计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
63(9): 2463−2477, 2026

一种面向分离式内存的用户态异步数据通路

李权熹　崔慧敏　冯晓兵　王晨曦
（中国科学院计算技术研究所　北京　100190）
（中国科学院大学 北京 100049）
（liquanxi20z@ict.ac.cn）

A Userspace Asynchronous Data Path for Disaggregated Memory

Li Quanxi, Cui Huimin, Feng Xiaobing, and Wang Chenxi
（*Institute of Computing Technology*, *Chinese Academy of Sciences*, *Beijing* 100190）
（*University of Chinese Academy of Sciences*, *Beijing* 100049）

Abstract　Disaggregated memory architectures are rapidly emerging as a significant trend for next-generation data centers, promising to enhance resource utilization and scalability. Existing system software often employs rapid context switching between lightweight threads to generate concurrent remote data access requests, thereby masking the microsecond-level latency inherent in disaggregated memory. However, this approach introduces substantial thread scheduling and communication overhead, consuming a significant amount of CPU resources that would otherwise be dedicated to computation and thus creating a new performance bottleneck. To address this challenge, a userspace asynchronous data path for disaggregated memory is proposed. The proposed data path focuses on data prefetching and eviction, eliminating the high-overhead multi-threaded concurrency model in favor of an asynchronous I/O mechanism to achieve the overlap of computation and network I/O in fine-grained tasks. By combining adaptive batching with dynamic thread management strategies, CPU overhead is minimized while ensuring the timeliness and concurrency of remote memory requests, thereby effectively hiding the latency of disaggregated memory access. A prototype system is implemented on top of AIFM and evaluated with four typical application workloads. Experimental results show that, compared with AIFM, the proposed approach reduces the CPU overhead of data prefetching and eviction by 73% while simultaneously increasing the average end-to-end application throughput by 38%.

Key words　disaggregated memory; asynchronous I/O; scheduling system; runtime system; data center

摘　要　分离式内存架构正在迅速成为下一代数据中心，有望提高资源利用率和可扩展性。现有系统软件往往采用轻量级线程的高速切换以生成并发的远端数据访问请求，来掩盖分离式内存的微秒级延迟。然而，这一方法产生了巨大的线程调度和通信开销，消耗了大量本应用于计算的CPU资源，形成了新的性能瓶颈。针对这一挑战，设计了一种面向分离式内存的用户态异步数据通路。该通路面向数据预取和逐出，摒弃高开销的多线程并发模型，转而利用异步I/O机制实现细粒度任务中计算与网络I/O的交叠。通过结合自适应批处理与动态线程管理策略，最大限度地降低了CPU开销的同时保障了远端内存请求及时性和并发性，从而掩盖分离式内存访问延迟。基于AIFM实现了原型系统，并在4个典型应用负载下进行了评估。实验结果表明，与AIFM相比，该方法在将数据预取和逐出的CPU开销降低73%的同时，将端

收稿日期：2025-12-05；**修回日期**：2026-04-13
基金项目：国家重点研发计划项目（2024YFE0204100）
This work was supported by the National Key Research and Development Program of China (2024YFE0204100).
通信作者：王晨曦（wangchenxi@ict.ac.cn）

到端应用吞吐量平均提升了 38%。

关键词 分离式内存;异步 I/O;调度系统;运行时系统;数据中心

中图法分类号 TP316; TP393.0

DOI: 10.7544/issn1000-1239.202550871 **CSTR:** 32373.14.issn1000-1239.202550871

近年来,随着大数据与人工智能应用的广泛部署,内存容量与访问带宽已成为制约数据中心服务器性能的关键瓶颈。在传统的计算架构中,计算与内存资源紧密耦合,导致其弊端日益凸显,2 方面问题日益突出:其一,处理器与内存之间的性能差距持续扩大,“内存墙”问题愈发显著;其二,数据中心普遍存在“搁浅内存”(stranded memory)现象,大量服务器内存资源利用率低下,这不仅造成硬件资源的严重浪费,也显著提高了数据中心的总体拥有成本。

为应对上述挑战,分离式内存(disaggregated memory)架构应运而生。其中,基于 RDMA(remote direct memory access)技术或无损以太网的方案,因其能够直接复用当前数据中心广泛部署的网络硬件设施而备受关注,展现出巨大的应用潜力。该架构借助高速互联技术将内存资源从计算节点中分离,构建统一的内存池。计算节点可按需动态地挂载与访问远端内存资源,从而打破传统架构中计算与内存的固定绑定,显著提升了资源利用率、系统扩展性与部署灵活性。本工作关注基于用户态运行时的分离式内存系统,其被普遍认为是当前兼具高性能与高灵活性的主流实现路径。如图 1 所示,在这种架构下,数据以对象为粒度,根据其访问热度被分别存放于本地内存池(热数据)与远端内存池(冷数据),应用程序访问远端数据时需通过网络进行数据传输。

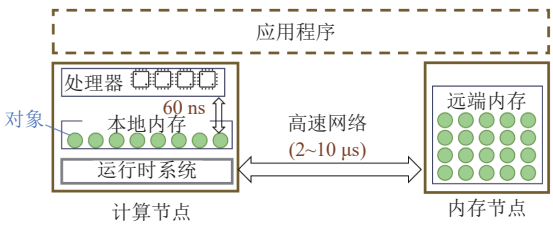


Fig. 1 User-level runtime based disaggregated memory system

图 1 基于用户态运行时的分离式内存系统

为了掩盖远端内存的访问延迟,现有远端内存数据通路往往使用大量轻量级用户态线程来产生大量并发的远端内存访问请求。图 2 展示了访问远端内存的 3 条数据通路。除了应用线程按需访问数据外,另外 2 条数据通路发生于运行时系统内的数据预取线程以及冷数据逐出线程。

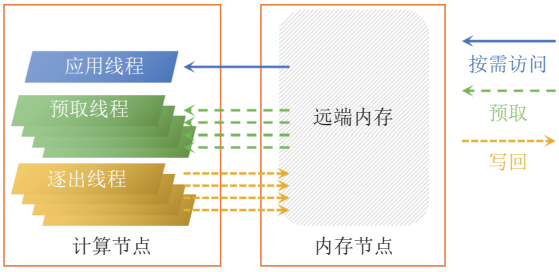


Fig. 2 Three data paths for remote memory access

图 2 远端内存访问的 3 个数据通路

1)预取线程。处理器访问远端内存池需要跨越网络,从而产生微秒级的延迟。为缓解这一访问延迟带来的影响,分离式内存系统会在后台对数据进行预取,通过预测应用程序未来的内存访问行为,在实际访问发生前将数据从远端内存池异步加载至本地,从而提升程序执行效率。

2)逐出线程。由于本地内存大小受限,分离式内存系统定期进行数据逐出。当本地内存池产生内存压力时,逐出线程会扫描本地内存池,区分数据冷热,随后将冷数据逐出到远端内存池并释放对应的本地内存。

以下将预取线程和逐出线程统称为分离式内存系统运行时中的服务线程。为了避免阻塞应用程序,服务线程通常在后台与应用并发执行。然而,在现有系统实现中,这些后台线程往往会产生巨大的 CPU 开销,限制了计算资源的有效利用。以代表性系统 AIFM[1] 为例,在运行访存密集型应用 KMeans 时,尽管在 CPU 资源充足的情况下,其预取线程和逐出线程能实现准确、及时的预取和逐出,但预取和逐出消耗的 CPU 资源可以分别达到 KMeans 应用线程的 187% 和 72%,造成了巨大的开销。究其原因,AIFM 为保证预取和逐出的及时性和高吞吐,采用多个后台线程并发完成访存分析、预取请求生成和数据逐出等工作(详见 1.2 节)。然而,我们发现,这种多线程并发的设计带来了显著的线程调度与通信开销。性能剖析表明,此类开销可占到服务线程总 CPU 开销的 45% 以上。

针对上述问题,本文提出一种面向分离式内存的用户态异步数据通路。本工作的核心目标是在保

障服务任务吞吐量和及时性的同时，最大限度地降低其 CPU 资源消耗，将主要的 CPU 资源让出给应用线程。具体而言，需要应对如下 2 个关键挑战：

挑战 1. 如何以最少的 CPU 消耗实现及时预取和逐出。我们发现，对于数据预取和逐出，执行的主要瓶颈在于网络 I/O 和线程调度，而非计算本身。单个预取和写回任务所需的 CPU 时间极短，若为每个任务调度独立线程，则线程管理开销过大。反之，若将所有预取或逐出任务集中于单一后台线程中批量处理，虽能节约 CPU 资源，但会延误请求的发出时机，且缺乏算力的可扩展性。

挑战 2. 如何动态调整服务线程数量，保证应用性能不受影响。若采用固定数量的后台服务线程，系统将难以适应负载波动。我们以预取为例：当应用程序对远端内存访问较少或访问模式难以预测时，预取器应尽可能节约计算资源；而在多个应用线程呈现高强度顺序访问时，则需多个预取线程并行工作以保证预取及时性。例如，在特定应用中，当配置多个并行计算线程并限制本地内存容量时，预取器需相应增加线程数以维持性能。

我们进一步分析可以发现，无论是预取还是逐出任务，均由大量独立且包含网络 I/O 的小任务组成：对于预取器，任务可以拆解为解析内存踪迹并生成预取、发出预取请求以及在请求完成后更新元数据 3 部分；对于逐出器的写回阶段，同样可以将任务拆分为发现待写回的脏对象、发出写回请求以及在请求完成后更新元数据 3 部分。同时，这些小任务每次执行时间在 1 μs 以内，且彼此之间存在紧密数据依赖。因此，本工作的核心设计思想是将相关联的细粒度任务聚合于同一线程中执行，并利用异步网络原语实现 I/O 与计算的重叠。为保障任务的及时性，本工作通过监控任务等待延迟，自适应地调节网络请求的批处理规模。同时，系统依据应用程序本地命中率或内存压力等指标进行压力反馈，动态调整服务线程数量。

本文的主要贡献包括以下 2 点：

1）提出一种基于异步 I/O 与自适应批处理的轻量级分离式内存数据通路，在保证预取和逐出任务的及时性前提下，显著降低 CPU 资源开销；

2）设计并实现一种自适应服务线程管理机制，实现任务的计算资源自动扩展与负载均衡。

我们在 AIFM 的基础上实现了原型系统，并在多个典型应用中进行了全面性能评估。实验结果表明，在 8 个 CPU 核心，本地内存配置为应用所用内存资源 50% 的场景下，相较于基准系统 AIFM，本工作平均将端到端应用吞吐量提升了 38%。本工作所提出的异步数据通路，亦可与其他基于用户态运行时的分离式内存系统[1-4]以及其他分离式内存预取策略[5]等相互补充，共同推动该架构的进一步发展。

1 研究背景和相关工作

1.1 分离式内存

分离式内存是一种新型的数据中心内存架构，其核心思想在于将内存资源从计算节点中物理分离，通过高速互联网络（如 InfiniBand[6]，CXL（compute express link）[7] 或无损以太网）构建一个独立的、可共享的内存池。计算节点可以按需动态地挂载和访问远端内存池中的数据。这种架构显著提升了资源的利用率和部署的灵活性，有效解决了传统服务器架构中普遍存在的“搁浅内存”问题。

目前，远端访存数据通路主要分为 3 类：基于硬件的[8-11]、基于内核页面交换的[12-15]以及基于用户态运行时的[1-4]。其中，以 CXL 为代表的硬件技术可以实现缓存行级别的低延迟访问，但相关技术尚未成熟，无法实现商业应用。基于页面交换的技术由于读写放大、用户态-内核态上下文切换和页表维护等原因，性能开销巨大且难以感知应用访问模式的语义。

本工作主要面向基于用户态运行时的分离式内存系统，如图 1 所示。其中，数据以对象为粒度进行管理。当应用程序访问对象时通过运行时屏障（barrier）检查对象位置，当对象未命中在本地时通过 InfiniBand，DPDK（data plane development kit）[16] 等用户态网络栈将数据取回到本地内存。访问远端对象一般会产生 2~10 μs 的延迟，相比本地内存（约 60 ns）有数量级的差距。此类系统中，预取和逐出服务执行在用户态，并以对象为粒度执行。因此可以更好地感知应用程序语义，如按照数据结构定制预取器策略。同时，细粒度执行能最大限度避免读写放大，但精细管理带来了更重的运行时负担。

1.2 AIFM 系统及其后台服务线程的实现

我们的工作基于 Ruan 等人[1] 提出的运行时分离式内存系统 AIFM，其设计具有代表性。为保证数据预取和逐出等后台服务的高吞吐与及时性，AIFM 采用了一种高并发的多线程处理模型。具体而言，AIFM 在后台维护着 2 类服务线程（分别用于预取和逐出），并为每类服务并发执行多达 16 个甚至更多的线程。其核心目标是让每个线程都能独立地发出网络请求，

通过大规模的并行网络访问来最大化数据吞吐量。AIFM 借助了高效的内核旁路(kernel-bypass)网络栈和用户态线程(user-level thread)技术[17]来避免网络访问和线程调度时高昂的用户态-内核态上下文切换开销。用户态线程的调度完全在用户空间进行，其切换成本远低于内核线程，使计算与网络 I/O 得以细粒度交叠。然而，与本文提出的系统相比，AIFM 在预取和逐出数据通路上 CPU 开销极高，主要开销体现在线程调度和线程间通信上。

1)AIFM 的预取器。AIFM 的预取器架构如图 3 所示，其中包含 1 个模式匹配线程以及 1 组预取线程池。其中，模式匹配线程读取访存踪迹，并使用有限状态机(finite state machine, FSM)对后续访存进行预测。预取线程池中包含 16 个工作线程，默认处于挂起状态。

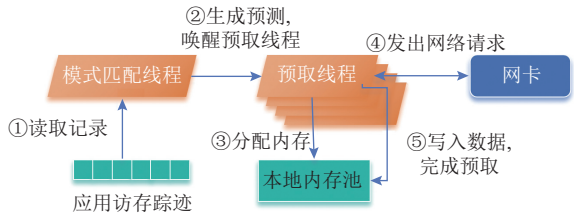


Fig. 3 Architecture of AIFM prefetcher

图 3 AIFM 的预取器架构

数据结构开发者使用预取接口捕获应用程序中的访存踪迹，例如访问的数组的下标等，并将其写入到预取器的 1 个环形缓冲区中，并唤醒模式匹配线程。每当模式匹配线程生成 1 个新的访存预测结果，将轮换地唤起 1 个预取线程并通过共享内存传递预取对象地址。随后，预取线程为对象分配本地内存、发出同步网络访问，并在预取完成后重新挂起。

2)AIFM 的逐出器。AIFM 的逐出器包括 1 个主线程和 1 组工作线程池。逐出器的执行包含 4 个阶段：①当本地内存资源紧张时，主线程选取部分内存区域进行回收；②主线程从线程池启动多个工作线程，每个线程并发地扫描一部分被选中的内存区域，识别对象冷热并将冷对象标记为待逐出；③主线程等待应用线程同步，避免在对象逐出时出现数据竞争；④主线程再次启动多个工作线程，如图 4 所示，并发将标记为待逐出的对象写回到远端内存池。

1.3 相关工作

1.3.1 用户态任务调度

现代数据中心和云计算环境中，高并发、低延迟的网络服务对任务调度系统提出了严苛的要求。在操作系统内核中执行网络和任务调度虽然提供了良好的隔离性和资源管理能力，却因频繁的系统调用和上下文切换导致较高的性能开销。

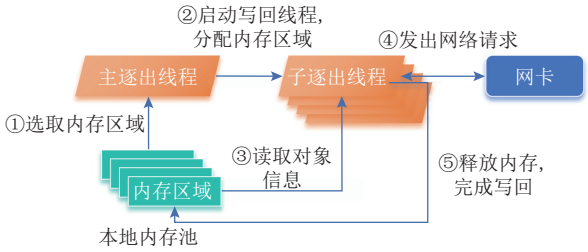


Fig. 4 Architecture of AIFM evacuator (writeback phase)

图 4 AIFM 的逐出器架构（写回阶段）

用户态任务调度的核心思想是在用户空间实现一套轻量级的调度器来管理用户态线程。线程切换开销远小于内核线程，且调度策略可以由应用程序自行决定。已经有许多用户态调度的相关工作[17-24]。其中，Shenango[17]通过定期检测应用的 CPU 资源压力并在应用进程之间快速重新分配 CPU 核心，从而提高 CPU 利用效率；Caladan[18]解决多任务同时运行时的资源干扰问题，通过微秒级资源监控和用户态线程调度实现应用间的性能隔离；Aspen[24]通过用户态中断实现快速的抢占式调度等。上述工作普遍面向微秒级以上任务。本工作关注分离式内存的后台服务，此类任务具有极短(亚微秒级)计算时间和更密集的网络访问，并伴随更强的任务间数据依赖，仅依赖用户态线程调度无法满足性能要求。

1.3.2 数据预取

数据预取技术已被广泛研究，其实现横跨硬件[25-30]、编译器[31-32]、操作系统内核[5,33-35]和用户态运行时等多个层次。硬件预取器集成在 CPU 中，对软件透明，擅长处理规律性内存访问，但难以应对不规则模式，且缺乏应用语义。编译器通过静态分析插入预取指令，但受限于静态分析的准确性，难以处理动态访问。操作系统内核中可以使用预取隐藏 I/O 延迟，并可以在分离式内存架构中应用于远程内存访问[5]，却存在页面粒度粗、读写放大、难以根据应用对象语义定制等缺陷。

本工作所属的用户态运行时预取是当前分离式内存领域的研究热点，因为它能结合应用语义，实现更灵活、更精确的预取。本文把预取器看作一类后台服务，关注其中的任务通信和调度问题，与各类访存模式预测和预取策略的工作是正交的。

1.3.3 用户态分离式内存系统

近年来，学术界在 AIFM 提出的用户态、对象粒度的分离式内存方案基础上提出了许多新的改

进。Mira[2]引入了程序行为导向的协同设计，通过静态分析与动态剖析智能调整缓存配置与预取策略；TrackFM[3]利用编译器自动识别及转换堆对象，消除了AIFM手动修改代码的负担；Beehive[4]提出基于“pararoutine”的执行模型，挖掘应用线程内部的异步性以最大化计算与通信的重叠，从而有效掩盖微秒级访存延迟。另外，HiTrain[36]面向大模型训练中的异构内存卸载，针对内核态I/O的系统调用与上下文切换开销问题，设计了异步优化器驱动的存储-计算流水线调度，实现GPU计算与I/O的有效重叠，该工作从大模型训练场景印证了用户态异步数据通路在相邻系统优化领域的重要价值，与本文的设计思路形成呼应。

然而，上述工作主要聚焦于上层策略决策或应用线程的访存延迟掩盖，并未涉及预取和逐出等服务线程的执行效率改进。本文提出的异步数据通路与上述工作正交且互补，我们专注于提升运行时内部服务线程的执行效率，消除了冗余的线程调度与线程间通信开销，能为上层策略模型提供更高效的底层支撑。

2 研究动机

在本节中，我们基于一组实验分析AIFM在分离式内存中后台服务线程的性能瓶颈，展示依靠多线程来实现并发预取和逐出开销巨大，且无法完全满足及时性要求。

1）实验设置

我们使用KMeans应用作为示例。KMeans算法将大量数据点按照相对距离划分为多个簇，它迭代式地对每个数据点与所有聚类中心进行距离计算，并随后重新计算每个簇的质心。因此，它需要在每次迭代中重复地、顺序地扫描整个数据集以读取数据点。这一访存特征非常适合预取器工作，同时会产生大量需要逐出器写回的脏数据。

我们在AIFM系统上实现了KMeans计算程序。计算服务器和内存节点经由一对100 Gbps Mellanox ConnectX-5 InfiniBand网卡相连。在我们的实验中，KMeans消耗了8 GB的内存，其中4 GB配置为本地内存，其余数据存放在远端内存池。作为对比，我们还展现了该应用在不使用远端内存下的性能。

为了展现服务线程消耗的CPU资源，我们展现单个KMeans计算线程下的结果。我们按照AIFM的默认设置，配置1个模式匹配线程和16个预取线程，并为应用分配了4个CPU。

我们在预取器和调度器的路径上插入性能剖析代码，分别统计预取器在模式匹配、对象管理、线程调度、线程间通信以及网络栈上的开销。

2）实验观察

如图5所示，KMeans应用在不使用远端内存的情况下每轮迭代需要2.55 s。在不启用预取器的情况下，每轮迭代需要高达32.3 s，这是因为此应用的计算访存比很低，即程序在数据访问上花费的时间远超于在计算上花费的时间，因此内存访问延迟是主要性能瓶颈。运行时系统取回每个对象平均需要9 μs的延迟，而在对象上的计算仅需700 ns。启用预取器可以将每轮迭代时间缩短至5.45 s，达到完全使用本地内存的214%。虽然预取器的引入显著缩短了每轮迭代的执行时间，但一方面应用吞吐并未提高到和只使用本地内存相似的水平，另一方面，预取器额外消耗了10.3 s的CPU时间，平均消耗了189%的CPU时间才能支撑1个KMeans计算线程，导致了显著的CPU资源开销。同时，逐出器也在后台消耗了71%的CPU。这部分资源本可用于应用计算，从而限制了系统的整体吞吐。

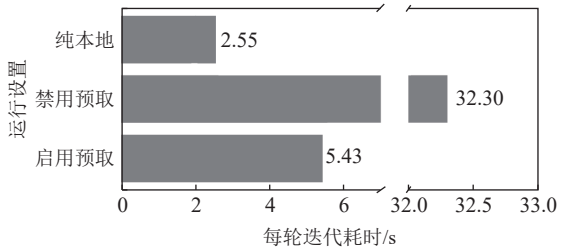


Fig. 5 Running performance for KMeans under three settings

图5 KMeans在3种设置下的运行性能

图6和图7分别展现了对预取器和逐出器运行过程的开销分解。我们对结果进行分析，得到如下2个观察：

1）后台服务的主要开销在于调度与线程间通信。我们发现，由于AIFM预取器和逐出器对任务并发性和及时性的追求，它启动了大量的线程来完成预取

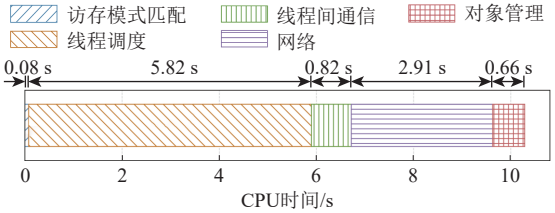


Fig. 6 Breakdown of CPU resources consumed by prefetcher

图6 预取器消耗的CPU资源分解

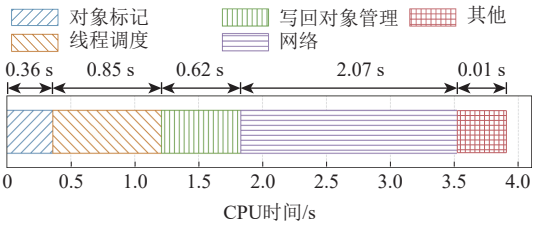


Fig. 7 Breakdown of CPU resources consumed by evacuator

图 7 逐出器消耗的 CPU 资源分解

和写回操作，使得几乎每个任务都伴随着线程的调度和线程间通信。对于预取任务，线程之间的调度开销达到了 CPU 消耗的 56%，另外 8% 的时间用于模式匹配线程和预取线程之间的通信。对于逐出任务，任务间通信量很少，但线程调度开销占用了 22% 的总 CPU 时间。这些额外 CPU 开销显著的主要原因在于任务粒度细、数据依赖重。在 KMeans 算法每轮迭代中，共发出 5.2×10^6 次预取和写回，每个对象的预取和写回平均仅消耗 1.4 μs 的 CPU 时间在调度和线程间通信以外的计算上，这与线程间通信和调度的开销在同一量级。我们的分析表明，对于这种短时延、细粒度的任务，采用大规模线程并发的模型所引入的调度和同步开销过高，会大幅抵消其带来的并发收益。

2)后台服务并未完全满足及时性要求。我们注意到后台服务中预取任务具有极强的及时性要求。然而，即使付出高昂的开销，在启用预取器的情况下，KMeans 计算线程的每轮执行时间仍然比纯本地内存上的计算慢 114%，应用线程仍有高达 8% 的未命中率，这并非来自算力或网络访问的限制。预取器占用的下行网络带宽仅 16 Gbps，远低于 100 Gbps 的硬件最大网络带宽。另一方面，我们通过修改运行时代码，可以测得 16 个预取器线程在 3 个 CPU 核心上并发预取可达到的峰值带宽为 33.1 Gbps，远高于单个 KMeans 线程的带宽需求。造成这一现象的主要原因是线程调度的不确定性。KMeans 应用运行过程中，在多个 CPU 核心上同时并发执行着大量线程：应用线程、逐出线程、预取器的模式匹配线程、正在发出请求的预取线程以及等待请求完成的预取线程，线程之间并没有调度顺序的约束，因此被线程调度系统以最大化 CPU 利用率的形式任意调度。然而，对于及时性要求极高的预取任务而言，最先产生的请求往往比其他预取任务更具急迫性。依赖通用的线程调度系统无法直接满足这一需求。

3 自适应异步数据通路设计

图 8 为本工作的整体设计。本系统旨在为分离式内存架构提供轻量级且自适应的后台服务调度。我们将带网络请求的细粒度的后台服务(如数据预取和逐出)任务分解为 3 个主要执行阶段：任务准备、请求发出和任务完成。为了提高任务处理的并发性并优化 CPU 开销，每个服务线程都在内部交织执行这些不同阶段的细粒度任务，其核心设计理念围绕批处理和异步机制展开。

3.1 后台服务的任务分解与执行

我们将带网络请求的细粒度后台服务任务分解为任务准备、请求发出和任务完成 3 个阶段。对于分离式内存系统中的预取和写回任务，我们分别讨论其实现。

1)任务准备。服务模块首先从应用线程产生的访存踪迹缓冲区读取记录(对于预取任务)或从堆上读取对象信息(对于逐出任务)。随后，它根据预设逻辑，如模式匹配分析访存模式以预测未来访存地址，或分析对象头元数据以确定对象状态，生成相应的服务任务。

2)请求发出。生成任务后，服务模块开始执行任务，为预取任务分配本地内存空间并修改对象的元数据，将对象锁定。随即利用异步通信原语向远端内存池发出数据加载或写回的网络请求。

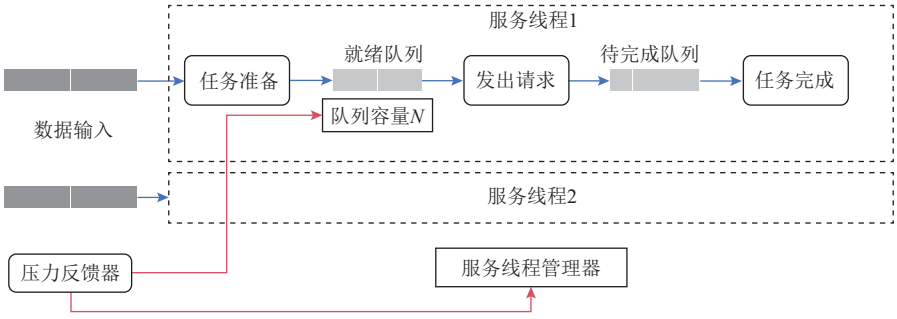


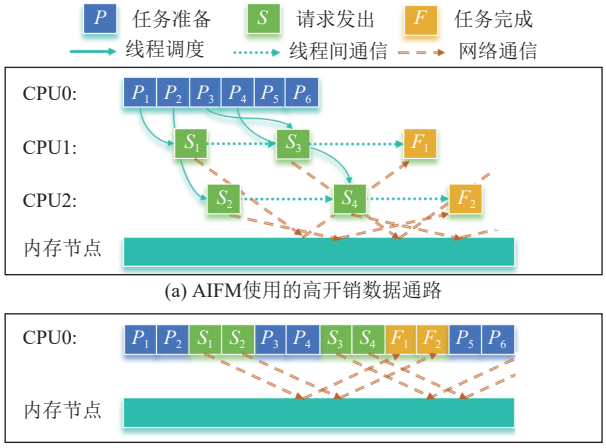
Fig. 8 Overall design of system

图 8 系统整体设计

3)任务完成。当网络请求完成后，服务模块会更新对应的对象状态，并释放网络和内存资源。

3.2 服务线程中细粒度任务调度

图 9(a)展示了 AIFM 系统中使用的数据通路。我们以图 9(a)中第 1 个任务为例，(P_1, S_1, F_1)描述了其 3 个阶段，其中准备阶段 P_1 在 CPU0 上的主线程执行，随后通过线程间通信，将请求发出阶段 S_1 与任务完成阶段 F_1 交由线程池中的某个线程完成。S_1 在 CPU1 上执行时网络请求使用同步通信原语发出，请求发出后阻塞线程的继续执行，触发线程切换并挂起线程，增加调度开销。当 S_1 发出的网络请求被内存节点回复后，线程调度器再把线程调起，在 CPU1 上继续执行 F_1。由此可见，AIFM 在其中引入了大量的线程间通信和线程切换开销。在有些情况下，由于 AIFM 使用的用户态线程调度器以追求 CPU 高占用率为目标进行负载均衡策略而不关注任务的相关性，同一任务的请求发出和任务完成阶段可能被调度到不同的核心处理，额外增加了数据在核间搬运的开销。



(a) AIFM使用的高开销数据通路

(b) 本工作使用的异步数据通路和细粒度任务管理

Fig. 9 Comparison of data path between our work and AIFM

图 9 本工作和 AIFM 的数据通路对比

然而，这些任务阶段完全可以在单个线程内交织执行，优化 CPU 开销的同时不影响任务的并发性，只需要我们引入异步通信原语并进行细粒度的任务调度管理。为了进一步优化任务执行开销，我们尽量将同一阶段的多个任务成批执行(详见 3.4 节)。本工作的设计主旨围绕批处理和异步进行，具体流程如算法 1 所示。

算法 1. 服务线程的自适应调度和负载均衡。

① while *WorkerRunning* do
　　/* 分阶段流水线执行 3 阶段任务 */
② 　*PrepareTasks*(N);
③ 　*IssueRequests*();
④ 　*CompleteTasks*();
　　/* 自适应调节任务队列长度 N */
⑤ 　$P \leftarrow$ *GetPressure*();
⑥ 　if $P > \theta$ then
⑦ 　　$N \leftarrow \max(1, N \times (1-\delta))$;
⑧ 　else if $P < \theta'$ then
⑨ 　　$N \leftarrow N \times (1+\delta)$;
⑩ 　end if
　　/* 负载均衡 */
⑪ 　if *InputSize*() > *Limit* then
⑫ 　　*WorkerState* ← OVERLOADED;
⑬ 　else
⑭ 　　while *NoRemainedTasks*() do
⑮ 　　　*WorkerState* ← IDLE;
⑯ 　　　*StealWorks*();
⑰ 　　end while
⑱ 　　*WorkerState* ← BALANCED;
⑲ 　end if
⑳ end while

服务线程首先从输入数据源(例如访存踪迹缓冲区或堆)批量读取数据。完成任务准备后，它会将生成的任务发布到任务队列中。此阶段持续进行，直到没有新的任务产生或已生成的任务数量达到队列容量 N。

随后，服务线程进入下一阶段，从任务队列中读取所有新生成的任务。它处理这些任务，并利用异步通信原语发出多个并发网络请求，然后将这些请求放入等待队列。

最后，服务线程尝试轮询网卡缓冲区，查询已完成的网络请求。一旦检测到请求完成，它会从等待队列中弹出对应的任务，执行任务的完成阶段。线程会循环执行上述流程，以实现多个任务之间网络和计算的并发。

图 9(b)展示了本工作利用异步 I/O 进行的流水线式处理。服务线程对任务进行逐批处理，批大小即为算法中的队列容量 N。在图 9(b)中，批大小 N=2。第 2 批的 2 个任务的准备阶段 P_3, P_4 相继执行后再成批发出请求(S_3, S_4)。从图 9(b)中可以看到，异步的网络请求并未阻塞线程的继续执行，网络传输与前一批次任务的完成阶段(F_1, F_2)和后一批次任务的准备阶段(P_3, P_4)异步执行。在此过程中，同一任务的 3 个阶段始终被一个线程执行。

由此，我们不但避免了冗余的线程调度和线程

间通信、节约了 CPU 资源，还可以保证预取任务完成的有序性。越早产生的任务将被越早放入任务队列，因此，这些任务请求也就越早被发出和完成。

就绪队列的容量 N 决定每个工作周期之间的间隔。N 过小不利于批处理优化，N 过大则任务的排队时间会过长，导致任务执行不及时。为了平衡高吞吐和低延迟这一对矛盾的目标，我们设计了一种基于迟滞效应的自适应反馈控制机制，该机制通过监控服务压力 P 动态调节批大小 N。如算法 1 中行⑤~⑩所示，我们在算法中引入了双阈值机制，构建了一个迟滞区间。当任务压力高于阈值θ时，将容量 N 减少δ以减小排队延迟；当任务压力低于θ'时，则将容量 N 增大δ以提高批处理大小；当压力处于$[\theta',\theta]$之间时，系统保持当前状态不变，以过滤负载的微小抖动。参数δ作为控制回路中的增益系数决定系统对负载变化的响应速度，较大的δ能快速适应负载突变，但增加超调风险；较小的δ则能提供更平滑的调节曲线，但可能导致响应滞后。

对于预取任务，预取器定期监测应用程序的访存未命中率，将访存与预测模式匹配但未命中的频率作为任务压力；对于逐出任务，任务压力设定为应用程序内存分配时本地内存容量低于水位线的频率。我们通过实验经验设定上述阈值为$\theta = 5\%$, $\theta' = 2\%$, $\delta = 25\%$。我们在 4.6 节展示了一个参数敏感性分析实验。为了确保系统在负载波动时的稳定性，我们在调节任务队列长度时设置了最小时间间隔$t = 10\ \mu s$，避免频繁的调整导致系统震荡。

3.3 自适应线程管理

我们根据负载情况启动多个服务线程，并将需要追踪的访存踪迹缓冲区或需要逐出的内存区域分配给各线程处理。为了实现服务模块的可扩展性，同时提高 CPU 资源利用率，1 个服务线程管理器监控线程的任务压力，调整服务线程的数量并进行负载均衡。

简单地说，我们通过监测线程内 CPU 实际利用率和未处理的输入数据大小，将服务线程分为“空闲”“平衡”“过载”3 种状态。“空闲”状态的线程会通过工作窃取偷取“过载”线程的任务来实现服务线程之间的负载均衡。具体过程如算法 1 中行⑪~⑲所示。

服务线程每执行一轮任务会判断一次自身状态。当预取线程未处理的访存踪迹数目超过就绪队列长度 N，或者逐出线程未处理的内存区域数目超过各线程的平均值，则认为这一线程进入过载状态；当服务线程一段时间内未能收到新输入时，则认为 CPU 开始进入空闲状态。对于服务线程之间的工作窃取，由于单个任务粒度过细，服务线程的工作窃取以整个踪迹缓冲区或内存区域为基本单位。一个空闲的服务线程根据其吞吐以及 CPU 空闲状态的时间比例计算出它可以额外承受的工作量。随后，它遍历各过载的服务线程，选择吞吐量最高的线程进行工作窃取。图 10 展示了一个例子，其中预取线程 T_1 检测到自身进入空闲状态，选择过载状态的线程 T_2，对其负责的踪迹缓冲区 B_4 进行“窃取”，从而平衡各线程之间的负载。

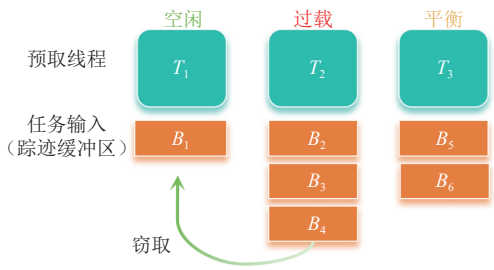


Fig. 10 Load balancing of service threads via work stealing

图 10 通过工作窃取实现服务线程负载均衡

最后，服务线程管理器会定期监视各线程状态，增减线程数量，实现 CPU 资源的弹性。如果不存在过载线程，而各线程的 CPU 空闲率之和超过 100% 时应减少服务线程数。此时，服务线程管理器选取 1 个最为空闲的服务线程，将其输入平摊给其他线程，然后令此线程结束执行。如果线程过载无法通过工作窃取解决，则启动一个新的线程来增加计算资源。

通过上述设计，可以实现为分离式内存的服务任务自适应地分配 CPU 资源，并在各线程之间维持负载均衡。

3.4 批处理优化

在 3.1~3.3 节所述的设计的基础上，我们通过进一步改进预取器和逐出器的工作过程以节约 CPU 开销。

预取和逐出请求的发出包括修改对象状态以及将网络请求写入网卡缓冲区等步骤。由于每个线程都会累积不超过 N 个请求后再发出请求，可以在内存分配和网络请求的步骤上实现批处理，从而进一步降低 CPU 开销。同样地，在轮询网卡缓冲区以及为对象分配和释放内存的过程也可以进行批完成，上述操作可以显著减少内存分配、对象管理和网络连接过程中的冗余检查和锁定。

3.5 实 现

我们在 AIFM 的基础上实现了本工作的原型

系统。

1)异步网络请求。AIFM基于shenango用户态线程库[14]，其中集成了基于DPDK的网络接口，并与调度器深度耦合。我们改进了其中的网络接口，添加了异步网络请求原语。此外，我们进一步放松了网络连接上的锁开销，在服务线程创建之初，从网络连接池中获取一个独占的连接，并在此后的异步网络访问过程中不再额外加锁。

2)服务线程管理器。我们在shenango调度器中插入了一段代码，用于及时唤起服务线程和调整线程数量。对于预取线程，当应用代码触发对象未命中，AIFM会发出网络请求并挂起线程。此时，服务线程管理器会查看是否有预取器负责此线程的踪迹缓冲区，并产生对应的压力反馈。此外，有压力的预取线程会被临时提升为高优先级，以避免CPU抢占导致的预取失效。对于逐出线程，我们沿用了AIFM的内存区域选择和对象标记逻辑，仅在对象写回阶段接入服务线程管理器。当对象逐出进入到写回阶段时，服务线程管理器根据内存压力启动数个逐出线程。

3)减轻预取过程中的锁竞争。我们对预取器的实现细节进行进一步优化，以避免锁定产生的冗余线程挂起。如果预取任务的执行不够及时，或者预取目标正在被其他线程并发访问，此时会出现对该对象状态的数据竞争。AIFM通过在发出预取数据请求之前锁定对象来避免竞争。本工作针对预取场景改进了这一点。在预取器中使用try_lock而非lock来锁定对象。若对象正在被其他线程并发访问，try_lock失败，预取线程将直接绕过这一预取任务。

4 实验评估

在本节中，我们对本文提出的系统进行全面的性能评估。我们的评估旨在回答以下5个关键问题：

1)相比于基准系统(AIFM)，本工作在预取和逐出的吞吐量和可扩展性方面表现如何。(4.2节)

2)本工作能否有效提升真实应用在分离式内存环境下的端到端性能。(4.3节)

3)本工作在降低CPU开销，特别是线程调度和通信开销方面的效果如何。(4.4节)

4)本工作是否是普适的，面向AIFM以外的其他系统是否同样可以实现性能优化。(4.5节)

5)本工作在调度系统中引入了一些额外参数，这些参数的选择是否合理。(4.6节)

4.1 实验配置和方法

我们的实验平台由1台计算服务器和1台内存服务器组成。2台服务器各搭载2块Intel Xeon Gold 6342 CPU和256 GB内存，并通过100 Gbps Mellanox ConnectX-5 InfiniBand网络适配器直连。

为了评估系统在不同访存模式下的性能，我们选取了4种具有代表性的数据密集型应用作为评测负载，分别包括：

1)KMeans[37]。一种迭代式的聚类算法，具有连续、顺序的数据访问模式和高脏写回数据压力，对预取和逐出数据通路提出了高要求。

2)PageRank[38]。经典的图计算算法，其访存模式包含大规模的流式读取和部分非规则间接寻址。

3)Snappy[39]。数据压缩应用，包含流式输入和压缩后的流式输出。与KMeans相比，Snappy的计算复杂度略高，可以验证本工作的数据通路改进，不仅提高了峰值网络带宽还改善了CPU执行效率。

4)DataFrame[40]。模拟数据分析场景中的表格操作，涉及对特定列的跨步访问。代表了现代数据科学栈中最常见的交互式分析负载，用于测试系统在真实数据分析场景下的通用性和响应速度。

各应用配置和数据集如表1所示。

Table 1 Memory Configurations and Datasets of Applications

表1 应用内存配置和数据集

应用负载	内存总量/GB	数据集
KMeans	8	随机生成
PageRank	3	根据PBBS图生成器[41]生成
DataFrame	4	在线飞行数据集[42]
Snappy	8	从维基百科数据集[43]中摘录

4.2 微基准测试：预取器的吞吐量

4.2.1 实验设计

我们以预取器为例，通过微基准测试来评估经过服务线程调度框架改进后的最大数据吞吐能力和扩展性。我们移除了应用计算逻辑，让预取器持续地从远端内存服务器拉取数据，并测量其能达到的最大网络带宽。我们为预取器分配1~4个专用的CPU核心，以观察其性能扩展性。

4.2.2 结果分析

如图11所示，本工作在所有CPU核心配置下均展现出优于AIFM的吞吐性能。

在单核心配置下，AIFM的吞吐量为22.8 Gbps，而本工作达到了46.8 Gbps，高出105%。这表明本工作

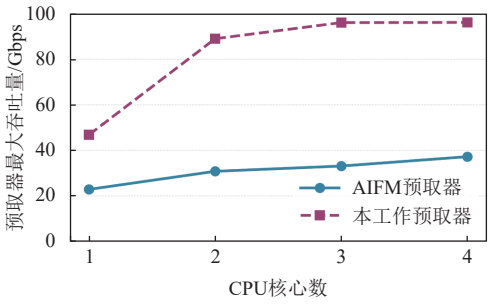


Fig. 11 Peak throughput of prefetcher under different numbers of CPU cores

图 11 不同 CPU 核心数下预取器峰值吞吐量

对预取任务的轻量化调度设计和异步 I/O 机制能更高效地利用单个核心进行数据处理和网络通信。

随着核心数的增加, 2 个系统的吞吐量都得到了提升。当使用 4 个核心时, 本工作预取器的吞吐量达到了 96.1 Gbps, 接近了 100 Gbps 的物理链路带宽, 而 AIFM 预取器的吞吐量为 37.2 Gbps, 且随着 CPU 核心数上升增长乏力。这证明了本工作实现的预取器不仅自身效率高, 还具备出色的垂直扩展能力, 其内部的线程管理和批处理机制能够有效地将负载分配到多个核心。而 AIFM 预取器由于其固定线程数的配置并不具备足够的扩展性。

4.3 应用端到端性能

4.3.1 实验设计

我们评估了本工作在 4 种真实应用负载下的端到端性能。我们为每个应用配置本地内存大小为总内存使用量的 50%, 即应用需要将另外 50% 的数据存放在远端内存池中。我们测量了应用的运行总时间(对于 DataFrame 和 Snappy)或每轮迭代的平均时间(对于 KMeans 和 PageRank)。

为了全面评估本工作的性能, 我们将其与以下 3 种情况进行比较:

1)纯本地。理想情况, 所有数据均放置在本地内存中, 代表了性能的上限。

2)禁用预取。所有远端数据均按需同步读取, 代表了性能的下限, 用于衡量分离式内存访问延迟带来的原始影响。

3)AIFM。代表了当前最先进的用户态预取系统, 作为我们的主要比较基准。

4.3.2 结果分析

如图 12 所示, 本工作在所有测试应用中均取得了最佳性能。

首先, 与纯本地基准相比, 禁用预取使得应用性能急剧下降, 运行时间平均增加了 548%, 这凸显了预取服务隐藏远端访存延迟的必要性。其中, 在 KMeans 的减速尤为严重, 禁用预取运行时间达到纯本地的 16.6 倍, 这是因为 KMeans 的计算访存比最高。

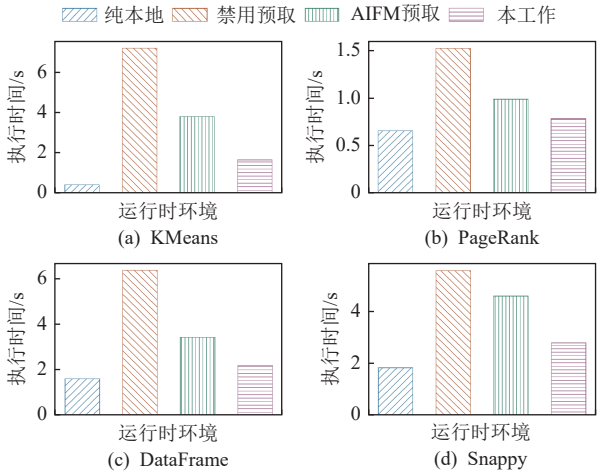


Fig. 12 End-to-end performance of applications

图 12 应用端到端性能

AIFM 通过预取显著改善了性能, 平均减少了 62% 的执行时间, 但后台服务占用大量 CPU, 导致性能仍显著差于纯本地的情景。因此, 本文提出的系统在 AIFM 的基础上实现了进一步的性能提升。在 KMeans 中, 本工作的执行时间仅为 1.67 s, 比 AIFM 快 56%, 达到了纯本地性能的 74%。对于 PageRank, DataFrame, Snappy 等其他应用, 本工作相比 AIFM 预取的性能提升分别为 20%, 36%, 39%。这一结果强有力地证明, 本工作通过降低自身开销, 将更多的 CPU 资源留给应用计算, 从而实现了更高的应用整体吞吐量。

为了进一步验证本工作对应用程序尾延迟的影响, 我们利用 DataFrame 和 Snappy 构建了 2 个负载。其中对于 DataFrame 应用, 我们在在线飞行数据集的基础上构建了细粒度的范围查询, 模拟了真实数据中心中常见的交互式分析场景; 对于 Snappy 应用, 我们设计了一个压缩数据请求服务, 每个查询将获取并压缩 1.8~3.2 MiB 的随机数据。我们统计了数万次并发查询请求的端到端响应时间, 并绘制了延迟累积分布函数(cumulative distribution function, CDF)曲线, 如图 13 所示。

观察图 13 中的延迟分布形态可以发现, 本工作在服务线程中引入批处理并未损害应用线程的延迟指标, 反而在关键的长尾延迟控制上优于 AIFM 系统。一方面, 本工作的后台服务 CPU 效率优于 AIFM, 因为 AIFM 在高负载下, 大量的后台服务线程与应用线程争抢 CPU 核心, 导致部分请求的处理时间发生剧

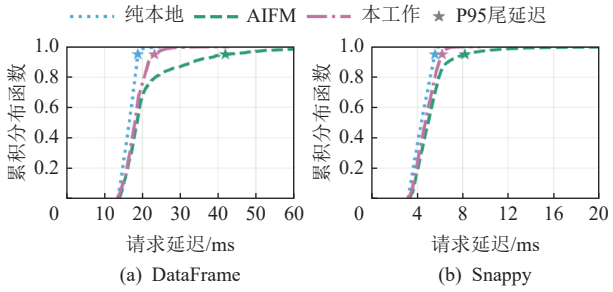


Fig. 13 CDF of request latency on DataFrame and Snappy

图 13 DataFrame 和 Snappy 上请求延迟的累积分布函数

烈抖动，在 CDF 图上表现为 P95 尾部曲线的显著“拖尾”。相比之下，本工作基于异步 I/O 和自适应批处理算法可以提供更稳定的运行时后台服务，从而使应用得以稳定处理请求。实验数据显示，本工作在保持吞吐量优势的同时，在 2 个应用中将 P95 尾延迟相比 AIFM 分别降低了 45% 和 25%，表明本工作有益于数据中心应用对服务质量稳定性（quality of service, QoS）的要求。

4.4 服务线程开销分析

4.4.1 实验设计

为了进一步确认本工作性能优势来源于减少后台服务的 CPU 占用，我们对 AIFM 和本工作的服务线程中的 CPU 时间开销进行了分析。我们基于 x86-64 平台的 RDTSC 指令，构建了性能剖析工具并插入到线程调度器源码中，测量了后台服务线程所占 CPU 时间。

4.4.2 结果分析

如图 14 所示，本工作成功地大幅削减了后台服务的计算开销。平均将预取服务的开销减小 78%，将逐出服务的开销减小 37%。

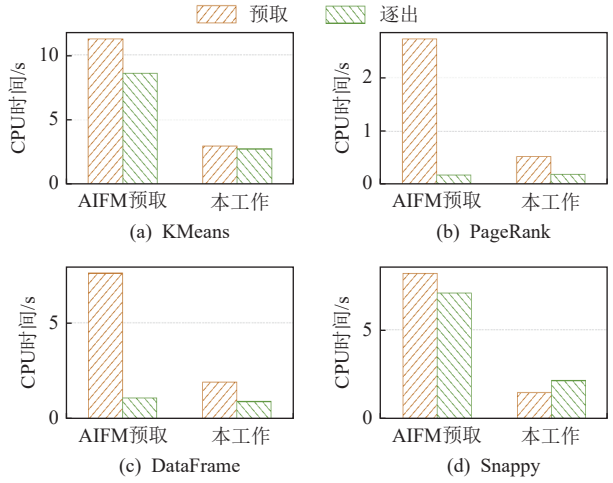


Fig. 14 Breakdown of service threads cost

图 14 服务线程开销分解

值得注意的是，对于 PageRank 和 DataFrame 两个应用，逐出服务开销很低，且本工作对逐出的性能改进不明显。这主要是因为 2 个应用在静态数据集上执行分析，没有产生脏数据。因此，逐出服务只需要将内存释放，不产生写回任务。

对于 KMeans 和 Snappy，应用产生了大量脏数据和写回任务，因此我们可以将逐出服务的开销分别降低 68% 和 70%。这一降低比例不如预取服务，因为逐出服务包含冷热数据识别和待逐出数据标注的开销，这一部分不能被我们的异步数据通路加速。

这一性能分析解释了本工作为何能带来更高的应用性能，也证明了我们设计的轻量级、自适应预取框架在资源效率上的巨大优势。

4.5 普适性验证

为了验证本工作提出的异步数据通路设计的普适性，以及在这一领域最新进展下的竞争力，我们将本文提出的设计分别移植到了 Beehive[4] 和 TrackFM[3] 系统中。Beehive 是近年来分离式内存系统的代表作之一，其使用 RDMA 网络栈，在应用的并发控制方面相比 AIFM 有了显著改进；TrackFM 基于 AIFM，利用编译器技术消除手动修改代码的负担，运行时部分和 AIFM 差异不大。我们分别对比了原版与集成了本工作的 Beehive，TrackFM 框架在 50% 本地内存配置下的性能表现。

实验结果如图 15 所示，我们以纯本地内存的执行时间为基准（归一化为 1）进行对比。如图 15(a)，在 Beehive 框架下，对于 KMeans 应用，集成本工作后的性能与原版 Beehive 基本持平。这是因为 KMeans 计算访存比极低，100 Gbps 的物理网络带宽已成为绝对瓶颈，单纯的 CPU 端优化无法突破这一物理限制。然而，对于 PageRank，DataFrame，Snappy 这 3 个计算访存比稍高的应用，本工作成功将 Beehive 的端到端执行时间平均降低了 7.7%。这一提升幅度小于在

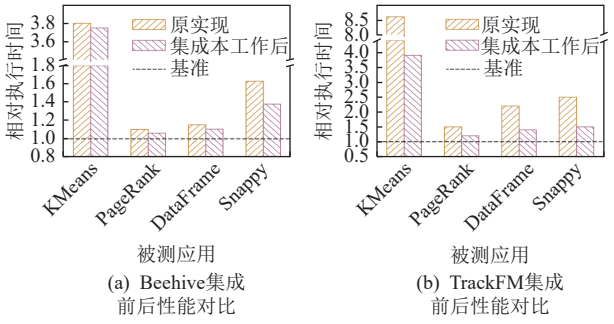


Fig. 15 Performance comparison of Beehive and TrackFM before and after integration of our work

图 15 集成本工作前后 Beehive 与 TrackFM 性能对比

AIFM 上的收益, 主要原因是 Beehive 本身的基础性能更好。如图 15(b), TrackFM 框架的基础性能比 Beehive 差, 因此本工作在 TrackFM 下相对优化效果更好, 平均降低了 37% 的执行时间。这是因为 TrackFM 相对 AIFM 的改进主要体现在可编程性而非性能上。

实验结果可以证明, 本工作在 AIFM 以外基于用户态运行时的分离式系统上同样可以进一步挖掘性能潜力, 从而逼近纯本地内存的理想性能。

4.6 参数敏感性分析

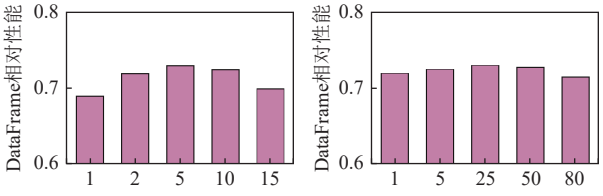
本工作使用的自适应控制依赖于θ,θ',δ等参数。压力阈值θ,θ'和调节步长δ共同决定了系统对服务压力变化的响应灵敏度及调节幅度, 其目的是在批处理带来的“高吞吐”与请求积压导致的“高延迟”之间寻求最佳的权衡点: 批大小过大虽能提升 CPU 效率, 但会损害预取和逐出的及时性; 而批大小过小则无法充分利用批处理优势。

本工作通过监测服务线程的输入压力进行调节。压力阈值θ,θ'决定了相应灵敏度, 当压力偏离区间$[\theta',\theta]$时才会触发对批大小的修改。显然, 压力阈值设定过低会导致系统对微小的负载波动表现出过度敏感, 从而使批大小发生剧烈抖动, 频繁的调整开销会抵消批处理的收益, 同时偶然抖动下批大小可能过高或过低, 偏离合理范围。压力阈值设定过高会导致系统对负载变化的响应变得迟滞, 一方面可能在应用产生较大预取压力时仍维持较大的批大小 N 从而导致积压; 另一方面也可能在低负载下维持过低的 N 导致 CPU 效率下降。

参数δ则决定了负载变化时批大小的调节步长直接影响自适应系统的收敛速度。面对负载快速切换的场景, δ过低(过于平缓的调节)使得 N 无法及时适应新的压力状态, 造成了短暂的滞后; 而δ过高会使得 N 在极值之间大幅跳变, 系统容易陷入震荡状态, 难以精确稳定在最佳工作点。2 种情况均会导致批大小 N 的设置在一段时间内偏离最优, 从而产生性能损耗。

我们以 DataFrame 负载为例进行了敏感性测试。我们构建了 1 组计算访存比各异的操作序列, 并将每个查询的平均执行时间控制在 1 ms 左右, 以此模拟细粒度任务下的快速负载波动, 考察超参数对端到端平均应用吞吐量的影响。

实验中, 我们将θ的取值设定为 1%~15% 中的若干值, 并始终将下阈值θ'设定为上阈值θ的 40% 以维持一定的迟滞区间。实验结果如图 16(a)所示, 其中以 DataFrame 操作相对纯本地执行下的平均吞吐量作为相对性能。如图 16(a)所示, 端到端吞吐量随θ的变化呈现先升后降的趋势, 并在$\theta=5\%$达到最佳。当$\theta=1\%$时系统对微小的负载波动表现出过度敏感, 当$\theta=15\%$时系统对负载变化的响应出现显著迟滞, 均导致端到端性能的下降, 符合我们之前的分析。



(a) 压力阈值的敏感性分析 (b) 调节步长的敏感性分析

Fig. 16 Parameter sensitivity analysis

图 16 参数敏感性分析

图 16(b)显示δ在 1%~80% 的范围内取值下的相对性能变化, 可以观察到δ在 5%~50% 范围内吞吐量随δ的变化波动不大, 但在$\delta=25\%$处达到最优。在极小($\delta=1\%$)或极大($\delta=80\%$)的取值下分别造成了一定的批大小调节滞后或大幅跳变, 可以观察到性能损耗, 符合前述分析结果。

最后, 参数的最优取值在一定程度上受负载和硬件影响。例如, 当网络环境波动较大时, 预取线程的负载可能会出现频繁抖动, 将参数θ适度调高可能使系统更容易过滤掉抖动, 性能更好; 当负载变化更加平缓, 例如以秒级发生变化时, 将参数δ适度减小会使调节步幅更小, 更容易得到最优的批大小。

5 讨 论

5.1 设计的普适性

本工作主要关注基于用户态运行时的分离式内存后台服务系统实现。本项研究最重要的洞察在于揭示了: 对于延迟敏感且计算量极小的细粒度任务, 传统的、通过增加线程来追求高并发的“暴力”并行化方法, 其收益会被线程调度、同步和通信的开销迅速抵消。我们在原型系统上的成功实践表明, 采用异步 I/O 来重叠计算与网络通信, 并通过自适应批处理将多个细粒度操作聚合成一个粗粒度任务, 是打破这一开销瓶颈的关键。

尽管本工作聚焦于分离式内存场景, 但其核心设计原则, 即对细粒度并发任务的管理和优化具有更广泛的适用性。在微服务架构、函数计算(function as a service, FaaS)以及其他需要处理大量短时请求的现代计算范式中, 请求处理的调度开销同样是一个关

键的性能瓶颈。我们提出的自适应批处理和动态线程管理策略，可以被借鉴用于优化这些场景下的任务调度器和运行时系统。

5.2 局限性

我们也认识到当前工作存在以下局限性：

1）自适应参数的敏感性。系统中的自适应算法依赖于一系列预设阈值。这些参数是在我们的实验平台上凭经验调优得到的，其最优值可能随硬件环境（如网络延迟、CPU 核心数）和工作负载特性的变化而变化。系统在不同环境下的普适性和参数自适应能力有待进一步研究。

2）缺乏多应用下全局资源视图。当前的线程管理策略主要关注单个应用的内部状态，在多个应用或多个租户共享计算资源的环境中，可能会出现网络和 CPU 资源竞争。一个全局的、跨应用的预取资源管理器可能会带来更优的系统级性能。

我们将以上问题的解决留作未来工作。

6 结　论

针对现有分离式内存访问因过度使用多线程并发而导致 CPU 开销高昂的问题，本文提出并实现了一种面向分离式内存的用户态数据通路。与依赖大规模线程并发的传统设计不同，本文的核心思想在于利用异步 I/O 和自适应批处理技术，以最小化的调度开销实现计算与网络通信的深度重叠。实验结果有力地证明，本工作在显著降低预取和逐出数据通路 CPU 资源消耗的同时，大幅提升了应用的端到端吞吐量。本研究不仅为构建高效的分离式内存系统提供了一种新的解决方案，更重要的是，它验证了一种资源高效的设计范式：对于细粒度、延迟敏感的任务，精巧的异步化与批处理协同设计是比“线程换性能”的暴力并行化更优越的路径。

作者贡献声明：李权熹负责方法设计、实验验证和论文撰写；崔慧敏和冯晓兵提出指导意见并对论文进行了修改和校正；王晨曦负责方法设计和修改论文。

参 考 文 献

[1] Ruan Zhenyuan, Schwarzkopf M, Aguilera M K, et al. AIFM: High-performance, application-integrated far memory[C]//Proc of the 14th USENIX Symp on Operating Systems Design and Implementation (OSDI 20). Berkeley, CA: USENIX Association, 2020: 315−332

[2] Guo Zhiyuan, He Zijian, Zhang Yiying. Mira: A program-behavior-guided far memory system[C]//Proc of the 29th Symp on Operating Systems Principles (SOSP 23). New York: ACM, 2023: 692−708

[3] Tauro B R, Suchy B, Campanoni S, et al. TrackFM: Far-out compiler support for a far memory world[C]//Proc of the 29th ACM Int Conf on Architectural Support for Programming Languages and Operating Systems (ASPLOS 24), Volume 1. New York: ACM, 2024: 401−419

[4] Li Quanxi, Huang Hong, Liu Ying, et al. Beehive: A scalable disaggregated memory runtime exploiting asynchrony of multithreaded programs[C]//Proc of the 22nd USENIX Symp on Networked Systems Design and Implementation (NSDI 25). Berkeley, CA: USENIX Association, 2025: 167−187

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

[6] NVIDIA. InfiniBand Network[EB/OL]. [2025-11-15]. https://docs.nvidia.com/doca/sdk/infiniband-network/index.html

[7] CXL Consortium. Compute express link specification[EB/OL]. [2025-11-15]. https://computeexpresslink.org

[8] Calciu I, Imran M T, Puddu I, et al. Rethinking software runtimes for disaggregated memory[C]//Proc of the 26th ACM Int Conf on Architectural Support for Programming Languages and Operating Systems (ASPLOS 21). New York: ACM, 2021: 79−92

[9] Li Huaicheng, Berger D S, Hsu L, et al. Pond: CXL-based memory pooling systems for cloud platforms[C]//Proc of the 28th ACM Int Conf on Architectural Support for Programming Languages and Operating Systems (ASPLOS 23), Volume 2. New York: ACM, 2023: 574−587

[10] Gouk D, Lee S, Kwon M, et al. Direct access, high-performance memory disaggregation with DirectCXL[C]//Proc of the 2022 USENIX Annual Technical Conf (USENIX ATC 22). Berkeley, CA: USENIX Association, 2022: 287−294

[11] Guo Zhiyuan, Shan Yizhou, Luo Xuhao, et al. Clio: A hardware-software co-designed disaggregated memory system[C]//Proc of the 27th ACM Int Conf on Architectural Support for Programming Languages and Operating Systems (ASPLOS 22). New York: ACM, 2022: 417−433

[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 Z, et al. Can far memory improve job throughput?[C]//Proc of the 15th European Conf on Computer Systems (EuroSys 20). New York: ACM, 2020: 1−16

[14] Qiao Yifan, Wang Chenxi, Ruan Zhenyuan, et al. Hermit: Low-latency, high-throughput, and transparent remote memory via feedback-directed asynchrony[C]//Proc of the 20th USENIX Symp on Networked Systems Design and Implementation (NSDI 23). Berkeley, CA: USENIX Association, 2023: 181−198

[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] DPDK Project. DPDK – The open source data plane development kit accelerating network performance[EB/OL]. [2025-11-15]. https://www.dpdk.org/

[17] Ousterhout A, Fried J, Behrens J, et al. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads[C]//Proc of the 16th USENIX Symp on Networked Systems Design and Implementation (NSDI 19). Berkeley, CA: USENIX Association, 2019: 361−378

[18] Fried J, Ruan Zhenyuan, Ousterhout A, et al. Caladan: Mitigating interference at microsecond timescales[C]//Proc of the 14th USENIX Symp on Operating Systems Design and Implementation (OSDI 20). Berkeley, CA: USENIX Association, 2020: 281−297

[19] Anderson T E, Bershad B N, Lazowska E D, et al. Scheduler activations: Effective kernel support for the user-level management of parallelism[J]. ACM Transactions on Computer Systems, 1992, 10(1): 53−79

[20] Iyer R, Unal M, Kogias M, et al. Achieving microsecond-scale tail latency efficiently with approximate optimal scheduling[C]//Proc of the 29th Symp on Operating Systems Principles (SOSP 23). New York: ACM, 2023: 466−481

[21] Seo S, Amer A, Balaji P, et al. Argobots: A lightweight low-level threading and tasking framework[J]. IEEE Transactions on Parallel and Distributed Systems, 2017, 29(3): 512−526

[22] Shiina S, Iwasaki S, Taura K, et al. Lightweight preemptive user-level threads[C]//Proc of the 26th ACM SIGPLAN Symp on Principles and Practice of Parallel Programming (PPoPP 21). New York: ACM, 2021: 374−388

[23] Zhang I, Raybuck A, Patel P, et al. The Demikernel datapath OS architecture for microsecond-scale datacenter systems[C]//Proc of the 28th Symp on Operating Systems Principles (SOSP 21). New York: ACM, 2021: 195−211

[24] Guo Linsong, Zuberi D, Garfinkel T, et al. The benefits and limitations of user interrupts for preemptive userspace scheduling[C]//Proc of the 22nd USENIX Symp on Networked Systems Design and Implementation (NSDI 25). Berkeley, CA: USENIX Association, 2025: 1015−1032

[25] Byna S, Chen Yong, Sun Xianhe. A taxonomy of data prefetching mechanisms[C]//Proc of the 2008 Int Symp on Parallel Architectures, Algorithms, and Networks (I-SPAN 08). Piscataway, NJ: IEEE, 2008: 19−24

[26] Joseph D, Grunwald D. Prefetching using Markov predictors[C]//Proc of the 24th Annual Int Symp on Computer Architecture (ISCA 97). New York: ACM, 1997: 252−263

[27] Sherwood T, Sair S, Calder B. Predictor-directed stream buffers[C]//Proc of the 33rd Annual ACM/IEEE Int Symp on Microarchitecture. New York: ACM, 2000: 42−53

[28] Jain A, Lin C. Linearizing irregular memory accesses for improved correlated prefetching[C]//Proc of the 46th Annual IEEE/ACM Int Symp on Microarchitecture. New York: ACM, 2013: 247−259

[29] Zhu Huaiyu, Chen Yong, Sun Xianhe. Timing local streams: Improving timeliness in data prefetching[C]//Proc of the 24th ACM Int Conf on Supercomputing (ICS 10). New York: ACM, 2010: 169−178

[30] Khan M, Sandberg A, Hagersten E. A case for resource efficient prefetching in multicores[C]//Proc of the 43rd Int Conf on Parallel Processing (ICPP 14). Piscataway, NJ: IEEE, 2014: 101−110

[31] Kolli A, Saidi A, Wenisch T F. RDIP: Return-address-stack directed instruction prefetching[C]//Proc of the 46th Annual IEEE/ACM Int Symp on Microarchitecture. New York: ACM, 2013: 260−271

[32] Rabbah R M, Sandanagobalane H, Ekpanyapong M, et al. Compiler orchestrated prefetching via speculation and predication[J]. ACM SIGARCH Computer Architecture News, 2004, 32(5): 189−198

[33] Cao Pei, Felten E W, Karlin A R, et al. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling[J]. ACM Transactions on Computer Systems, 1996, 14(4): 311−343

[34] 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

[35] Kaplan S F, McGeoch L A, Cole M F. Adaptive caching for demand prepaging[J]. ACM SIGPLAN Notices, 2002, 38(Supplement 2): 114−126

[36] Tang Zhihang, Lan Hao, Liu Zhengguo, et al. HiTrain: Heterogeneous memory offloading and I/O optimization for large language model training[J]. Journal of Computer Research and Development, 2026, 63(3): 627−639 (in Chinese)
(汤志航, 兰颢, 刘政国, 等. HiTrain: 面向大模型训练的异构内存卸载与 I/O 优化[J]. 计算机研究与发展, 2026, 63(3): 627−639)

[37] McQueen J B. Some methods of classification and analysis of multivariate observations[C]//Proc of the 5th Berkeley Symp on Mathematical Statistics and Probability. Berkeley, CA: University of California, 1967: 281−297

[38] Page L, Brin S, Motwani R, et al. The PageRank citation ranking: Bringing order to the web[R]. Stanford, CA: Stanford Infolab, 1999

[39] Google. Google's fast compressor/decompressor[EB/OL]. [2025-11-15]. https://github.com/google/snappy

[40] Moein H. C++ DataFrame for statistical, financial, and ML analysis[EB/OL]. [2025-11-15]. https://github.com/hosseinmoein/DataFrame

[41] Anderson D, Blelloch G E, Dhulipala L, et al. The problem-based benchmark suite (PBBS), v2[C]//Proc of the 27th ACM SIGPLAN Symp on Principles and Practice of Parallel Programming (PPoPP 22). New York: ACM, 2022: 445−447

[42] Bulter22. Airline on-time performance data[EB/OL]. [2025-11-15]. https://www.kaggle.com/datasets/bulter22/airline-data

[43] Mahoney M. Large text compression benchmark[EB/OL]. [2025-11-15]. https://www.mattmahoney.net/dc/text.html



Li Quanxi, born in 1998. PhD. His main research interest includes building systems for data centers.
李权熹, 1998 年生。博士。主要研究方向为数据中心系统构建。



Cui Huimin, born in 1979. PhD, professor, PhD supervisor. Her main research interests include compiler construction, compiler optimization, and heterogeneous computing.

崔慧敏, 1979 年生。博士，研究员，博士生导师。主要研究方向为编译器构建、编译优化、异构计算。



Feng Xiaobing, born in 1969. PhD, professor and PhD supervisor. Senior member of CCF. His main research interests include compiler optimization and binary translation.

冯晓兵, 1969 年生。博士，研究员，博士生导师。CCF 高级会员。主要研究方向为编译优化、二进制翻译。



Wang Chenxi, born in 1990. PhD, associate professor, PhD supervisor. His main research interests include building systems for emerging architectures.

王晨曦, 1990 年生。博士，副研究员，博士生导师。主要研究方向为新兴架构下的系统构建。