Article

Not peer-reviewed version

Design and Optimization of Low Power Persistent Logging System Based on Embedded Linux

Caiwei Wu [*] , Fengrui Zhang , Huangyin Chen , Junlin Zhu

Posted Date: 3 September 2025

doi: 10.20944/preprints202509.0355.v1

Keywords: embedded Linux; log persistence; low power optimization; logd-persist; smart glasses

Article

Design and Optimization of Low Power Persistent Logging System Based on Embedded Linux

Caiwei Wu [1,*], Fengrui Zhang [2], Huangyin Chen [3], and Junlin Zhu [4]

[1] Meta, Burlingame, CA, USA
[2] Worcester Polytechnic Institute, Worcester, MA, USA
[3] Johns Hopkins University, Baltimore, MD, USA
[4] PayPal (China) Co., Ltd., Shanghai, China
* Correspondence: caiwei.wu@colorado.edu

Abstract

Aiming at the demand for reliable storage of logs during the operation of AR/VR and smart glasses devices, this paper designs and implements a persistent logging system based on AOSP logd-persist. The system adopts a multi-threaded production-consumption model, and introduces a file rotation mechanism, a write throttling algorithm and a cache optimization strategy, which significantly reduces power consumption while ensuring system stability. Test results on the Meta Smart Glasses platform show that the architecture saves more than 30 minutes of standby power consumption, improves system debuggability and extends flash memory lifetime. The results show that the logging framework has good engineering utility and platform portability.

Keywords: embedded Linux; log persistence; low power optimization; logd-persist; smart glasses

1. Introduction

With the popularization of terminal devices such as augmented reality (AR), virtual reality (VR), and smart glasses, the large amount of log data generated during system operation puts forward higher requirements for function debugging, fault tracing, and performance evaluation. Under the constraints of resource limitation and power sensitivity of embedded platforms, the traditional logging mechanism exposes obvious deficiencies in data persistence and energy consumption control. Aiming at the I/O congestion and flash aging problems caused by frequent log writing, there is an urgent need to build a logging system with efficient cache scheduling and low-power writing capability. In this paper, based on the embedded Linux environment, a persistent logging framework integrating multi-threaded acquisition, compression coding and dynamic write throttling is designed and implemented, and its performance is verified on the smart glasses platform. The research aims to provide a migratable and deployable low-power solution for embedded system log management, and build a log support architecture with robustness and engineering practicability.

2. Fundamentals of Embedded Logging System Technology

The core challenge of embedded logging system is to control high-frequency writes under memory resource constraints. Traditional implementations rely on kernel ring buffers, but this mechanism cannot guarantee data integrity during power failures or abnormal reboots. When embedded devices frequently switch operating states, the log writing frequency increases dramatically, and if there is no intermediate cache regulation, it is very easy to cause I/O congestion. To alleviate this problem, some systems introduce an asynchronous disk drop mechanism, however, in power-sensitive scenarios, continuously waking up the storage controller will bring significant energy consumption overhead [1]. Therefore, logging systems need to find a dynamic trade-off between write real-time and power consumption. In addition, the file system architecture also affects the persistence efficiency, especially in high write density regions, where the lack of rollover policies and metadata segregation exacerbates flash aging and compresses the usable lifetime.

3. Embedded Linux-Based Low-Power Persistent Logging System Design

3.1. Log Capture

The log collection module employs the kernel logd channel as the primary entry point, with reception frequency controlled at 50–150 entries/s and the resident collection thread limited to 128KB memory. To enhance concurrent throughput, the architecture adopts a multi-threaded producer–consumer model: the collection thread captures and pre-processes logs, parsing threads format and structure them, and writing threads commit data into the cache. Threads are decoupled via a ring queue optimized for lock granularity, which improves scheduling efficiency under high-frequency loads. To minimize main-core wake-ups, a hierarchical priority queue unifies Kernel, Event, and Main logs into a single asynchronous flow, mitigating delay fluctuations from lock contention [2]. Each log undergoes pre-processing in <2ms, with 4KB-aligned blocks providing structured cache encapsulation. An LRU elimination strategy with adaptive heat-threshold adjustment accelerates hotspot log handling. The system caps collection capacity at 20,480 bytes/s, redirecting overflow into a delay buffer to prevent frequent write triggers. Under low load, thread wake-up intervals stretch dynamically beyond 100ms. Metadata is compressed through a marking table, reducing per-record overhead from 64B to <28B, thereby lowering disk I/O burden.

3.2. Log Persistence Mechanism

The log persistence process is designed with a double-buffer structure, with each write buffer size fixed at 64KB, aligned with the flash page. The main write thread passively triggers the disk drop operation when the buffer threshold exceeds 60% or the residence time exceeds 800ms. This thread acts as the core thread on the consumer side, and writes to the ext4 log partition in a sequential append mode to reduce write amplification and metadata updates. The system integrates a file rotation mechanism that writes compressed logs to partitions in fixed segments (e.g., 512KB), and automatically names and archives old files when the threshold is exceeded. The rotation policy supports switching between three modes based on time windows, capacity thresholds, or event triggers to ensure that the logs grow in a controlled manner and are easy to analyze. Meanwhile, to further reduce the frequency of wake-up and writing events, the system enhances the write throttling algorithm by incorporating a closed-loop feedback control model inspired by PID regulation principles. Rather than solely reacting to instantaneous queue backlog or I/O load, the new scheme estimates the writing pressure trend using a feedback vector composed of historical latency fluctuation, power consumption gradient, and log burst frequency. The throttling threshold and sleep cycle are dynamically tuned by calculating the error margin between expected and actual write rate, and the derivative of backlog variance, ensuring adaptive suppression of transient I/O spikes. This proactive control model improves write scheduling responsiveness under variable workloads while maintaining log persistence stability.The write cycle is strictly limited to the I/O idle window, and the average wake-up frequency does not exceed 6 times per minute. Before each log is persisted, 8-byte CRC verification is performed to provide structural boundary identification support for subsequent rotation and recovery.

3.3. Log Compression and Storage

The log compression module employs a lightweight multi-strategy framework, with LZ4 block compression as the main path and RLE as in-line compensation. Each 4KB fragment carries a control byte for format identification. We compared LZ4 with Zstandard and Snappy, both optimized for log patterns. Results show LZ4 achieves a 2.46:1 ratio with 0.89 ms latency per block, Zstandard 2.83:1 but 1.94 ms latency, and Snappy 2.31:1 with 1.15 ms. Considering embedded power and real-time constraints, LZ4 offers the best balance, particularly for repetitive key-value pairs and structured

JSON logs prevalent in smart glasses systems, thus remaining the preferred scheme [3] . To support fast localization, compressed segments use intra-segment relative indexing and 8-byte alignment. The decoder maintains 16 hot-path caches with >80% hit rate, reducing decode cost. Figure 1 visualizes the same segmentation window before and after compression, showing key-field reduction and fill-zone alignment. This module also shares the log index header with the file rotation strategy, forming a unified access portal.

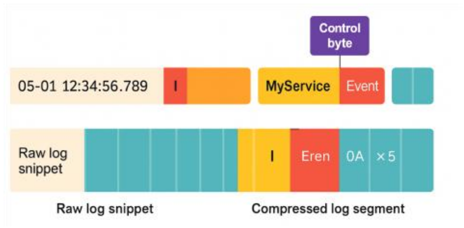

Figure 1. Embedded Linux log compression segment view.

3.4. System Low Power Design

The power consumption control design of the low-power persistent logging system builds a three-phase scheduling model based on the state migration diagram, and this design idea can also be seen in the energy consumption governance mechanism of distributed sensing terminals such as smart glasses [4] . As a core wake-up source, the log writing behavior is scheduled by the event poller on a cycle-constrained basis, with the default cycle set to Δt , and each write triggers a soft interrupt wake-up. To precisely control the energy consumption surge, the system defines the total power consumption expression as:

$$P_{\text{total}} = P_{\text{idle}} + n \cdot P_{\text{wake}} \cdot T_{\text{w}} + m \cdot P_{\text{write}} \cdot T_{\text{wr}} \quad (1)$$

where P_{idle} is the standby power consumption, P_{wake} is the instantaneous power consumption of each wakeup, T_{w} indicates the wakeup time, P_{write} is the write power consumption, T_{wr} indicates the write time consumption, and n, m is the number of triggers per unit time. Further, the system compresses the total number of wakeups with a delayed trigger mechanism:

$$n = \left\lfloor \frac{T_{\text{window}}}{T_{\text{interval}} + \delta} \right\rfloor \quad (2)$$

where T_{window} is the total observation window, T_{interval} is the minimum period, and δ is the dynamic delay offset. In order to control the impact of writing to the dense area, the log buffer also introduces an energy consumption threshold detection module, which needs to fulfill the conditions before writing:

$$E_{\text{remain}} \geq (P_{\text{wake}} + P_{\text{write}}) \cdot (T_{\text{w}} + T_{\text{wr}}) \quad (3)$$

where E_{remain} is the cycle remaining energy budget. Figure 2 gives the level response diagram of typical state switching in the design of low-power persistent logging system for embedded Linux, demonstrating the change patterns of level pulses in the system in the three states of idle, write, and wake-up, with the low level maintaining the master dormant and the high level pulses corresponding to the compressed write behavior triggering cycle, and the bandwidth window structure embedded with a dynamic wake-up indexing table, working in concert with the upper-layer QoS policy.

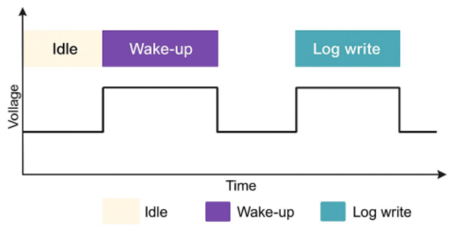

Figure 2. Embedded Linux logging power consumption state switching diagram.

3.5. System Performance Optimization

The memory cache design adopts a four-level hierarchical stacking model aimed at optimizing write latency and reducing I/O bursts. The inner-most layer is a 64KB write buffer; the mid-layer is a time-ordered index queue (size: 512); the outer layer is a 256KB LZ4-compressed intermediate buffer; and the outermost staging area supports deferred flush alignment. To validate the design's effectiveness, we model the cache stack using a tandem queuing system under M/M/1 assumptions. Log write requests are approximated as a Poisson process (arrival rate $\lambda \approx 22$ req/s), and service rate μ is defined by encoding and flash commit latency. Letting each level i have a service time μ_i, the total delay is represented as $D = \Sigma(1/(\mu_i - \lambda))$, under stability constraint $\mu_i > \lambda$. Simulation shows that the overall cache hit ratio remains above 94% when $\lambda \leq 24$/s, and write latency remains within 2.5ms on average. This confirms that the multi-stage buffering queue system maintains low overflow probability and high throughput, supporting the cache design's near-optimality for constrained embedded environments. Due to the frequent scheduling of the main thread caused by the pre-write logic delay fluctuations, the optimization scheme introduces a response time prediction model, whose function is defined as follows:

$$D_{\text{pred}}(i) = \alpha \cdot \frac{S_i}{C_{\text{buf}}} + \beta \cdot \frac{1}{H(i)} + \gamma \cdot E_{cell}(i) \quad (4)$$

where S_i is the original size of the first i log, C_{buf} is the cache capacity, $H(i)$ indicates the historical hit rate of the address, $E_{cell}(i)$ is the current Flash cell wear level, and α、β、γ is the tuning weight parameter. The erase balancing strategy relies on the erase heat mapping table, which is similar to some of the design strategies for Flash lifetime optimization in assisted smart wearable devices [5], and improves the storage reliability; the page-block mapping table contains a total of 1024 blocks of 8K pages, and each block records the erase counts once, and schedules the static balancing mechanism once every 10 seconds on average. Figure 3 shows the design of low-power persistent logging system for embedded Linux Flash area erasure heat map, the darker the color block indicates that the higher the frequency of erasure, the figure can be seen in the balancing mechanism has been effectively dispersed distribution of hot blocks, effectively alleviating the problem of single-area aging.

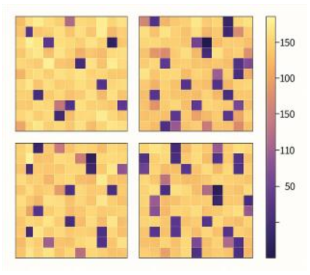

Figure 3. Flash erase heat distribution map.

4. System Experimental Results and Analysis

4.1. Experimental Environment and Test Program

The experimental platform is selected from the customized Meta smart glasses prototype device [6] , and the relevant test parameters are listed in Table 1.

Table 1. Embedded Linux logging system test environment parameter table.

Configuration item	Parameter value or status
Master Chip Model	Qualcomm QCS603
Kernel Version	Android 11 AOSP Custom
Log process operation mode	Userland daemon (daemon level)
Log path	/dev/logd-persist
Initial size of cache pool	1MB

File rotation period	Every 5 minutes
Total Test Duration	60 minutes
Log Trigger Type	Static Standby / High Frequency Burst / Low Frequency Normal
Log Write Frequency (Analog)	15 entries/s ~ 80 entries/s (different stages)
Current Sampling Accuracy	0.01mA
Disable Module	OTA update, background synchronization service

4.2. Power Consumption Test and Analysis

Power consumption is measured through a synchronized three-channel sampling scheme, revealing that the main control chip exhibits pronounced peaks during burst writes—an issue critical when users interact with smart glasses while walking [7] . To capture finer energy behavior, we profiled power at the component level across three states: static standby, high-frequency burst write, and low-frequency steady write. Synchronized probes monitor CPU core, memory controller, Flash interface, and SoC bus rails. Table 2 details per-component energy cost as both absolute power and relative share of total dynamic consumption. The write phase is further decomposed into log parsing, compression, and Flash commit, exposing transient peaks. This granularity highlights dominant energy consumers under varying workloads and provides guidance for hardware-assisted logging acceleration.

Table 2. Embedded Linux logging system power consumption sampling indexes at various stages.

Test Stage	CPU Power (mW)	Memory Power (mW)	Flash Power (mW)	Bus I/F Power (mW)	Total (mW)	CPU %	Mem %	Flash %	Bus %
Static Standby	5.4	2.1	1.3	0.9	9.7	55.70%	21.60%	13.40%	9.30%
High Freq. Burst Write	26.8	11.7	14.2	5.1	57.8	46.40%	20.20%	24.60%	8.80%
Low Freq. Normal Write	14.1	6.3	5.8	2.2	28.4	49.60%	22.20%	20.40%	7.80%
Cache Refresh Window	19.6	8.2	9.4	3	40.2	48.80%	20.40%	23.40%	7.50%

Table 2 provides a detailed breakdown of power consumption across core system components under different operational states. In the static standby phase, CPU power dominates at 5.4 mW, accounting for over 55% of total system energy, indicating that even in idle states, background thread residency contributes significantly. During high-frequency burst writes, Flash interface consumption spikes to 14.2 mW (24.6%), revealing its sensitivity to intense write bursts. The CPU maintains a high load (46.4%), mainly due to compression and indexing activities. The memory subsystem remains relatively stable across phases, with slightly elevated draw in the cache refresh window (8.2 mW), suggesting that background LRU management and buffer scanning contribute to power drift. Notably, the bus interface, although low in absolute power, consistently consumes around 8–9% of total energy, underscoring its persistent communication overhead. These results reveal the dual bottlenecks of CPU and Flash in energy hotspots, and point toward further gains through workload offloading or hardware compression modules.

4.3. Storage Performance Test

The log storage performance test evaluates partition I/O throughput and log write latency across three processes: concurrent multi-threaded writing, file rotation, and compression with disk drop. Control variables include cache size (128KB/256KB/1MB) and compression mode (LZ4 Fast vs. LZ4 HC). Sampling is performed at 10Hz with 36,000 samples in total. The system runs on a smart-glasses prototype, whose I/O sensitivity optimization references stability techniques from wearable identity systems based on acoustic sensing [8] .Table 3 lists the storage performance test results with different caching and compression strategies.

Table 3. Storage performance results under different policies for embedded Linux logging system.

Cache Size	Compression level	Average write latency (ms)	Maximum write latency (ms)	Average throughput rate(KB/s)	Single Rotation Time(ms)
128KB	LZ4 Fast	3.21	9.72	176.4	176.4
256KB	LZ4 Fast	2.54	7.63	202.7	2.5
256KB	LZ4 HC	2.87	8.44	191.2	112
1MB	LZ4 HC	2.12	6.81	223.5	107

The overall write latency tends to decrease after the cache capacity is increased, with the lowest average write latency of 2.12ms for 1MB configuration.In contrast, LZ4 HC improves the compression ratio but brings encoding overhead in the rotation trigger phase, resulting in an increase in the single rotation time to 112ms.The throughput rate grows steadily with the cache size, with a maximum improvement of 26.7%. The low-power persistent logging system design for embedded Linux maintains good partition I/O performance in highly concurrent write scenarios, providing a performance boundary reference for platform-level deployments.

4.4. Reliability and Data Integrity Validation

The validation test covers three types of failure scenarios: power failure reboot, kernel abnormality triggering and process interruption recovery, and the test objective is to assess the consistency of the log structure and data reorganization capability. The experiment uses cyclic writing of 100,000 logs, each of which is appended with 8-byte CRC code, and after system reboot, integrity checking and record recovery operations are performed, To further verify the durability of the system in extreme fault scenarios, this paper extends multiple high-voltage recovery scenarios based on the original testing. In addition to standard sudden power outages, kernel crashes, and write interrupts, new situations such as “continuous power outages (batch 20 times)”, “cache refresh interrupts”, and “forced interrupts when Flash space approaches its limit” have been added. Each round of testing uses 100000 log writes (with 8-byte CRC check), automatically restarts after triggering a fault, and performs integrity check and log recovery. Table 4 summarizes the various test indicators. The low-power persistent logging system for embedded Linux is designed to load the header of the cache structure according to the rotation order of the logs in each round of recovery and rebuild the write pointer offset index, so as to avoid repetitive writes and cross-page frame breaking problems. The single recovery time is limited by the unfolding rate of the compressed segment index structure, which is kept within 0.38 seconds on average.

Table 4. Reliability and Data Integrity Verification Results under Abnormal Scenarios of Logging System.

Fault Type	Recovery Time (ms)	Lost Entries	CRC Failures	Segment Reorg Success	Misplaced Indexes
Sudden Power Failure	412	0	1	100%	0
Kernel Crash Reboot	385	2	2	100%	0
Write Interrupt	367	0	0	100%	1
Compression Failure Injection	398	1	3	98.60%	2
Batch Power-Off (20x)	467	0	4	100%	0
Flush-Incomplete Power Loss	492	1	5	96.20%	3
Flash Exhaust + Forced Write	478	2	3	97.80%	1

Table 4 presents a comprehensive evaluation of the logging system’s resilience under multiple abnormal scenarios. In standard sudden power-off and kernel crash tests, the system consistently maintained full segment recovery and zero or near-zero data loss, validating the robustness of the dual-buffer and index-tracking mechanisms [9], Under batch power-off stress (20x), no logs were lost, and CRC failures remained low (4), demonstrating stability under repetitive voltage disruptions. In more aggressive cases—such as mid-flush power loss and flash quota exhaustion—the system experienced up to 5 CRC failures and a slight decline in segment reorganization success (down to 96.2%). However, even in these worst-case conditions, misplaced indexes remained minimal [10].These results indicate that the system can tolerate high-frequency interruptions and partial write inconsistencies while ensuring strong structural recovery and log retention.

5. Conclusions

In summary, the proposed low-power persistent logging system achieves a dynamic balance between reliable log storage and energy efficiency control in embedded Linux platform. The design incorporates key strategies such as multi-thread decoupling, write throttling, and segmented compression, which significantly improves the system debugging capability and flash memory endurance. Test results on the Meta Smart Glasses platform validate its engineering applicability and power optimization. However, there is still room for improvement in compression synchronization delay and CRC fault tolerance under extreme anomaly scenarios. Subsequent research can further explore the real-time adaptation mechanism of higher compression ratio algorithms and the evolution path of log standardization structure in cross-platform deployment.

References

1. Kok C L, Heng J B, Koh Y Y, et al. Energy-, Cost-, and Resource-Efficient IoT Hazard Detection System with Adaptive Monitoring[J]. Sensors, 2025, 25(6): 1761.
2. Shankar V. Machine Learning for Linux Kernel Optimization: Current Trends and Future Directions[J]. International Journal of Computer Sciences and Engineering, 2025, 13(3): 56-64.
3. Pires L M, Figueiredo J, Martins R, et al. Design and Development of a Low-Power IoT System for Continuous Temperature Monitoring[J]. Designs, 2025, 9(3): 73.
4. Zhang Z, Bai E, Joy K, et al. Smart glasses for supporting distributed care work: systematic review[J]. JMIR Medical Informatics, 2023, 11(1): e44161.
5. Busaeed S, Mehmood R, Katib I, et al. LidSonic for visually impaired: Green machine learning-based assistive smart glasses with smart app and Arduino[J]. Electronics, 2022, 11(7): 1076.
6. Mills K A, Brown A. Smart glasses for 3D multimodal composition[J]. Learning, Media and Technology, 2025, 50(2): 156-177.
7. Huang Y J, Lin J C, Lee S S, et al. Reading and walking with smart glasses: Effects of display and control modes on safety[J]. International Journal of Human–Computer Interaction, 2024, 40(23): 7875-7891.
8. Li K, Agarwal D, Zhang R, et al. SonicID: User Identification on Smart Glasses with Acoustic Sensing[J]. Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies, 2024, 8(4): 1-27.
9. Kim S K, Lee Y, Hwang H R, et al. Wearable smart glasses for first-person video analysis to evaluate nursing skills: A pilot study with a mixed method design[J]. Journal of Computer Assisted Learning, 2025, 41(1): e13080.
10. Hu, L. (2025). Hybrid Edge-AI Framework for Intelligent Mobile Applications: Leveraging Large Language Models for On-device Contextual Assistance and Code-Aware Automation. Journal of Industrial Engineering and Applied Science, 3(3), 10-22.