计算机研究与发展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202330538
61(3): 557−570, 2024

基于超低延迟 SSD 的页交换机制关键技术

王紫芮　蒋德钧
（中国科学院计算技术研究所先进计算机系统研究中心　北京　100190）
（处理器芯片全国重点实验室（中国科学院计算技术研究所）　北京　100190）
（中国科学院大学计算机科学与技术学院　北京　100049）
（wangzirui22z@ict.ac.cn）

Key Techniques of Swapping Mechanism Based on Ultra-Low Latency SSD

Wang Zirui and Jiang Dejun
（*Research Center for Advanced Computer Systems*, *Institute of Computing Technology*, *Chinese Academy of Sciences*, *Beijing* 100190）
（*State Key Lab of Processors* (*Institute of Computing Technology*, *Chinese Academy of Sciences*), *Beijing* 100190）
（*School of Computer Science and Technology*, *University of Chinese Academy of Sciences*, *Beijing* 100049）

Abstract　With the rapid increase of memory-intensive applications, memory capacity is playing an increasingly prominent role in application requirements. However, particle density puts constraints on the DRAM memory capacity scalability. The swapping mechanism, known as a common memory-expansion technology, is to temporarily store less-used memory pages in devices to expand memory. In the past, the disk's read/write speed was the main limit to prevent the wide adoption of the swapping mechanism. In recent years, with the rapid development of ultra-low latency SSDs, the swapping mechanism can take advantage of its low-latency read and write characteristics to improve the efficiency of swapping. The I/O stack of the swapping approach, however, has a significant software overhead with low I/O latency. We analyze and evaluate the Linux swapping mechanism using ultra-low latency SSDs and design Ultraswap, a swapping mechanism based on ultra-low latency SSDs. Ultraswap adds the processing of polling requests to the Linux I/O stack and reduces the I/O merging and scheduling overhead to achieve a lightweight I/O stack. Based on Ultraswap's I/O stack, the swap-in and swap-out paths of the kernel swapping mechanism are further optimized. By optimizing the handling of faulted pages and direct memory recycling, the time overhead on the critical path of the swapping mechanism is reduced. The results show that Ultraswap can improve the average performance by 19% compared with Linux swapping mechanism; with 20% of local memory, Ultraswap can achieve a 33% performance improvement, effectively reducing the time overhead on the critical path of the swapping path.

Key words　swapping; I/O stack; ultra-low latency SSD; polling; NVMe SSD

摘　要　随着内存密集型应用的快速发展，应用对单机内存容量的需求日益增大.然而，受到颗粒密度的限制，内存容量的扩展度较低.页交换机制是进行内存扩展的经典技术，该机制通过将较少使用的内存页面暂存在存储设备，以达到扩展内存的目的.过去页交换机制由于慢速磁盘的读写速度限制，无法被广泛应用.近年来，得益于超低延迟固态硬盘（solid state drive, SSD）的快速发展，页交换机制可以利用其低延迟的读写特性，提升页交换效率.然而，在低 I/O 延迟的情况下，传统页交换机制的 I/O 栈存在巨大的软件

收稿日期：2023-06-21；**修回日期**：2024-01-02
基金项目：国家自然科学基金重大研究计划项目 (92270202)；中国科学院战略性先导科技专项 (XDB44030200)
This work was supported by the Major Research Plan of the National Natural Science Foundation of China (92270202) and the Strategic Priority Research Program of Chinese Academy of Sciences (XDB44030200).
通信作者：蒋德钧（jiangdejun@ict.ac.cn）

开销. 首先对使用超低延迟 SSD 的 Linux 页交换机制进行测试与分析, 发现现有页交换机制的主要瓶颈在于发送请求时存在队头阻塞问题、I/O 合并和调度开销, 以及内核返回路径上的中断处理和直接内存回收开销. 基于分析结果, 提出基于超低延迟 SSD 的页交换机制 Ultraswap.Ultraswap 在 Linux I/O 栈的基础上增加对轮询请求的处理, 并降低 I/O 合并与调度开销, 实现轻量级的 I/O 栈. 基于 Ultraswap 的 I/O 栈, 对内核页交换机制的换入与换出路径进一步优化. 通过优化对缺页、直接内存回收的处理, 降低页交换机制关键路径上的时间开销. 实验结果表明 Ultraswap 在应用测试场景下相比 Linux 页交换机制能够提升 19% 的平均性能; 在可使用内存比例为 20% 的情况下, Ultraswap 可达到 33% 的性能提升.

关键词 页交换; I/O 栈; 超低延迟 SSD; 轮询; NVMe SSD

中图法分类号 TP316

随着内存密集型应用的广泛兴起, 日益增长的数据量和对高性能应用的追求使得对内存的需求迅速增长. 然而, 随着摩尔定律的放缓, 内存难以获得更高的存储密度和更低的颗粒价格[1]. 因此, 如何有效地进行内存扩展成为急需解决的问题.

页交换机制是用于内存扩展的经典技术. 当内存不足时, 该机制通过将较少使用的内存页面保存在存储设备, 从而达到扩展内存的目的. 当应用需要访问不在内存中的页面时, 则通过触发缺页异常, 将在存储设备的页面读入内存中的指定地址, 以确保应用的正常运行. 页交换机制由内核提供, 且对应用程序透明.

在过去, 页交换机制的性能受到慢速设备读写的限制. 目前广泛应用的固态硬盘(solid state drive, SSD)具有容量大、带宽高的特性, 读写延迟相比磁盘的 10 ms 降低到 100 μs[2]. 近年来, 随着 3D XPoint[3] 和 Z-NAND[4] 等非易失性存储器(non-volatile memory, NVM)技术的发展, 存储设备和 CPU 的性能差距不断缩小. 如今的超低延迟 SSD(ultra-low latency SSD), 例如英特尔 Optane SSD[5]、三星 Z-SSD[6] 等, 可以提供低于 10 μs 的延迟和高于 6 GBps 的 I/O 带宽, 相比磁盘的读写速度提高了 1 个量级. 因此, 使用超低延迟 SSD 为页交换设备, 可以更加高效地对数据进行访问和操作, 降低页交换对应用整体性能的影响.

然而, 随着高速设备的发展, I/O 延迟被降低到微秒级别, 设备的读写性能不再是限制页交换性能的瓶颈, 反而传统上被认为是轻量级的内核软件开销, 愈发不可忽视. 页交换机制较为复杂, 涉及到内核的交换逻辑层(页面回收与缺页异常处理)以及 I/O 栈(多队列块层与设备驱动层). 交换逻辑层需要完成对交换页面的选择并对交换缓存进行处理, 而 I/O 栈需要进行读写请求的提交和完成工作. 由于目前没有工作对现有的 Linux 页交换机制进行分析, 本文希望通过实验和分析, 可以为设计高效的页交换机制提供设计指导.

针对超低延迟 SSD, 学术界利用其低延迟的特点进行了 I/O 栈设计, 例如 AIOS[7], FlashShare[8] 等. 这部分工作通过降低内核块层开销, 提升整体性能. 然而, 尽管现有页交换机制和文件系统共享一部分 I/O 栈, 文献 [7−8] 工作集中于对文件页进行 I/O 优化, 而忽视页交换时匿名页的读写特征以及内核时间开销情况, 导致无法有效降低页交换机制的软件开销. 据我们所知, 本文是利用超低延迟 SSD 对页交换机制进行优化的工作.

本文首先通过对 Linux 页交换机制源码进行分析, 分析评估页交换内核路径上各操作的影响. 测评从页换入和页换出 2 方面对 Linux 页交换机制进行分析, 探究页交换机制在使用超低延迟 SSD 时的软件层面的开销特征, 对页交换过程中交换逻辑层、多队列块层和驱动层的延迟进行比较. 基于当前 Linux 页交换机制的性能特征, 分析影响页交换机制性能的因素. 测试结果表明, Linux 的 I/O 栈软件开销可达到内核总时间开销的 64%. 主要原因在于发送请求时存在队头阻塞(head-of-line blocking)问题、I/O 合并和调度开销, 以及内核返回路径上的中断处理和直接内存回收开销.

基于测试实验结果, 本文设计了基于超低延迟 SSD 的页交换机制 Ultraswap.Ultraswap 在 Linux I/O 栈的基础上增加了对轮询请求的处理, 并通过直接调用 NVMe(non-volatile memory express)驱动的方式, 降低了 I/O 合并与调度开销, 从而实现了轻量级的 I/O 栈. 利用 Ultraswap 的 I/O 栈, 本文对内核页交换机制的换入与换出路径进一步优化. 在缺页异常方面, Ultraswap 为每个 CPU 分配多个请求队列, 从而混合使用轮询和中断请求, 分离关键页面、预取页面及回收页面, 解决请求发送时的队头阻塞问题; 在内存回收方面, 考虑到处理缺页异常的关键路径上可能触发直接内存回收, 影响缺页异常的返回, 本文将

直接内存回收的过程进行迁移，降低关键路径上的时间开销.

本文的主要贡献包括 4 个方面：

1）基于超低延迟 SSD 进行 Linux 页交换机制的分析和性能测评，分析内核各操作的时间占比，归纳影响页交换机制的主要原因；

2）设计和实现基于超低延迟 SSD 的页交换机制 Ultraswap，实现轻量级的内核 I/O 栈，降低页交换的软件开销；

3）提出内核页交换机制的换入与换出路径优化方法，优化页交换关键路径，降低内核软件延迟；

4）结果表明 Ultraswap 在应用测试场景下相比 Linux 页交换机制能够提升 19% 的平均性能，与理想化情况相比，Ultraswap 能够将 Linux 页交换机制与理想化情况的性能差距减少 95%.

1 相关工作

本节主要介绍目前业界对于页交换机制和超低延迟 SSD 的相关工作. 在页交换机制部分，主要介绍学术界在使用不同存储设备，例如远程直接内存访问（remote direct memory access, RDMA）[9]、硬盘的情况下，针对页交换机制的软件层开销进行的优化工作，包括预取算法、I/O 栈优化等；在超低延迟 SSD 部分，将介绍部分超低延迟 SSD 的性能参数，以及基于超低延迟 SSD 进行的内核软件优化工作.

1.1 页交换机制

为了满足用户的需求，或者满足内存密集型应用程序的需求，内核会将较少使用的内存页面换出到存储设备，这相当于提供更多的内存，这种机制称为页交换机制. 当内存不足时，内核会扫描页面链表，选择较少使用的匿名页面，将其写到存储设备上的交换分区. 当需要使用该页面时，触发缺页异常，根据换出时的信息记录，发送读请求将该页面从存储设备读入内存.

在使用不同存储设备的情况下，页交换机制的上层软件交换逻辑（例如页面回收算法、页面预取算法）相同，而读写请求在经过多队列块层之后会通过不同的驱动发送到设备，例如发送到 NVMe SSD 上的读写请求需要使用 NVMe 驱动接口函数，而发送到远端内存的读写请求使用 RDMA 接口函数. 根据存储设备的不同，本文将使用硬盘作为交换分区的页交换机制称为本地页交换机制，使用远端内存作为交换分区的页交换机制称为远端页交换机制.

1）本地页交换机制. 页交换机制已经被研究了很多年. 与磁盘相比，SSD 的性能有极大的提升，基于 SSD 的页交换机制已成为解决内存扩展问题的主流方案.FlashVM[10] 利用闪存的特性，对内核的页交换机制实现整体优化. 在页换入路径上，利用闪存相对磁盘更低的寻道时间，FlashVM 使用步幅预取以最大限度地减少不需要的页面内存污染. 在页换出路径上，FlashVM 以比 Linux 更精细的粒度限制页面回写速率，从而更好地实现页面从内存到闪存的拥塞控制. SPAN[11] 提出重新构建系统执行分页和预读的机制，利用 NVM 设备提供的高并行度和高性能，将触发缺页异常的关键页面与预取的页面分开，将预取任务分配给单独的线程处理，从而降低关键路径上的等待时延. 此外，SPAN 提出了一种复杂的算法来预测预取页面的未来访问情况，从而减少从 NVM 设备中预取的无用页面数量.

2）远端页交换机制. 在 RDMA 网络和新兴硬件的支持下，可以利用集群内的空闲资源，以改善和平衡资源利用率. 为了实现内存分解，文献 [12−15] 探索了使用远端内存而不是本地 SSD 进行页交换，但远端内存的性能通常受到网络速度和 CPU 开销的限制. Infiniswap[12] 是一种专为 RDMA 网络设计的远端内存分页系统. Infiniswap 将每台计算机的交换空间划分为多个 slabs 并将它们分布在多台机器的远端内存中，通过负载均衡收集空闲内存供应用程序使用. Fastswap[13] 也是为 RDMA 网络设计的远端内存分页系统，其通过在关键路径上区分关键页面和预取页面来防止队头阻塞. 此外，Fastswap 不采用块 I/O 操作，而是以页粒度和类似内存（RAM-like）的方式访问远端内存. Leap[14] 是一种基于 RDMA 的在线预取解决方案，可以最大限度地减少关键路径中的远端内存访问总数. Canvas[15] 是一个重新设计的交换系统，它完全隔离了远端内存应用程序的交换路径，允许每个应用程序拥有其专用交换分区、交换缓存、预取程序和 RDMA 带宽.

基于本地页交换机制的工作主要利用内核提供的块层接口，对页交换过程中的交换逻辑部分，包括预取和页面回收流程进行优化. 基于远端页交换机制的工作主要关注于集群内的空闲资源改善和平衡资源利用率. 表 1 将 Ultraswap 与文献 [10−15] 所述的工作进行对比，本文利用超低延迟 SSD，在较新版本的内核上进行页交换机制的优化，希望通过分析页交换过程中交换逻辑层、多队列块层和驱动层各部分的延迟情况，进一步降低页交换机制的软件开销，

从而提升应用的整体性能.

Table 1 Comparative Analysis of Swapping Mechanism in Related Works

表 1 页交换机制相关工作对比

系统	内核版本号	存储设备	I/O
FlashVM[10]	2.6	传统 SSD	SATA
SPAN[11]	3.13	超低延迟 SSD 模拟器	模拟实现
Infiniswap[12]	3.13	远端内存	RDMA
Fastswap[13]	4.11	远端内存	RDMA
Leap[14]	4.4	远端内存	RDMA
Canvas[15]	5.5	远端内存	RDMA
Ultraswap（本文）	5.4	超低延迟 SSD	NVMe

1.2 超低延迟 SSD

在过去的几十年中，各种工作研究了如何采用新的数据存储技术，包括相变存储器（phase-change memory, PCM）、自旋转移扭矩磁性随机存储器（spin-transfer torque RAM, STT-RAM）和可变电阻式存储器（resistive RAM, ReRAM）等，以建立快速的非易失性存储器.

随着 Z-NAND, 3D XPoint 等新型存储技术的出现，新一代的超低延迟 SSD，例如三星 Z-SSD 和英特尔 Optane SSD，将 4 KB 大小的随机读延迟从 10 ms 降低至 10 μs，与内存的性能差距从 10 万倍降低至 100 倍，进一步缩小了内存和外存之间的性能差距. 以英特尔 Optane SSD 为例，3D XPoint 技术是英特尔和美光科技联合开发的一种非易失性存储器技术，Optane SSD 的控制器能够将单个 4 KB 的 I/O 分布在多个 3D XPoint 内存通道上，从而同时利用多个内存芯片处理单个 I/O. Optane SSD 的第一代产品为 P4800x，可以分别提供 2 400 MBps 的顺序读取速度和 2 000 MBps 的顺序写入速度. 在随机读写方面，P4800x 提供 550kIOPS 的顺序读取性能和 500kIOPS 的随机写入性能.

近年来，业界在利用超低延迟 NVMe 设备消除块层 I/O 延迟方面进行了一些工作. 这些工作分别通过软件[7-8,16-18]和硬件[19-20]层面的优化来降低 I/O 访问延迟和优化 I/O 调度. AIOS[7]在超低延迟 SSD 的基础上提出了异步 I/O 栈的概念，将 I/O 路径中的同步操作用异步操作取代，以便将与 I/O 有关的 CPU 操作与设备 I/O 同时进行. FlashShare[8]进行了整体的跨堆栈设计，使得应用能够直接利用超低延迟 SSD 设备的低延迟优势，并满足应用的不同服务级别要求. FlashShare 通过对存储软件栈的数据结构进行扩展，将应用的属性传递给从内核到 SSD 固件的所有层，从而实现 I/O 请求从提交到执行，再到完成的所有过程优化. Harris 等人[16]的工作系统地利用超低延迟存储设备研究了所有可用的 Linux I/O 完成机制，包括中断、轮询和混合轮询机制. Whitaker 等人[17]的工作对 I/O 调度程序进行研究，对使用超低延迟 SSD 情况下 I/O 调度的性能和能效进行评估.

本节先后对近年来学术界针对页交换机制和超低延迟 SSD 的优化工作进行了介绍. 现有的页交换机制针对 SSD 和远端内存的 I/O 特性进行软件层面的优化，而针对使用超低延迟 SSD 的单机场景下的页交换机制存在研究空白. 对于使用超低延迟 SSD 的工作，这类工作主要针对文件系统的 I/O 操作进行优化，而对于页交换操作缺乏针对性. 本文首先针对超低延迟 SSD 的特性优化 Linux 页交换机制，实现开销更低的页交换机制 Ultraswap，包括实现轻量级的 I/O 栈以及更为优化的交换路径.

2 页交换机制分析

本节将首先介绍页交换机制的整体流程，然后分别对页交换机制的页换入路径与页换出路径进行分析并给出测试结果.

图 1 为页交换机制整体架构图，页交换机制由多层组成，包括交换逻辑层、多队列块层以及设备驱动层. 交换逻辑层提供页换入与页换出过程中，页面选择算法的实现以及对交换缓存的处理；多队列块层

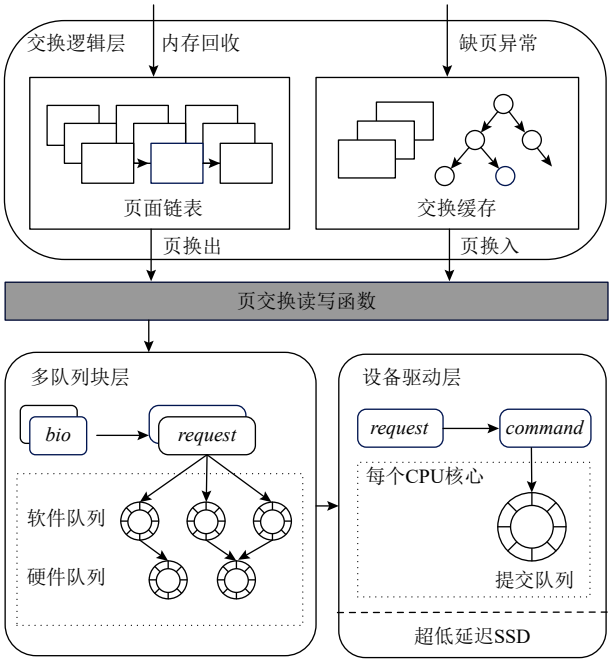


Fig. 1 Overview of the swapping mechanism frame diagram

图 1 页交换机制整体架构图

提供操作系统级别的块请求/响应管理和 I/O 合并与调度机制；设备驱动层负责处理设备上 I/O 请求的提交和完成.

2.1 页换入路径

当应用访问到不在内存中的页面时，会触发缺页异常，从用户态进入内核态. 此时只有从交换分区中将该页面取回内存后，才能返回用户态继续执行. 算法 1 描述了页换入算法的整体流程.

算法 1. 页换入算法.

① function *do_swap_page*( )
② *page*←查找交换缓存;
③ if(交换缓存中不存在页面)
④ *page*←*swapin_readahead*( );
⑤ end if
⑥ 为页面生成页表项;
⑦ 添加页面反向映射关系;
⑧ end function
⑨ function *swapin_readahead*( )
⑩ *ra_info*←获得需要换入的页面范围;
⑪ 开启块层蓄流机制;
⑫ for(每一个 *ra_info* 中的页面 *page*)
⑬ *page_allocated*←为 *page* 分配交换缓存并判断是否成功;
⑭ if(*page_allocated*==true)
⑮ *swap_readpage*(*page*);
⑯ end if
⑰ end for
⑱ 结束块层蓄流机制;
⑲ 将 *page* 插入页面链表;
⑳ end function
㉑ function *swap_readpage*(*page*)
㉒ *bio*←生成块层页面 I/O 信息，记录回调函数为 *end_read_func*;
㉓ *submit_bio*(*bio*);
㉔ end function

1)交换逻辑层. 当触发缺页异常，内存需要进行页换入时，内核会调用函数 *do_swap_page* 进行处理. 该函数会首先在交换缓存中对触发缺页异常的关键页进行查询(算法 1 的行②). 交换缓存是另一种形式的页面缓存，用于在选择页面的操作和实际执行页面 I/O 的机制之间充当缓冲者. 对页换入过程而言，当一个存在多对映射的页面被读入内存，该页面将被保存在交换缓存中. 当需要该页面的另一个映射触发缺页异常，内核可以直接利用交换缓存中的页面与该应用的页表建立映射，而不需要从设备再一次读入该页面，从而减少设备 I/O 的次数. 当所有需要该页面的页表建立对应映射之后，交换缓存释放该页面.

如果交换缓存中没有找到对应页面，内核会调用函数 *swapin_readahead* 向设备发送读请求. 为了减少发生缺页异常的次数，内核将发生缺页异常的关键页面附近的页面一并取回，即进行预取操作. 对于使用 SSD 的情况，由于 SSD 比磁盘拥有更好的随机读性能，Linux 内核中将取回与关键页的物理地址临近的几个页面，而非在设备上连续的几个页面. 在设备完成页面的读请求之后，内核为取回的缺页创建页表项及反向映射，完成缺页异常(算法 1 的行⑥⑦).

在函数 *swapin_readahead* 中，内核会首先获取包含关键页地址的一个地址范围，即需要取回的所有页面范围. 对于这个地址范围内的每一个页面，内核首先查找该页面是否已经存在交换缓存中. 若不存在，则为该页面分配页框并发送读请求(算法 1 的行⑬～⑯). 在所有页面的请求发送完成后，将页面加入到对应的页面链表，使得取回的页面能真正地被内核管理.

2)多队列块层. 内核通过调用函数 *swap_readpage* 向设备发送每个页面的读请求. 对每个页面，内核为该页面生成一个 *bio* 结构体，记录该读操作完成后的回调函数 *end_read_func*，调用函数 *submit_bio* 将该页面的读请求发送到 Linux 的块层.

图 2 表示向多队列块层和 NVMe 驱动发送 I/O 请求的处理流程. 在块层中，函数 *submit_bio* 提交的 *bio* 结构体会被转换为 *request* 结构体，并将 *request* 结构体插入每个核对应的请求队列中，在队列中执行 I/O 合并和调度操作.*request* 结构体依次通过每个核对应的软件队列和硬件队列，进入设备驱动层.

I/O 合并和调度操作是将符合条件的多个 I/O 请求合并成单个 I/O 请求进行处理. 在 Linux 内核中，在从 *bio* 结构体到向驱动发送 *command* 的过程中，主要会发生 2 次 I/O 合并，分别为蓄流过程和调度过程.

每个进程都有一个蓄流队列. 如果进程向通用块层发送 I/O 请求前开启了蓄流功能，请求会被保存在队列中进行合并，直到泄流开启，请求才会被发送到下层的调度器中. 蓄流过程的另外一个功能是生成 *request* 结构体. 当进程发送一个 *bio* 结构体，会首先尝试将 *bio* 结构体合并进蓄流队列里的 *request* 结构体，如果无法合并，则生成一个新的 *request* 结构体. 在泄流的过程中，请求会被发送到下层调度算法

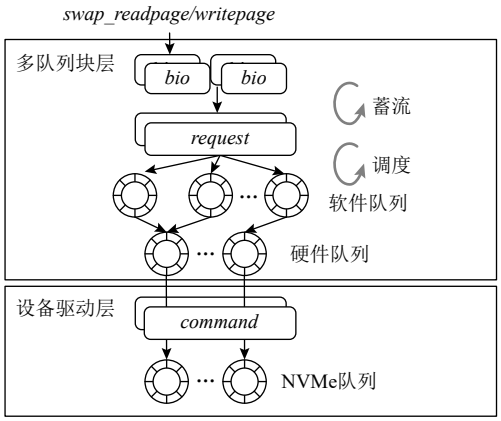


Fig. 2 Diagram of multi-queue block layer & NVMe driver

图 2 多队列块层及 NVMe 驱动图

的队列中, 实现进一步合并. 电梯调度算法的主要目的是将 *request* 结构体对硬件的访问顺序化, 并发送到下层的驱动层执行 I/O 请求.

3)设备驱动层. 在硬件队列, 内核使用函数 *nvme_queue_rq* 将请求分配给设备驱动层. 每个 NVMe 队列都由一个提交队列和一个完成队列成对组成. 每个 *request* 结构体在提交的过程中, 会被转变成 *command* 结构体发送到提交队列, 进行设备 I/O. 完成 I/O 请求的服务后, NVMe 控制器通过向主机发送消息信号中断(message signaled interrupts, MSI)来通知内核数据传输完成.

2.2 页换出路径

当内存不足的时候, 内核需要进行页面回收, 从而释放页帧, 使得有空闲的内存可以被使用. 页面回收过程需要确定哪些页可以从内存换出且对内核性能影响较小.

页面回收过程主要在 2 种情况下触发: 1)如果进行内存分配时没有足够的空闲内存, 立即触发内存回收; 2)内核存在周期性回收内存机制, 对系统内存进行周期性检查. 在页交换过程中, 缺页异常同样会涉及内存分配, 因此在完成页换入的关键路径上, 内核会对空闲内存的剩余情况进行检查. 如果此时空闲内存剩余不足, 则触发直接页面回收.

在触发页面回收之后, 需要选择内存中的页面进行回收. 在每个内存节点, 内核采用最近最少使用(least recently used, LRU)链表指向物理内存页面. 根据局部性原理, 页面链表假定最近最少使用的页面在较短的时间内不会频繁使用, 因而可以选出需要回收的页面. 内核会对链表进行扫描, 刷新活跃链表和不活跃链表. 内核通过扫描不活跃链表, 主要进行 2 个操作: 为可以回收的页面发送写请求和释放写请求完成的页面. 算法 2 展示了页换出算法的整体流程.

算法 2. 页换出算法.

① function *shrink_page_list*(页面链表)
② while(页面链表非空)
③ *page*←获得页面链表中的 1 个页面;
④ if(页面不在交换缓存中)
⑤ 分配页面交换槽位;
⑥ 解除页面映射;
⑦ *swap_writepage*(*page*);
⑧ end if
⑨ if(页面完成回写)
⑩ 直接释放页面;
⑪ end if
⑫ end while
⑬ end function
⑭ function *swap_writepage*(*page*)
⑮ *bio*←生成块层页面 I/O 信息, 记录回调函数 *end_write_func*;
⑯ *submit_bio*(*bio*);
⑰ end function

对于链表中的页面, 如果没有被保存在交换缓存中, 则需要对该页进行回收操作(算法 2 的行④～⑥). 内核会为该页面分配交换分区的空闲槽位, 并将其加入交换缓存. 函数 *add_to_swap* 根据交换分区的槽位位图, 为换出页选择一个空闲槽位, 并将交换分区和槽位信息保存在该页面的结构体中, 便于页换入时查找页面. 内核在将页面放入交换缓存且为该页面分配槽位之后, 会解除该页面与用户空间的映射并调用函数 *swap_writepage* 向块层发送写请求.

扫描匿名页表的第 2 个目的是回收页框(算法 2 的行⑨～⑪). 对于没有回写完成的页面, 内核会继续将其保留在不活跃链表中. 对于已经回写完成的页面, 内核将释放页框, 释放该页面的锁, 完成页面回收流程.

在函数 *swap_writepage* 中, 与页换入机制类似, 内核首先为需要进行 I/O 的页面分配 *bio* 结构体, 调用函数 *submit_bio* 将其向 Linux 的块层发送. 页换出机制在块层和设备驱动层的操作与页换入机制相同.

2.3 页交换机制测试

结合 2.1 节对页换入路径与 2.2 节对页换出路径的描述, 表 2 对页换入路径的操作进行整理, 并对每个操作的内核时间占比进行统计.

表 2 的测试结果表明, 在使用超低延迟 SSD 的情况下, 页换入路径上多队列块层的时间可达到整

Table 2 Performance Analysis of Swapping Mechanism in Read Path

表 2 页换入机制性能分析

层级	操作	内核时间占比/%	总计/%
交换逻辑层	查找交换缓存	0.5～1.5	24～25
	生成页表项、建立映射	1.5～8	
	查找、分配交换缓存	11～17.5	
	刷新链表	3.5～5.5	
多队列块层	块层蓄流机制	2.5～14	55～64
	提交读请求	24～40	
	中断完成处理	12.5～26	
设备驱动层	设备 I/O	12～20	12～20

注：Linux 中预取范围为 0～8 页.

体内核时间的 64%, 是设备延迟的 5.2 倍. 表 3 对页换出路径的操作进行分析. 测试结果表明, 页换出路径上多队列块层的内核时间占比达到 56.9%, 是设备 I/O 延迟的 5.0 倍. 以执行 kmeans 应用为例, 该应用使用 sklearn 库对 1 500 万个样本进行分类. 在全内存的情况下, 应用执行完成的时间为 238 s. 当可使用内存为所需内存的 60% 时, 应用执行完成的时间增加至 527 s, 增加的时间即为执行页交换机制的时间. 其中, 块层所占时间为 162 s, 占页交换机制执行时间的 56%.

Table 3 Performance Analysis of Swapping Mechanism in Write Path

表 3 页换出机制性能分析

层级	操作	内核时间占比/%	总计/%
交换逻辑层	获取链表页面	0.6	31.8
	分配交换缓存	5.5	
	解除映射	16.3	
	释放页框	9.4	
多队列块层	提交写请求	28.0	56.9
	中断完成处理	28.9	
设备驱动层	设备 I/O	11.3	11.3

对于页换入路径, 主要问题在于 Linux 页交换机制的队头阻塞问题、缺页异常的关键路径上的中断处理过程, 以及 I/O 合并和调度机制导致内核软件开销过大.

首先, 内核根据需要预取的页面数量(Linux 中默认预取 8 页), 将整个内核空间地址按照特定的聚集(cluster)进行划分. 当确认预取页面范围时, 内核根据关键页面的虚拟地址, 选择包含该页面的 cluster, 将该 cluster 中的页面按照地址从小到大的顺序发送读请求. 然而, 这产生了一个问题, 触发缺页异常的关键页面可能在这个 cluster 的任何位置. 与此同时, 现在 Linux 中的多队列块层为每个 CPU 只提供了 1 个队列[21], 关键页的请求可能排在预取页的请求之后, 产生队头阻塞问题, 因而缺页的读请求可能需要等待几十微秒才会被发送[13].

其次, 现在 Linux 中的多队列块层为每个 CPU 提供一个中断队列, 即当 I/O 完成后会发送中断信号来通知 I/O 完成. 因为缺页异常返回前需要保证关键页的 I/O 完成, 因此中断会发生在缺页异常的关键路径上, 为缺页异常的整体过程增加了 13%～26% 内核时间开销.

此外, 内核会将每个页面的读请求加入到蓄流队列. 在所有页面的请求发出之后, 再集中泄流到驱动队列. 多队列块层的合并操作虽然可以减少 I/O 操作的总数, 但是合并过程需要消耗许多 CPU 周期来搜索软件队列, 进一步延长了关键页读请求的等待时间开销.

与页换入路径类似, 页换出机制在块层同样值得优化. 然而, 由于页换出路径部分并不需要区分关键页面与其他页面. 因此, 我们可以考虑去除 I/O 合并和调度机制, 实现更轻量级的写请求. 值得一提的是, 在触发缺页异常的页面需要进行内存分配时, 会对空闲内存的情况进行检查. 如果空闲内存不足, 内核会进行直接页面回收, 影响缺页异常的返回, 延长应用的等待时间.

3 Ultraswap 设计

本节介绍基于超低延迟 SSD 开发的内核页交换机制 Ultraswap, 该机制包括一个轻量级的 I/O 栈, 该 I/O 栈将 Linux 中的块层进一步简化, 进而降低页交换机制的软件开销. 此外, 本节还将介绍 Ultraswap 对于交换路径的优化工作. 图 3 展示了 Ultraswap 的整体设计图.

Ultraswap 主要进行 2 方面改进:

1) 在 I/O 栈部分, 首先取消了 I/O 合并和调度机制. 读请求会被直接发送到驱动层进行设备 I/O, 从而降低请求的等待时间. 其次, 考虑到 Linux 块层的队头阻塞问题, Ultraswap 的 I/O 栈实现了对中断和轮询请求的处理. 一方面, 中断与轮询请求使用 2 个不同的队列, 可以避免队头阻塞问题. 另一方面, 考虑到中断对关键路径的影响, 使用轮询请求的方式可以降低关键路径上的中断开销.

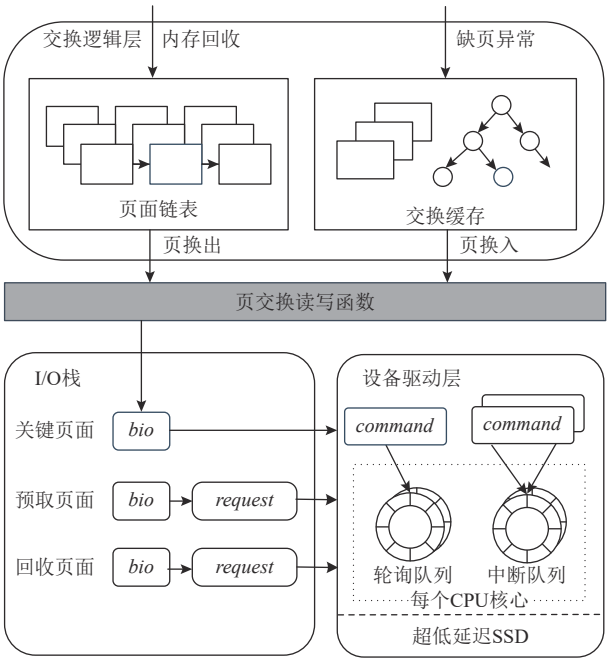


Fig. 3 Overview design drawing of Ultraswap

图 3 Ultraswap 整体设计图

2) 在交换路径部分主要进行了 2 部分优化. 首先, 在页换入过程中, 区分触发缺页异常的关键页面与预取页面, 使用轮询请求处理关键页面, 并优先发送关键页面的读请求. 另一方面, 增加对关键路径上直接内存回收的处理. 考虑到内存不足时直接内存回收的处理会影响缺页异常的完成, 将直接内存回收提交给一个专门的 CPU 进行处理, 从而降低应用的等待时间.

3.1 I/O 栈设计

在 Linux 页交换机制中, 所有页的读写请求都会通过多队列块层被放入当前 CPU 对应的中断队列. 对 Ultraswap 而言, 想要实现轻量级的 I/O 栈, 首先需要解决 Linux 中的 I/O 合并和调度问题, 其次需要实现对轮询请求的支持.

图 4 展示了 Ultraswap 的 I/O 栈中对请求的发送流程图. 图 4 箭头①为 Linux 的通用块层的流程. 当页交换机制希望提交 I/O 请求时, 会先构造 *bio* 结构体. *bio* 结构体会首先与队列中已有的 *request* 结构体进行合并, 如果不能合并则生成新的 *request* 结构体. 队列中的 *request* 结构体经过调度之后, 被放入当前 CPU 的软件队列和硬件队列. 之后, *request* 结构体转变为 *command* 结构体放入对应 NVMe 中断队列, 进行设备 I/O.

由于利用超低延迟 SSD 进行存储的 I/O 调度程序要么无济于事, 要么显著增加请求延迟, 同时还会

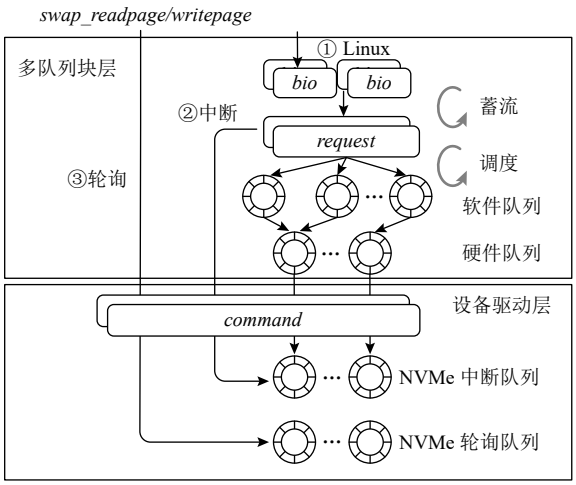


Fig. 4 Overview of request sending in Ultraswap

图 4 Ultraswap 请求发送流程图

对吞吐量和能效产生负面影响[16]. 因此, 我们考虑直接由 *bio* 结构体构造 *request* 结构体, 并最终直接构造 *command* 结构体放入驱动队列, 进行设备 I/O, 从而尽可能地减少内核软件开销(图 4 箭头②).

事实上, 对于 Ultraswap 中的中断情况, 在设备 I/O 完成进入中断处理函数时, 会根据完成中断的请求找到其 *request* 结构体, 并对 *request* 中的每个 *bio* 结构体调用请求处理函数. 因此, 在发送中断请求的函数中, Ultraswap 需要保留 *bio* 结构体和 *request* 结构体, 仅仅去掉不必要的 I/O 合并和调度操作.

对于轮询请求, 则可以使用更为简单的方式. 轮询完成的处理过程并不需要使用 *request* 结构体和 *bio* 结构体. 因此, 轮询请求函数可以根据需要进行读写的页面地址, 直接构造 *command* 结构体向 NVMe 轮询队列发送请求(图 4 箭头③).

针对上述描述, Ultraswap 的 I/O 栈提供了 3 种请求函数, 图 5 为这 3 种函数的示意图. 图 5(a)为轮询请求. 当请求发送到设备后, CPU 会一直进行轮询, 直到设备 I/O 完成. 图 5(b)为中断请求处理过程, 在发送请求之后, 如果 I/O 完成, 内核会收到中断信号并进入中断处理过程. 考虑到轮询处理过程会大量占用 CPU, 因而图 5(c)将轮询请求的发送过程和轮询过程进行拆分, 使得在发送轮询请求后可以进行其他步骤的处理后再轮询等待 I/O 完成.

3.2 交换路径优化设计

在处理缺页异常的过程中, 主要会碰到 4 个问题:

1) 页面的读请求在 NVMe 队列中会按地址顺序处理, 由于每个 CPU 只有一个队列, 这导致关键页的读请求前可能存在其他预取页的请求, 从而导致队头阻塞问题.

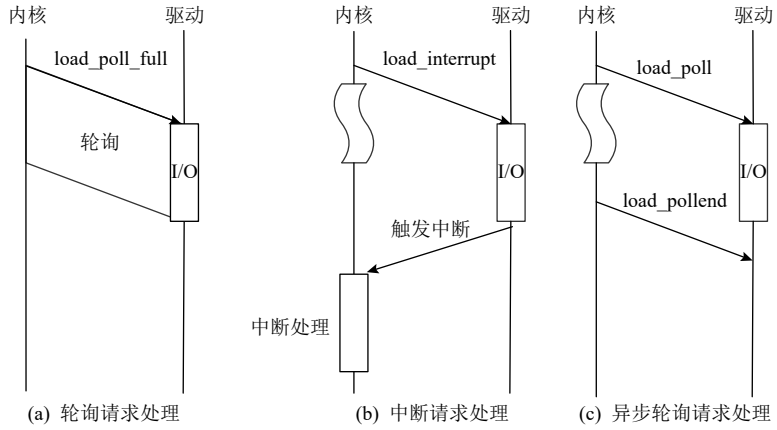


Fig. 5 Illustration of Ultraswap API functions

图 5 Ultraswap 接口函数示意图

2) Linux 的页交换机制存在 I/O 合并和调度操作，导致关键页的读请求可能会被合并或调度到别的请求之后，这个问题已经在 Ultraswap 的 I/O 栈实现中被优化.

3) 现在的 Linux 块层队列均为中断队列，在缺页异常完成之前需要处理关键页的 I/O 中断，延后了缺页异常返回的时间.

4) 在关键页处理完成之后，内核需要进行空闲内存检查. 如果空闲内存不足，需要进行直接内存回收操作，待内存回收操作结束后才会继续执行应用.

图 6(a)展示了 Linux 页交换机制对缺页异常的处理过程. 当触发缺页异常，内核会按照地址顺序依此发送读请求，并在块层进行蓄流. 请求会在蓄流结束后被送到设备进行 I/O.I/O 完成后，设备通过触发中断令 CPU 对请求进行处理. 如果此时空闲内存不足，Linux 还将会进行直接页面回收. 在上述步骤完成后，Linux 页交换机制才会从内核态返回用户态，继续执行应用.

针对上述问题，Ultraswap 的交换路径优化设计如下：

首先，在缺页异常的过程中，为了解决队头阻塞问题，Ultraswap 选择给每个 CPU 分配 2 个队列，用于

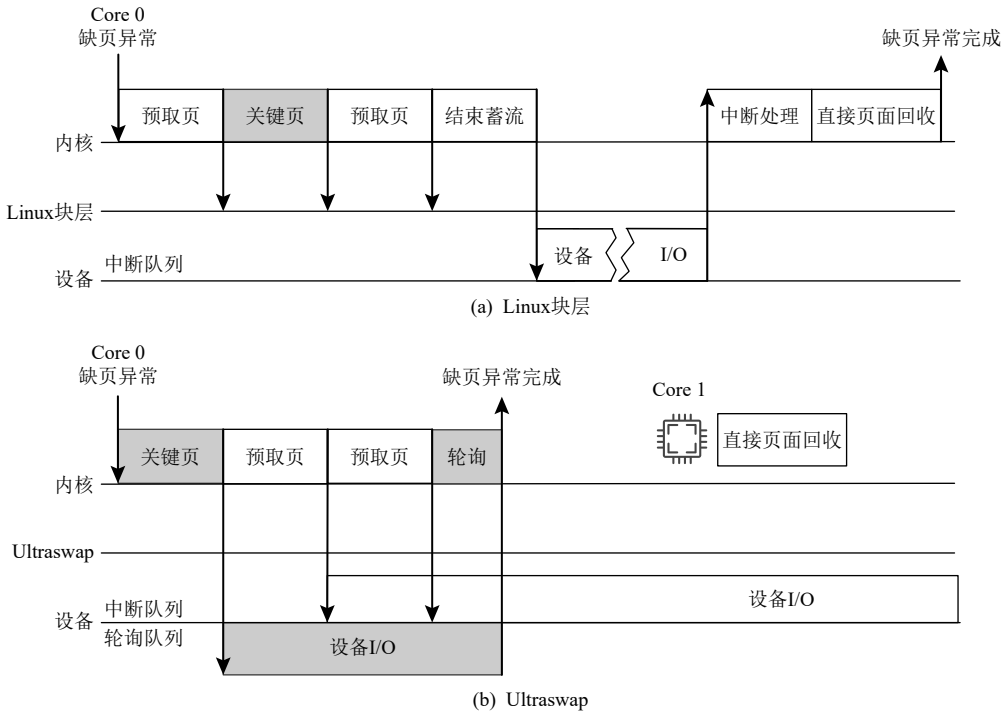


Fig. 6 Overview of page fault and prefetching process

图 6 缺页异常和预取流程示意图

分别处理关键页和预取页. 同时，使用 2 个队列分别处理关键页和预取页使得 Ultraswap 能够使用不同的方法处理关键页和预取页. 考虑到在缺页异常返回前，需要保证关键页的 I/O 完成，且关键页的 I/O 完成越快越好，因此 Ultraswap 使用轮询的方式处理关键页，这样可以减少缺页异常的关键路径上中断带来的影响. 对于预取页，在缺页异常过程中并不需要保证预取页的 I/O 完成，因此 Ultraswap 采用中断的方式处理预取页的读请求.

其次，通过修改内核，本文对关键页和预取页进行分别处理. 当进入函数 *swapin_readahead* 时，会首先发送关键页的轮询请求至设备的 NVMe 驱动队列. 接下来对于所有需要预取的页面，内核会依次发送对应该页面的中断请求. 当发送完中断请求之后，内核会进行轮询操作，等待关键页的 I/O 完成，如图 6(b)所示. 通过先发送关键页的读请求，我们可以在设备读入关键页的同时，进行预取页的页框分配和发送预取页的读请求，从而可以充分利用设备的 I/O 时间. 事实上，在我们的实验中发现，绝大多数轮询请求的 I/O 在发送完成预取请求之后已经完成，因此对 CPU 开销的影响可以忽略不计.

对于页面回收过程中 Ultraswap I/O 栈请求函数的选择，由于内核可以将 I/O 未完成的页面重新加入页面链表，选择将上一次页面回收过程中被加入页面链表的页框释放，并不需要等待本次回收页面的 I/O 完成. 因此，对于回收页面，Ultraswap 同样采用中断的方式发送请求.

此外，在缺页异常的处理过程中读取页面后，内核会检查内存是否超过内存水位线. 如果有多余的页面，页面将被直接回收. 直接内存回收发生在缺页异常的上下文中，造成相当一部分的时间开销. 因此本文将直接页面回收过程指定给一个专门的 CPU.当内存不足时，内核会唤醒指定 CPU 来处理页面回收，从而当前 CPU 可以返回用户态继续执行应用. 本文设置的内存水位线为内存限制的 70%. 如果使用的内存超过了内存限制，则会直接在关键路径上实现内存回收，而不进行迁移. 图 6(b)展示了 Ultraswap 对缺页异常优化后的处理过程. 当触发缺页异常，内核首先会将关键页的轮询请求直接发送到设备的轮询队列. 接下来，内核会依次将预取页的请求发送到设备的中断队列. 当预取页的请求发送完成，内核会对轮询队列进行轮询操作，直到关键页的请求完成. 随后，如果空闲内存不足，Ultraswap 会调用另一个 CPU 进行页面回收操作. 在上述步骤完成后，Ultraswap 即可从内核态返回用户态，继续应用的执行.

4 测评与分析

本节对 Ultraswap 进行性能测评. 首先测试 Ultraswap 的 I/O 接口函数性能，然后对 Ultraswap 进行应用场景测试. 近年来，一些工作[12−15]利用新型高速设备提出页交换机制的优化，这些工作主要是利用 RDMA 技术，但是没有工作针对超低延迟 SSD 进行优化. 考虑到 RDMA 与超低延迟 SSD 使用的 I/O 栈差异较大，因此本文选择与 Linux 页交换机制进行比较. 最后针对在缺页异常和内存回收方面提出的优化方法进行评测，评估其有效性.

4.1 测试环境

本文测试使用 1 台带有英特尔 Optane SSD 的服务器进行页交换机制性能测试，其硬件指标如表 4 所示.

Table 4 Test Environment for Swapping Mechanism

表 4 页交换机制测试环境

设备	型号
CPU	Intel Xeon Gold 6226 CPU 2.70 GHz
DRAM	Samsung DDR4 16GB × 4
存储设备	375GB Intel Optane P4800x
操作系统	CentOS Linux release 7.9.2009
内核版本	Linux 5.4.161

为了更灵活地控制内存使用以及触发页交换机制，本文使用了 Linux 控制组(control groups, cgroups). cgroups 可以用来限制和控制一个进程组的资源，包括 CPU、内存和设备 I/O 等. 本文使用 cgroups 对测试过程中可使用的物理内存进行限制，超过物理内存的部分会通过页交换机制被放入交换分区.

4.2 读写性能测试

本节主要对 Ultraswap 的 I/O 栈接口函数进行性能测试，并将其与 Linux 中的页交换读写函数进行比较. 此外，利用 fronswap 接口，本文还实现了以 DRAM 为存储设备的 I/O 栈，该 I/O 栈通过直接进行 memcpy 操作将页面交换到内存中不被 cgroups 限制的另一块区域. 这种方式将 I/O 需要的软硬件开销降至最低，因此本文将以 DRAM 为存储设备的页交换方式作为页交换的理想化情况并进行测试.

本节分 3 步对 Ultraswap(轮询)、Ultraswap(中断)和 DRAM 这 3 种 I/O 栈进行读写测试. 1)分配一个 100 MB 的 char 类型数组，利用 cgroups 工具将可使用

的物理内存限制到 50 MB. 2)对这个数组的每一项先写入一遍字符, 触发页换出过程. 3)遍历数组的每一项, 触发页换入过程并进行正确性验证. 该测试均在单线程环境下完成, 统计每部分操作的延迟.

图 7 展示了在 3 种 I/O 栈中发送读/写请求的延迟. 测试结果表明, 在单线程环境下, 2 种 Ultraswap 的轮询请求处理过程相比 Linux 在读/写操作上分别提升了 78% 和 72%. 原因在于 Ultraswap 发送轮询请求的过程中可以完全绕过块层, 直接通过 NVMe 驱动向设备发送 I/O 请求. 此外, 当 CPU 轮询到请求完成时, 可以直接调用请求完成处理函数, 减少了中断和上下文切换带来的时间开销. 2 种 Ultraswap 的中断请求处理过程相比 Linux 在读/写上仅提升了 18% 和 13%. 原因在于: 一方面, Ultraswap 仅对发送中断请求的部分进行优化; 另一方面, 虽然 Ultraswap 在发送请求的过程中减少了 Linux 块层中的 I/O 合并和调度操作, 但是为了保证中断处理, 仍然需要分配 *bio* 和 *request* 结构体, 造成额外开销.

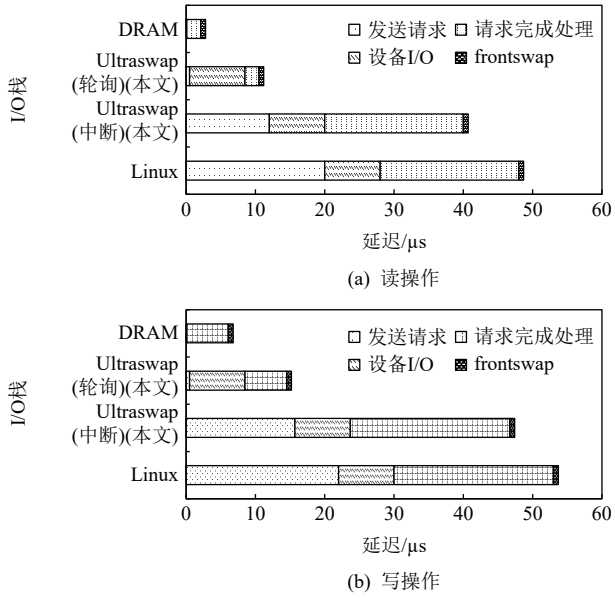


Fig. 7 Results of read and write performance

图 7 读写性能测试结果

4.3 应用负载测试

本节对 4.2 节中 3 种 I/O 栈进行性能测试. 测试包括 5 个应用, 分别为 quicksort, kmeans, tensorflow, pagerank, linpack. quicksort 应用使用 C++标准库对 8 GB 的整数进行快速排序; kmeans 应用使用 sklearn 库对 1 500 万个样本进行分类; tensorflow 应用对用于基准测试的 inception 样例进行计算; pagerank 应用使用包含 65.5 万个节点和 750 万条边的数据集进行测试; linpack 应用是一个线性代数性能基准测试, 使用英特尔提供的二进制文件. 表 5 展示这 5 个应用需要使用的内存和核数.

Table 5 Basic Information of Application

表 5 应用基本情况

应用	内存/GB	核数
quicksort	8.2	1
kmeans	2.9	1
tensorflow	2.1	2
pagerank	4.8	3
linpack	1.8	4

根据应用需要的内存上限, 将可使用的内存比例限制到 100%, 80%, 60%, 40%, 20%, 记录应用执行完成的时间, 多次执行并计算平均时间. 当可使用的内存比例为 100% 时, 应用不触发页交换机制. 则当对内存比例进行限制, 增加的应用完成时间可以认为是页交换机制的执行时间. 本文将可使用内存比例为 100% 的应用平均执行时间作为基准时间, 对不同内存比例下的应用执行时间进行标准化处理.

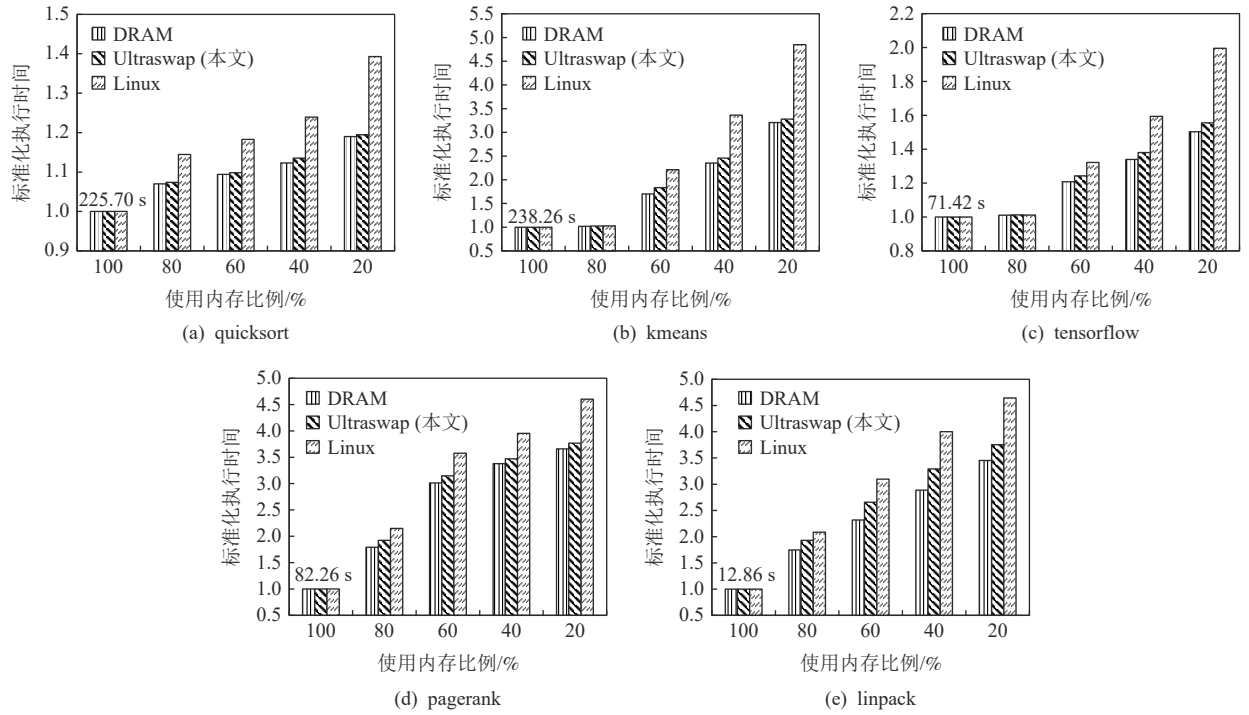
图 8 展示 3 种 I/O 栈的应用测试结果, 当可使用内存比例为 100% 时应用不触发页交换机制, 我们将此时的应用执行时间标注在图上. 测试结果表明, 在 5 种应用上 Ultraswap 均表现出较好的页交换性能. quicksort, kmeans, tensorflow, pagerank, linpack 应用的平均性能提升为 9%, 19%, 10%, 13%, 15%. 当可使用内存比例为 20% 时, 这 5 个应用可达到 15%, 33%, 22%, 18%, 19% 的性能提升. 与理想化情况相比, Ultraswap 将 Linux 页交换机制与理想情况的差距缩小至 5%, 25%, 18%, 22%, 40%, 这得益于 Ultraswap 的 I/O 栈和交换路径优化设计.

4.4 优化有效性测试

4.4.1 缺页异常优化测试

Ultraswap 中对触发缺页异常的关键页面使用轮询请求, 对预取页面和回收页面使用中断请求. 本节对使用纯中断 Ultraswap-interrupt-only 和纯轮询请求 Ultraswap-poll-only 的 2 种 I/O 栈进行性能测试, 并与 Ultraswap 的优化方案进行比较, 以说明 Ultraswap 在缺页异常方面优化的合理性和有效性.

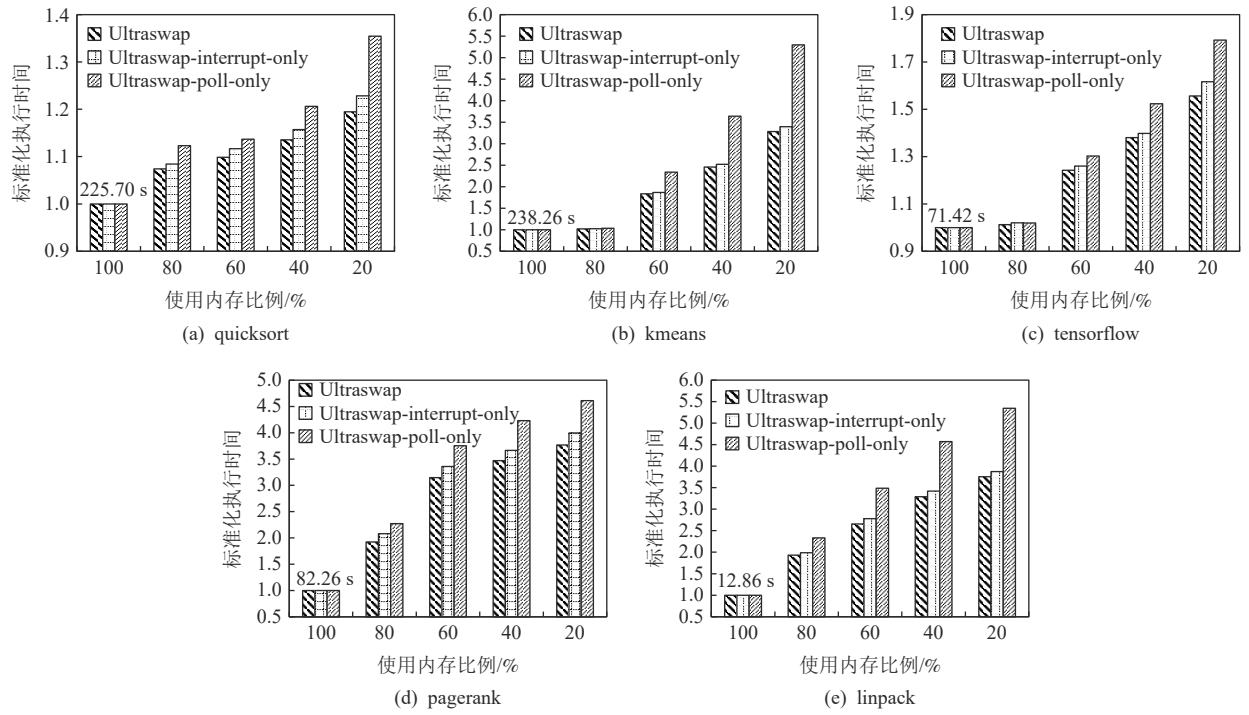
图 9 的测试结果表明, 与 Ultraswap 相比, 在 quicksort, kmeans, tensorflow, pagerank, linpack 应用上 Ultraswap-interrupt-only 的时间分别增加了 2%, 2%, 2%, 7%, 4%, 主要原因在于轮询请求减少了关键路径上的中断处理和上下文切换时间, 且 CPU 处理完请求后即



注：使用内存比例 100% 时所标注的时间为该条件的应用执行时间，且其余条件下的应用执行时间按该时间进行标准化.

Fig. 8 Results of application performance tests

图 8 应用性能测试结果



注：使用内存比例 100% 时所标注的时间为该条件的应用执行时间，且其余条件下的应用执行时间按该时间进行标准化.

Fig. 9 Optimization results of Ultraswap for page fault handling

图 9 Ultraswap 缺页异常优化结果

可返回应用.

Ultraswap-poll-only 的应用性能则更差, 在 quicksort, kmeans, tensorflow, pagerank, linpack 应用的时间分别增加了 6%, 35%, 7%, 20%, 33%. 与受内存限制影响

较小的应用 quicksort 和 tensorflow 相比，受内存限制影响较大的应用 kmeans, pagerank, linpack 受到纯轮询的 I/O 栈影响更为明显. 本文认为主要原因在于内核的内存回收机制更倾向于使用异步操作而不是同步操作. 首先，在内存回收过程中，如果是异步操作，那么在写请求发送完成后，页面会在不活跃页面链表上挂起；等到第 2 次进入内存回收过程时，如果该页面的 I/O 请求已经完成，则内存可以直接被回收，因而页面链表上始终存在等待请求完成的页面. 而对于同步操作，页面链表上的每个可以回收的页面会在发送写请求后，等待设备 I/O 完成并且释放页框，因而在内存回收之后不活跃页面链表上的页面大量减少，从而会从活跃链表迁移更多的页面进入不活跃链表. 总而言之，纯轮询的 I/O 栈会导致更多的页面交换，从表 6 可以看出，在 linpack 应用执行过程中，Ultraswap-poll-only 相比 Ultraswap 会触发 1.15 倍的缺页异常次数.

Table 6 Number of Page Faults in Linpack with Ultraswap's I/O Stack

表 6 Ultraswap I/O 栈在 Linpack 应用中触发缺页异常次数

内存比例/%	Ultraswap	Ultraswap-poll-only
80	11.9×10^4	12.5×10^4
60	24.1×10^4	27.1×10^4
40	32.4×10^4	37.0×10^4
20	40.0×10^4	46.3×10^4

4.4.2 内存回收优化测试

本节测试 Ultraswap 中对内存回收路径的优化效果. 本测试将应用的可使用内存比例设置为 50%，对迁移直接页面回收过程前后的内核时间占比进行记录和统计，从而评估对内存回收路径的优化效果. 表 7 显示，优化页面回收路径可以显著减少内核时间，使内核时间减少多达 35.3%.

Table 7 Performance of Memory Reclaimation Optimization

表 7 内存回收优化性能 s

应用	优化前内核时间	优化后内核时间
quicksort	52.3	37
kmeans	413	267
tensorflow	33.2	28.5
pagerank	103	82.3
linpack	95	87.5

5 总 结

页交换机制是用于内存扩展的经典技术，该机制通过将较少使用的内存页面保存在存储设备，从而达到扩展内存的目的. 过去页交换机制一直受限于慢速磁盘的读写速度，因而无法广泛应用. 近年来，随着超低延迟 SSD 的快速发展，页交换机制可以利用其低延迟的读写特性来改善页交换效率.

本文首先对 Linux 页交换机制的页换入和页换出路径进行分析和评测. 测试结果表明，在使用超低延迟 SSD 的情况下，Linux 块层和设备驱动层的软件开销可达到内核时间开销的 64%. 主要原因在于块层中存在队头阻塞、I/O 合并和调度开销，以及关键路径上的中断时间开销问题. 同时，当触发缺页异常需要分配内存空间时，如果当前内核的空闲内存不足，会触发直接内存回收，同样会在缺页异常的关键路径上造成额外的时间开销.

基于上述问题，本文设计了内核页交换机制 Ultraswap.Ultraswap 通过直接使用 NVMe 驱动、避免 I/O 合并和调度的方式实现了轻量级的 I/O 栈，能够处理读写函数的轮询和中断请求. 在内核方面，Ultraswap 从缺页异常和内存回收 2 个方面进行优化，通过拆分请求队列，混合使用轮询和中断请求、转移直接内存回收的方式，降低关键路径上的时间开销.

本文将 Ultraswap 与 Linux 页交换机制进行对比来比较性能，结果表明 Ultraswap 在应用测试场景下相比 Linux 能够提升 19% 的平均性能；与理想化情况相比，Ultraswap 能够将 Linux 与理想化情况的性能差距减少 95%.

作者贡献声明：王紫芮提出文章思路、方案设计并撰写论文；蒋德钧提出指导意见并修改论文.

参 考 文 献

[1] Lee S H. Technology scaling challenges and opportunities of memory devices [C/OL] //Proc of the 62nd IEEE Int Electron Devices Meeting. Piscataway, NJ: IEEE, 2016[2023-12-25].https://ieeexplore.ieee.org/document/7838026

[2] Harris B, Altiparmak N. Ultra-low latency SSDs' impact on overall energy efficiency [C/OL] //Proc of the 12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'20). Berkeley, CA: USENIX Association, 2020[2023-12-25].https://www.usenix.org/system/files/hotstorage20_paper_harris.pdf

[3] Intel. 3D XPoint: A breakthrough in non-volatile memory technology [EB/OL]. 2023[2023-12-25].https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html?wapkw=3D-Xpoint

[4] Samsung. Ultra-low latency with Samsung Z-NAND SSD [EB/OL]. 2017[2023-12-25].https://download.semiconductor.samsung.com/resources/brochure/Ultra-Low%20Latency%20with%20Samsung%20Z-NAND%20SSD.pdf

[5] Intel. Intel Optane DC SSD series [EB/OL]. 2023[2023-12-25].https://www.intel.com/content/www/us/en/products/details/memory-storage/data-center-ssds/optane-dc-ssd-series.html

[6] Samsung. The ultra-low latency SSD, Z-SSD [EB/OL]. 2018[2023-12-25].https://semiconductor.samsung.com/newsroom/tech-blog/the-ultra-low-latency-ssd-z-ssd

[7] Lee G, Shin S, Song W, et al. Asynchronous I/O stack: A low-latency kernel I/O stack for ultra-low latency SSDs [C] //Proc of the 2019 USENIX Annual Technical Conf (ATC'19). Berkeley, CA: USENIX Association, 2019: 603−616

[8] Zhang Jie, Kwon M, Gouk D, et al. FlashShare: Punching through server storage stack from kernel to firmware for ultra-low latency SSDs [C] //Proc of the 13th USENIX Symp on Operating Systems Design and Implementation (OSDI'18). Berkeley, CA: USENIX Association, 2018: 477−492

[9] Mitchell C, Geng Yifeng, Li Jinyang. Using one-sided RDMA reads to build a fast, CPU-efficient key-value store [C] //Proc of the 2013 USENIX Annual Technical Conf (ATC'13). Berkeley, CA: USENIX Association, 2013: 103−114

[10] Saxena M, Swift M M. FlashVM: Virtual memory management on flash [C/OL] //Proc of the 2010 USENIX Annual Technical Conf (ATC'10). Berkeley, CA: USENIX Association, 2010[2023-12-25]. https://dl.acm.org/doi/abs/10.5555/1855840.1855854

[11] Fedorov V, Kim J, Qin Mian, et al. Speculative paging for future NVM storage [C] //Proc of the 2017 Int Symp on Memory Systems. New York: ACM, 2017: 399–410

[12] Gu Juncheng, Lee Y, Zhang Yiwen, et al. Efficient memory disaggregation with Infiniswap [C] //Proc of the 14th USENIX Symp on Networked Systems Design and Implementation (NSDI'17). Berkeley, CA: USENIX Association, 2017: 649−667

[13] Amaro E, Branner-Augmon C, Luo Zhihong, et al. Can far memory improve job throughput [C/OL] //Proc of the 15th European Conf on Computer Systems (EuroSys'20). New York: ACM, 2020[2023-12-25]. https://dl.acm.org/doi/pdf/10.1145/3342195.3387522

[14] Maruf H A, Chowdhury M. Effectively prefetching remote memory with Leap [C] //Proc of the 2020 USENIX Annual Technical Conf (ATC'20). Berkeley, CA: USENIX Association, 2020: 843−857

[15] Wang Chenxi, Qiao Yifan, Ma Haoran, et al. Canvas: Isolated and adaptive swapping for multi-applications on remote memory [C] //Proc of the 20th USENIX Symp on Networked Systems Design and Implementation (NSDI'23). Berkeley, CA: USENIX Association, 2023: 161−179

[16] Harris B, Altiparmak N. When poll is more energy efficient than interrupt [C] //Proc of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'22). New York: ACM, 2022: 59–64

[17] Whitaker C, Sundar S, Harris B, et al. Do we still need IO schedulers for low-latency disks [C] //Proc of the 15th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'23). New York: ACM, 2023: 44−50

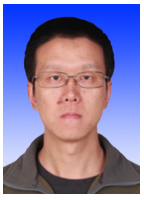
[18] Tu Yaofeng, Han Yinjun, Jin Hao, et al. UStore: Unified storage system for advanced hardware[J]. Journal of Computer Research and Development, 2023, 60(3): 525−538(in Chinese)
(屠要峰, 韩银俊, 金浩, 等. UStore: 面向新型硬件的统一存储系统[J]. 计算机研究与发展, 2023, 60(3): 525−538)

[19] Tavakkol A, Sadrosadati M, Ghose S, et al. Flin: Enabling fairness and enhancing performance in modern NVME solid state drives [C] //Proc of the 45th Annual Int Symp on Computer Architecture (ISCA'18). Piscataway, NJ: IEEE, 2018: 397−410

[20] Lee G, Jin W, Song W, et al. A case for hardware-based demand paging [C] //Proc of the 47th Annual Int Symp on Computer Architecture (ISCA'20). Piscataway, NJ: IEEE, 2020: 1103−1116

[21] Bjørling M, Axboe J, Nellans D, et al. Linux block IO: Introducing multi-queue SSD access on multi-core systems [C/OL] //Proc of the 6th Int Systems and Storage Conf. New York: ACM, 2013[2023-12-25].https://kernel.dk/systor13-final18.pdf

Wang Zirui, born in 2000. PhD candidate. Her main research interest includes storage system.
王紫芮, 2000 年生. 博士研究生. 主要研究方向为存储系统.



Jiang Dejun, born in 1982. PhD, associate professor, PhD supervisor. Member of CCF, ACM, IEEE. His main research interests include storage architecture, storage system, and distributed system.
蒋德钧, 1982 年生. 博士, 副研究员, 博士生导师. CCF, ACM, IEEE 会员. 主要研究方向为存储体系结构、存储系统、分布式系统.