduce or reshape loop blocks, potentially requiring parameter-level adaptation. We will investigate this aspect in future work.

Author Contributions: Conceptualization, H.K., J.L. and H.B.; Methodology, H.K. and J.L.; Software, H.K. and J.L.; Validation, J.L.; Formal analysis, H.K., J.L. and H.B.; Investigation, H.K. and J.L.; Resources, H.B.; Data curation, H.K. and J.L.; Writing—original draft, H.K., J.L. and H.B.; Writing—review & editing, H.K., J.L. and H.B.; Visualization, H.K. and H.B.; Supervision, H.B.; Project administration, H.B.; Funding acquisition, H.B. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported in part by the National Research Foundation of Korea (NRF) under Grant RS-2025-02214322 and the Institute of Information & Communications Technology Planning & Evaluation (IITP) under Grant RS-2022-00155966 (Artificial Intelligence Convergence Innovation Human Resources Development (Ewha University)) funded by the Korean government (MSIT).

Data Availability Statement: The original contributions presented in this study are included in the article. Further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

References

- Chen, H.; Zhang, J.; Du, Y.; Xiang, S.; Yue, Z.; Zhang, N.; Cai, Y.; Zhang, Z. Understanding the Potential of FPGA-based Spatial Acceleration for Large Language Model Inference. *ACM Trans. Reconfigurable Technol. Syst.* **2024**, *18*, 1–29. [CrossRef]
- Miao, X.; Oliaro, G.; Zhang, Z.; Cheng, X.; Jin, H.; Chen, T.; Jia, Z. Towards Efficient Generative Large Language Model Serving: A Survey from Algorithms to Systems. *ACM Comput. Surv.* **2025**, *58*, 1–37. [CrossRef]
- Chen, L.; Feng, D.; Feng, E.; Wang, Y.; Zhao, R.; Xia, Y.; Xu, P.; Chen, H. Characterizing Mobile SoC for Accelerating Heterogeneous LLM Inference. In Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, Seoul, Republic of Korea, 13–16 October 2025; pp. 359–374.
- Kwon, S.; Bahn, H. Memory Reference Analysis and Implications for Executing AI Workloads in Mobile Systems. In Proceedings of the IEEE International Conference on Electrical and Information Technology, Malang, Indonesia, 14–15 September 2023; pp. 281–285.
- Apple A17 Pro 3nm iPhone 15 Pro. Available online: <https://www.tomshardware.com/news/apple-a17-pro-3nm-iphone-15-pro> (accessed on 12 October 2025).
- Qualcomm. Snapdragon 8 Gen 3 Mobile Platform Product Brief. Available online: https://docs.qualcomm.com/bundle/publicresource/87-71408-1_REV_C_Snapdragon_8_gen_3_Mobile_Platform_Product_Brief.pdf (accessed on 12 October 2025).
- Samsung Revamped NPU Architecture in Exynos 2400 for AI Boost. Available online: <https://www.androidheadlines.com/2023/10/samsung-revamped-npu-architecture-in-exynos-2400-for-ai-boost.html> (accessed on 12 October 2025).
- Apple Machine Learning Research. Core ML On-Device Llama. Available online: <https://machinelearning.apple.com/research/core-ml-on-device-llama> (accessed on 12 October 2025).
- PyTorch. ExecuteTorch Llama Demo for Android. Available online: <https://docs.pytorch.org/executorch/0.7/llm/llama-demo-android.html> (accessed on 12 October 2025).
- Das, B.; Amini, M.; Wu, Y. Security and Privacy Challenges of Large Language Models: A Survey. *ACM Comput. Surv.* **2025**, *57*, 1–39. [CrossRef]
- Wang, Z.; Yang, J.; Qian, X.; Xing, S.; Jiang, X.; Lv, C.; Zhang, S. MNN-LLM: A Generic Inference Engine for Fast Large Language Model Deployment on Mobile Devices. In Proceedings of the 6th ACM International Conference on Multimedia in Asia Workshops, Auckland, New Zealand, 3–6 December 2024; pp. 1–7.
- Kwon, G.; Bahn, H. Quantifying the Effectiveness of Cloud and Edge Offloading: An Optimization Study on Energy Efficiency of Mobile Real-Time Systems. In Proceedings of the IEEE International Conference on Artificial Intelligence in Information and Communication, Fukuoka, Japan, 18–21 February 2025; pp. 427–432.
- Lee, J.; Lim, S.; Bahn, H. Analyzing Data Access Characteristics of AIoT Workloads for Efficient Write Buffer Management. *IEEE Internet Things J.* **2025**, *12*, 31601–31614. [CrossRef]
- Park, S.; Bahn, H. WANE: Workload Adaptive Neuro-Genetic Engine for Container Usage Prediction. In Proceedings of the IEEE International Conference on Advances in Electrical Engineering and Computer Applications, Dalian, China, 16–18 August 2024; pp. 311–318.

15. Liu, Z.; Yuan, J.; Jin, H.; Zhong, S.; Xu, Z.; Braverman, V.; Chen, B.; Hu, X. KIVI: A Tuning-Free Asymmetric 2bit Quantization for KV Cache. *arXiv* **2024**, arXiv:2402.02750. [\[CrossRef\]](#)
16. Wang, J.; Han, J.; Wei, X.; Shen, S.; Zhang, D.; Fang, C.; Chen, R.; Yu, W.; Chen, H. KVCache Cache in the Wild: Characterizing and Optimizing KVCache Cache at a Large Cloud Provider. *arXiv* **2025**, arXiv:2506.02634. [\[CrossRef\]](#)
17. Kwon, W.; Li, Z.; Zhuang, S.; Sheng, Y.; Zheng, L.; Yu, C.; Gonzalez, J.; Zhang, H.; Stoica, I. Efficient Memory Management for Large Language Model Serving with PagedAttention. In Proceedings of the 29th ACM Symposium on Operating Systems Principles (SOSP), Koblenz, Germany, 23–26 October 2023; pp. 611–626.
18. Hu, J.; Huang, H.; Wang, W.; Wang, H.; Hu, T.; Zhang, Q.; Feng, H.; Chen, X.; Shan, Y.; Xie, T. EPIC: Efficient Position-Independent Caching for Serving Large Language Models. *arXiv* **2024**, arXiv:2410.15332.
19. Gim, I.; Chen, G.; Lee, S.; Sarda, N.; Khandelwal, A.; Zhong, L. Prompt Cache: Modular Attention Reuse for Low-Latency Inference. In Proceedings of the Machine Learning and Systems (MLSys), Santa Clara, CA, USA, 13–16 May 2024.
20. Yao, J.; Li, H.; Liu, Y.; Ray, S.; Cheng, Y.; Zhang, Q.; Du, K.; Lu, S.; Jiang, J. CacheBlend: Fast Large Language Model Serving for RAG with Cached Knowledge Fusion. In Proceedings of the 20th ACM European Conference on Computer Systems (EuroSys), Rotterdam, The Netherlands, 30 March–3 April 2025; pp. 94–109.
21. Jin, C.; Zhang, Z.; Jiang, X.; Liu, F.; Liu, S.; Liu, X.; Jin, X. RAGCache: Efficient Knowledge Caching for Retrieval-Augmented Generation. *ACM Trans. Comput. Syst.* **2025**, *44*, 2. [\[CrossRef\]](#)
22. Zeng, Z.; Lin, B.; Hou, T.; Zhang, H.; Deng, Z. In-Context KV-Cache Eviction for LLMs via Attention-Gate. *arXiv* **2024**, arXiv:2410.12876.
23. Park, J.; Jones, D.; Morse, M.; Goel, R.; Lee, M.; Lott, C. KeyDiff: Key Similarity-Based KV Cache Eviction for Long-Context LLM Inference in Resource-Constrained Environments. *arXiv* **2025**, arXiv:2504.15364.
24. Hu, C.; Huang, H.; Hu, J.; Xu, J.; Chen, X.; Xie, T.; Wang, C.; Wang, S.; Bao, Y.; Sun, N.; et al. MemServe: Context Caching for Disaggregated LLM Serving with Elastic Memory Pool. *arXiv* **2024**, arXiv:2406.17565. [\[CrossRef\]](#)
25. Zhu, Y.; Falahati, A.; Yang, D.; Amiri, M. SentenceKV: Efficient LLM Inference via Sentence-Level Semantic KV Caching. *arXiv* **2025**, arXiv:2504.00970.
26. Sun, H.; Chang, L.; Bao, W.; Zheng, S.; Zheng, N. ShadowKV: KV Cache in Shadows for High-Throughput Long-Context LLM Inference. *arXiv* **2024**, arXiv:2410.21465.
27. Kim, J.; Kim, J.; Kwon, S.; Lee, J.; Yun, S.; Song, H. KVzip: Query-Agnostic KV Cache Compression with Context Reconstruction. *arXiv* **2025**, arXiv:2505.23416. [\[CrossRef\]](#)
28. Tian, Y.; Wang, Z.; Peng, Y.; Yuan, A.; Wang, Z.; Yi, B.; Liu, X.; Cui, Y.; Yang, T. KeepKV: Eliminating Output Perturbation in KV Cache Compression for Efficient LLMs Inference. *arXiv* **2025**, arXiv:2504.09936. [\[CrossRef\]](#)
29. Feng, Y.; Lv, J.; Cao, Y.; Xie, X.; Zhou, S. Identify Critical KV Cache in LLM Inference from an Output Perturbation Perspective. *arXiv* **2025**, arXiv:2502.03805. [\[CrossRef\]](#)
30. Jaillet, P.; Jiang, J.; Mellou, K.; Molinaro, M.; Podimata, C.; Zhou, Z. Online Scheduling for LLM Inference with KV Cache Constraints. *arXiv* **2025**, arXiv:2502.07115. [\[CrossRef\]](#)
31. Bang, F. GPTCache: An Open-Source Semantic Cache for LLM Applications Enabling Faster Answers and Cost Savings. In Proceedings of the 3rd Workshop for Natural Language Processing Open Source Software (NLP-OSS), Singapore, 6 December 2023; pp. 212–218.
32. Gill, W.; Elidrisi, M.; Kalapatapu, P.; Ahmed, A.; Anwar, A.; Gulzar, M.A. MeanCache: User-Centric Semantic Caching for LLM Web Services. In Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS), Milano, Italy, 3–7 June 2025; pp. 1298–1310.
33. Li, J.; Xu, C.; Wang, F.; Riedemann, I.; Zhang, C.; Liu, J. SCALM: Towards Semantic Caching for Automated Chat Services with Large Language Models. In Proceedings of the IEEE/ACM 32nd International Symposium on Quality of Service (IWQoS), Guangzhou, China, 19–21 June 2024; pp. 1–10.
34. Yan, J.; Ni, W.; Chen, L.; Lin, X.; Cheng, P.; Qin, Z.; Ren, K. ContextCache: Context-Aware Semantic Cache for Multi-Turn Queries in Large Language Models. *Proc. VLDB Endow.* **2025**, *18*, 5391–5394. [\[CrossRef\]](#)
35. Gill, W.; Cechmanek, J.; Hutcherson, T.; Rajamohan, S.; Agarwal, J.; Gulzar, M.; Singh, M.; Dion, B. Advancing Semantic Caching for LLMs with Domain-Specific Embeddings and Synthetic Data. *arXiv* **2025**, arXiv:2504.02268. [\[CrossRef\]](#)
36. Liu, X.; Atalar, B.; Dai, X.; Zuo, J.; Wang, S.; Lui, J.; Chen, W.; Joe-Wong, C. Semantic Caching for Low-Cost LLM Serving: From Offline Learning to Online Adaptation. *arXiv* **2025**, arXiv:2508.07675. [\[CrossRef\]](#)
37. Haqiq, K.; Jahan, M.; Farimani, S.; Masoom, S. MinCache: A Hybrid Cache System for Efficient Chatbots with Hierarchical Embedding Matching and LLM. *Future Gener. Comput. Syst.* **2025**, *170*, 107822. [\[CrossRef\]](#)
38. Iyengar, A.; Kundu, A.; Kompella, R.; Mamidi, S. A Generative Caching System for Large Language Models. *arXiv* **2025**, arXiv:2503.17603. [\[CrossRef\]](#)
39. Samsung Mobile Press. Galaxy S22. Available online: <https://www.samsungmobilepress.com/media-assets/galaxy-s22> (accessed on 12 October 2025).

40. Samsung Galaxy AI. Available online: <https://www.samsung.com/us/galaxy-ai> (accessed on 12 October 2025).
41. Google Mediapipe Samples. Available online: <https://github.com/google-ai-edge/mediapipe-samples> (accessed on 8 November 2025).
42. Apple S9 Technical Specifications. Available online: <https://support.apple.com/en-us/111833> (accessed on 12 October 2025).
43. Samsung Exynos W1000 Wearable Processor. Available online: <https://semiconductor.samsung.com/processor/wearable-processor/exynos-w1000/> (accessed on 12 October 2025).
44. Qualcomm Snapdragon W5+ Gen 1 Wearable Platform. Available online: <https://www.qualcomm.com/products/mobile/snapdragon/wearables/snapdragon-w5-plus-gen-1-wearable-platform> (accessed on 12 October 2025).
45. Lee, J.; Lim, S.; Bahn, H. Analyzing File Access Characteristics for Deep Learning Workloads on Mobile Devices. In Proceedings of the 5th IEEE International Conference on Advances in Electrical Engineering and Computer Applications, Dalian, China, 16–18 August 2024; pp. 417–422.
46. Kim, J.; Choi, J.; Kim, J.; Noh, S.H.; Min, S.; Cho, Y.; Kim, C. A Low-Overhead High-Performance Unified Buffer Management Scheme That Exploits Sequential and Looping References. In Proceedings of the 4th USENIX Conference on Operating System Design & Implementation (OSDI), San Diego, CA, USA, 22–25 October 2000; pp. 119–134.
47. Vietri, G.; Rodriguez, L.; Martinez, W.; Lyons, S.; Liu, J.; Rangaswami, R.; Zhao, M.; Narasimhan, G. Driving Cache Replacement with ML-based LeCaR. In Proceedings of the 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage), Boston, MA, USA, 9–10 July 2018.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.