# AppFlow：面向移动与车载系统中大型应用冷启动的内存调度

Xiaochen Li西北工业大学
xiaochenli@mail.nwpu.edu.cn

Sicong Liu*西北工业大学
scliu@nwpu.edu.cn

Bin Guo西北工业大学
guob@nwpu.edu.cn

Yu Ouyang
中国西北大学
ouyangyu@stumail.nwu.edu.cn

Fengmin Wu
西北工业大学
fenny@mail.nwpu.edu.cn

Yuan Xu
西北工业大学
npuyuan
@mail.nwpu.edu.cn

Zhiwen Yu哈尔滨工程大学，西北工业大学
zhiwenyu@nwpu.edu.cn

arXiv:2603.17259v1 [cs.OS] 18 Mar 2026

## 摘要

像端侧大语言模型和富媒体编辑器这类 GB 级大型应用正成为新一代趋势，但它们对内存和输入/输出有着巨大的需求，尤其是在多任务处理期间，这会导致设备回收内存或终止进程，使热应用变成冷启动。挑战不在于存储这些应用，而在于快速、准确地启动它们。对用户而言，1 秒是可用性的临界点，但我们的测量结果显示，86.6% 的 GB 级冷启动耗时超过 1 秒。此外，Android Vitals 仅将 ≥5s 标记为缓慢，揭示了巨大的满意度差距。现有优化方案彼此独立设计，且相互冲突。例如，预加载可减少输入/输出停顿，但会消耗紧缺的内存，且其效果会被内存回收抵消；而内存回收和终止进程虽能释放内存，却会牺牲后台存活能力，导致反复冷启动。我们的关键洞见是，尽管多任务处理使运行时行为变得复杂，每个应用的文件访问模式仍然可以预测。挑战在于如何利用这种可预测性，即在不耗尽内存的情况下进行预加载，在不抵消优化收益的情况下进行内存回收，并有选择地终止进程以保留后台存活能力。我们

我们提出 AppFlow，这是一种基于预测的系统级调度器，集成了选择性文件预加载器、自适应内存回收器和上下文感知进程终止器。AppFlow 在无需修改应用的情况下，跨 Android 框架和 Linux 内核实现，使 GB 级应用的冷启动延迟降低 66.5%（例如，2s→690ms），并在为期 100 天的测试中确保 95% 的启动在 1 秒内完成，显著提升了响应速度和多任务体验。

## CCS 概念

• **以人为中心的计算** → **普适与移-动计算**; • **计算机系统组织** →**嵌入式系统**；**可用性**；**嵌入式软件**。

## 关键词

GB 级大型应用冷启动，移动内存调度器

**ACM 参考格式：**

Xiaochen Li、Sicong Liu、Bin Guo、Yu Ouyang、Fengmin Wu、Yuan Xu 和 Zhiwen Yu。2026。AppFlow：移动与车载系统中大型应用冷启动的内存调度。载于第 *32* 届年度移动计算与网络国际会议（*MobiCom '26*），*2026*年*10*月*26*日至*30*日，美国德克萨斯州奥斯汀。ACM，美国纽约州纽约市，15 页。
https://doi.org/10.1145/3795866.3796690

*通讯作者：scliu@nwpu.edu.cn

本作品采用知识共享署名 4.0 国际许可协议授权。*MobiCom '26*，美国德克萨斯州奥斯汀© 2026 版权所有者/作者保留所有权利。ACM
ISBN 979-8-4007-2505-0/2026/10
https://doi.org/10.1145/3795866.3796690

## 1 引言

GB 级大型应用已成为移动设备上的新一代趋势，给内存和输入/输出资源带来了巨大压力。如今，热门应用包括由大语言模型驱动的助手 [7], 图像转文本生成器 [45], 剪映等富媒体编辑器 [4], TikTok 等流媒体平台 [42], 以及

PUBG Mobile 等 3D移动游戏 [32], 也都采用 GB 级内存占用。值得注意的是，Google Play 上下载量排名前十的应用中，如今有八款超过 1GB [2, 39]。尽管移动设备资源有限，用户通常仍会同时打开多个任务。例如，*Lucy* 可能会使用 TikTok 和 Red-Note 规划行程，同时让端侧大语言模型在后台保持运行。反复切换会导致应用频繁冷启动，打断她的操作流程。更糟的是，大语言模型瞬时产生的高内存占用可能导致系统终止后台应用，使多任务体验变得支离破碎、令人沮丧。在车载系统中，驾驶员会冷启动 Google 地图等应用进行导航，或启动行人检测应用以获得驾驶辅助；每一秒的延迟都会降低体验，甚至影响行车安全。这种常见的多任务场景使内存和输入/输出资源不堪重负。

在这种压力下启动 GB 级大型应用，操作系统必须回收内存，方法包括交换缓存页面 [28,29]或终止后台进程 [1]。两种方式都会牺牲 后台应用的响应能力：先前的热应用会退化为冷启动。当*Lucy* 切回 Instagram 时，系统必须从存储中重新加载超过 1GB 的资源文件，常常会卡顿 2 秒以上，达到 1 秒可用性临界点 [30] 的两倍；超过这一临界点时，用户会觉得应用冻结或崩溃。

关键在于，这并非个别情况，而是日常难以避免的现象。现代智能手机常驻内存的应用只有 20 个 [14]（车辆等其他设备上甚至更少），而用户平均安装 80+ [3],，因此[3 4] 次启动从设计上就会是冷启动。举例来说，在移动设备上始终运行着 15 个 GB 级应用（例如 TikTok、小红书和 PUBG）时，例如在 Google Pixel 8 上，86.6% 的重新启动会超过 1 秒的可用性临界点。简言之，用户每天不可避免地会多次遭遇这些“应用冻结时刻”（例如，在我们的 100 天测试中每天最多 7 次），破坏多任务处理体验中无缝启动应用的流畅感。

尽管付出了大量努力，在不牺牲后台存活能力的情况下减少冷启动延迟仍然极其困难。以往工作只解决了瓶颈的一方面：I/O 预加载 [6, 33] 能够减少停顿，但会消耗稀缺的随机存取存储器，并且很快因激进的内存回收而失效；而内存回收 [24, 29] 可以释放空间，却往往会驱逐预加载页，从而抵消 I/O 带来的收益。核心难点在于，预加载与内存回收/终止进程是彼此孤立设计的，且两者天然存在冲突。据我们所知，此前尚无工作对二者进行联合优化。为解决这些问题，我们面临两项挑战：

• 挑战 ***#1***：决定何时、预加载什么以及如何预加载，以平衡冷启动延迟与多任务处理，并非易事。GB 级启动涉及数千次小型、分散的读取，导致输入/输出碎片化，使输入/输出吞吐量远低于峰值（见第 2.2.1 节）。以往的方案要么预加载所有文件 [20, 21, 31],，这对于 GB 级应用来说代价过高，因为这会耗尽内存并导致内存回收或进程被终止；要么按需加载 [6, 16, 33], ，虽然可以节省内存，却常常跟不上 CPU 执行的速度，使文件输入/输出成为瓶颈，令 GB 级冷启动更加缓慢。此外，预加载会与内存回收争夺输入/输出带宽，其额外的内存占用也会损害后台存活能力。

•挑战 ***#2***：对于 GB 级冷启动而言，要在不影响后台存活能力或抵消预加载收益的情况下及时回收内存是难以实现的。实际上，冷启动会使内存分配陷入停顿，甚至阻塞预加载；然而，现有解决方案（例如安卓的低内存杀手（LMK） [1] 以及后续研究 [24, 26, 29] ）尝试通过回收内存页 [24, 26] 或终止进程 [1]来解决这一问题。不过，这两种方法都存在不足，因为它们尽管成本并不对称，却仍将文件支持的内存页和匿名内存页一视同仁 [26]。此外，它们还不可避免地以牺牲新应用的响应能力为代价换取后台存活能力，使问题始终未能得到解决。

为解决这些问题，我们提出了 AppFlow，这是一种全系统调度器，可在普通移动设备和车载设备上联合管理预加载、页面回收和进程终止。

• 首先，我们对 GB 级启动进行了分析，发现了结构性的输入/输出失衡：即，尽管小文件仅占总大小的 3%，却会中断大文件传输并打乱顺序读取，从而主导启动延迟（见第 2.2.1 节）。基于这一发现，我们设计了选择性文件预加载器，用于决定何时、预加载哪些内容以及如何预加载。它会在启动前预加载小文件，以尽量减少内存占用并避免启动初期停顿；同时在启动过程中使用大块数据流式读取大文件，以最大化吞吐量。毫秒级背包问题求解器还会为每个应用选择截止阈值（见第 4.1 节）。

•其次，我们将文件支持页与匿名页面回收解耦，以利用两者成本的不对称性：文件支持页可以快速以大块连续内存进行回收，而匿名页则需要缓慢回写（第 2.2.2 节）。基于这一特点，AppFlow 集成了自适应内存回收器，在内存压力下优先回收文件支持页，否则转而回收匿名页，并为预加载数据添加标记以防止其被过早逐出（见第 4.2 节）。此外，它还包含上下文感知应用终止器，只会终止运行时间较长的后台应用（见第 4.3 节）。

我们实现了 AppFlow，覆盖 Android 框架和 Linux 内核层，无需修改应用层。我们在三个平台上进行了广泛评估，包括商用智能手机和车载系统，并使用了 60+ 款热门应用及多种工作负载。与 Android 操作系统和三种最先进的基线方法相比，AppFlow 将 GB 级冷启动延迟最高降低了 66.5%（例如，2s→690ms）。不同于牺牲后台响应能力的既有系统，AppFlow 在为期 100 天的测试中，即使并发度提高 1.29× 倍，仍能确保 95% 的启动在 1 秒内完成。这些提升得益于 2.35× 倍的输入/输出吞吐量

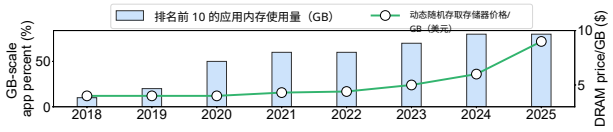



**图 1：过去几年中，Google Play 排名前 10 的 GB 级应用占比显著增长，同时动态随机存取存储器价格也快速上涨。**

以及降低 67.9% 的内存压力（以 Linux 内核直接回收次数衡量）。我们的主要贡献总结如下。

- 据我们所知，这是首个联合优化 GB 级冷启动和多任务存活率的方案。它使 95% 的启动和重新启动延迟低于 1 秒（可用性临界点）。
- 我们设计了 AppFlow，这是一种系统级调度框架，利用可预测的应用访问模式，将选择性文件预加载、自适应内存回收和上下文感知进程终止统一起来。
- 实验表明，与最先进的方案相比，AppFlow 显著降低了 GB 级应用的冷启动延迟，并通过提高 I/O 吞吐量和更高效的内存回收，增强了应用的后台存活能力。

## 2 背景与动机

### 2.1 应用启动流程简介

快速应用启动对用户体验至关重要 [23, 36, 38,40]，尤其是对于 GB 级、内存密集型应用（例如，移动游戏、大语言模型驱动的应用）而言；它们受硬件 I/O 和内存限制影响，冷启动延迟显著。我们注意到，先前的研究 [30]表明，应用启动延迟超过 *1* 秒便会达到**可用性临界点**，使用户怀疑网络或系统发生故障。为了系统地分析应用启动瓶颈，我们首先概述其基本机制、开销，以及底层硬件和操作系统的限制。

*2.1.1 GB* 级大型移动应用的发展趋势。过去十年间，Google Play 商店中的应用体积显著增大 [39, 44, 46]。排名前 10 的热门应用中，有 8 款体积超过 1GB [2, 39]，，排名前 100 的应用中约有 30% 体积超过 1GB。这些 GB 级应用主要分为三 种类型。*i)* 端侧 *AI* 应用，例如大语言模型驱动的助手 [7] 和图像转文本生成器 [45]，，会在本地运行人工智能模型，以实现私密且低延迟的推理，从而增加随机存取存储器的使用量。*ii)* 富媒体应用，例如视频编辑器剪映 [4] 和流媒体平台 TikTok [42]，，会集成高分辨率及多模态输入，从而增加内存占用。*iii)* 手机游戏，例如PUBG Mobile 等图形密集型游戏会加载大量 3D 资源，因此往往会产生较大的内存占用。

*2.1.2 GB* 级大型应用冷启动问题。移动应用通常有三种启动模式：即冷启动、温启动和热启动

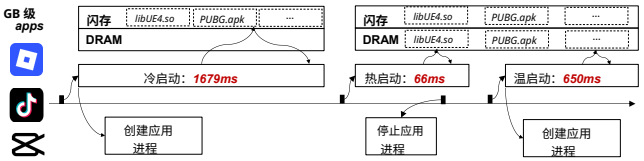



**图 2：GB 级应用启动进度示意图。**

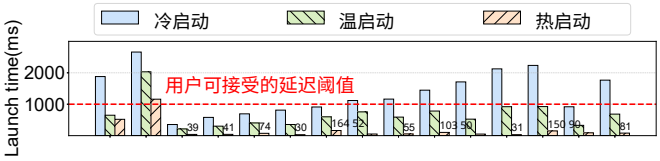



**图 3：15 款热门 GB 级应用在冷、温、热启动模式下的启动延迟示意图。**

启动，每种模式给用户的体验都不同。如图 2 所示，在热启动中，应用已在随机存取存储器中运行，所有数据也已就绪，因此几乎可以瞬间启动。温启动 发生在应用进程已停止但缓存仍保留时；其启动速度比热启动慢，但仍然相当快 [6]。在冷启动中，内存中没有任何内容，因此系统必须将 APK 文件（例如，PUBG.apk）和共享库（例如，libUE4，等等）从闪存加载到动态随机存取存储器中，因此启动速度最慢 [29]。我们在 Pixel 8 智能手机上，针对 15 个 GB 级应用的三种启动模式进行了测试，发现 33% 的冷启动通常耗时超过 1 秒（见图 3）。

造成这种延迟的主要有两个因素。首先，冷启动会产生数十个输入/输出请求，争抢有限的移动设备动态随机存取存储器带宽，从而拖慢每次读取。其次，移动用户通常会同时运行多个 GB 级应用（例如，视频播放器 TikTok [42],或基于大语言模型的助手 Google AI Edge Gallery [7]），直到随机存取存储器占满或电池耗尽。为了给新应用腾出内存，安卓/Linux 会终止后台进程 [1]或回收缓存页面 [11]，，使原本可能热启动的应用变成温启动或冷启动。动态随机存取存储器价格持续上涨 [5, 15, 43]，对内存扩容造成了重大限制（见图 1），因此难以增加可保持活跃状态、以便热启动的应用数量。现有研究 [14] 表明，移动设备可保持约 20 个应用处于活跃状态，但普通用户通常安装约 80 个应用 [3]，，导致 75% 的应用需要冷启动，这凸显了冷启动发生的频繁程度及其影响。

此外，频繁的多任务切换会使冷启动延迟问题更加突出。例如，用户从 TikTok 切换到 Line 再切回来时，只有在这两个应用都留在随机存取存储器中时，才能立即加载；否则，每次加载都将是一次缓慢的冷启动。

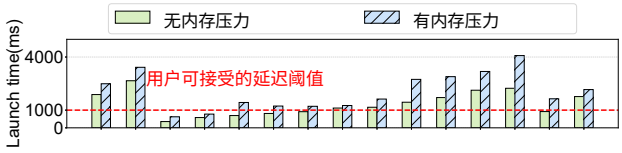



**图 4：内存可用性对应用启动时间的影响：比较有 10 个后台应用（<1GB 空闲内存）与无后台应用（>2GB 空闲内存）时的冷启动延迟。**

一些系统 [14, 19, 28] 采用激进的内存管理策略，以尽可能增加可热启动的应用数量。然而，典型的移动设备 DRAM 一次只能容纳三个 GB 级应用。为过多的大型应用和后台服务保留状态与缓存会超出这一限制。

*2.1.3* 移动内存管理。高效的内存管理对于在资源受限的移动设备上为新应用启动留出内存空间（即动态随机存取存储器），并减少 GB 级应用冷启动延迟至关重要。移动操作系统通常采用两种内存管理策略：*i*）内存回收 [26]（丢弃文件支持页，并交换匿名页以释放 RAM），以及*ii*）进程终止 [29]（终止后台应用）。

具体而言，当空闲 RAM 低于阈值时，内核的内存回收进程（kswapd）会执行两阶段清理：*i*）丢弃文件支持页（代码/资源），这些内容之后可以重新加载 [40]。*ii*）将匿名页交换到闪存，然后释放 RAM [18, 25, 27]。然而，写入闪存的速度很慢。移动设备上有限的输入/输出带宽总会导致内存回收跟不上内存分配，尤其是在内存压力较大时。在我们的测试中（图 4），后台有十个应用时启动一个新应用，耗时最长可达无后台应用时的 2.2× 倍，导致 86.6% 的启动耗时超过 1 秒。如果内存回收和交换因交换空间有限或内存压力过大而无法释放足够的 RAM，操作系统就会终止后台进程，以回收其整个工作集。虽然这能为新应用腾出内存空间，但会导致被终止的运行中应用在下次使用时冷启动，降低用户体验。为了进一步减少冷启动延迟，文件预加载 [31, 33, 47] 是一种正交技术。通过在启动前 [31]或启动期间 [33], 将关键二进制文件和数据加载到内存中，系统可以让输入/输出与其他任务并行执行，从而缩短启动时间。然而，要在保持后台应用处于热状态的同时加快内存回收，以实现快速冷启动，并非易事。

## 2.2 观察

本小节介绍我们的关键观察，揭示了优化机会。

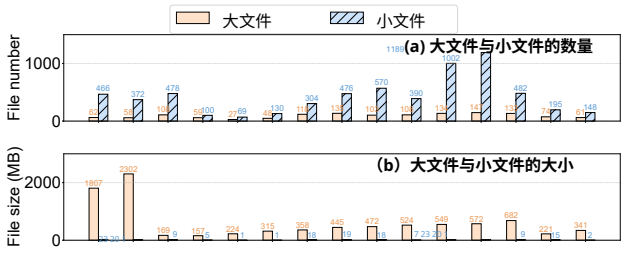



**图 5：在 GB 级大型应用启动期间，小型 ≤128 KB 文件的访问次数是 >128 KB 大文件访问次数的 4.7× 倍，但仅占其内存的 2.2%。**

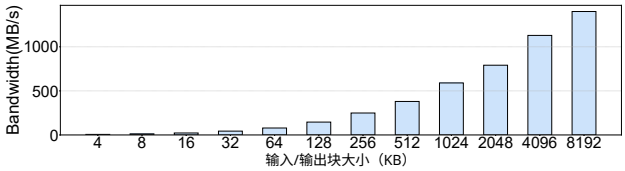



**图 6：I/O 带宽随 I/O 块大小变化。**

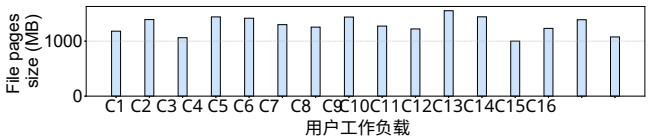
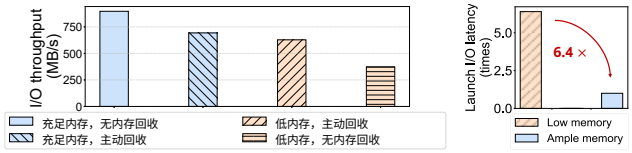



**图 7：16 种代表性用户工作负载 [41] 中多样的文件页大小。例如，C1 是一个订餐场景，通常发生在下午，用户会同时启动 WhatsApp 等社交应用和 TikTok 等媒体应用。**

*2.2.1* 空闲 *I/O* 与热集文件。如上所述，高效预加载能够隐藏冷启动 I/O 延迟。为此，我们有三项发现。首先，通过实证我们发现，在 24 款应用中，97.2% 的冷启动 I/O 等待涉及可预测的静态数据（例如，UE4 [34]引擎资源或大语言模型权重）。尤为重要的是，我们观察到，在应用切换期间，79% 的 DRAM I/O 带宽处于未使用状态。这表明存在适合预加载的空闲窗口。其次，我们还对 15 款 GB 级应用进行了分析，每款应用启动十余次。结果显示，GB 级应用实际上依赖许多小型热集文件，而这些文件的总大小很小。例如，如图 5 所示，TikTok [42] 启动期间会访问 1,002 个小文件，数量是大文件的 7.47× 倍。它们仅占用 23MB 内存，不到大文件所占内存的 4%。第三，增大 I/O 块大小可提升大文件的吞吐量，却会浪费小文件的带宽；减小块大小适合微小文件，却会限制吞吐量并增加延迟。以结构化方式调整 I/O 块大小，可以显著提升 I/O 吞吐量（见图 6）。




（a）预加载 I/O 吞吐量。 （b）I/O 延迟。

**图8：预加载-回收干扰分析。（a）内存回收会降低预加载的吞吐量；（b）回收预加载文件会导致 GB 级应用启动期间的 I/O 延迟增加 6.4× 倍。**

*2.2.2* 页面差异化回收成本。回收成本因页面类型而异。匿名页的回收成本较高，因为它们需要将数据写入闪存（例如，swap_writepage()）；而文件支持页的回收成本较低，因为无需写入即可将其丢弃。现有回收策略在两者之间均匀地交替处理，但这种交错方式迫使系统先等待耗时的写入完成，才能释放易于回收的页面。此外，我们对 16 种典型用户工作负载的分析 [41]（见图 7）显示，可回收的文件支持页通常超过 1000 MB，无论社交应用呈非线性增长，还是媒体应用呈线性增长。

**观察**：不同页面类型需要采用不同的内存回收策略。在高内存压力下，首先丢弃文件支持页可立即释放内存，避免 I/O 延迟。随后可在需求高峰过后处理匿名页面回写，以确保更快的内存回收。

*2.2.3* 时序性杀进程影响。 当仅靠内存回收无法释放足够的随机存取存储器时，操作系统的低内存杀手（LMK）可能会终止后台进程以腾出内存空间。然而，这种简单粗暴的方法会带来问题。首先，杀死用户可能很快重新打开的应用会扰乱工作流。现有用户行为研究表明，92.5%的应用会在30分钟内被再次访问 [41], 但 LMK 仍会杀死其中62%的高概率应用，导致重复 I/O 和代价高昂的冷重启。其次，移动用户往往会在不同时间关注特定类别的应用（例如，上午使用生产力工具，晚上使用媒体应用）。

**观察**：将时间局部性纳入终止决策，可显著减少不必要的终止操作，节省输入/输出，并减轻用户的不适。

*2.2.4* 预加载与回收干扰。 GB 级应用的冷启动延迟源于两个紧密耦合的瓶颈，即，有限的 *I/O* 带宽会减慢关键文件的读取，而较少的空闲内存则会阻碍新的内存分配和预加载。加速冷启动意味着同时加载文件和释放随机存取存储器，但这两个操作往往会相互争夺资源，抵消彼此的收益。其主要原因来自两个方面：

- **I/O 争用**：回收匿名页时，内存回收会增加输入/输出写入流量，与预加载读取产生竞争。
- **页面驱逐**：回收文件支持页时，内存回收可能会驱逐预加载页，迫使应用在启动期间重新读取这些页面。

为量化它们之间的干扰，我们首先测试四种内存条件下的输入/输出吞吐量：*i*）充足内存，无回收；*ii*）充足内存，主动回收；*iii*）低内存，主动回收；以及*iv*）低内存，无回收（图 8a）。情况（iv）的输入/输出吞吐量最差，因为停滞的内存分配导致存储未得到充分利用。即使内存充足，启用内存回收（情况 ii）仍会降低吞吐量，因为匿名页回收产生的写入流量会与预加载读取竞争。接下来我们测量预加载后额外增加的输入/输出延迟，以量化它们在页面驱逐方面的干扰，并分别考察充足内存和低内存条件（图 8b）。在低内存条件下，启动时的输入/输出延迟增加了 6.4×，这表明许多预加载页在回收期间被驱逐，因而必须重新读取。

**观察**：如果缺乏适当的协调，内存回收和预加载可能会相互抵消各自的收益。这一观察表明，仅有可预测的启动时依赖关系还不够。必须协调预加载与内存回收，以减少争用并避免预加载内容被立即驱逐。这促使我们设计一个全系统调度器，(i) 利用空闲 I/O 窗口，(ii) 在内存限制下优先处理热集文件，以及 (iii) 执行内存回收时不抵消预加载带来的收益。

# 3 概述

## 3.1 问题形式化

我们的目标是在确保共享相同输入/输出和内存资源的后台应用能够在 1*s* 内完成重新启动 （已知的响应性阈值为 [30]），即每 $T_i < 1$，从而保持流畅的用户体验的同时，尽可能缩短 GB 级应用的冷启动延迟。冷启动延迟$T_i^{cold}$分解如下：

$$T_i^{cold} = T_i^{I/O} + T_i^{cpu} + T_i^{alloc} \quad (1)$$

其中， $T_i^{I/O}$ 表示从闪存加载数据到动态随机存取存储器所需的时间，这是每个应用启动时间的主要组成部分。 $T_i^{cpu}$ 是应用初始化时间， $T_i^{alloc}$表示内存分配时间，它取决于应用启动时系统的可用内存。由于 $T_i^{cpu}$ 取决于每个应用内部的初始化逻辑，而我们无法在系统层面修改该逻辑，因此我们的目标是减少输入/输出和内存延迟。在移动设备上，总

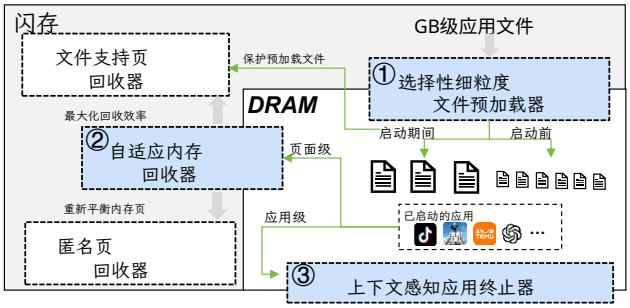



**图 9：AppFlow 的工作流。**

内存使用量可表示为：

$$M = M^{sys} + M^{aval} + \sum_{i} p_i + \sum_{j \in \mathcal{J}} m_j \tag{2}$$

其中，$M^{sys}$ 表示系统使用量（例如，UI 渲染引擎），$M^{aval}$ 表示可用于内存分配的可用 RAM。$p_i$ 表示为待启动应用 $i$ 预加载的内存。J 是后台应用集合，$m_j$ 表示每个应用 $j \in$ J 占用的内存。假设应用 $i$ 在启动期间必须读取 $M_i^{file}$ 字节的文件，并分配 $M_i^{sum}$字节的 RAM。设 $v^{I/O}$ 为 I/O 吞吐量，$v^{rec}$ 为内存回收吞吐量，二者分别取决于 I/O 块大小和可用内存（参见第 2.2 节）。那么，两个关键子延迟为：

$$T_i^{I/O} = \frac{M_i^{file} - p_i}{v^{I/O}} \tag{3}$$

$$T_i^{alloc} = \frac{Max\{(M_i^{sum} - M^{aval}), 0\}}{v^{rec}} \tag{4}$$

这里，$p_i$、$M^{aval}$、$v^{I/O}$和 $v^{rec}$是我们可以调整以提升启动性能的设计变量。我们将可行设计空间 F 定义为：

$$\mathcal{F} = \{(p_i, M^{aval}, v^{I/O}, v^{rec}) \mid \text{C1, C2}\} \tag{5}$$

$$\text{C1:} \quad M^{sys} + M^{aval} + \sum_{i \in \mathcal{I}} p_i + \sum_{j \in \mathcal{J}} m_j \leq M, \tag{6}$$

$$\text{C2:} \quad T_i^{\text{cpu}} + T_i(p_i, M^{aval}) \leq 1s, \ \forall\, i \in app\ set \mathcal{J}. \tag{7}$$

输入/输出与内存分配的总延迟为：

$$T_i(p_i, M^{aval}) = \frac{M_i^{file} - p_i}{v_{I/O}} + \frac{Max\{(M_i^{sum} - M^{aval}), 0\}}{v_{rec}} \tag{8}$$

随着内存受限设备上的应用日益达到 GB 级（见第 2.1.1 节），$C1$ 会严格限制 $p_i$ 和 $M^{aval}$。在如此严格的内存限制下（$C2$），要确保 J 中每个正在运行的应用都满足低于 1 秒的启动时限，这使得对 $M^{aval}$、$v^{I/O}$和 $v^{rec}$进行交互式调优极具挑战性。

## 3.2 解决方案概述

为解决上述棘手问题，我们提出了 AppFlow，这是一个专为移动设备设计的系统级内存调度框架。如图 9 所示，AppFlow 包含三个紧密耦合的组件，它们在不同粒度和优化维度上运行，即预加载、内存回收和进程终止，共同构成一套全面的解决方案。具体而言，选择性文件预加载器（文件级）会主动利用可用的空闲内存和输入/输出，预加载关键文件页，将冷启动延迟最多降低 55.2%（见第 4.1 节）。当空闲内存不足时，自适应内存回收器（页面级）会根据实时 I/O 负载调整回收速率，从后台应用回收干净页面，以维持预加载器的效率；同时，它会标记并跳过刚刚获取的页面，避免无谓的重新加载（见第 4.2 节）。此外，所提出的回收器能够感知预加载器，会有意跳过预加载器刚刚加载的页面，从而避免代价高昂的重新加载。当主动修剪仍不足以解决问题时，AppFlow 会启用上下文感知应用终止器（进程级），其决策不再局限于 Android 的 LRU 列表。它会针对长期运行且内存占用庞大的应用；这类应用重新启动后的占用空间会恢复至基线，因此可避免重复终止，并在长期测试中将冷启动时间缩短 29.8%（见第 6.6 节）。这些组件协同工作，使 AppFlow 即使在内存压力大、并发度高的情况下，也能确保 GB 级大型应用快速启动。正如我们将在第 6 节展示的那样，AppFlow 将三类 GB 级大型应用的冷启动延迟最多降低 66.5%，同时将后台并发度提高 2.6×。此外，在为期 100 天的测试中，AppFlow 确保 95% 的启动在 1 秒内完成，这表明，能够协同调度预加载、页面回收和进程终止的系统级内存调度器，对于提升移动设备的响应能力至关重要。

# 4 系统设计

## 4.1 选择性文件预加载器

在动态随机存取存储器受限的移动设备上，现代 GB 级应用在冷启动期间经常因阻塞式 *I/O* 而停滞，例如中央处理器任务会等待从闪存读取文件并将其载入动态随机存取存储器。正如第 2.2.1 节所述，预加载可以填补这些空档，但为了避免过度占用内存，我们必须仔细决定预加载的时机（时间）、的内容（哪些文件）以及的方式（如何调整 I/O 块大小）。通过调节这三个维度，我们可以让输入/输出与空闲时段重叠。

**现有方法的局限性**。以往的研究难以兼顾启动速度和内存效率。首先，在启动前预加载会提前加载所有必需文件 [13, 31,47],，从而消除 I/O 延迟，但会导致内存使用量过高，这在移动设备上往往难以接受。例如，如图 10a 所示，仅预加载三个热门应用——例如，PUBG（0.5 GB）、原神（0.43 GB）和 Gemini（1.8 GB）——就需要约 3GB 内存，而 Pixel 8 手机

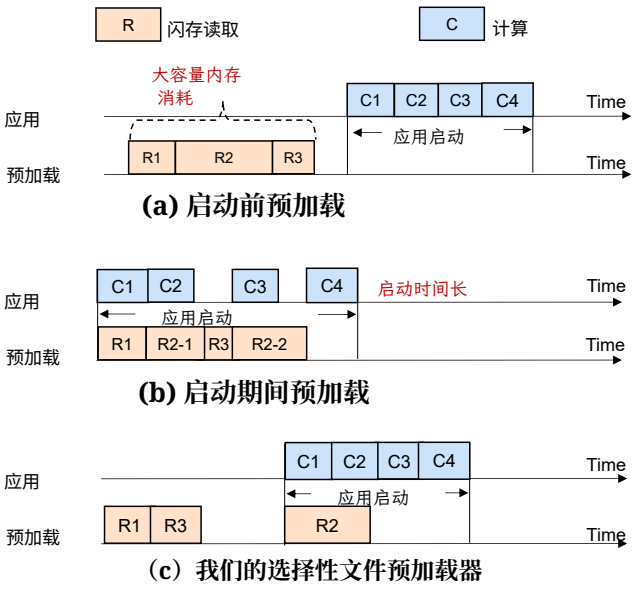



**图 10：多种预加载方法示意图。**

通常只分配不到 1GB 的内存用于预加载 [29]。其次，在启动期间预加载通过在运行时按需加载文件来节省内存，但当文件访问落后于 CPU 执行时，仍会受到 I/O瓶颈的影响。如图 10b 所示，由于 R2 和 R3 的 I/O时间较长，计算步骤 C3 和 C4 明显延迟。

这些局限性要求采用细粒度方法，使预加载时间与文件特征相匹配。实际上，GB 级应用的启动关键文件分为两种类型：*i)* 小而频繁的文件（例如，$<128KB$）占用的内存极少，但如果不尽早预加载，就会导致频繁的随机I/O。*ii)* 大型且对*I/O* 带宽要求高的文件（例如，$\geq 128KB$），其加载时间取决于I/O 带宽。预加载这些文件能够提升性能，但会占用大量内存。如第 2.2.1 节所述，在 GB 级应用中，小而频繁的文件占据了大部分启动I/O（91%），且其占用空间不大（占总量的 ≈3%；见图 5）。此外，我们发现，块大小较大（即 ≥128KB）的大文件，其I/O 带宽最高可比块大小较小时快 22.8× 倍。（图 6）。

基于这些考虑，为了降低启动延迟且不超出内存限制，我们提出了一种分两个阶段的时间分段、文件与应用类型感知预加载策略。其核心思想是在启动之前，以较小的 I/O 块大小预加载 I/O 高频文件，从而以最小的内存开销减少对延迟至关重要的阻塞；并在启动期间，以较大的 I/O 块大小预加载其余大文件，从而在时间限制内最大化 I/O 效率。具体而言，AppFlow 采用两种互补策略：

• 启动前，频率优先。此阶段侧重于在受限的内存预算内，尽可能多地加载频繁访问的小文件。它会预加载所有小文件（$<s_i$，由各个应用 $i$决定）以及大文件中频繁访问的热数据段。这些文件只会带来极少的内存开销，却能消除高频输入/输出停顿。

• 启动期间，吞吐量优先。应用启动后，此阶段会积极使用较大的块大小（$\geq s_i$）加载大文件，以最大化大文件的 I/O 吞吐量。然而，与前台 I/O 重叠会带来新的挑战，即预加载器与应用本身之间的 I/O 争用。为了解决这一问题，我们将预加载器作为独立的低优先级、独立进程运行，确保其绝不会阻塞或延迟前台 I/O 请求。

选择最优的应用专属截止大小 $s_i$，以区分小文件（$< s_i$）和大文件，这既有必要，也颇具挑战。截止大小较大 $s_i$ 会增加启动前的内存占用，而较小的截止大小 $s_i$则会导致过多小文件未被预加载，从而限制启动期间的输入/输出。我们将 $s_i$ 的选择表述为一个离散优化问题：

$$\arg\ \min T_i^{I/O} = \frac{M_i^{file} - p_i}{v_i^{I/O}}, \quad p_i = P(s_i), \quad v_i^{I/O} = V(s_i)$$
$$s.t. \quad \sum_i p_i \leq M_p$$

其中，$P$（$s_i$）是小于 $s_i$的文件（预加载）的大小，$V$（$s_i$）是流式读取其余文件时所达到的输入/输出带宽。$P$和 $V$都会随 $s_i$ 增大。由于输入/输出块大小是 2 的幂（即4KB、8KB、等等），我们仅针对这些离散大小对 $P$和 $V$进行分析。我们将选择问题建模为多重选择背包问题：每个应用在预加载预算 $M_p$（设为 100MB，见第 6.5.1 节）内，选取一个预先计算的 $s_i$。该问题规模较小，可在 < 0.1毫秒内求解（见第 6.5.3 节）；只有当应用的启动配置发生变化时（例如，启动了一个 GB 级大型应用），我们才会重新优化 $s_i$，因此运行时开销可以忽略不计，且配置始终接近最优。

## 4.2 自适应内存回收器

即使进行了预加载优化，可用内存不足仍可能阻碍启动，尤其是在系统必须先回收内存才能满足新的内存分配请求时。为解决这一问题，本模块将重点转向缩短应用的内存分配时间 $T^{alloc}$，同时避免在移动设备内存限制下导致预加载效率下降。

如第 2.1.2 节所述，当可用内存 $M^{aval}$充足时，$T^{alloc}$ 可以忽略不计。当 $M^{aval}$较低时，启动期间的内存分配请求可能会被阻塞，直到系统回收足够的内存。因此，$T^{alloc}$ 主要取决于内存回收速度 $v^{rec}$。形式上，可近似表示为 $T^{alloc} = {}^{Max}_{\{(M^{sum}-M^{aval}),\ 0\}} v^{rec}$。其中，$M^{sum}$ 表示应用启动期间的总内存需求。

**现有技术概述**。基于安卓/Linux 的移动系统（例如，智能手机、可穿戴设备、机器人技术）依赖

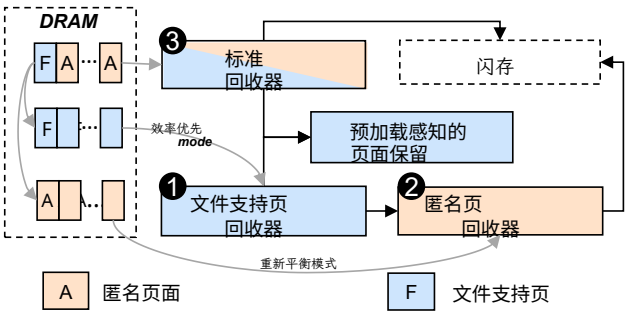



**图 11：自适应内存页回收器。**

**表 1：不同 GB 级大型应用中的匿名（Anon.）和文件支持的（File.）内存大小（MB）。**

| 类型 | Apps | 匿名 | 文件支持的 | 类型 | Apps | 匿名 | 文件。 |
|---|---|---|---|---|---|---|---|
| 多媒体 | Photoshop | 636 | 379 | 设备端 V/LLM | Qwen2.5 | 307 | 1637 |
| | Snow | 679 | 370 | | Gemma | 710 | 1925 |
| | 小红书 | 778 | 403 | Game | PUBG | 636 | 392 |
| | TikTok | 616 | 434 | | Roblox | 856 | 281 |

内核的后台回收器线程 kswapd，在 $M^{aval}$ 较低时释放内存，此期间应用启动会被阻塞。kswapd 主要针对两种内存页，即，从闪存映射而来的文件支持页，以及在运行时动态生成的匿名页。kswapd 在这两种类型之间交替进行均衡处理，试图大致均衡地释放文件支持页和匿名页。然而，我们观察到，在高内存压力下，由于两者在内存回收成本方面存在根本性的异质性和不对称性（见第 2.2.2 节），这种方式效率低下。如图 11 的第 3 部分所示，均衡交替回收会使快速的文件支持页回收受制于缓慢的匿名页处理。

*4.2.1* 页面类型感知内存回收。 为了提高内存回收效率，我们将文件支持页和匿名页的回收过程解耦。在冷启动期间，我们首先仅回收文件支持页，无需额外的输入/输出或中央处理器开销即可立即释放随机存取存储器，从而提升 $v^{\text{rec}}$ 并降低内存分配延迟 $T^{\text{alloc}}$。然而，我们观察到，由于大多数 GB 级大型应用平均产生的匿名页比文件支持页多 1.92×（见表 1），仅优先回收文件页会导致回收饥饿：此时只剩匿名页，内存分配也会停滞。

因此，我们引入一种自适应、感知内存页类型的策略，根据运行时压力切换模式：i）在高压力阶段（例如冷启动期间），AppFlow 仅回收文件支持页，以实现最高速度；ii）其他情况下，则回收匿名页，以便在未来突发负载到来前清理代价高昂的内存区域。关键挑战在于，无需修改应用或开发者介入，如何检测高压力时段。

我们利用内核的 页面分配速率 作为一种有效、开销低且与应用无关的信号。通过对 _alloc_pages_nodemask()进行插桩，并使用一个定期轮询的简单计数器（例如，每 100 毫秒一次），我们计算出 $n_{\text{alloc}}$，即上一个时间间隔内分配的内存页数。

如果某个阶段满足以下条件，我们就将其归类为内存敏感阶段：

$$M^{aval} < M^{F} \quad \& \quad n^{alloc} > N^{alloc}$$

其中，$M^{F}$ 是系统特定阈值，$N^{\text{alloc}}$ 设为 12800（第 6.5.1 节），$M^{aval}$ 是当前空闲 RAM。检测到内存密集型条件后，AppFlow 会切换到效率优先模式，仅回收文件支持页（图 11 中的 1）。压力缓解后，它会切换到重新平衡模式，恢复正常的文件页到匿名页回收比例（图 11 中的 2）。最后，AppFlow 会退回标准回收方法（图 11 中的 3）。这个全自动的多阶段控制器以极低的开销运行，无需修改应用，因此适合实际部署。

*4.2.2* 预加载感知的页面保留。在应用启动期间仅回收文件支持的页面，可能会在预加载数据使用前将其驱逐。为使这些页面保留在随机存取存储器中，直到应用实际使用它们，我们引入了一种预加载感知的保留策略，对两个预加载阶段采取不同处理。核心思想是，每个预加载阶段对页面保留的需求各不相同。启动前读取的文件数量较少，只需短暂驻留；我们每十秒轻触一次其页面，以此让它们保持活跃。一次扫描约需 10 毫秒，仅增加 0.1% 的开销。GB 级大型应用启动后，这些页面便不再纳入后续扫描，可以被回收。相比之下，启动期间读取的文件数量要多得多，定期扫描的成本过高。因此，我们在内存回收路径 shrink_page_list 中执行预加载文件检查。如果某个页面映射到预加载文件，回收器就会跳过该页面，并将其标记为活跃，以避免再次扫描。选择性文件预加载器通过 /proc 发布预加载文件列表，并在应用启动完成后将其清除。因此，预加载页会一直保留在动态随机存取存储器中，直到启动完成，不会被过早驱逐。

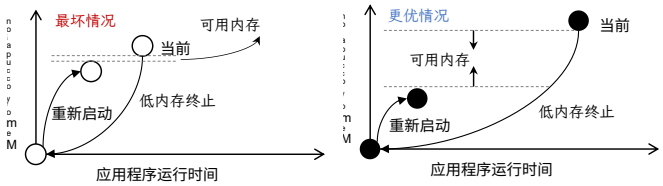
算法 1 展示了该流程。每 100ms，AppFlow 读取 $n_{\text{alloc}}$ 和 $M^{aval}$，然后选择仅回收文件支持页或全面内存回收策略（第 4 行）。在回收文件支持页期间，AppFlow 会避免回收预加载页（第 5 行）。文件支持页回收完成后，将调用匿名页面回收器，以重新平衡页面类型比例（第 6 行）。正如我们将在第 6.4.2 节中展示的，这种自适应方案既能快速恢复内存，又能实现长期均衡的内存回收，从而防止匿名页面堆积和饥饿，进而降低启动延迟和冷启动频率。

## 4.3 上下文感知型进程终止器

即使采用自适应页面回收（第 4.2 节），当只剩下匿名页时，系统有时仍会诉诸进程终止

**算法 1** 自适应内存回收器

**初始化**: 阈值 $M^F$，$N^{alloc}$；**每隔 100 毫秒** 更新 $n^{alloc} \leftarrow$AllocRate()。
**对于每个**内存分配事件**执行**
  $M^{aval} \leftarrow$FreeMem()
  **如果** $M^{aval} < M^F \wedge n^{alloc} > N^{alloc}$ **则**
    ReclaimFilePages(**排除**受保护文件)
    ReclaimAnonPages()
  **结束 if**
**结束 for**




**(a) 终止最近使用的应用** **(b) 终止长期运行的应用**

**图12：多种终止方法示意图。**

两种常见情形会导致匿名页占优，迫使系统采用效率低下的后备方案。

• 突发分配。在大规模、反复进行内存分配时，例如例如大语言模型推理，系统会延迟写入匿名页（5×比丢弃文件页更慢），因此匿名页会不断累积。根据我们对三类应用跟踪数据的分析（第4.2节），匿名页与文件页的峰值比例为1.92:1，导致文件页耗尽，剩下的只有可回收的匿名页。

• 回收容量限制。匿名页回收本身也有上限，受交换分区大小（例如，Pixel 8 上为 2GB）或 zRAM 容量（动态随机存取存储器的 25~30%）限制。达到上限后，匿名页回收会停滞，系统便会转而丢弃文件页，直到文件页耗尽并触发进程终止。

随着应用活动增加，匿名页也会随之增长，因此这些情况无法避免。要避免不必要的进程终止并维持性能和稳定性，智能的终止器必须同时考虑当前页面类型失衡状况和回收容量限制。

**内核低内存杀手的局限**。当只剩下匿名页时，安卓/Linux 会调用低内存杀手（LMK），按照从后台到前台的顺序终止应用。如第 2.2.3 节所述，每次终止都会迫使应用完全冷启动重启，像 PUBG 这样的应用通常需要 2 秒以上 [32], ，随后又会立即造成内存压力（例如，如图 12a 所示，释放了 1200MB，但重启时需要 1100MB），使系统陷入终止—重启循环。这是因为 LMK 盲目选择应用，忽略了应用的重启成本、再次访问可能性和净内存收益。

我们提出了一种新颖的上下文感知应用终止机制，该机制利用了两种运行时模式：即长期后台膨胀和短期近期性。具体而言，我们发现，长期运行的应用（例如社交媒体应用）的内存占用会比基线状态增加 30~50%（即启动后、尚未产生任何运行时缓存时立即使用的最小内存），但在重新启动时会恢复到基线状态，因为进程会清除所有累积的缓存和临时内存分配。利用这种膨胀与重置现象，AppFlow 优先从这些应用回收内存，从而获得大量且可靠的可用内存。相反，最近使用的应用重用概率较高，且内存占用也较大。我们不会依据启发式规则终止它们，而是推迟终止，以避免重新启动带来的大量内存分配。如图 12b 所示，AppFlow 根据净释放内存指标 $\Delta M = M^{\text{current}} - M^{\text{relaunch}}$，对终止操作排序，确保每次终止都能最大限度地实现持久的内存恢复，同时不影响稳定运行的应用。对于最近使用的应用，AppFlow 会转而采用考虑近期性的延迟策略，让预加载与内存回收的空闲阶段重叠。正如我们将在第 6.4.2 节所示，这种两阶段策略可减少 30% 的终止事件，在积极回收内存与无缝交互之间取得平衡。

# 5 实现

我们在安卓这一应用最广泛的移动操作系统之一上实现了 AppFlow。Android 操作系统通常分为三层：应用层、Android 框架和 Linux 内核。AppFlow 是一种横跨 Android 框架和 Linux 内核的内存管理机制。它不对应用层进行任何修改，因此 GB 级应用无需更改任何代码即可受益于其加速效果。具体而言，选择性文件预加载器和上下文感知进程终止器修改了 Android 框架中的 1,672 行代码，以控制预加载和进程终止；而自适应内存回收器修改了 Linux 内核中的 1,107 行代码，以管理内存回收。由于自适应内存回收器必须避免回收选择性文件预加载器预加载的文件，这两个模块通过 /proc 文件系统进行通信。

# 6 实验

## 6.1 实验设置

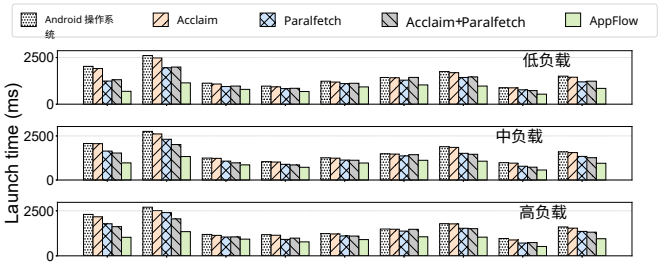
**平台**。我们在两款商用智能手机（Google Pixel 7 和 Pixel 8）以及一套车载系统上对 AppFlow 进行了原型实现。该车载系统由车辆供电，并由 Raspberry Pi 4B 负责运算；所有设备均运行最新的 Android 15 操作系统（OS） [9]。为模拟内存受限的低端移动设备，我们通过修改 arm64_memblock_init()，将两款智能手机的可见动态随机存取存储器缩减至 6GB 和 8GB。车辆动态随机存取存储器默认为 4GB。

**任务与工作负载**。我们使用真实世界的*100*天跟踪数据和模拟工作负载评估 AppFlow。真实世界的*100*天数据集AppFlow-100D 涵盖工作日、周末和节假日，涉及 60 多款应用（例如，TikTok、WhatsApp、相机）和 10,000+ 次多任务切换，从而捕捉具有代表性的用户工作负载。为收集 AppFlow-100D，我们开发了一款可在后台持续运行的无界面安卓应用。它利用 UsageStatsManager 记录 100 天的应用使用序列，之后自动终止，并提供一个面向用户的仪表板，供参与者在上传前查看跟踪数据。我们严格落实隐私保护措施：尽可能减少本地数据，仅保留元数据（软件包名称、时间戳、内存状态），并严格排除所有个人身份信息和应用内容。数据收集完成后，我们在同一设备上重放应用切换跟踪数据，以评估 AppFlow。模拟工作负载包含若干 GB 级应用（例如，Gemma、TikTok、Roblox、等）和 15 款低内存应用（例如，ChatGPT、Instagram、Zoom、等）。每项测试中，选取一款 GB 级应用，在三种场景下进行冷启动：（i）五款低内存应用（低负载）；（ii）十五款低内存应用（中负载）；（iii）十五款低内存应用加两款 GB 级应用（高负载）。为确保后台工作负载一致，我们首先终止所有后台应用并清除文件缓存。接着，我们使用自动化脚本启动负载生成应用，并将它们置于后台。负载建立后，我们执行 am start -W 命令启动目标应用并测量其启动时间。

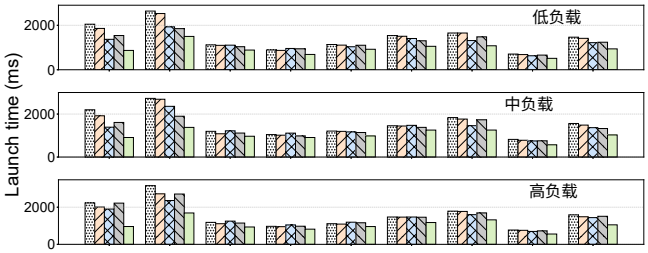
**基线方法**。为确保评估全面且公平，我们选择了四种具有代表性的基线方法，涵盖预加载和回收策略的设计空间：

- **Android 操作系统** [9] 执行内存回收，但不进行任何预加载。这构成了基础系统基线。
- **Paralfetch** [33]是一种先进的**预加载**方法，通过调度输入/输出序列，将小块合并为更大的块，从而提高启动速度。
- **Acclaim**[26] 是一种先进的**内存回收**方法，通过增大批次大小来加速内存回收，从而同时降低频率和延迟高负载条件下冷启动的。
- **Acclaim + Paralfetch** 结合了上述最先进的技术。据我们所知，目前尚无联合优化方案，因此这一基线展示了简单组合的局限性。

**指标**。 我们通过以下指标评估 AppFlow：(i)用户感知的 指标，即也就是，GB 级应用的冷启动延迟和启动频率，通过 adb 命令 [8] 捕获；以及 (ii) 中间 指标，




**(a) 在 Pixel8-8GB 上：低、中、高负载**




**(b) 在 Pixel8-6GB 上：低、中、高负载**

**图 13：GB 级冷启动延迟对比。**

即，通过 logcat [10] 或内核钩子获取输入/输出吞吐量和内存压力事件（直接回收、低内存杀手），以捕捉用户体验和系统行为。

## 6.2 GB 级应用冷启动性能

我们在两种配置下（即，内存上限分别为 6 GB 和 8 GB 的 Google Pixel 8 设备），对四种基线方法和 AppFlow 的表现进行测试：在三种工作负载下（即，低、中、高负载，如第 6.1 节所述），测试八款 GB 级应用的也就是冷启动时间，包括基于大语言模型的 Qwen2.5-2.0GB、基于视觉语言模型的 Gemma-2.6GB、TikTok-1.1GB、Snow-1.1GB、PUBG-1.2GB等。结果如图 13 所示。首先，在所有设备配置、工作负载和 GB 级应用中，AppFlow 的冷启动延迟平均比原生 Android 短 33.7~43.6%。其次，即使在低负载下，AppFlow 也优于最先进的预加载基线方法 Paralfetch；通过在 GB 级启动期间维持更高的输入/输出吞吐量，其冷启动延迟平均降低了 29.2%，如图 13a 所示。第三，在内存压力下，AppFlow 仍保持优势：在高负载下，内存受限的 Pixel 8（6 GB）上的启动延迟最多降低 57%（图 13b）。第四，简单地将预加载与内存回收结合起来可能会导致性能下降；例如，在高负载下，Paralfetch + Acclaim 使 Gemma 的启动时间比单独使用 Paralfetch 慢 14.4%，比单独使用 Acclaim 慢 4.8%。

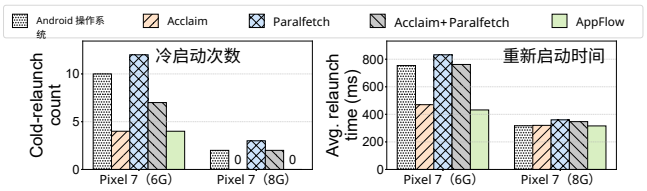



**图14：17 个应用的多任务处理性能：冷启动后重新启动次数（左）和平均重新启动时间（右）。**

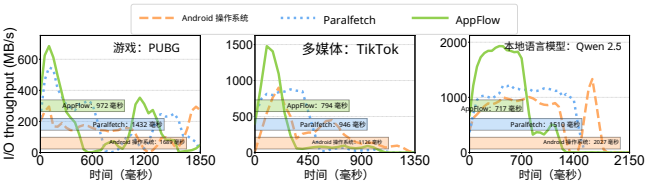



**图15：三款 GB 级大型应用的输入/输出吞吐量和冷启动延迟：PUBG（左）、TikTok（中）、通义千问 2.5（右）。**

在高负载下，Paralfetch 单独使用时的启动速度比两者组合快，Acclaim 单独使用时也快。AppFlow 通过保护预加载页缓解了这一问题，其启动速度比这种组合快 56.7%。

**总结。** AppFlow 在不同设备、工作负载和应用规模下均实现了最低的冷启动延迟，同时避免了简单基线组合的缺陷，展现出在真实移动系统中的稳健性和适用性。

## 6.3 多任务处理性能

我们在高负载下（参见第 6.1 节），于 Pixel 7（6/8 GB）上比较 AppFlow、Android 操作系统、[9], Paralfetch [33], 和 Paralfetch+Acclaim 的多任务性能。两个 GB 级应用（即，TikTok-1.1GB、Snow-1.1GB）和 15 个低内存应用各启动两次，以模拟在不同应用间切换；在第二次运行中，我们测量平均重新启动延迟和冷重新启动次数。图 14 展示了结果。首先，通过在应用和内存页两个层面快速释放内存，避免频繁触发低内存杀进程事件，AppFlow 相比 Android 操作系统多保留了 1.85× 个后台应用（即，从 7/17 个增至 13/17 个），并将平均启动时间缩短了 37.6%。其次，AppFlow 避免了预加载效果退化：在 Pixel 7（6GB）上，Paralfetch 的冷重启次数比 Android 操作系统多 20%，而 AppFlow 则减少了 60%。这是因为预加载带来的内存压力会触发过度的应用终止。第三，通过保护预加载页，同时确保回收页面不太可能被重新加载，AppFlow 的冷启动次数比 Paralfetch+ Acclaim 少 42.8%。

**摘要。** AppFlow 在内存压力较高时仍能维持多任务响应能力，降低重新启动开销。

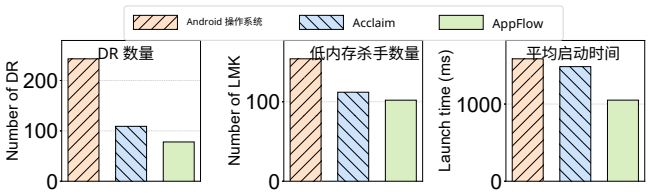



**图 16：内存压力缓解对比：内核直接回收次数（左）、低内存杀进程事件（中）和平均启动延迟（右）。**

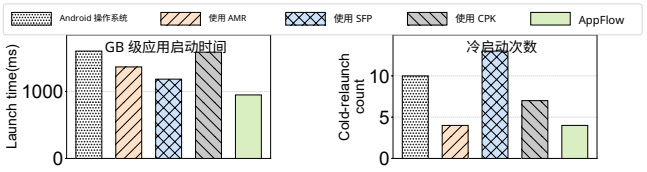
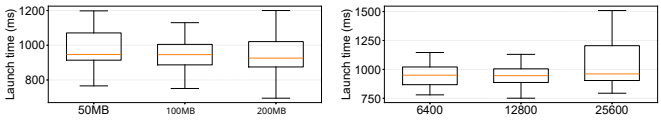



**图 17：消融研究：GB 级大型应用启动时间（左）和冷启动次数（右）。**

## 6.4 理解性能提升

*6.4.1 GB* 级应用启动期间的输入/输出吞吐量。 我们在 Google Pixel 8（8GB）上，以低负载测量 GB 级冷启动期间的输入/输出吞吐量，以展示选择性文件预加载器 如何加速冷启动（图 15）。AppFlow 通过两阶段、感知文件与应用的预加载策略，超越 Android 操作系统 [9] 和 Paralfetch [33]：启动前预取小文件以减少阻塞，启动期间则以更大的块流式读取大文件，从而最大限度地提升输入/输出效率。在 Qwen2.5 中，其吞吐量分别提高了 1.92×和 1.58×，使各类 GB 级大型应用的启动速度提升了 29.4%~64.6%。**摘要。**通过使预加载与文件访问模式相匹配，AppFlow 为可扩展的冷启动加速提供了一条有原则的路径。

*6.4.2 GB* 级应用启动期间的内存回收效率。为评估 AppFlow 缓解内存压力的效果，我们在 Google Pixel 8（6 GB）上，以高负载运行 GB 级应用启动测试，测量内核直接回收（DR）、低内存杀进程事件（LMK）以及平均启动延迟（图 16）。与 Android 操作系统相比，AppFlow 将内核 DR 次数减少了 67.9%，从而减少内存分配停顿和启动延迟；同时将低内存杀进程事件减少了 33.7%，使更多后台应用得以保留。这些改进得益于其自适应内存回收器，它通过有选择地处理内存页来加速内存回收；也得益于其上下文感知应用终止器，它优先终止内存密集型应用，避免反复触发 LMK。**总结。**通过结合细粒度内存回收与智能应用终止，AppFlow 将内存压力转化为高效的资源管理，使系统在 GB 级工作负载下仍能保持响应灵敏。




**(a) 预加载大小 $M^p$** **(b) 分配阈值 $N^{alloc}$**

**图 18：AppFlow 中超参数的影响。**

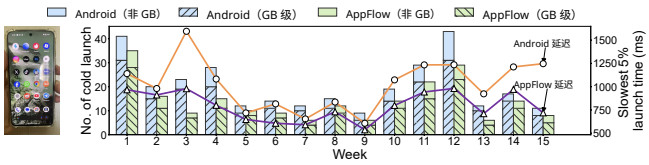


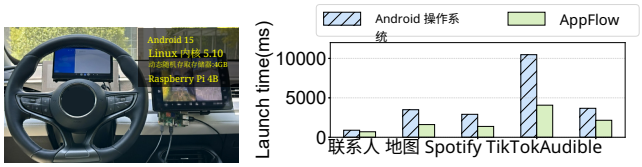

**图 19：AppFlow 在为期 100 天的测试中的性能。**

## 6.5 微基准测试

*6.5.1* 消融研究。我们在 Pixel 8（6GB）上评估 AppFlow 的各个模块，以量化它们对冷启动和多任务性能的贡献（图 17）。首先，与 Android 操作系统相比，自适应内存回收器（AMR）将冷启动次数减少了 60%，但启动时间仅缩短了 14.7%。其次，选择性文件预加载器（SFP）将启动时间缩短了 26.1%，但由于增加了内存压力，冷启动次数增加了 30%。第三，上下文感知进程终止器（CPK）通过终止内存密集型进程来避免反复终止进程，使冷启动次数减少了 30%。第四。集成这三个模块后，它们相互补充，在多保留 1.85× 个后台应用的同时，将冷启动延迟降低了 40.8%。

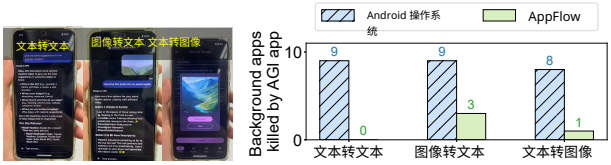
*6.5.2* 超参数设置。 我们探索了 AppFlow 中不同的超参数设置。首先，启动前预加载大小 $M^p$。该参数控制启动前预加载的数据量（第 4.1 节）。如图 18a 所示， $M^p$ = 100 MB 可实现最佳平衡。较小的值无法涵盖关键小文件（速度慢 6.5%），而较大的值则会浪费内存并增加启动时间方差。其次，分配阈值 $N^{alloc}$。它决定何时触发自适应内存回收器（第 4.2 节）。如图 18b 所示， $N^{alloc}$ = 12800 可获得最佳结果；较大的阈值会延迟内存回收，并导致启动期间最多减速 19.9%。

*6.5.3* 额外开销分析。尽管 AppFlow 带来了诸多好处，但也引入了两项额外开销，即启动前预加载所占用的内存，以及计算预加载大小所需的 CPU 时间。首先是内存。预加载量上限为 100 MB（在 4 GB 设备上占动态随机存取存储器的 <5%），这些内存页会在内存压力下被回收，因此不会影响性能。其次是 *CPU* 时间，它仅在 GB 级大型应用改变状态时才会产生（两次状态变化间隔数分钟）；即使有 60 个应用，计算也能在 <0.1 毫秒内完成，实际开销同样可以忽略不计。




**（a）车载测试平台** **（b）应用启动性能**

**图20：AppFlow 的车载性能。**




**（a）端侧 AGI（b）并发性能**

**图21：AGI工作负载下的性能。**

## 6.6 案例研究

为验证 AppFlow 的实用性和普适性，我们在各种场景中开展了真实世界案例研究。

**100天多任务处理**。在日常使用中，用户每天会在社交、效率和媒体应用之间习惯性地切换数十次，持续造成内存压力，此时启动延迟的每一秒都会影响用户感知到的响应速度。为此，我们在 Google Pixel 8（8G）上测试了为期 100 天的60+真实应用启动跟踪记录，并将 Android 操作系统与 AppFlow 进行比较。如图 19 所示，GB 级应用占用户经历的所有冷启动的 75.7%。AppFlow 将平均冷启动延迟降低了 23%，并将 GB 级应用的冷启动次数减少了 31.6%，这证明 AppFlow 在日常多任务处理中能够持续带来长期收益。例如，在第三周，用户进行旅行规划时，Android 系统会冷启动 TikTok 和小红书，导致频繁卡顿。相比之下，使用 AppFlow 时，这些应用均可热启动，从而确保流畅的用户体验。

**车载系统**。 当驾驶员启动车辆时，信息娱乐系统会随车辆点火系统一同启动，清除所有进程，并迫使导航或音乐等应用进行冷启动，使冷启动延迟变得十分明显。我们在比亚迪海豹车辆上部署了最新的Android操作系统15和AppFlow。该车辆配备了一个拥有4 GB内存的Raspberry Pi 4B（容量小于典型智能手机），并由车辆点火系统直接供电（图20a），确保每次启动都是真正的冷启动。随后，我们测量了五款热门应用的冷启动延迟，包括例如，TikTok-1.1GB [42]（图20b）。与Android操作系统相比，AppFlow将冷启动延迟降低了53.4%（4.3秒→2.0秒）。这表明，在资源受限的汽车环境中，AppFlow具有显著优势，其中

系统响应速度会直接影响每次车辆启动时的用户体验。

**端侧 AGI**。端侧 AGI 应用 [7, 37] 代表着一种新兴趋势，它们由端侧视觉/大语言模型驱动，可提供保护隐私的离线生成式智能。然而，其庞大的内存占用严重制约了多任务处理。例如，用户使用图像转文本应用撰写 Instagram 帖子后切回 Instagram 时，Instagram 往往会被终止，从而打断工作流。为复现这一场景，我们在 Google Pixel 8（8G）上启动一个 AGI 应用，随后切换到后台应用（图 21）。使用 AGI 后，Android 操作系统会终止 8~9 后台应用，而 AppFlow 将这一数量限制在 0~3，使多任务存活率从 66.6% 提升至 100%。这样，用户使用端侧 AGI 服务后，便能快速切换到其他应用，无需等待漫长的冷启动。

**摘要**。在长期（100 天）的日常使用、由点火驱动的车载汽车系统以及新兴的设备端通用人工智能工作负载中，AppFlow 始终能够降低启动延迟并避免冷重启，展现出其在多样化移动环境中的稳健性，以及作为下一代移动智能基础的潜力。

## 7 相关工作

### 7.1 移动应用启动优化

应用启动时间是关键的用户体验指标 [13, 23, 30]。以往工作通过两种方式应对这一问题：要么缓存更多应用以实现热启动 [16, 28, 31, 47]（借助交换空间 [19, 48], 数据压缩 [22, 28]），要么加速冷启动 [24, 26]（通过文件预加载 [6, 33, 47] 和更快的内存回收 [24, 26]）。然而，对于 GB 级工作集 [39], ，缓存无法覆盖大多数应用，因此冷启动难以避免 [14],；而冷启动加速器会争抢稀缺的输入/输出资源和内存，造成资源浪费 [6, 40]。AppFlow 首次实现了 GB 级冷启动加速，同时保留后台存活能力，其方法是紧密结合预加载、内存回收和应用终止机制。

### 7.2 移动设备上的预加载

预加载分为两种类型，即基于预加载时机的启动前预加载 [20, 21, 31]和启动期间预加载 [6, 16, 33] 。启动前方案会预测用户接下来要打开的应用，并将其工作集提前载入随机存取存储器（RAM） [20, 21, 31]。例如，MAPLE [17] 会结合用户上下文提示大语言模型进行预测，但其 52% 的命中率意味着每次预测失败都会浪费数百兆字节的内存，而且无法改善冷启动延迟；对于 GB 级大型应用而言，这是不可接受的权衡。启动期间方案则完全跳过预测。用户点击图标后，系统便利用空闲I/O 带宽流式加载应用所需的内存页 [6, 16, 33]。这些内存页可以被可靠识别，例如 Paralfetch [33] 通过离线分析可覆盖其中 97.2% 的内存页。AppFlow 采用按需的启动期间方案，但更进一步：它会选择性地预加载文件，以较小的内存开销提升启动速度，同时与内存回收机制无缝协作。

### 7.3 移动内存回收

当空闲 RAM 低于阈值 [35] 时，就会触发内存回收。先前的工作主要沿着两个方向展开：决定回收什么/何时回收（例如，Acclaim [26] 会优先处理后台应用 [12, 48]），以及缩短回收耗时 （例如，SWAM 会在动态随机存取存储器中压缩内存页，而不是将其写入闪存 [14, 24, 29]）。而我们的工作侧重于具备预加载感知能力的目标选择，因为淘汰错误的内存页可能抵消预加载带来的收益。

## 8 结论

为解决受限内存与输入/输出条件下 GB 级大型移动应用日益增长的冷启动延迟问题，AppFlow 旨在确保前台与后台启动均能在 1 秒内完成。它通过三个协同模块实现这一目标：针对延迟关键文件的选择性文件预加载器、保护预加载数据的自适应内存回收器，以及上下文感知应用终止器。这些模块共同减少停顿、高效回收内存，并维持多任务处理能力。评估结果表明，AppFlow 可显著降低冷启动延迟，同时即使在高负载工作场景下，也能将重新启动控制在 1 秒以内。在为期 100 天的日常跟踪数据、车载系统以及新兴的端侧 AGI 场景中，AppFlow 均能持续降低延迟并避免冷启动重启，展现出其在各种真实场景中的稳健性。虽然 AppFlow 对小型、低输入/输出应用（例如，日历）的提升有限，但不会造成性能退化。未来，设备端应用（尤其是本地人工智能代理）的规模和输入/输出占用将持续增长，而动态随机存取存储器容量受不断上涨的动态随机存取存储器价格限制；在这种情况下，AppFlow 能够有效应对日益突出的输入/输出延迟瓶颈。在未来的工作中，我们将引入用户行为预测，将预加载与回收从按应用转向按活动进行，从而进一步加快启动速度。

## 9 致谢

本工作部分得到了国家重点研发计划（编号：2024YFB4505502）、国家杰出青年科学基金（62025205）以及国家自然科学基金（62522215、62532009、62472354）的资助。

# 参考文献

[1] 安卓开源项目。2025年。低内存杀手守护进程。https://source.android.com/docs/core/perf/lmkd 访问日期：2025-07-25。[2] AppMagic。2025年。实时应用商店排名 - AppMagic。https://appmagic.rocks/top-charts/live-store-rankings。访问日期：2025-07-27。[3] BuildFire。2021年。移动应用下载统计与使用统计（2025年）。https://buildfire.com/app-statistics/ 访问日期：2025-06-29。[4] 剪映。2025年。剪映视频编辑器。https://www.capcut.com/。访问日期：2025-07-29。[5] James Carbone。2022年。动态随机存取存储器价格上涨将趋缓。Electronics Sourcing。https://electronics-sourcing.com/2022/05/12/dram-price-increases-will-ease 访问日期：2025-07-27。[6] Shaleen Garg、Jian Zhang、Rekha Pitchumani、Manish Parashar、Bing Xie 和 Sudarsun Kannan。2024年。Crossprefetch：加速现代存储的输入/输出预取。载于 *第29届 ACM 编程语言与操作系统架构支持国际会议论文集，第1卷*。102–116。[7] Google AI Edge。2025年。Google AI Edge Gallery：展示设备端 ML/GenAI 用例的图库。https://github.com/google-ai-edge/gallery 访问日期：2025-07-25。[8] Google Android Developers。2025年。安卓调试桥接器（adb）。https://developer.android.com/tools/adb。访问日期：2025-08-01。[9] Google Android Developers。2025年。Android15——关于安卓平台版本15。https://developer.android.com/about/versions/15。访问日期：2025-08-01。[10] Google Android Developers。2025年。Logcat | Android Developers。https://developer.android.com/tools/logcat。访问日期：2025-08-01。[11] Mel Gorman。2004年。*理解 Linux 虚拟内存管理器*。第352卷。Prentice Hall Upper Saddle River。[12] Weichao Guo、Kang Chen、Huan Feng、Yongwei Wu、Rui Zhang 和 Weimin Zheng。2015年。 *marsː* 通过闪存感知的页面交换提升移动应用重新启动速度。*IEEE Trans. Comput.* 65，3（2015），916–928。[13] Max Hort、Maria Kechagia、Federica Sarro 和 Mark Harman。2021年。移动应用性能优化综述。*IEEE Transactions on Software Engineering* 48，8（2021），2879–2904。[14] Jiacheng Huang、Yunmo Zhang、Junqiao Qiu、Yu Liang、Rachata Ausavarungnirun、Qingan Li 和 Chun Jason Xue。2024年。通过前台/后台感知的 GC-交换协同设计，让移动设备运行更多应用并实现更快的热启动。载于 *第29届 ACM 编程语言与操作系统架构支持国际会议论文集，第3卷*。654–670。[15] Francisco Jeronimo。2025年。全球内存短缺危机：市场分析及其对2026年智能手机和 PC 市场的潜在影响。https://www.idc.com/resource-center/blog/global-memory-shortage-crisis-market-analysis-and-the-potential-impact-on-the-smartphone-and-pc-markets-in-2026/。IDC 博客（市场与趋势），发表于2025-12-18，访问日期：2026-01-03。[16] Yongsoo Joo、Junhee Ryu、Sangsoo Park 和 Kang G Shin。2011年。{FAST}：在{Solid-State}固态硬盘上快速启动应用。载于 *第9届 USENIX 文件与存储技术会议（FAST 11）*。[17] Yonchanok Khaokaew、Hao Xue 和 Flora D Salim。2024年。MAPLE：利用大型语言模型嵌入进行移动应用预测。*ACM 交互式、移动、可穿戴与普适技术论文集* 8，1（2024），1–25。[18] Sang-Hoon Kim、Jinkyu Jeong 和 Jin-Soo Kim。2017年。面向应用的移动系统交换。*ACM Transactions on Embedded Computing Systems*（*TECS*）16，5s（2017），1–19。[19] Niel Lebeck、Arvind Krishnamurthy、Henry M Levy 和 Irene Zhang。2020年。终结毫无意义的终止：改进内存管理以移动操作系统。载于*2020*年*USENIX*年度技术会议（*USENIX ATC 20*）。873–887。[20] Joohyun Lee、Kyunghan Lee、Euijin Jeong、Jaemin Jo 和 Ness B Shroff。2016。移动系统中的上下文感知应用调度：用户接下来会做什么、不会做什么？载于*2016*年*ACM* 普适与泛在计算国际联合会议论文集。1235–1246。[21] Joohyun Lee、Kyunghan Lee、Euijin Jeong、Jaemin Jo 和 Ness B Shroff。2017。CAS：交互式移动系统中的上下文感知后台应用调度。*IEEE* 通信领域精选期刊35, 5 (2017), 1013–1029。[22] Changlong Li、Zongwei Zhu、Chao Wang、Fangming Liu、Fei Xu、Edwin H-M Sha 和 Xuehai Zhou。2025。Archer：具备{Page-Association-Rule} 感知能力、可提升移动设备高速响应的自适应内存压缩。载于第*23*届*USENIX* 文件与存储技术会议（*FAST 25*）。497–511。[23] Tong Li、Tong Xia、Huandong Wang、Zhen Tu、Sasu Tarkoma、Zhu Han 和 Pan Hui。2022。智能手机应用使用分析：数据集、方法与应用。*IEEE* 通信调查与教程24, 2 (2022), 937–966。[24] Wentong Li、Li-Pin Chang、Yu Mao 和 Liang Shi。2025。{PMR}：通过在移动设备上并行内存回收实现快速应用响应。载于*2025*年*USENIX*年度技术会议（*USENIX ATC 25*）。1569–1584。[25] Wentong Li、Dingcui Yu、Yunpeng Song、Longfei Luo 和 Liang Shi。2024。Elasticzram：重新审视移动设备上的交换用 zRAM。载于第*61*届*ACM/IEEE* 设计自动化会议论文集。1–6。[26] Yu Liang、Jinheng Li、Rachata Ausavarungnirun、Riwei Pan、Liang Shi、Tei-Wei Kuo 和 Chun Jason Xue。2020。Acclaim：通过自适应内存回收改善 Android 系统中的用户体验。载于*2020*年*USENIX*年度技术会议（*USENIX ATC 20*）。897–910。[27] Yu Liang、Riwei Pan、Tianyu Ren、Yufei Cui、Rachata Ausavarungnirun、Xianzhang Chen、Changlong Li、Tei-Wei Kuo 和 Chun Jason Xue。2022。{CacheSifter}：筛选缓存文件，提升移动设备性能并延长使用寿命。载于第*20*届*USENIX* 文件与存储技术会议（*FAST 22*）。445–459。[28] Yu Liang、Aofeng Shen、Chun Jason Xue、Riwei Pan、Haiyu Mao、Nika Mansouri Ghiasi、Qingcai Jiang、Rakesh Nadig、Lei Li、Rachata Ausavarungnirun 等。2025。Ariadne：一种热度感知且大小自适应的压缩交换技术，可快速重新启动应用并降低移动设备的中央处理器使用率。载于*2025*年*IEEE* 高性能计算机体系结构国际研讨会（*HPCA*）。IEEE，1588–1602。[29] Geunsik Lim、Donghyun Kang、MyungJoo Ham 和 Young Ik Eom。2023。SWAM：重新审视交换和 oomk，以提升移动设备上的应用响应能力。载于第*29*届移动计算与网络年度国际会议论文集。1–15。[30] Jakob Nielsen。1994。可用性工程。Morgan Kaufmann。[31] Abhinav Parate、Matthias Böhmer、David Chu、Deepak Ganesan 和 Benjamin M Marlin。2013。面向智能手机的实用预测与预取，加快应用访问速度。载于*2013*年*ACM* 普适与泛在计算国际联合会议论文集。275–284。[32] PUBG Corporation。 [n. d.]。PUBG Mobile 官方网站。https: //www.pubgmobile.com/en-US/home.shtml。访问日期：2025-08-05。[33] Junhee Ryu、Dongeun Lee、Kang G Shin 和 Kyungtae Kang。2023。面向个人{计算/通信}设备的快速应用启动。载于第*21*届*USENIX* 文件与存储技术会议（*FAST 23*）。425–440。[34] Andrew Sanders。2016。虚幻引擎 *4* 简介。AK Pe- ters/CRC Press。

[35] Mohit Saxena 和 Michael M Swift。2010。{FlashVM}：闪存上的虚拟内存管理。载于*2010*年*USENIX*年度技术会议（*USENIX ATC 10*）。[36] Zhihao Shen、Kang Yang、Wan Du、Xi Zhao 和 Jianhua Zou。2019。DeepAPP：用于移动应用使用预测的深度强化学习框架。载于第*17*届嵌入式网络传感器系统会议论文集。153–165。[37] ShiftHackZ。2025。Stable-Diffusion-Android：Stable Diffusion人工智能客户端（安卓）。https://github.com/ShiftHackZ/Stable-Diffusion-Android。访问日期：2025-09-01。[38] Choonsung Shin、Jin-Hyuk Hong 和 Anind K Dey。2012。智能手机移动应用使用情况的理解与预测。载于*2012*年*ACM*普适计算会议论文集。173–182。[39] Hadlee Simons。2025。2025年，你的手机究竟需要多少随机存取存储器?
h
t
t
p
s://www.androidauthority.com/how-much-ram-do-i-need-phone-3086661/。访问日期：2025-07-27。[40] Sam Son、Seung Yul Lee、Yunho Jin、Jonghyun Bae、Jinkyu Jeong、Tae Jun Ham、Jae W Lee 和 Hongil Yoon。2021。{ASAP}：通过自适应预分页实现快速移动应用切换。载于*2021*年*USENIX*年度技术会议（*USENIX ATC 21*）。365–380。[41] Yuan Tian、Ke Zhou、Mounia Lalmas 和 Dan Pelleg。2020。从移动应用使用模式中识别任务。载于第*43*届*ACM SIGIR*国际会议：信息检索研究与开发信息检索。2357–2366。[42] TikTok。2025年。TikTok——让每一天精彩。https://www.tiktok.com/ 访问日期：2025-07-25。[43] 集邦咨询。2025年。DRAM现货价格趋势。集邦咨询DRAM价格趋势（DRAM现货价格）。https://www.trendforce.com/price/dram/ dram_spot 最后更新：2025-07-25 访问日期：2025-07-27。[44] u/DeckerSU 等人。2019年。安卓应用的正常内存占用是多少? https://www.reddit.com/r/androiddev/comments/ atfccg/what_normal_memory_footprint_for_an_android_app/ 访问日期：2025-07-27。[45] Whitespace Global。2025年。Collective AI：安全的离线人工智能助手。苹果应用商店（美国）。https://apps.apple.com/us/app/collective-ai/ id6608983600 版本 1.3.0（2025-10-02）。访问日期：2026-01-13。[46] 维基百科贡献者。2025年。安卓版本历史 – 维基百科。https://en.wikipedia.org/wiki/Android_version_history。https://en. wikipedia.org/wiki/Android_version_history 访问日期：2025-07-27。[47] Tingxin Yan、David Chu、Deepak Ganesan、Aman Kansal 和 Jie Liu。2012年。利用预测性用户上下文快速启动移动设备上的应用。载于第*10*届移动系统、应用与服务国际会议论文集。113–126。[48] Xiao Zhu、Duo Liu、Kan Zhong、Jinting Ren 和 Tao Li。2017年。SmartSwap：移动系统中兼具高性能与良好用户体验的交换机制。载于第*54*届设计自动化年会（*2017*）论文集。1–6。