

Article

RankProp: Learning and Propagating Priorities for Cached Apps in Android Memory Management

Jaehwan Lee ¹ and Yeunwoong Kyung ^{2,*}¹ Department of Computer Science and Engineering, Kongju National University, Cheonan 31080, Republic of Korea; jhnlee@kongju.ac.kr² Department of Electronic Engineering, Seoul National University of Science and Technology, Seoul 01811, Republic of Korea

* Correspondence: ywkyung@seoultech.ac.kr

Abstract

Modern smartphones keep background apps to reduce latency during frequent app switching. Under memory pressure, Android terminates cached processes and reclaims pages, yet neither path represents when each resident app is likely to return. This paper presents RankProp, which learns an eviction order over the current resident app set and propagates through Android's memory management. Unlike its predecessor, which scores apps independently, RankProp uses machine learning to train a neural network ranking model that produces an eviction order over all resident apps at each launch event. Android projects the order onto a cached process to reclaim priority and derives a hint for page reclaim. Across 279 user app usage traces, RankProp enhances the aggregate hit ratio from 0.9172 for the calibrated predecessor to 0.9252 across logical app cache capacities from 5 to 15. In an Android emulator with synthetic apps and sustained memory pressure, propagation to both reclaim paths reduced mean launch time by 4.71 percent relative to Android's stock recency policy. Within this scope, mobile operating systems' reclaim paths can consume priorities derived from the app eviction ranking without replacing their native decision units.

Keywords: intelligent memory management; artificial intelligence integration; Android memory management; prediction of relaunch distance; resident app ranking

1. Introduction

Smartphone users frequently switch among various apps, such as messaging, browsing, camera, navigation, and banking, at short intervals. As app switching becomes more frequent, the performance perceived by users depends on more than the time required for a cold launch. When the execution state of a recently used app remains in main memory, the system can restore the previous screen and task context with low latency. If the state has been discarded by process termination or aggressive page reclaim, however, the next switch incurs launch latency and stalls that users can perceive. Consequently, deciding which app states to preserve in limited main memory is central to memory management in mobile operating systems [1–4].

Relaunch latency can be reduced by keeping apps after they leave the foreground as background or cached processes. Under memory pressure, Android frees memory by terminating selected cached processes, reclaiming pages that have not been accessed recently,



Academic Editor: Juan-Carlos Cano

Received: 23 June 2026

Revised: 23 July 2026

Accepted: 25 July 2026

Published: 28 July 2026

Copyright: © 2026 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 \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

or compressing anonymous pages into zRAM. Within Android, the low memory killer daemon (lmkd) selects victims at process granularity, whereas multi-gen least recently used algorithm (MGLRU) [5] ages and selects pages for reclaim from their access histories. These mechanisms handle low memory events robustly, but neither path represents which resident apps are likely to return soon [1,5,6]. This limitation can determine whether a later app switch is served as a hot launch or requires a full relaunch. A policy that follows recency alone may keep an app that was opened briefly over an app whose last use was earlier but whose use recurs regularly. Moreover, a signal about a future launch cannot guide both reclaim paths consistently unless Android propagates it with a shared meaning.

Research on memory management for app switching has explored app usage prediction, victim selection that uses app state, and swapping. Prediction approaches estimate the probability of each app's next launch from app sequences and temporal context [7–10]. More recent work estimates relaunch distance, which counts the distinct app identities observed before the same app is launched again. An eviction policy that uses relaunch distance provides a more direct signal than prediction of only the next app because it estimates how far each resident app is from future relaunch. M2ARP [2] introduced relaunch distance for cached app eviction and used the least recently used (LRU) policy as a fallback when prediction was unavailable or unreliable. A calibrated relaunch prediction [11] later placed predicted relaunch distance and recency fallback on one runtime scale. Recent systems also reduce reclaim cost through swap coordination, parallel page reclaim, background process freezing, and compressed memory management [3,12–16]. These systems improve particular stages of memory management, but they do not construct a relative order over the current resident apps for Android reclaim paths. Because the calibrated relaunch prediction scores each app independently, it also omits the composition of the resident set at the moment of reclaim. The remaining problem is therefore to convert the calibrated relaunch score into an order over the current resident set. This study asks whether that order can improve cached app hit ratio and launch behavior when Android projects it safely onto its operating system's reclaim paths.

This paper introduces RankProp, a framework that learns a total order over the apps currently resident in memory and propagates that order to Android memory management as a common control signal. RankProp constructs this order with a residual neural ranker that combines calibrated relaunch scores with the current state of each candidate app. The resulting order is then consumed through separate deterministic mechanisms for process reclaim and page reclaim. For process reclaim, the corresponding mechanism projects the order onto the subset of cached processes that are eligible for termination. For page reclaim, the other mechanism aggregates the order to compute a bounded scan hint for each app sandbox. These components allow Android to apply a learned ordering consistently across two distinct reclaim paths while preserving the constraints of each path.

The performance evaluation examines ranking quality through trace replay and runtime behavior through experiments in an Android emulator. In trace replay from 279 users' app usage history and app-level logical cache capacities ranging from five to fifteen apps, RankProp's ranking model achieves an aggregate hit ratio of 0.9252, compared with 0.9172 for the calibrated relaunch prediction. In the emulator experiments, the complete RankProp configuration reduces mean launch time by 4.71% relative to Android LRU. This configuration also reduces the measured physical memory footprint while increasing the amount of data stored in swap, indicating a change in memory placement rather than a uniform reduction in memory consumption. Together, these results suggest that a common ranking over resident apps can improve relaunch behavior when Android applies the ranking to its memory management subsystem.

The main contributions of this paper are as follows:

- We define a ranking model that starts from a calibrated relaunch score and learns a bounded correction from the composition of the current resident set.
- We design a propagation path that applies the app eviction order within Android app processes and derives a bounded hint for page reclaim mechanism.
- We evaluate the ranker with LSApp, the app launch trace dataset used in this study, and evaluate the propagation design with Android emulator replay under controlled memory pressure.

The remainder of this paper is organized as follows. Section 2 provides background on Android memory reclaim and app relaunch distance. Section 3 presents the resident app ranking model that converts calibrated relaunch scores into an eviction order over the current resident set. Section 4 describes the architecture that propagates the resulting ranking through the Android framework and Linux kernel. Section 5 presents the experimental setup and evaluates ranking quality, emulator performance, and inference overhead. Section 6 reviews related work on app usage prediction, mobile memory management, and learned replacement policies. Section 7 concludes the paper with the main findings, limitations, and directions for future work. Appendix A provides the full experimental setup.

2. Background

RankProp uses a calibrated estimate of future app reuse to influence both process reclaim and page reclaim. This section establishes the native semantics of these paths and then defines the inherited score.

2.1. Android Memory Reclaim

After an app leaves the foreground, Android may keep its process as a cached process. The cached process preserves execution state and can shorten a later launch. Under memory pressure, Android reclaims memory at either the process level or the page level. Process reclaim terminates a cached process and discards its state. A later return then requires a full relaunch. Page reclaim keeps the process alive but removes selected pages from physical memory. Accessing a reclaimed page causes a refault. The kernel may then reload the page from a file or from swap. The paths act on different units and impose different recovery costs.

For low memory process killing, Android protects each process according to its current importance. Relevant changes in an app lifecycle cause `system_server` to reevaluate the process that hosts its components. This evaluation also follows process dependencies because one process may support work performed by another. The framework uses this information to compute both a process state and an out-of-memory (OOM) adjustment value. It then sends the adjustment to `lmkd` through the `LMK_PROCPRIO` command. The daemon records the value in `/proc/<pid>/oom_score_adj` and tracks the process at the corresponding priority. Lower values provide stronger protection from termination. Processes that support foreground or perceptible work therefore receive stronger protection than cached processes. Within the cached class, the baseline policy uses the least recently used order to assign relative levels of protection [2]. The `lmkd` daemon monitors memory pressure and can use pressure stall information (PSI) to detect delays caused by memory shortage [6]. When current memory conditions require a kill, `lmkd` determines the minimum adjustment value that is eligible for termination. It then considers processes at or above that threshold, beginning with those that have weaker protection. Current importance determines which work Android protects at a given moment. LRU recency further distinguishes processes within the cached class. However, recency reflects past use and

does not explicitly estimate the distance to the next relaunch. This limitation motivates the relaunch distance signal defined next.

At the kernel level, page reclaim uses access recency to select individual pages. The conventional Linux LRU approximates recency with active and inactive lists. This representation records page age at a coarse granularity. Multi-Gen LRU (MGLRU) refines this representation by dividing reclaimable pages into multiple generations ordered by access recency [5]. During aging, MGLRU promotes recently accessed pages to the youngest generation. During eviction, it scans the oldest generation first. Refault feedback adjusts subsequent protection when an evicted page is accessed again. These mechanisms improve the identification of cold pages within a reclaim scope. Android assigns a distinct Linux user identifier (UID) to each app sandbox [17]. When per-app memory cgroups are enabled, processes associated with that UID can be grouped in a memory cgroup [18,19]. The memory cgroup accounts for the group's memory. It also provides a target for reclaim. RankProp applies reclaim control at this scope. MGLRU then selects individual pages within the selected memory cgroup.

2.2. App Relaunch Distance

Relaunch distance measures how many distinct apps appear before the same app returns [2]. Let $L = (\ell_1, \dots, \ell_T)$ denote a chronological launch sequence. For a launch at position j , let $j^+(j)$ denote the first later position that contains the same app. The relaunch distance is

$$d_j = \begin{cases} |\{\ell_s \mid j < s < j^+(j)\}|, & \text{if } j^+(j) \text{ exists,} \\ -1, & \text{otherwise.} \end{cases} \quad (1)$$

A finite value counts the distinct app identities that intervene before the return. The value -1 means that no later launch appears within the observation horizon, not that the app will never return. For example, in $L = (A, B, C, B, D, A)$, the first launch of A has distance 3 because B , C , and D intervene. A larger finite distance denotes later reuse.

The exact value of d_j is determined by launches that occur after position j . Existing work therefore estimates d_j when ℓ_j is launched [2]. The predictor uses the history observed through position j and evaluates only the app that has just launched. It emits a finite distance estimate $\hat{d}_j^{\text{fin}}$. Later work adds a probability p_j^{nr} that no return appears within the observation horizon [11]. Let d_{clip} denote the largest finite relaunch distance represented by the predictor. $d_{\text{nr}} = d_{\text{clip}} + 1$ represents a return that is not observed within the current data split. This bounded representation compresses the long tail of relaunch distance and prevents rare large values from dominating the regression loss. The signal computed at launch is

$$m_j = (1 - p_j^{\text{nr}})\hat{d}_j^{\text{fin}} + p_j^{\text{nr}}d_{\text{nr}}. \quad (2)$$

A larger m_j represents a later expected return. The system stores this output for the app's current launch. A later reclaim decision reads the output from each app's most recent launch instead of rerunning the predictor over the resident set.

The resident set can contain apps with and without usable relaunch distance. Relaunch distance estimation should compare both cases through the distinct age $k_t(a)$, which counts the distinct app identities launched since app a last appeared. A launch resets the launched app's age to zero. It increases another resident app's age only when the launched identity has not already been counted. This update uses observed launches and requires no predictor inference. Because distinct age and relaunch distance use the same counting unit, the fallback function $b(k)$ maps age to an expected clipped relaunch distance. The function is monotone, so a larger age cannot produce a smaller fallback value. At decision time t , $m_t(a)$ is the latest stored output, $\tilde{k}_t(a)$ is the clipped age, and $\xi_t(a)$ in-

icates whether the stored output may be used. The indicator is zero when no output is available or when the app has not met the prediction eligibility rule. The weight $\alpha_t(a) \in [0, 1]$ controls interpolation between the fallback and the stored estimate. The inherited score is

$$\rho_t(a) = b(\tilde{k}_t(a)) + \xi_t(a)\alpha_t(a) \left[m_t(a) - b(\tilde{k}_t(a)) \right]. \quad (3)$$

Without a usable output, $\rho_t(a)$ equals the fallback. Otherwise, it interpolates between the fallback and the stored launch-time estimate. The stored estimate remains fixed until the app launches again. The fallback is recomputed from the current age as distinct launches accumulate. Therefore, predicted and fallback apps can be compared without predicting every resident app at every launch. RankProp inherits a frozen model from the calibrated relaunch distance predictor [11], leaving Equations (2) and (3) unchanged. However, this calibrated score does not by itself determine a reclaim action. First, $\rho_t(a)$ scores each app independently, whereas a reclaim decision compares the apps that are resident at the same time. Second, a common app priority cannot control both paths in the same way because process reclaim terminates a process, whereas page reclaim selects individual pages. These two requirements motivate the RankProp design in the next section.

3. Resident App Ranking Model

We propose RankProp, which constructs an eviction order of the resident candidates. The inherited predictor evaluates each app's relaunch score only at its launch, so its score does not account for the composition of the current resident set or the context of the reclaim event. RankProp addresses this limitation with a compact neural ranker that applies a correction to eligible candidates. A scoring head evaluates each candidate using its local state. The following subsections define the ranking input, scoring rule, and learning objective.

3.1. Ranking Input Representation

At ranking update t , let $\mathcal{I}_t = \{1, \dots, n_t\}$ index the resident candidates $\mathcal{R}_t = \{a_i \mid i \in \mathcal{I}_t\}$. Each a_i denotes one app identity and its current resident state. The index i serves only as an identifier. RankProp combines candidate state with a symmetric summary of the resident set.

The calibrated score $\rho_t(a_i)$ provides the base eviction signal. RankProp normalizes this score and appends predictor state and policy state:

$$s_{i,t}^{\text{base}} = \text{clip}\left(\frac{\rho_t(a_i)}{d_{\text{nr}}}, 0, 1\right), \quad (4)$$

$$\mathbf{x}_{i,t} = [s_{i,t}^{\text{base}}, s_{i,t}^{\text{raw}}, r_{i,t}^{\text{rd}}, p_{i,t}^{\text{nr}}, \alpha_{i,t}, \bar{k}_{i,t}, \bar{b}_{i,t}, e_{i,t}, \ell_{i,t}^{\text{LRU}}]^T \in [0, 1]^9.$$

In (4), the predictor group contains the raw score $s_{i,t}^{\text{raw}}$, the finite relaunch estimate $r_{i,t}^{\text{rd}}$, the nonreturn probability $p_{i,t}^{\text{nr}}$, and the combination weight $\alpha_{i,t}$. The policy group contains distinct age $\bar{k}_{i,t}$, fallback score $\bar{b}_{i,t}$, app eligibility $e_{i,t}$, and the position in Android's baseline victim order $\ell_{i,t}^{\text{LRU}}$. The base score increases with expected relaunch distance. A larger base score gives greater eviction priority. Every feature comes from the launch history and resident metadata available at update t .

A candidate score must account for the current competitors. RankProp therefore pools selected channels across the resident set:

$$\begin{aligned}
\mathbf{v}_{i,t} &= [s_{i,t}^{\text{base}}, s_{i,t}^{\text{raw}}, r_{i,t}^{\text{rd}}, p_{i,t}^{\text{nr}}, \alpha_{i,t}, \bar{b}_{i,t}, \ell_{i,t}^{\text{LRU}}]^T, \\
\mathcal{V}_t &= \{\mathbf{v}_{i,t} \mid i \in \mathcal{I}_t\}, \\
\mathbf{z}_t &= [\text{mean}(\mathcal{V}_t); \text{sd}_0(\mathcal{V}_t); \min(\mathcal{V}_t); \max(\mathcal{V}_t)] \in \mathbb{R}^{28}, \\
\mathbf{g}_t &= [g_t^n, g_t^{\log n}, g_t^{\text{elig}}, g_t^{\text{pressure}}, g_t^{\text{target}}]^T \in [0, 1]^5, \\
g_t^{\text{elig}} &= \frac{1}{n_t} \sum_{i \in \mathcal{I}_t} e_{i,t}, \\
\mathbf{h}_{i,t} &= [\mathbf{x}_{i,t}; \mathbf{z}_t; \mathbf{g}_t] \in \mathbb{R}^{42}.
\end{aligned} \tag{5}$$

In (5), sd_0 computes the uncorrected standard deviation across the resident set. The symmetric operators make $\mathbf{z}_t$ independent of candidate enumeration. The vector $\mathbf{g}_t$ contains the linear set size, logarithmic set size, eligibility ratio, normalized memory pressure level, and normalized reclaim target. The two runtime context values are set to zero in the offline trace replay because that trace does not contain Android pressure or reclaim target observations. The resulting $\mathbf{h}_{i,t}$ has 42 coordinates and is the input to the residual head.

3.2. Scoring and Eviction Ordering

The base score defines the default eviction order. RankProp learns an additive correction that adapts this order to the current resident composition. A shared function f_θ evaluates each candidate vector $\mathbf{h}_{i,t}$. RankProp computes the correction and final score as follows:

$$\begin{aligned}
\Delta_{i,t} &= \tanh(f_\theta(\mathbf{h}_{i,t})), \\
\hat{s}_{i,t} &= \text{clip}(s_{i,t}^{\text{base}} + e_{i,t}\Delta_{i,t}, 0, 1).
\end{aligned} \tag{6}$$

In (6), the hyperbolic tangent restricts $\Delta_{i,t}$ to $[-1, 1]$. The clipping operator keeps $\hat{s}_{i,t}$ within $[0, 1]$. When $e_{i,t} = 0$, the final score equals the base score. The shared head and symmetric pooling preserve permutation equivariance.

The candidates are sorted in decreasing score order. Larger scores receive greater eviction priority. Stable sorting preserves the baseline order for tied scores. The final score represents relative priority within the current ranking update. The committed scores define the order consumed by the reclaim path. A change in resident composition can change a candidate's rank at the next update.

3.3. Learning the Residual Ranker

Training replays each user trace in launch order for every logical capacity $C \in \{5, \dots, 15\}$. A sample is recorded only when the requested app misses a full resident set. At launch record t , the replay updates the history, the prediction for ℓ_t , eligibility, and distinct ages. It then snapshots the current residents before the base policy selects a victim and inserts ℓ_t . Each candidate uses the prediction stored at its latest launch, its updated age, and its current LRU rank. Training starts from an empty state. Validation and test first replay earlier splits to restore state, then collect samples. Samples from all training traces and capacities train one global residual ranker.

The inherited predictor is trained with oracle relaunch distance generated from future events inside the corresponding training split. Its finite target is clipped at d_{clip} , and non-return is encoded as d_{nr} . This bounded target reduces the dynamic range of the regression problem. The residual ranker uses different offline supervision: the set of victims selected by Belady-MIN at each sampled miss of a full cache. Belady-MIN also defines the clairvoyant upper bound in the offline evaluation. Neither form of future information is available during deployment. Relaunch distance counts distinct intervening app identities, whereas the residual target uses next occurrence measured in launch records. Repeated launches

therefore occupy separate target positions. For a sampled miss at position t , let T_s denote the final launch record and let $H_t = T_s - t$ denote the number of remaining future records at t . Define

$$\begin{aligned}\mathcal{H}_t &= (\ell_{t+1}, \dots, \ell_{T_s}), \\ q_{i,t}^* &= \min(\{h \in \{1, \dots, H_t\} \mid \ell_{t+h} = a_i\} \cup \{H_t + 1\}), \\ \mathcal{B}_t &= \arg \max_{i \in \mathcal{I}_t} q_{i,t}^*, \\ y_{i,t} &= \begin{cases} |\mathcal{B}_t|^{-1}, & i \in \mathcal{B}_t, \\ 0, & \text{otherwise.} \end{cases}\end{aligned}\quad (7)$$

In Equation (7), $H_t + 1$ marks a candidate with no recurrence in the partition suffix. The set $\mathcal{B}_t$ contains the victims selected by Belady–MIN for that suffix [20]. The hard target assigns them equal mass.

Let $\mathcal{T}$ contain sampled misses with at least two candidates and at least one eligible candidate. For each $t \in \mathcal{T}$, let $\mathcal{E}_t = \{i \in \mathcal{I}_t \mid e_{i,t} = 1\}$. RankProp compares the hard target with a softmax over the final scores and minimizes the following listwise objective [21]:

$$\begin{aligned}\hat{y}_{i,t} &= \frac{\exp(\hat{s}_{i,t}/\tau_s)}{\sum_{j \in \mathcal{I}_t} \exp(\hat{s}_{j,t}/\tau_s)}, \\ \mathcal{L}(\theta) &= -\frac{1}{|\mathcal{T}|} \sum_{t \in \mathcal{T}} \sum_{i \in \mathcal{I}_t} y_{i,t} \log \hat{y}_{i,t} \\ &\quad + \frac{\beta}{|\mathcal{T}|} \sum_{t \in \mathcal{T}} \frac{1}{|\mathcal{E}_t|} \sum_{i \in \mathcal{E}_t} \Delta_{i,t}^2, \quad \tau_s = 0.1, \quad \beta = 0.01.\end{aligned}\quad (8)$$

The softmax in Equation (8) covers every resident candidate. Cross entropy shifts probability toward the optimal tie set. Ineligible candidates keep their base scores and anchor eligible candidates. The quadratic term limits residual magnitude.

4. Rank Propagation Architecture

4.1. Architecture Overview

Section 3 produces an eviction score for each resident candidate. Sorting these scores yields the resident order. Android reclaims memory at process and page levels, so RankProp converts this order into a separate input for each path. The process input reorders comparable `lmd` candidates, while the page input sets the reclaim bias of the corresponding UID. Figure 1 places both conversions between `system_server` and the existing Android and Linux reclaim components. This arrangement lets one resident ranking guide both paths while each path preserves its own decision unit. When the framework has no ranking to publish, the consumers use their baseline inputs.

A successful activity launch triggers rank propagation in Android's ActivityManagerService (AMS). RankProp snapshots the live cached set. A dedicated background thread updates the prediction for the launched app. The same thread then scores the resident snapshot with stored candidate predictions. RankProp atomically commits only a complete result for the current request. The commit schedules a full `oom_score_adj` recomputation over the LRU process list, using the activity update. `OomAdjuster` rebuilds the live safe set and normalizes the committed protection scores within that set. It then projects the resulting order onto Android's cached `oom_score_adj` range. After the update, the page reclaim path then receives a bounded per-app hint through `lmd`. Thus, inference follows the launch, while both reclaim paths consume the committed result at a later stage. `logd` records optional framework and `lmd` events, whereas kernel tracepoints record scan hint activity.

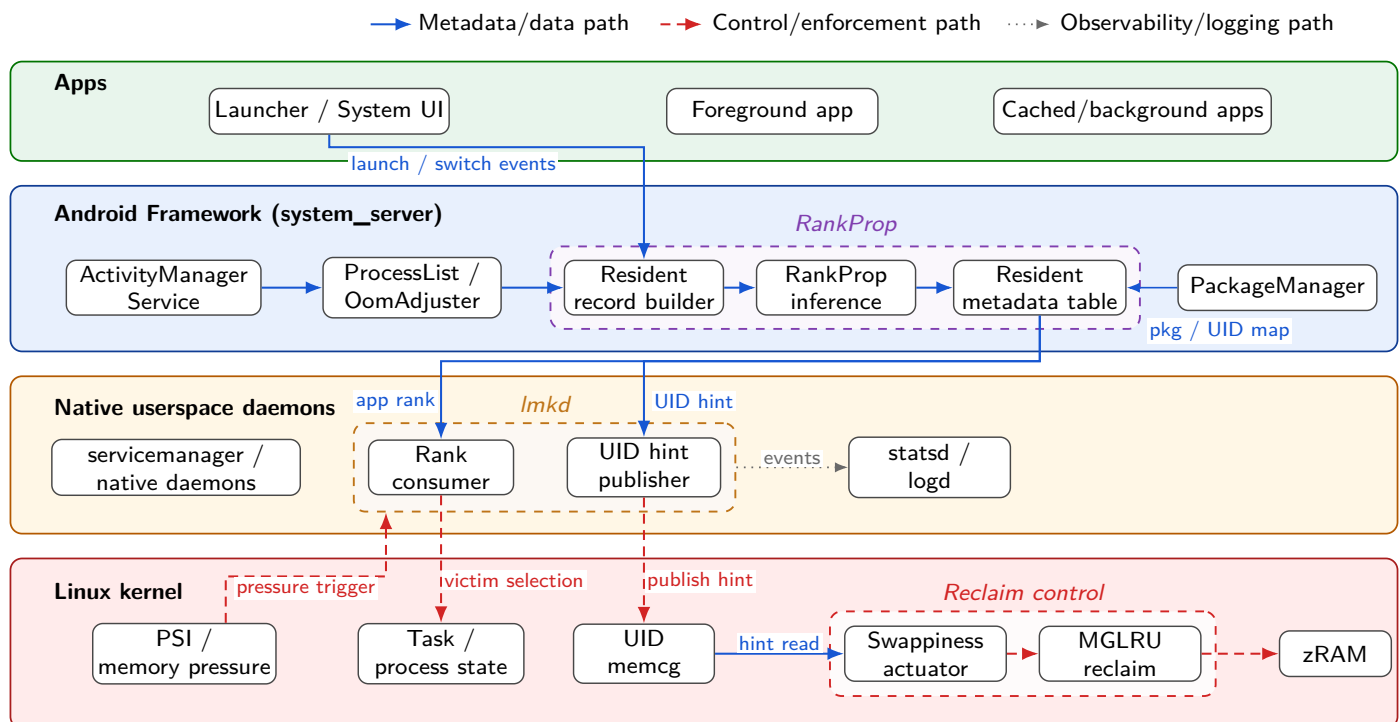


Figure 1. RankProp architecture across the Android Open Source Project (AOSP) and the Linux kernel. Blue arrows carry metadata. Red dashed arrows carry control and enforcement. Gray dotted arrows carry observability events.

4.2. Rank Propagation to Process Reclaim

The process branch runs during the full out-of-memory adjustment (OOMAdj) pass that follows a score commit. It selects safe cached processes from the canonical AMS LRU list. Eligibility excludes protected or persistent processes. It also excludes connection groups and above-client dependencies. Let $m_t \geq 2$ denote the number of eligible processes. Let $r_{i,t} \in \{0, \dots, m_t - 1\}$ denote the rank of process i , where zero has the highest eviction priority. RankProp maps this rank to

$$\text{adj}_{i,t} = 999 - \text{round}\left(\frac{99r_{i,t}}{m_t - 1}\right). \quad (9)$$

Equation (9) maps process ranks onto the OOM adjustment interval reserved for cached processes, from CACHED_APP_MIN_ADJ (900) to CACHED_APP_MAX_ADJ (999). Within this interval, a larger adjustment value indicates a higher eviction priority. Equal scores preserve the original LRU order by maintaining recency information. The projection changes cached adjustment values while the canonical LRU list remains unchanged. The low memory killer daemon receives these values through Android's existing process priority protocol. Consequently, projected `oom_score_adj` values provide the RankProp input to `lmkd` victim selection. Partial OOMAdj updates hold the most recent complete ranking.

4.3. Rank Propagation to Page Reclaim

A surviving app can still lose pages during global reclaim. RankProp therefore derives a UID scan hint after each full OOMAdj pass. Any foreground, perceptible, persistent, or system process suppresses the shared UID. For each accepted UID, the controller keeps the shortest expected relaunch distance and the strongest process protection rank.

Let $v_1, \dots, v_n$ denote the eligible UIDs sorted from strongest process protection to highest eviction priority. Let $d_{(1)} \leq \dots \leq d_{(n)}$ denote their clipped expected relaunch

distances after an independent sort. This pairing assigns longer distances to less protected UIDs. With median $\tilde{d}$, the controller computes

$$\begin{aligned} x_j &= \max(d_{(j)} - \tilde{d}, 0), \\ R &= \sqrt{\frac{1}{n} \sum_{j=1}^n x_j^2}, \\ G &= \text{clip}\left(1 - \frac{\sum_j x_j^2}{(\sum_j x_j)^2}, 0, 1\right) \times \text{clip}\left(\frac{R}{d_{\text{clip}}}, 0, 1\right), \\ S_j &= \frac{e^{x_j/R} - 1}{e^{x_{\max}/R} - 1}, \\ Q_{v_j} &= \text{round}(1024 \times \text{clip}(GS_j, 0, 1)). \end{aligned} \quad (10)$$

Here, $x_{\max} = \max_j x_j$, and the terms G and S_j set the hint strength. The value Q_{v_j} uses Q10 fixed point units, in which 1024 represents 1.0. Therefore, hints apply only to less protected UIDs in the upper tail.

`ProcessList` sends updated hints, then `lmkd` writes each value to the UID's cgroup interface [19]. During global reclaim by `kswapd`, MGLRU computes $N_v^{\text{eff}} = N_v^{\text{base}} + \left\lceil N_v^{\text{base}} \min(Q_v, Q_{\max}) / 1024 \right\rceil$, where N_v^{eff} denotes the effective number of pages to be scanned for reclaim and N_v^{base} denotes the default scan amount. Therefore, MGLRU scans more pages from apps with lower protection priority during reclaim.

4.4. Lightweight Native Execution

The launch callback posts inference through the Java Native Interface (JNI) to the RankProp inference thread. This placement keeps model execution outside the AMS main path and the reclaim path. The relaunch prediction stage runs once for the launched app, and the ranking stage reuses the resulting features for resident candidates. That update recomputes the deterministic calibrated features and evaluates only the residual head. When candidate rows share the same prefix of input sequence length, the runtime evaluates the prefix once and applies the final event separately to each candidate. These reuse points confine repeated computation to the state that changed.

5. Evaluation

The evaluation addresses two questions. First, we examine whether RankProp's ranking model improves the resident app order derived from app launch histories. Second, we examine whether Android benefits when that order is propagated to process reclaim and page reclaim under memory pressure. We answer the first question with model quality evaluation and the second with an Android emulator under memory pressure. The following setup defines the trace preparation and the protocol used in each experiment.

5.1. Experimental Setup

Both evaluation stages use LSApp, the app launch trace dataset, as the workload source [22]. Here, a user denotes one anonymous LSApp user and that user's chronological app launch trace. It does not denote an Android user account, a managed profile, or several accounts that run concurrently on the emulator. Each user trace is split and replayed independently. We remove consecutive foreground records of the same app so that each remaining record represents one launch event. The processed trace contains 198,270 launch events from 291 users and 87 apps. We divide each user trace in chronological order into training, validation, and test segments with nominal ratios of 70%, 10%, and 20%, respectively. Labels that require future observations are constructed sepa-

rately within each segment. A user is eligible for evaluation only when all three segments are nonempty. This criterion produces 279 users whose traces contain 198,219 launch events. During test replay, each policy reconstructs its online state from preceding events, while the reported metrics use the test splits. The test replay contains 39,901 launch events from the 279 evaluation users. The Transformer vocabulary contains 83 identifiers: 81 app tokens, PAD, and UNK. Apps that are absent from the training vocabulary are mapped to UNK during validation and test replay. Table 1 summarizes the trace and label profile.

Table 1. Trace profile and label preparation used by the inherited predictor and RankProp.

Item	Configuration
Trace source	LSApp trace dataset
Preprocessing	Remove consecutive foreground records of the same app
Processed trace	198,270 launches, 291 users, and 87 app identities
Chronological split	Training, validation, and test partitions with nominal ratios of 70%, 10%, and 20%
Relaunch-distance label	Number of distinct app identities before the next launch of the same app
Finite-distance profile	$d_{50} = 2$, $d_{90} = 7$, and $d_{99} = 16$
Clipping and non-return	$d_{\text{clip}} = 16$ and $d_{\text{nr}} = 17$
Model vocabulary	81 app tokens, a padding token (PAD), and an unknown token (UNK); total size 83
Evaluation population	279 users and 39,901 test launches per policy and capacity

The offline replay isolates the quality of app replacement from behavior inside Android. We compare five policies. Each policy serves a distinct role. LRU provides the recency baseline. We adapted the Static Re-reference Interval Prediction (SRRIP) policy proposed in [23] for app cache slots to assess whether a simple score policy could compete with our learning model. We denote this variant App-SRRIP. It assigns each resident app a 2-bit re-reference prediction value. The direct predecessor of RankProp [11] is denoted as Calibrated-RD. RankProp's ranking model adds a learned residual that represents the current resident set and is denoted as RankProp-ranker. Belady-MIN [24] uses the future launch oracle to define the offline upper bound. It evicts the resident app whose next access occurs farthest in the remaining trace. The five policies distinguish recency, a simple scoring rule, predicted relaunch distance, resident set correction, and the ground truth, respectively. We sweep a fully associative logical cache from $C = 5$ to $C = 15$. This range represents constrained through moderately provisioned resident sets and reveals how replacement value changes as capacity increases. Each resident app occupies one logical slot so that the replay isolates ranking quality from heterogeneous physical footprints. At each capacity, the primary outcome is the hit ratio computed for each user and then averaged across users. The aggregated hit ratio is the arithmetic mean of the values for every launch event over users and capacities. Belady-LRU gap recovery measures how much of the hit ratio gap above LRU an online policy recovers. Kendall's τ_b measures agreement between the complete resident order and the order induced by future access distance. We also report the Belady-optimal decision ratio (BODR), following the use of per-eviction decision ratios in learned cache replacement [25]. BODR credits a decision when the selected victim belongs to the set of residents tied for the farthest next access. We pool Kendall's τ_b and BODR over all full cache eviction decisions across users and capacities. Together, the four metrics measure cache outcome, recovered headroom, the complete resident order, and the chosen victim. Appendix A gives the full policy and measuring procedure. The trace replay is deterministic for a fixed split, and the primary emulator configurations were repeated three times.

To evaluate how the proposed ordering mechanism operates within Android's app execution and memory management paths, we replay app launches on an emulator that runs a modified Android Open Source Project (AOSP) system image under sustained memory pressure. This replay measures how the order in which Android considers processes for reclaim, together with the hints supplied to page reclaim, affects app launches. We compare four policies, denoted as Android-LRU, Calibrated-RD-P, RankProp-P, and RankProp-Full. Android-LRU preserves the platform's recency order. Calibrated-RD-P sends the calibrated order to process reclaim. RankProp-P sends the RankProp order to the same path. RankProp-Full additionally publishes the UID hint consumed by MGLRU. The contrast between RankProp-P and RankProp-Full measures the added effect of the app order hint to the operating system's paging mechanism. The emulator also replays the test splits from the 279 eligible user traces. The guest runs AOSP 16.0.0_r4 on an ARM64 virtual device with four virtual CPU cores and 3 GiB of RAM. A background allocator holds 1024 MiB throughout replay to sustain memory pressure. The LSApp trace provides app identities but does not provide real package names. A deterministic mapper therefore assigns the 87 identities to synthetic apps before any policy run begins. The mapper uses launch frequency and user coverage to represent prevalence. It uses transition entropy to represent switching diversity. The resulting catalog contains 34 small packages, 32 medium packages, 16 large packages, and 5 huge packages. Each package allocates retained Java objects and native buffers. It creates bitmap storage and mapped files, performs small SQLite and file operations, and then touches its assigned pages during launch and resume. We keep the software image and launch sequence fixed across policies. Appendix A.4 reports the complete platform, workload, and metric definitions.

5.2. Model Quality Evaluation

The LSApp trace replay provides each model's eviction-level performance. Figure 2 reports the hit ratio $HR(C)$ and Belady-LRU gap recovery $BGR(C)$ at each cache capacity. RankProp-ranker shows the highest hit ratio among the policies at 0.925169, except for Belady-MIN. LRU, App-SRRIP, and Calibrated-RD report 0.913322, 0.915880, and 0.917168, respectively. RankProp-ranker achieves higher hit ratio than LRU by 1.1847%p. It exceeds App-SRRIP by 0.9289%p and exceeds Calibrated-RD by 0.8001%p. RankProp-ranker also produces 5200 fewer cache evictions than LRU, 4077 fewer than App-SRRIP, and 3512 fewer than Calibrated-RD. Belady-MIN reaches 0.949630, which remains 2.4461%p above RankProp-ranker. The result therefore recorded a higher measured value over all three online baselines. RankProp-ranker exceeds LRU at every capacity, with a margin of 2.10%p at $C = 5$ and 0.12%p at $C = 15$. RankProp-ranker also exceeds Calibrated-RD throughout the sweep. App-SRRIP is marginally higher at $C = 13$ and $C = 14$. RankProp recovers between 15.07%p and 29.33%p of the Belady-LRU gap at every capacity. The recovery peaks at $C = 7$, and its arithmetic mean across the sweep is 21.40%. These results show that RankProp provides a broad capacity-level benefit, with the largest recovered headroom under higher memory pressure.

Hit ratio records the cache outcome but does not describe the complete resident order. Table 2 provides a complementary aggregate view by pooling test requests for hit ratio and full-cache eviction decisions for Kendall's τ_b and the Belady-optimal decision ratio. Decision-weighted Kendall's τ_b provides this second view by comparing the policy order with future access distance. RankProp-ranker records a Kendall value of 0.385347. LRU, App-SRRIP, and Calibrated-RD record 0.310476, 0.284722, and 0.348042, respectively. RankProp-ranker outperforms LRU by 0.074871, App-SRRIP by 0.100625, and Calibrated-

RD by 0.037305. RankProp-ranker therefore produces the strongest complete resident order among the online policies in the aggregate replay.

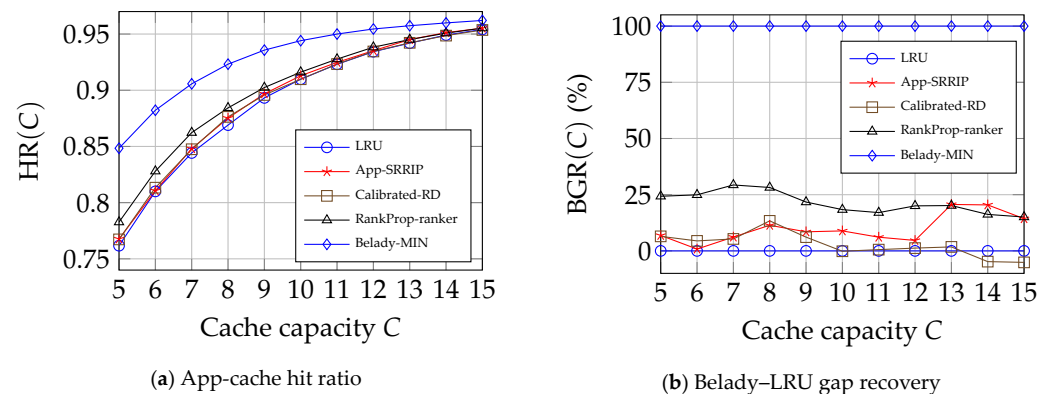


Figure 2. Trace replay results across cache capacities. The hit ratio is computed for each user and then macro-averaged across the 279 users. Belady-LRU gap recovery is computed from the unrounded macro-average hit ratios and is reported as a percentage.

Table 2. Aggregate trace replay quality pooled over $C = 5$ – 15 and 279 users. Hit ratio is the arithmetic mean over all app launches. Kendall's τ_b and BODR are the arithmetic mean over full cache eviction decisions. Evictions is the total number of full cache victim selections.

Policy	Hit Ratio	Kendall's τ_b	BODR	Evictions
LRU	0.913322	0.310476	0.400218	37,537
App-SRRIP	0.915880	0.284722	0.418932	36,414
Calibrated-RD	0.917168	0.348042	0.435716	35,849
RankProp-ranker	0.925169	0.385347	0.514395	32,337
Belady-MIN	0.949630	1.000000	1.000000	21,601

Kendall's τ_b evaluates the complete order, whereas BODR evaluates the selected victim. BODR credits a decision when the selected app belongs to the set tied for the farthest next access. RankProp-ranker records a decision-weighted BODR of 0.514395. Calibrated-RD, App-SRRIP, and LRU record 0.435716, 0.418932, and 0.400218, respectively. RankProp-ranker shows higher BODR than LRU by 0.114177. It also outperforms App-SRRIP by 0.095463 and Calibrated-RD by 0.078679. More than half of RankProp-ranker's full-cache evictions therefore select a member of the Belady-optimal victim set. Together, the aggregate hit ratio, Kendall's τ_b , and BODR show that RankProp-ranker improves app-level cache behavior, complete-order quality, and victim quality.

5.3. Performance Evaluation in the Android Emulator

ActivityManager (AM) total time measures completed launch latency. Proportional set size (PSS) estimates the process's proportional physical footprint, whereas proportional swapped set size (SwapPSS) estimates its proportional footprint in swap. The P95 column reports the 95th percentile of AM total time.

The Android evaluation asks whether the learned prediction of app victim order improves launch behavior when Android applies it to the process and page reclaim paths. Table 3 compares the complete configurations.

Table 3. Android launch and memory outcomes. RankProp-Full uses $Q_{\max} = 1024$.

Configuration	Mean AM (ms)	P95 AM (ms)	Mean PSS (MiB)	Mean SwapPSS (MiB)
Android-LRU	327.24	829.03	266.49	87.30
Calibrated-RD-P	316.81	775.67	253.07	95.06
RankProp-P	315.84	778.33	254.36	94.21
RankProp-Full	311.84	727.72	251.61	97.14

RankProp-Full records the lowest mean AM total time and the lowest P95 AM total time. Relative to Android-LRU, it reduces the mean from 327.24 ms to 311.84 ms. This difference is 15.40 ms, or 4.71%. RankProp-Full also reduces P95 from 829 ms to 728 ms, which is a difference of 101 ms. RankProp-Full lowers the mean by 4.97 ms relative to Calibrated-RD-P and lowers the mean by 4.00 ms relative to RankProp-P. RankProp-Full also records the lowest mean PSS, at 251.61 MiB. This value is 14.88 MiB below Android-LRU, 1.46 MiB below Calibrated-RD-P, and 2.75 MiB below RankProp-P. Mean SwapPSS moves in the opposite direction and reaches 97.14 MiB under RankProp-Full. The primary comparison therefore combines lower launch latency and a smaller proportional physical footprint with a larger proportional swapped footprint.

The second experiment varies $Q_{\max}$, which bounds the Q10 scan boost that RankProp-Full supplies to MGLRU. The RankProp policy and the remaining emulator settings stay fixed. The evaluated caps are 256, 512, 768, and 1024. As shown in Table 4, increasing $Q_{\max}$ from 256 to 1024 reduces mean AM total time from 315.58 ms to 311.84 ms. P95 is not monotonic across the intermediate caps, but it reaches its minimum of 728 ms at 1024. Mean PSS reaches its minimum of 250.29 MiB at 768 and rises by 1.32 MiB at 1024. Mean SwapPSS increases from 94.05 MiB at 256 to 97.14 MiB at 1024. We use $Q_{\max} = 1024$ in the primary comparison because it yields the lowest mean and P95 launch time in this sweep, while accepting the observed memory-placement tradeoff.

Table 4. Effect of the maximum Q10 scan cap in RankProp-Full.

$Q_{\max}$	Mean AM (ms)	P95 AM (ms)	Mean PSS (MiB)	Mean SwapPSS (MiB)
256	315.58	781.03	254.07	94.05
512	312.91	759.06	253.01	96.16
768	311.91	777.23	250.29	96.79
1024	311.84	727.72	251.61	97.14

5.4. Inference Overhead

The RankProp-ranker uses a $42 \rightarrow 32 \rightarrow 16 \rightarrow 1$ multilayer perceptron (MLP). The network contains 1921 trainable parameters, including the bias terms. Its three linear layers require $42 \times 32 + 32 \times 16 + 16 \times 1 = 1872$ multiply–accumulate operations (MACs) for one resident candidate. The computation for a ranking update with N candidates is therefore

$$\text{MAC}_{\text{RankProp}}(N) = 1872N. \quad (11)$$

The count includes the products in the three linear layers. Bias additions and Gaussian error linear unit (GELU) evaluations are excluded. Dropout is disabled during inference. The measured resident sets contained 8 to 10 candidates, which corresponds to 14,976 to 18,720 MACs per update. The mean candidate count was 9.53. At the observed mean, Equation (11) gives 17,840.16 MACs per update. The operation count therefore grows linearly with the resident candidate count.

Table 5 reports the overhead latency. The RankProp ranker required 24.27 μs on average and 26.10 μs at P95. Candidate preparation and calibrated prediction increased the native batch mean to 299.90 μs . The complete JNI call required 340.48 μs on average and

818.25 μ s at P95. Its mean remained below 0.35 ms. RankProp executes this work on a `HandlerThread` that runs at background priority. The launch callback posts the work to that thread, while `ActivityManager` and `OOMAdj` consume the most recently committed order. Model inference therefore does not block the synchronous app launch path.

Table 5. Inference latency from 100 ranking updates during trace replay. The resident sets contained 8–10 candidates, with a mean of 9.53. The native batch includes the calibrated relaunch predictor and candidate preparation.

Boundary	Mean (μ s)	P50 (μ s)	P95 (μ s)	P99 (μ s)
RankProp ranker	24.27	16.00	26.10	154.24
Native batch	299.90	148.50	798.35	1476.65
Complete JNI call	340.48	210.00	818.25	1494.64

5.5. Discussion

Taken together, the two evaluation stages connect the quality of the resident order with the behavior observed after Android consumes that order. In the trace replay, RankProp-ranker records the highest pooled hit ratio, Kendall's τ_b , and BODR among the online policies. The capacity sweep also shows that RankProp outperforms LRU and Calibrated-RD throughout $C = 5$ –15. The emulator configurations then clarify where the runtime improvement appears. The transition from Android-LRU to Calibrated-RD-P provides most of the reduction in mean launch time. RankProp-P lowers the mean from 316.81 ms to 315.84 ms, while its P95 rises from 776 ms to 779 ms. RankProp-Full lowers the mean to 311.84 ms and P95 to 727.72 ms. The clearest additional runtime benefit therefore appears when Android applies the RankProp order to both process reclaim and page reclaim.

The $Q_{\max}$ sweep shows that the strongest launch result does not minimize every memory measure. A cap of 1024 gives the lowest mean and P95 AM time, whereas a cap of 768 gives the lowest mean PSS. Mean SwapPSS increases as the cap grows. This pattern indicates a tradeoff in memory placement. The ranker required 24.27 μ s on average, and the complete JNI call required 340.48 μ s on average. The inference work runs asynchronously and does not block the launch path. Because each complete sweep on the emulator required approximately 20 h, we limited the evaluation to the selected AOSP emulator configuration, a catalog of 87 synthetic packages, and a sustained memory-pressure workload holding 1024 MiB. Physical devices and other workloads may therefore produce different performance gains. The replay evaluation of model quality provides indirect results about performance under memory pressure, while direct validation on physical devices and across a broader range of workloads remains future work.

6. Related Work

The closest studies to RankProp predict future app use. FALCON and Practical Prediction infer likely future launches from context and launch history [7,8]. Predicting the Next App and AppUsage2Vec model the next launch from usage sequences or learned app representations [9,10]. These methods mainly support prediction or prefetching for the next launch. M2ARP changes the target from the next app to the relaunch distance of the app that has just launched [2]. It uses a long short-term memory (LSTM) regressor to estimate that distance. At eviction time, M2ARP removes the resident app with the largest current estimate. When a usable estimate is unavailable, the policy reverts to LRU. A calibrated relaunch predictor maps the learned estimate and the recency fallback onto one comparable runtime scale [11]. This calibration makes the resident candidates comparable. However, each score remains independent of the other apps that are resident when

reclaim occurs. RankProp learns a bounded eviction order from the composition of the current resident set by using the calibrated relaunch score.

Mobile memory systems complement prediction by changing a particular stage of reclaim. SmartLMK selects a process by considering its value and the amount of memory that termination can release [1]. Application-Aware Swapping chooses which anonymous pages should enter swap [26]. ASAP, an adaptive prepagging system, prepares pages that a returning app is expected to access [4]. SWAM coordinates swap capacity and cleanup with process termination [3]. MemSaver moves pages from background apps before memory scarcity forces termination [27]. A study on swap allocation assigns resources to individual apps and manages movement between zRAM and flash [28]. Parallel Memory Reclaim (PMR) reduces kernel reclaim latency by shrinking pages in parallel and issuing larger write requests [12]. Ice freezes a background process when its default activity would interfere with the foreground app [13]. Each system improves one decision or execution stage. None constructs app eviction rank from its launch context over the current resident set and propagates through the operating system memory management path.

Research on compressed memory and the Android Runtime (ART) moves the decision to pages, compression regions, or managed objects. Archer groups pages that tend to recur together into compression regions [14]. It adjusts the size of those regions at runtime. Ariadne classifies anonymous data by hotness [15]. It selects a compression size for each class. It also decompresses data that is likely to be accessed when an app returns. oaRAM preserves object information in compressed memory so that garbage collection can interpret compressed objects directly [29]. Silk records the hotness of managed objects during execution [16]. It groups objects that exhibit similar hotness. This placement guides kernel reclaim toward regions that contain colder objects. These methods improve what happens after memory has been assigned to an app. RankProp acts earlier by determining the relative protection of each resident app. The scopes are complementary. However, these methods do not supply the app order that RankProp propagates.

Learning has also been applied to replacement and migration outside Android app management. LeCaR adjusts the influence of recency and frequency from observed cache outcomes [30]. Learning Relaxed Belady predicts a relaxed future reuse distance for objects in a content distribution network [25]. Hawkeye and Parrot learn replacement decisions for hardware cache blocks from labels derived from Belady's policy [20,31]. 3L-Cache uses online gradient boosting to predict reuse for individual cache objects [32]. Its sampling and update procedures reduce the cost of learning during execution. ArtMem uses reinforcement learning to adapt page migration across memory tiers as the workload changes [33]. Its agent revises the migration policy from reward feedback. These methods act on cache objects, hardware blocks, or pages in memory tiers. RankProp instead orders resident Android apps. It performs no online exploration or reward update. It learns from offline labels. During deployment, separate Android consumers translate one relative order into an eviction order for lmkd and a bounded hint for MGLRU.

Table 6 compares representative methods by capability. The matrix identifies the decision signal and unit used by each method. It also shows whether the method has the capability of ordering the current resident set and where the result changes system behavior. A dash marks a capability outside the stated scope of a method.

App usage prediction methods produce an independent score or one eviction choice. Methods that manage compression, ART, or memory tiers act after the system assigns memory to an app. Neither group provides one relative order over the current resident set to both Android reclaim paths. RankProp addresses this gap by learning that order from the relaunch score and the composition of the current resident set. RankProp can therefore coexist with mechanisms that optimize compression or swap. It is architecturally comple-

mentary and could be evaluated together with systems that improve garbage collection or page migration.

Table 6. Capability comparison of representative methods for app prediction and memory management.

Method and Signal	Decision Unit	Orders Resident Apps	Process Path	Memory Management Path
M2ARP [2]	Cached app	No	App eviction	–
LSTM relaunch distance	Cached app	No	Priority score	–
Calibrated-RD [11]	Cached app	No	Priority score	–
Distance and recency	Compression chunk	–	–	zRAM
Ariadne [15]	Managed object	–	–	ART and compression
Hotness and locality	Object or region	–	–	Kernel reclaim
oaRAM [29]	Page in a memory tier	–	–	Page migration
Object metadata	Resident app set	Yes	lmkd	MGLRU
Silk [16]				
Object hotness				
ArtMem [33]				
Reward feedback				
RankProp				
Calibrated score and list-wise MLP				

7. Conclusions

RankProp converts an app's relaunch score into an eviction order over the current resident set and exposes priorities derived from that order to Android memory management. This conversion employs the calibrated relaunch predictor, while a residual ranker corrects each score by considering the current resident set context. The framework applies the order and translates it into cached process priorities and a bounded scan hint without replacing the native decision unit of the reclaim path. Across logical capacities from five to fifteen resident apps, RankProp recorded an aggregate hit ratio of 0.9252, compared with 0.9172 for the calibrated relaunch prediction. RankProp also increased Kendall's τ_b from 0.3480 to 0.3853 and the Belady optimal decision ratio from 0.4357 to 0.5144 relative to Calibrated-RD. These measures show that the residual ranker improved the resident ordering and the victim selection, which provides a direct explanation for the improvement in cache hit ratio. Applying the RankProp order only to process reclaim reduced the mean launch time by 0.97 ms relative to the corresponding calibrated relaunch prediction. When the same order was also applied to page reclaim, the mean decreased by a further 4.00 ms. The clearest additional runtime benefit therefore appeared when process and page reclaim paths consumed the learned order. The accompanying decrease in physical footprint and increase in swapped footprint indicate a change in the composition of the pages. Taken together, these results support the conclusion that resident app order can inform both process reclaim and page reclaim for intelligent memory management.

This study is limited to replay of LSApp traces and synthetic app in an Android emulator under sustained memory pressure, so it does not establish effectiveness on physical devices or across Android and kernel configurations. Future work should evaluate real apps on physical devices and additional pressure settings. That validation should use larger and more recent usage traces to determine whether the ranking remains effective across users, app repertoires, and changes in behavior. Energy and thermal measurements should determine whether the latency benefit justifies the cost of inference and the additional page activity. After these deployment properties are characterized, an online adaptation mech-

anism should update the ranking as user behavior changes while preserving the existing fallback behavior.

Author Contributions: Conceptualization, J.L. and Y.K.; methodology, J.L. and Y.K.; experimental design, J.L. and Y.K.; data curation, J.L. and Y.K.; writing—original draft preparation, J.L.; writing—review and editing, Y.K.; supervision, Y.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported in part by Seoul National University of Science and Technology and in part by the research grant of Kongju National University in 2023.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data analyzed in this study were derived from the publicly available LSApp dataset, available at GitHub: <https://github.com/aliannejadi/LSApp> (accessed on 1 May 2026).

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

Appendix A. Full Experimental Setup

Appendix A.1. Trace Preparation and Ground-Truth Labels

The preprocessing pipeline groups records by user and orders each group by launch time. It emits a launch only when the foreground app differs from the app in the preceding emitted row. The pipeline divides each resulting sequence into chronological training, validation, and test split before generating labels. For a launch at position j , the label generator locates the next position $k > j$ with the same app identity. It assigns the number of distinct identities between j and k as the relaunch distance. If the app does not appear again in the same partition, the generator assigns the non-return label. The target representation clips finite distances at 16 and encodes non-return as 17. This bounded representation compresses the long tail of finite relaunch distance and prevents rare large values from dominating the regression loss. Future launches are used only to construct offline labels and the Belady–MIN evaluation bound; they are not available to the deployed predictor or residual ranker. Table A1 reports the dataset preparation and target semantics.

Table A1. Dataset preparation and target semantics.

Item	Definition
Input fields	User identifier, launch timestamp, and app identity
Launch event	Foreground record whose app differs from the preceding emitted record
Sequence split	Per-user chronological training, validation, and test split
Finite target	Number of distinct identities before the next launch of the current app
Non-return target	No later launch of the current app within the same partition
Target representation	Finite distance clipped at 16; non-return encoded as 17
Vocabulary	81 app tokens, a padding token (PAD), and an unknown token (UNK)
Unknown identity	Map identities without a training token to UNK in validation and test
All-user test profile	39,922 launches from all 291 processed users
Evaluation workload	39,901 test launches from 279 eligible users

Appendix A.2. Offline Replay and Statistical Analysis

Each policy replays the training and validation split before test metrics are accumulated. The cache is fully associative, and each distinct resident app consumes one logi-

cal slot. For user u , policy M , and capacity C , the per-user hit ratio is $H_{u,M}(C)/R_u$. The capacity-specific macro average is

$$\text{HR}_M(C) = \frac{1}{|\mathcal{U}|} \sum_{u \in \mathcal{U}} \frac{H_{u,M}(C)}{R_u}, \quad (\text{A1})$$

where $H_{u,M}(C)$ is the number of test hits and R_u is the number of test requests for user u . Figure 2 uses this macro average. The aggregate hit ratio in Table 2 is the arithmetic mean over users and capacities. Across all capacities, define:

$$\text{HR}_M^{\text{pool}} = \frac{\sum_{C=5}^{15} \sum_{u \in \mathcal{U}} H_{u,M}(C)}{\sum_{C=5}^{15} \sum_{u \in \mathcal{U}} R_u}. \quad (\text{A2})$$

Belady-LRU gap recovery is computed from the unrounded macro-average hit ratios:

$$\text{BGR}_M(C) = 100 \frac{\text{HR}_M(C) - \text{HR}_{\text{LRU}}(C)}{\text{HR}_{\text{Belady-MIN}}(C) - \text{HR}_{\text{LRU}}(C)}. \quad (\text{A3})$$

At a miss that occurs when the cache is full, let $\mathcal{S}_d$ be the resident candidates and let $\Delta_d(i)$ be the number of future launch positions before candidate i appears again. We set $\Delta_d(i) = \infty$ when i does not return in the remaining trace. Kendall's τ_b compares the policy priority order with the weak order induced by Δ_d and retains ties. The set of victims that are optimal under Belady's rule is

$$\mathcal{V}_d^* = \arg \max_{i \in \mathcal{S}_d} \Delta_d(i). \quad (\text{A4})$$

For the policy-specific decision set $\mathcal{D}_M(C)$, the Belady-optimal decision ratio is

$$\text{BODR}_M(C) = \frac{1}{|\mathcal{D}_M(C)|} \sum_{d \in \mathcal{D}_M(C)} \mathbf{1}[v_d^M \in \mathcal{V}_d^*], \quad (\text{A5})$$

where v_d^M is the victim selected by policy M . Each Kendall's τ_b or BODR is weighted by the number of eviction decisions. The aggregate table pools these decision counts over all users and capacities. The eviction column reports the corresponding total number of full cache victim selections.

Table A2 describes the trace replay and aggregation configuration.

Table A2. Trace replay and aggregation configuration.

Item	Configuration
App-SRRIP	Two-bit re-reference prediction value; maximum 3; insertion 2; hit value 0; saturating aging; LRU victim tie-break
Warmup replay	Training and validation split
Test split replay	39,901 test launches from 279 eligible users
Capacity sweep	$C = 5, 6, \dots, 15$
Cache model	Fixed capacity, fully associative, one logical slot per resident app
Capacity hit ratio	Per-user hit ratio macro-averaged across 279 users
Pooled hit ratio	Hit and request counts pooled over users and capacities
Gap recovery	Percentage of the Belady-LRU gap recovered from unrounded macro-average hit ratios
Order outcome	Kendall's τ_b against future access distance; ties remain
Victim outcome	BODR; every victim tied for the farthest next access receives credit
Decision aggregation	Weighted by the number of full-cache eviction decisions
Evaluation software	Python 3.12 (Python Software Foundation, Wilmington, DE, USA), with NumPy 2.4.6, pandas 3.0.3, and PyTorch 2.12 (Meta Platforms, Inc., Menlo Park, CA, USA).

Appendix A.3. Model Configuration

The calibrated relaunch predictor contains 5722 trainable parameters. The RankProp residual ranker adds 1921 parameters. The complete model therefore contains 7643 shared parameters. Dynamic INT8 quantization is applied to linear layers after FP32 training. The other operations remain in floating point. For personalization, we select the 146 users whose event counts are at or above the median among the 291 users after preprocessing. The personalization learns three FP32 scalars for each selected user. The remaining users have zero-valued personalization terms. Table A3 reports the model, training, and deployment configuration.

Table A3. Model, training, and deployment configuration.

Component	Configuration
Base input	Vocabulary 83; context length 32; app embedding dimension 8; three numeric inputs projected to dimension 8
Base encoder	Model dimension 16; two Transformer blocks; four attention heads; feedforward network dimension 32; dropout 0.2
Calibration rule	Distance clip 16; non-return value 17; at least five prior launches for prediction eligibility
RankProp input	Nine candidate features; four pooled statistics over seven features; five global features; total dimension 42
Residual ranker	Multilayer perceptron $42 \rightarrow 32 \rightarrow 16 \rightarrow 1$; Gaussian error linear unit activation; dropout 0.1; residual scale 0.25
Training objective	Listwise cross entropy with temperature 0.1 and residual penalty 0.01
Ranker optimization	AdamW; batch size 4096; learning rate 2×10^{-3} ; weight decay 10^{-4} ; maximum 30 epochs; early stopping patience 4
Personalization	Three FP32 scalars for each of 146 users; AdamW; learning rate 0.005; maximum 5 epochs; L2 coefficient 0.01
Deployment	Dynamic INT8 for linear layers; remaining operations and personalization in FP32

Appendix A.4. Android Emulator and Synthetic Workload

Table A4 summarizes the platform and workload settings held constant across the Android emulator experiments.

Table A4. Android emulator and runtime configuration.

Item	Configuration
Host	macOS Tahoe 26.4.1; Apple M2 Max with 12 CPU cores; 96 GiB RAM
AOSP and target	AOSP 16.0.0_r4; sdk_phone64_arm64-userdebug
Guest	ARM64 virtual device; four virtual CPU cores; 3 GiB RAM
Pressure workload	Background allocator holding 1024 MiB throughout the primary sweep
Replay workload	39,901 test launches from 279 eligible users
Repeated sweep	Three repetitions per configuration in the primary comparison
Native candidate batch	Maximum 64 resident candidates
Cached oom_score_adj range	900–999
Score tie handling	Preserve the existing LRU order
Primary RankProp-Full cap	$Q_{\max} = 1024$ in Q10 units

The LSApp trace provides app identities but no installable packages or measured memory footprints. The mapper uses launch frequency to represent how often an identity appears. It uses user coverage to represent how broadly the identity appears across users. It uses transition entropy to represent the diversity of app switches. The deterministic catalog contains 34 small packages, 32 medium packages, 16 large packages, and 5 huge packages. The mapping remains fixed across configurations and repetitions. Each pack-

age keeps Java objects and native buffers. It creates bitmap storage together with clean and dirty file mappings. It performs SQLite and small file operations during launch. It touches selected pages during launch and resume so that the assigned footprint appears in the measured resident state.

Table A5 defines the Android metric scope and presentation.

Table A5. Android metric scope and presentation.

Metric	Scope	Presentation
Mean AM total	ActivityManager total time for completed launches	Arithmetic mean in milliseconds
P95 AM total	Upper tail of ActivityManager total time	95th percentile in milliseconds
PSS	Post-launch proportional physical footprint	Mean MiB
SwapPSS	Post-launch proportional swapped footprint	Mean in MiB

Appendix A.5. Synthetic App Profile

The LSApp trace provides app identities but does not provide installable packages or process footprints. We therefore map its 87 identities to independently installable synthetic apps. The mapping uses behavior from the training and validation partitions together with footprint quotas, app category, and popularity. Table A6 summarizes the four stored classes. The generated catalog spans target cached PSS values from 45 to 802 MiB, with a mean of 170.931 MiB. Its launch hot set spans 22 to 626 MiB, with a mean of 113.138 MiB. Its configured CPU work spans 44 to 306 ms, with a mean of 106.908 ms.

Table A6. Synthetic app classes for the 3 GB emulator profile.

Class	Packages	Target PSS (MiB)	Launch Hot Set (MiB)	CPU Work (ms)	First Draw (ms)
Small	34	40–85	18–51	30–90	500–650
Medium	32	90–190	50–133	55–145	600–800
Large	16	210–430	137–344	100–240	700–950
Huge	5	500–850	325–680	180–380	850–1000

All four classes execute the same activity implementation. The class changes allocation sizes, hot sets, CPU work, and runtime bounds. Each package allocates Java byte arrays and a connected object graph. It also allocates direct byte buffers and Android bitmaps. The package creates a clean mapping of a file and a dirty mapping of a writable file. SQLite operations and small file operations add bounded I/O work. The target cached PSS describes the intended process footprint rather than the size of one allocation or a guarantee of residency.

On a cold launch, the package materializes the configured Java, native, bitmap, and mapping components. It then performs the configured I/O and CPU work while touching pages from the launch hot set. When measured PSS remains below 85% of the target, it may add dirty mapped memory subject to a 256 MiB limit and the configured runtime bound. First draw waits until the launch worker completes. When the activity pauses, the package keeps its object and mapping references, and memory trim callbacks do not release them by default. A surviving process touches the configured resume hot set before completing a later resume. If Android terminates the process, the next launch materializes the complete profile again. Pages backed by clean files may be dropped, pages that are anonymous or dirty may be swapped, and `lmkd` may terminate the process.

References

- Kim, S.H.; Jeong, J.; Kim, J.S.; Maeng, S. SmartLMK: A Memory Reclamation Scheme for Improving User-Perceived App Launch Time. *ACM Trans. Embed. Comput. Syst.* **2016**, *15*, 47. [[CrossRef](#)] [[PubMed](#)]

2. Jaehwan Lee, S.P. An Efficient Memory Management for Mobile Operating Systems Based on Prediction of Relaunch Distance. *Comput. Syst. Sci. Eng.* **2023**, *47*, 171–186. [CrossRef]
3. Lim, G.; Kang, D.; Ham, M.; Eom, Y.I. SWAM: Revisiting Swap and OOMK for Improving Application Responsiveness on Mobile Devices. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking, Madrid, Spain, 2–6 October 2023*; ACM: New York, NY, USA, 2023; pp. 1–15. [CrossRef]
4. Son, S.; Lee, S.Y.; Jin, Y.; Bae, J.; Jeong, J.; Ham, T.J.; Lee, J.W.; Yoon, H. ASAP: Fast Mobile Application Switch via Adaptive Prepaging. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC 21)*; USENIX Association: San Francisco, CA, USA, 2021; pp. 365–380.
5. Linux Kernel Development Community. Multi-Gen LRU; The Linux Kernel Documentation. 2022. Available online: https://docs.kernel.org/admin-guide/mm/multigen_lru.html (accessed on 14 April 2026).
6. Android Open Source Project. Low Memory Killer Daemon. Android Open Source Project Documentation. 2026. Available online: <https://source.android.com/docs/core/perf/lmkd> (accessed on 26 May 2026).
7. Yan, T.; Chu, D.; Ganesan, D.; Kansal, A.; Liu, J. Fast app launching for mobile devices using predictive user context. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services, New York, NY, USA, 25–29 June 2012*; pp. 113–126. [CrossRef]
8. Parate, A.; Böhmer, M.; Chu, D.; Ganesan, D.; Marlin, B.M. Practical prediction and prefetch for faster access to applications on mobile phones. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing, Zurich, Switzerland, 8–12 September 2013*; pp. 275–284. [CrossRef]
9. Baeza-Yates, R.; Jiang, D.; Silvestri, F.; Harrison, B. Predicting The Next App That You Are Going To Use. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining, Shanghai, China, 2–6 February 2015*; pp. 285–294. [CrossRef]
10. Zhao, S.; Luo, Z.; Jiang, Z.; Wang, H.; Xu, F.; Li, S.; Yin, J.; Pan, G. AppUsage2Vec: Modeling Smartphone App Usage for Prediction. In *Proceedings of the 2019 IEEE 35th International Conference on Data Engineering (ICDE), Macao, China, 8–11 April 2019*; IEEE: New York, NY, USA, 2019; pp. 1322–1333. [CrossRef]
11. Lee, J.; Kyung, Y. A Calibrated Relaunch Distance Framework for App Eviction in Smartphone Memory Management. *Electronics* **2026**, *15*, 2415. [CrossRef]
12. Li, W.; Chang, L.P.; Mao, Y.; Shi, L. PMR: Fast Application Response via Parallel Memory Reclaim on Mobile Devices. In *Proceedings of the 2025 USENIX Annual Technical Conference (USENIX ATC 25), Boston, MA, USA, 7–9 July 2025*; pp. 1569–1584.
13. Li, C.; Zhu, Z.; Xue, C.J.; Liang, Y.; Ausavarungrun, R.; Shi, L.; Zhou, X. Freezing-based Memory and Process Co-design for User Experience on Resource-limited Mobile Devices. *ACM Trans. Comput. Syst.* **2025**, *43*, 1–29. [CrossRef]
14. Li, C.; Zhu, Z.; Wang, C.; Liu, F.; Xu, F.; Sha, E.H.M.; Zhou, X. Archer: Adaptive Memory Compression with Page-Association-Rule Awareness for High-Speed Response of Mobile Devices. In *Proceedings of the 23rd USENIX Conference on File and Storage Technologies (FAST 25), Santa Clara, CA, USA, 25–27 February 2025*; pp. 497–511.
15. Liang, Y.; Shen, A.; Xue, C.J.; Pan, R.; Mao, H.; Ghiasi, N.M.; Jiang, Q.; Nadig, R.; Li, L.; Ausavarungrun, R.; et al. Ariadne: A Hotness-Aware and Size-Adaptive Compressed Swap Technique for Fast Application Relaunch and Reduced CPU Usage on Mobile Devices. In *Proceedings of the 2025 IEEE International Symposium on High Performance Computer Architecture (HPCA), Las Vegas, NV, USA, 1–5 March 2025*; pp. 1588–1602. [CrossRef]
16. Yuan, Y.; Tan, Z.; Feng, D.; Wei, S.; Gan, J.; Xiao, Y.; Qi, W.; Zhang, J. Silk: Runtime-Guided Memory Management for Reducing Application Running Janks on Mobile Devices. *ACM Trans. Archit. Code Optim.* **2026**, *23*, 12. [CrossRef]
17. Android Open Source Project. Application Sandbox. Android Open Source Project Documentation. 2026. Available online: <https://source.android.com/docs/security/app-sandbox> (accessed on 26 May 2026).
18. Android Open Source Project. Cgroup Abstraction Layer. Android Open Source Project Documentation. 2026. Available online: <https://source.android.com/docs/core/perf/cgroups> (accessed on 26 May 2026).
19. The Linux Kernel Documentation. Control Group v2. Linux Kernel Documentation. 2026. Available online: <https://docs.kernel.org/admin-guide/cgroup-v2.html> (accessed on 26 May 2026).
20. Jain, A.; Lin, C. Back to the future: Leveraging Belady’s algorithm for improved cache replacement. In *Proceedings of the 43rd International Symposium on Computer Architecture, Seoul, Republic of Korea, 18–22 June 2016*; IEEE Press: New York, NY, USA, 2016; pp. 78–89. [CrossRef]
21. Cao, Z.; Qin, T.; Liu, T.Y.; Tsai, M.F.; Li, H. Learning to rank: From pairwise approach to listwise approach. In *Proceedings of the 24th International Conference on Machine Learning, Corvallis, OR, USA, 20–24 June 2007*; pp. 129–136. [CrossRef]
22. Aliannejadi, M.; Zamani, H.; Crestani, F.; Croft, W.B. Context-aware Target Apps Selection and Recommendation for Enhancing Personal Mobile Assistants. *ACM Trans. Inf. Syst.* **2021**, *39*, 29. [CrossRef]
23. Jaleel, A.; Theobald, K.B.; Steely, S.C.; Emer, J. High performance cache replacement using re-reference interval prediction (RRIP). *SIGARCH Comput. Archit. News* **2010**, *38*, 60–71. [CrossRef]
24. Belady, L.A. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.* **1966**, *5*, 78–101. [CrossRef]

25. Song, Z.; Berger, D.S.; Li, K.; Lloyd, W. Learning Relaxed Belady for Content Distribution Network Caching. In Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), Santa Clara, CA, USA, 25–27 February 2020; pp. 529–544.
26. Kim, S.H.; Jeong, J.; Kim, J.S. Application-Aware Swapping for Mobile Systems. *ACM Trans. Embed. Comput. Syst.* **2017**, *16*, 182. [[CrossRef](#)]
27. Challa, P.; Song, B.; Jiang, S. MemSaver: Enabling an All-in-memory Switch Experience for Many Apps in a Smartphone. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering, London, UK, 7–11 May 2024*; ACM: New York, NY, USA, 2024; pp. 267–275. [[CrossRef](#)]
28. Wei, Y.C.; Tsou, Y.C.; Chen, Y.C.; Chang, L.P. App-aware Swap Resource Allocation for Enhancing User-perceived Latency on Mobile Devices. *ACM Trans. Embed. Comput. Syst.* **2025**, *24*, 130. [[CrossRef](#)]
29. Li, X.; Yang, Z.; Wu, M.; Chen, H.; Zang, B. Object-Aware Memory Compression for Smartphones. *ACM Trans. Archit. Code Optim.* **2025**, *22*, 139. [[CrossRef](#)]
30. Vietri, G.; Rodriguez, L.V.; Martinez, W.A.; 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 18), Boston, MA, USA, 9–10 July 2018.
31. Liu, E.Z.; Hashemi, M.; Swersky, K.; Ranganathan, P.; Ahn, J. An imitation learning approach for cache replacement. In *Proceedings of the 37th International Conference on Machine Learning; ICML'20; JMLR.org*: New York, NY, USA, 2020.
32. Zhou, W.; Niu, Z.; Xiong, Y.; Fang, J.; Wang, Q. 3L-Cache: Low Overhead and Precise Learning-based Eviction Policy for Caches. In Proceedings of the 23rd USENIX Conference on File and Storage Technologies (FAST 25), Santa Clara, CA, USA, 25–27 February 2025; pp. 237–254.
33. Yi, X.; Du, H.; Wang, Y.; Zhang, J.; Li, Q.; Xue, C.J. ArtMem: Adaptive Migration in Reinforcement Learning-Enabled Tiered Memory. In Proceedings of the 52nd Annual International Symposium on Computer Architecture, Tokyo, Japan, 21–25 June 2025; pp. 405–418. [[CrossRef](#)]

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.