基于混合内存的Apache Spark缓存系统实现与优化

魏森，周浩然，胡创，程大钊

引用本文

魏森，周浩然，胡创，程大钊.基于混合内存的Apache Spark缓存系统实现与优化[J]．计算机科学，2023，50(6)：10-21.

WEI Sen，ZHOU Haoran，HU Chuang，CHENG Dazhao. Implementation and Optimization of Apache Spark Cache System Based on Mixed Memory [J]. Computer Science，2023，50(6)：10-21.

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.220900261

基于混合内存的 Apache Spark 缓存系统实现与优化

魏 森 周浩然 胡 创 程大钊
武汉大学计算机学院 武汉 430072
(weisen@whu.edu.cn)

摘 要 随着大数据时代数据规模的激增,内存计算框架得到了长足发展。主流内存计算框架 Apache Spark 使用内存来缓存中间结果,大幅度地提升了数据处理速度。同时,具有较快的读写速度和较大容量的非易失性存储器 NVM 在内存计算领域展现出了巨大的发展前景,使用 DRAM 和 NVM 构建 Spark 混合缓存系统成为一种可行方案。文中提出了一种基于 DRAM-NVM 混合内存的 Spark 缓存系统,该系统选择平面混合缓存模型作为设计方案,然后为缓存块管理系统设计了专用的数据结构,并提出了适用于 Spark 的混合缓存系统整体设计架构。另外,为了将频繁访问的缓存块保存在 DRAM 缓存中,提出了基于缓存块最小重用代价的混合缓存管理策略。首先从 DAG 信息中获取 RDD 的未来重用次数,未来重用次数多的缓存块将被优先保存在 DRAM 缓存中,并在缓存块迁移时考虑了迁移成本。设计实验表明,DRAM-NVM 混合缓存相比原有缓存系统的性能平均提升了 53.06%,对于相同的混合内存,所提策略相比默认缓存策略有平均 35.09%的提升。同时,使用文中设计的混合系统只需要 1/4 的 DRAM 和 3/4 的 NVM 作为缓存,就能达到全部 DRAM 缓存约 79%的性能表现。

关键词:Spark;缓存管理策略;NVM;混合内存
中图法分类号 TP311.13

Implementation and Optimization of Apache Spark Cache System Based on Mixed Memory

WEI Sen,ZHOU Haoran,HU Chuang and CHENG Dazhao
School of Computer Science,Wuhan University,Wuhan 430072,China

Abstract With increasing data scale in the "big data era",in-memory computing frameworks have grown significantly. The mainstream in-memory computing framework Apache Spark uses memory to cache intermediate results,which greatly improves data processing performance. At the same time,non-volatile memory (NVM) with fast read and write performance has great development prospects in the field of in-memory computing,so there is huge promise in building Spark's cache with a mix of DRAM and NVM. In this paper,a Spark cache system based on DRAM-NVM hybrid memory is proposed,which selects the flat hybrid cache model as the design scheme,and then designs a dedicated data structure for the cache block management system,and proposes the overall design architecture of the hybrid cache system for Spark. In addition,in order to save frequently accessed cache blocks in the DRAM cache,a hybrid cache management strategy based on the minimum reuse cost of cache blocks is proposed. First,the future reuse of RDD is obtained from the DAG information,and the cache blocks with high future reuse times will be stored in the DRAM cache first,and the migration cost is considered when the cache block is migrated. The design experiments show that the DRAM-NVM hybrid cache has an average performance improvement of 53.06% compared to the original cache system,and the proposed strategy has an average improvement of 35.09% compared to the default cache strategy for the same hybrid memory. At the same time,the hybrid system designed in this paper only needs 1/4 of the DRAM and 3/4 of the NVM as the cache,and the running time of the total DRAM cache can be achieved by 85.49%.

Keywords Spark,Cache management strategy,NVM,Hybrid memory

1 引言

近年来,随着数据规模的激增以及对可扩展性大数据处理的需求增加,各种大数据框架在工业和研究领域都得到长足发展。Apache Hadoop[1]通过引入被称为 MapReduce[2]的数据处理框架来解决分布式数据处理问题。

Spark 计算框架的巨大成功来源其于对内存资源的使用。Spark 中的数据计算集中在内存中,减少了磁盘 I/O,从而

到稿日期:2022-09-28 返修日期:2022-11-03
基金项目:之江实验室开放课题(K2022PI0AB01);湖北珞珈实验室专项基金资助项目(220100016)
This work was supported by the Zhejiang Lab Open Research Project (K2022PI0AB01) and Special Fund of Hubei Luojia Laboratory (220100016).
通信作者:程大钊(dcheng@whu.edu.cn)

避免了序列化成本并减少了大量 I/O 开销，且可以在内存空间中缓存部分中间结果，下次使用这些中间结果时可直接在缓存中访问。

然而，大数据时代不断增长的数据量给内存计算框架带来了巨大的压力，这些框架必须定期处理 TB 甚至 PB 的数据。因此，这些框架对可用内存有着很高的要求，内存不足可能导致一系列严重的功能和性能问题。然而，数据的增长速度超过了 DRAM 价格的下降速度，在这些数据分析集群中，内存仍然是一种稀缺资源。目前使用的 DRAM 内存技术已进入发展瓶颈，成本难以降低，存储密度难以增加，阻碍了内存计算规模的继续增长。同时，一种新型存储介质——非易失性存储器（Non-Volatile Memory，NVM）[3] 被认为有着巨大潜力。作为持久化存储设备，其性能远优于常见的块存储设备（硬盘及 SSD）；与 DRAM 相比，NVM 同样具备字节可寻址的特点，但读写延迟均高于 DRAM，其带宽可能低至 DRAM 的十分之一[4]。因此，在大多数系统中，NVM 无法替代 DRAM 作为内存直接使用。

异构存储是一种新兴的存储器架构。在异构存储中，具有不同技术的多个存储器组件组合在一起以构建主存储器。异构存储结构通常由高容量存储器（性能相对较差，如 NVM）和高性能存储器（容量较小，如 DRAM）组成。具体而言，Intel © Optane™ DC 持久内存[5] 作为一种 NVM 目前已经商业化并可以购买，其单位容量的价格比 DRAM 低 47%。虽然单纯地使用 NVM 没有任何效果，但是使用一定比例的 DRAM-PMM 系统可以显著减少 DRAM 的使用量并且更具成本效益。

因此，使用一小部分 DRAM 和大量 NVM 的混合内存架构被看作是更有前景的内存解决方案[6]。将 NVM 与 DRAM 混合用于大数据平台的缓存系统可以同时利用 NVM 字节可寻址、更高容量的优势和 DRAM 的低延迟高带宽的优势，提升混合内存的整体读写性能。

基于上述背景，本文提出了一种基于混合内存的 Spark 缓存系统，设计了适配 Spark 内存管理的混合内存系统，将 Spark 缓存数据块放置在混合内存上，并设计了两种缓存介质间的缓存块放置和迁移策略，让需要频繁访存的缓存块可以获得更好的响应，同时最大限度减少缓存块迁移带来的性能开销，在扩展缓存总容量的同时达到较好的缓存 I/O 性能。

本文的主要贡献如下：

（1）为 Apache Spark 设计了 DRAM-NVM 混合缓存结构。

（2）在混合内存系统的基础上，实现了与之相适配的缓存管理策略。

（3）实现了缓存块的未来重用次数的计算和更新。

（4）量化了数据块在混合内存之间的迁移成本，在需要进行混合缓存之间的迁移时，通过缓存块的读写时间等信息计算迁移缓存块的潜在性能收益，迁移收益为正时才进行缓存块的迁移。

本文第 2 节总结和分析了 Spark 缓存管理和混合内存的研究进展；第 3 节介绍了本文对 Spark 在混合内存上设计缓存系统的设计方案，包括 NVM-DRAM 混合缓存的结构设计，并为混合缓存系统设计了数据结构，最后提出了适用于 Spark 的混合缓存系统的整体设计架构；第 4 节介绍了基于最小重用代价的缓存管理策略的设计与实现，详细介绍了基于 DAG 的缓存的获取和使用，以及数据块迁移代价的计算方法，并提出了具体的数据块放置和迁移策略；第 5 节介绍了实验设计和实验结果分析，通过实验对比和分析，证明了使用 NVM 设计混合缓存对大数据系统的性能提升，并验证了本文设计的缓存管理策略在混合缓存场景下的有效性。相比默认缓存管理策略，本文设计的混合缓存系统提高了缓存命中率，缩短了运行时间。

2　相关工作

Spark 默认使用最近最少使用（Least Recently Used，LRU）缓存策略[7]，但是 Spark 的数据抽象弹性分布式数据集（Resilient Distributed Dataset，RDD）的缓存有着明确的数据依赖关系，表示为计算任务的有向无环图（Directed Acyclic Graph，DAG）。因此有研究者提出利用 DAG 的依赖关系来指导缓存管理的构想。此外，由于 Spark 平台是基于 Java 虚拟机（Java Virtual Machine，JVM）的，而内存替换的过程正好与 JVM 本身的垃圾清理（Garbage Collection，GC）过程相关，因此有研究者提出可以将 Spark 的内存管理与 GC 过程相结合。

2.1　基于 DAG 依赖关系的缓存管理策略

Xu 等提出的 MemTune[8] 是利用最早 DAG 依赖关系的 Spark 缓存管理系统。MemTune 动态调整 Spark 中计算任务和缓存的数据共享，并根据需要逐出/预取数据。

MemTune 只考虑当前运行任务的本地依赖信息。而另一些研究则是基于整个 DAG 的依赖信息来指导缓存管理，其中最有代表性的是最少引用计数（LRC）[9]，该方法遍历应用程序的 DAG，对每个数据块的再次引用进行计数，并在内存不足时驱逐具有最小引用计数的缓存数据块。

与这种思路相近的研究还有很多：文献[10]提出了最小有效引用计数（LERC），该研究表明仅仅追求单个数据块的更高缓存命中率并不一定会导致任务在并行环境中更快地完成；文献[11]提出了最少组合引用计数（LCRC），该研究发现 LRC 策略将具有较大引用计数的缓存块驻留在内存中虽然可以一定程度地提高系统性能，但是这些块在其整个生命周期中的某些阶段可能不会被访问，导致内存利用率降低，使用组合引用计数并驱逐具有最小组合引用计数的缓存块；文献[12]使用 RDD 的引用距离作为指标来指导缓存块的管理，实现了最大引用距离（MRD）策略。这些研究工作选择了 DAG 所反映出的缓存数据块的不同时空特性，并以此作为放置和迁移的依据。

近两年来也有很多其他工作。Zhao 等提出的 RDE[13] 可以充分利用应用程序的 DAG 信息来优化管理结果，考虑 RDD 的依赖性和参考序列，有效地排除了具有冗余特性的 RDD，并为传入数据块完善了内存。与 LRC 和 MRD 相比，RDE 的性能分别提高了 48% 和 20%。Song 等提出了较少争用管理策略[14]（MCM），以减少内存争用的负面影响。其优先满足任务的最小执行资源，并考虑竞争成本，用持久位置选择

算法动态选择最佳存储位置，以改善持久加速的效果。Chukonu[15]是一个原生大数据框架，其提出了一系列技术用于优化 DAG 程序编译时部分，如算子融合、向量化和压缩，以显著降低 Spark 集成开销。

2.2 基于 GC 的 Spark 混合内存管理系统

Wang 等提出的 Panthera[16]通过分析大数据平台 Spark 上的用户程序来推断数据块的粗粒度访问模式，修改后的 JVM 组件根据数据块的访问模式进行数据块的放置和迁移。Khan 等设计了一个平面混合方案[17]，以利用 NVM 缓存 RDD 块，并进行了一些架构优化，如用于块展开的动态内存分配、带抢占的异步迁移机制。该方法提出的混合存储器只使用了一部分 DRAM，但将运行性能的降低控制在平均 10% 左右。Chen 等提出了一种语义感知的全自动内存管理技术[18]，用于混合内存上的大数据处理。其基于大数据应用程序的内存访问模式实现了一种新的分析引导优化策略，在平均时间开销不到 1% 的情况下将能耗降低了 32%～53%。Qureshi 等提出了 DRAM buffer[19]系统。在 DRAM buffer 中，DRAM 用作 CPU 和仅由 PCM 组成的主存之间的缓冲区。Ramos 等提出了 RaPP[20]系统，其特点是硬件驱动的页面放置策略，该策略依赖于内存控制器 MC 来监控访问模式，并使用改进的多级队列调度算法（Multilevel Queues，MQ）在 DRAM 和 PCM 之间迁移页面。

此外，还有许多基于不同负载和场景的混合内存系统：Chen 等提出了 ATMem[21]系统，该系统是一种用于优化图形应用程序中数据放置的自适应运行时框架；Doudali 等提出了 Kleio[22]系统，它将现有的、轻量级的、基于历史的混合内存数据分层方法与基于深度神经网络的智能放置决策系统相结合，从而实现更显著的性能提升，同时降低了由此产生的系统资源开销。

综上，混合内存管理的关键问题是高效的数据放置和迁移策略，将关键数据或频繁访问的数据维持在访问延迟低的高速存储设备 DRAM 中，可以有效提高应用程序的运行性能。

3 基于混合内存的缓存系统设计方案

3.1 平面混合缓存结构设计

在 Spark 集群中，存在 Driver 和 Executor 两种组件。由于 Driver 主要起到管理和传递信息的作用，Executor 才是实际计算和缓存 RDD 的组件[23]，因此本文只关注 Executor 上的缓存管理。

由于 Spark 的内存管理系统依赖于 JVM 堆的内存管理，因此在设计混合内存时，不能直接将 DRAM 和 NVM 两种物理存储设备直接进行分层混合或者平面混合，需要在 Spark 原有的内存管理基础上进行设计。

对于 DRAM 内存，本文主要关注 RDD 缓存的部分。图 1 给出了Spark 在持久化级别为 MEMORY_AND_DISK 时的 RDD 缓存结构。在 Spark 中，缓存块的读写由块管理器 BlcokManager 控制。缓存块写入时，BlcokManager 会将缓存块放置在存储内存（MemoryStore 模块）中，如果存储内存空间不足且无法动态占用执行内存，则会根据默认的 LRU 策略驱逐部分缓存块到硬盘（DiskStore 模块）中。缓存块读取时，BlcokManager 可以从存储内存或者硬盘中查找并读取缓存块。

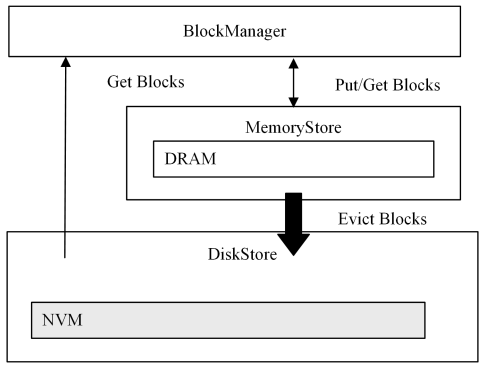


图 1 RDD 原有缓存结构

Fig. 1 Original cache structure of RDD

分层混合缓存模型是一种非侵入性的模块设计思路，也就是将 DRAM 缓存作为缓冲层引入块管理器 BlcokManager 和 NVM 缓存之间，如图 2 所示。由于其是分层组成，因此被称为分层混合缓存模型。在这种模型中，DRAM 缓存（MemoryStore模块）可以保持不变，NVM 缓存（NVMStore 模块）被放入 MemoryStore 之下。当读取目标缓存块时，直接从 DRAM 缓存中获取，若未在 DRAM 缓存中获取到该块，则在 NVM 缓存中查找并将其放置在 DRAM 缓存中；若在 NVM 缓存中也未找到，则重新计算该块。当放置目标缓存块时，直接将其放入 DRAM 缓存中，若 DRAM 缓存空间不足，则按照驱逐策略清理 DRAM 缓存，并将淘汰的缓存块驱逐到 NVM 缓存中；NVM 缓存空间不足时，根据缓存策略选择缓存块并驱逐。

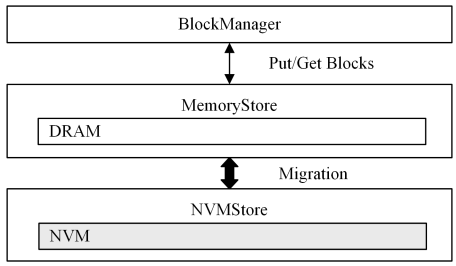


图 2 RDD 分层混合缓存模型

Fig. 2 RDD layer hybrid cache model

与分层混合模型不同，平面混合模型利用 NVM 作为 DRAM 的扩展，将 NVM 和 DRAM 平面组合在一起以实现 RDD 缓存，如图 3 所示。在平面混合模型中，DRAM 缓存和 NVM 缓存都将在同一个逻辑内存空间 HybridStore 模块中进行管理，BlcokManager 从 HybridStore 模块中读写缓存块。在 HybridStore 模块中缓存块，可以直接从 DRAM 缓存或 NVM 缓存中读写，并在 DRAM 和 NVM 之间实现双向的迁移机制，更细粒度地实现迁移策略[16]。要实现平面混合模型，需要在缓存系统中添加 HybridStore 模块，并在其中实现 MemoryStore 模块和 NVMStore 模块，还要设计两模块之间的迁移机制，因此平面混合模型要对原有的内存管理模型进行较大的改动。

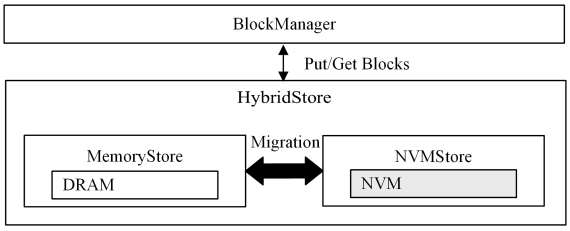


图 3　RDD 平面混合缓存模型

Fig. 3　RDD flat hybrid cache model

在 Spark 中同时利用 DRAM 内存和 NVM 作为缓存时，平面混合模型优于分层混合模型。原因有以下几点：

(1)分层混合模型存储空间利用效率低。分层混合内存系统需要额外的存储空间来存储元数据以跟踪 DRAM 和 NVM 中的数据块，该空间开销随着数据集规模的增大呈线性增长；而平面混合模型在系统中的使用方式与纯 DRAM 内存相同，不需要额外的空间来处理存储元数据。

(2)分层混合模型缓存块管理不够直接有效。分层混合模型的设计在每个缓存层次都要对 RDD 缓存块进行操作，也就是说，当缓存块在缓存策略中的权重变化跨多个缓存级别时，分层混合要求在每个层次之间依次迁移，这将极大地影响内存管理效率。

(3)分层混合内存管理开销过大。例如，在 Khan 等测试的分层混合内存系统中[17]，分配内存在展开过程中占用了 50%以上的时间，而在平面混合内存系统中，该代价通常不到 10%。

因此，本文选择平面混合缓存模型来设计 Spark 混合缓存系统，如图 3 所示。在本文设计的混合缓存系统中，RDD 的持久化级别为本文自行设计的“MEMORY_AND_NVM”。在缓存块的读取和写入时，都提供本文设计的 HybridStore 模块。在 HybridStore 模块中，管理 DRAM 缓存的 MemoryStore 模块和管理 NVM 模块的 NVMStore 模块是并列关系，两个模块都可以直接读取和写入缓存块，两者之间通过合适的缓存管理策略进行缓存块的驱逐和迁移。

3.2　混合缓存管理数据结构

3.2.1　缓存管理粒度 Block

本文为 RDD 缓存块设计了专门的数据结构 CacheBlockInfo 类，每个缓存块都会实例化一个该类，如图 4 所示。

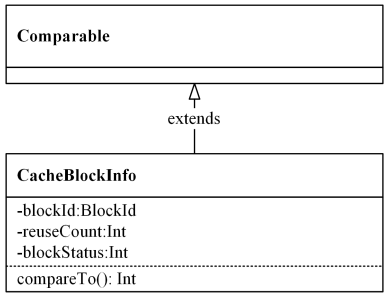


图 4　缓存块数据结构设计

Fig. 4　Design of cache block data structure

该类继承了 Comparable 类，类的主要元素包括：

(1)BlockId：标志缓存块信息。

(2)reuseCount：标志该缓存块在缓存管理中的权重。

(3)blockStatus：标志该缓存块的状态，表示该缓存块是否存在于 DRAM 或者 NVM 缓存中。

(4)compareTo 方法：根据权重，即 reuseCount 属性，决定两个缓存块的排序结果。

3.2.2　缓存管理集合 TreeSet

本文设计的缓存管理策略的大体思路是为每个缓存赋予一个权重值，权重值高的将被放置在 DRAM 缓存中，权重值低的将被驱逐到 NVM 缓存中。为了更高效地管理缓存块以及与本文设计的缓存管理策略相匹配，本文选择 Java TreeSet[24]作为缓存块集合的数据结构，如图 5 所示。TreeSet 是一种可变的集合数据结构，其特性是 Set 中的元素是有序的，元素排序依照构造时传入的比较器，也就是本文在缓存块数据结构中设计的 compareTo()比较器。

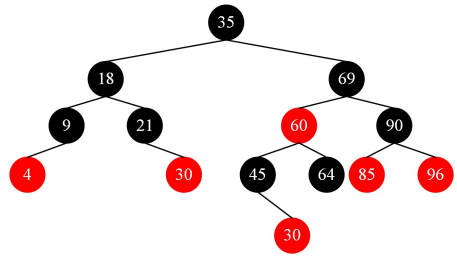


图 5　TreeSet 示例

Fig. 5　TreeSet example

TreeSet 底层通过红黑树(Red-Black Tree)[25]实现。红黑树是一种近似平衡的二叉查找树(Binary Search Tree)，其性质为每个节点的左右子树的高度差不会超过二者中较低的高度。

TreeSet 的好处是遍历该树时将得到一个有序序列，在根据权重驱逐缓存块时，遍历序列前面的就是权重最小的块[26]，该性质与本文对缓存块的管理策略契合。因此本文在每个 Executor 上维护了一个 BlockTreeSet，其数据结构类型为 Java TreeSet，其内容元素为缓存块数据结构 CacheBlockInfo，比较器为本文自定义的 CompareTo()，遍历 BlockTreeSet 将得到按权重由小到大排列的缓存块序列。需要注意的是，本文设计的 TreeSet 包含了多个层次，包括存储内存(DRAM 缓存)和 NVM 缓存，两者在使用时仅需通过块的 blockStatus 属性进行区分筛选，这样做的好处是避免维护两个有序的 TreeSet，降低排序和相互迁移的开销。

3.3　整体架构设计

如图 6 所示，本文的实现包括修改原有组件的接口和添加 4 个关键组件。其中 4 个关键组件以灰色模块表示，它们分别是：

(1)DAGAnalyzer，位于驱动节点 Driver 上，记录 DAGScheduler 提交时生成的原始 DAG，并根据 DAG 提供的 RDD 依赖关系生成每个 Stage 的缓存 RDD 依赖图。

(2) MasterCacheManager，位于驱动节点 Driver 上，根据缓存 RDD 依赖信息为每个 Stage 生成缓存 RDD 的权重值(也就是即将在第 4 节中介绍的 RDD 未来重用次数)，将其写入文件并分发给各个 Executor。

(3)ExecutorCacheManager，分布在每个执行节点 Executor 上，负责为 Executior 缓存空间中的缓存块计算权重值。

(4) HybridStore，分布在每个执行节点 Executor 上，根据缓存块的权重值实现放置和迁移策略的主要逻辑，根据权重值和迁移成本决定缓存块放置在 DRAM 缓存空间中还是 NVM 中，以及缓存块在混合缓存间的迁移。

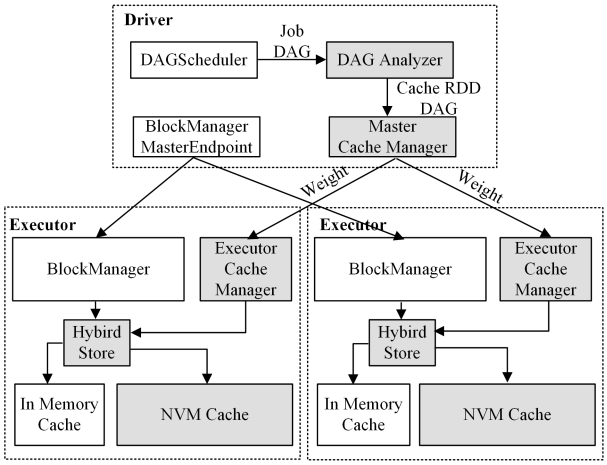


图 6 Spark 混合缓存系统架构图

Fig. 6 Spark hybrid cache system architecture

4 基于最小重用代价的混合缓存管理策略

4.1 基于最小重用代价的混合缓存策略优化

Spark 混合缓存系统的设计原则是将访问频繁的缓存块保持在 DRAM 缓存中，将访问不频繁的缓存块保持在 NVM 缓存中，而使用哪种指标来判断缓存块访问频繁便成为关键问题。

在基于内存的 Spark 缓存系统中，最直观的两种思路是使用最近最少使用策略 LRU 和最少使用策略(Least Frequently Used，LFU)[27]。LRU 和 LFU 分别将缓存块访问的新近度和数据访问频率作为指标来预测未来缓存块的访问频繁程度。而 Spark 中 RDD 有着明确的数据依赖性，也就是 DAG。图 7 给出了一个 DAG 示例，其中 RDD E 的计算依赖于 RDD D 和 RDD A，因此只有从 RDD B 计算出 RDD D 后，且 RDD A 已被计算时，才能开始计算 RDD E。RDD F 同理。

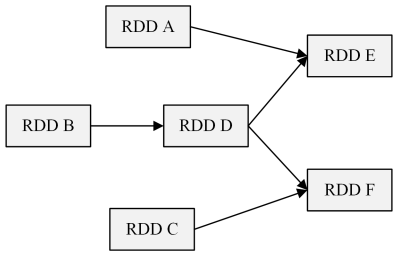


图 7 DAG 示意图

Fig. 7 DAG example

DAG 预测了 RDD 的未来访问模式，因此相比历史访问信息，DAG 可以更有效地指导缓存块的放置和驱逐。与 LFU 和 LRU 运用历史访问模式的指标类似，DAG 可以提供缓存 RDD 的未来引用次数和未来引用距离。

与 LFU 相对应的是未来重用距离，对于每个缓存 RDD，未来重用距离定义为在当前 Stage 和下一次使用该缓存 RDD 的 Stage 之间的 Stage 数量，也就是缓存 RDD 的未来重用的新近度，下次访问最近的缓存 RDD 应该放置在 DRAM 缓存中。与 LRU 相对应的是未来重用次数，对于每个缓存 RDD，未来重用次数定义为在当前 Stage 之后重新使用该缓存 RDD 的次数，也就是缓存 RDD 的未来重用的频率，未来重用次数最频繁的缓存 RDD 应该放置在 DRAM 缓存中。

为了探究哪种指标更有助于混合缓存系统达到更好的性能，本文首先分析了大数据系统的数据访问模式。以图 8 为例，本文跟踪了 Spark 在运行 PageRank 负载[28]时的缓存 RDD 数据访问模式，并对每个 RDD 的访问块数进行了标记。

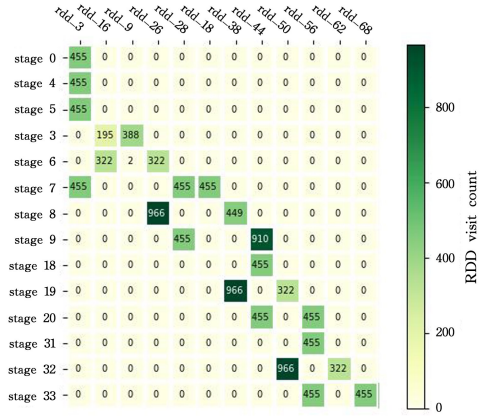


图 8 PageRank 的数据访问模式例图

Fig. 8 Example of data access mode of PageRank

从图 8 可以看出，首先，不是所有的缓存 RDD 都会被再次使用。从数据访问模式图中可以看出，部分缓存 RDD 仅在生成它的 Stage 时被使用，因此在下一个 Stage，这类缓存块没有必要存储在缓存空间中。其次，大多数缓存块的再次访问周期很短，很快就无法再次被引用。数据访问模式图中的大多数 RDD 的重用在 3 个 Stage 以内，生命周期很短。将这些不再被使用的缓存 RDD 长时间保存在缓存系统中会浪费缓存空间，降低命中率。对于未来重用次数为 0 的缓存块，可以直接将其从 DRAM 和 NVM 缓存中清除，及时释放缓存空间。另外，从纵坐标可以看出，Stage 的执行顺序并非按照 Stage 的序号排列，在不同的执行顺序下重用距离可能不同，因此未来重用距离并非一个好的指标，而执行顺序的不同并不影响重用次数的计算。

4.2 缓存块迁移代价

将 NVM 缓存中的缓存块迁移到 DRAM 缓存可以提升该缓存块的访问性能。但是，NVM 和 DRAM 之间的迁移也会导致整体的访问延迟增加，因此本文在 DRAM 缓存的性能提升和缓存块迁移成本之间进行权衡。在混合缓存系统中，缓存块的迁移包括从 DRAM 向 NVM 迁移和从 NVM 向 DRAM 迁移，本文将从这两个方面进行介绍。

4.2.1 由 DRAM 向 NVM 迁移

当目标缓存块需要缓存但 DRAM 缓存空间不足时，本文的原则是将未来重用次数最大的缓存块保留在 DRAM 中，将未来重用次数较少的缓存块保持在 NVM 中。因此，要比较的对象是准备缓存的缓存块 B_{toput} 和 DRAM 缓存中未来重用次数最少的缓存块 B_{low}(为满足 B_{toput} 的空间需求，B_{low} 可能是多个缓存块的集合，此时 B_{low} 的未来重用次数取缓存块集中

最高的作为该集合的未来重用次数）。如果B_{toput}的未来重用次数小于B_{low}，那么直接将其放置在 NVM 缓存中；如果B_{toput}与B_{low}的未来重用次数相等，就要考虑是将B_{toput}直接放置在 NVM 缓存中，还是将B_{low}迁移到 NVM 缓存中，再在 DRAM 缓存中放置B_{toput}。由于二者未来重用次数相同，这两种方案后续的访存性能是一致的，但两者的放置和迁移开销不同。将B_{toput}放置在 NVM 中的开销为：在 NVM 中写入B_{toput}。将B_{toput}放置在 DRAM 中，B_{low}迁移到 NVM 中的开销为：从 DRAM 清除B_{low}、在 NVM 中写入B_{low}、在 DRAM 中写入B_{toput}的迁移开销。显然，后者的开销远大于前者，因此，当B_{toput}和B_{low}未来重用次数相同时，应当选择将B_{toput}放置在 NVM 中，将B_{low}维持在 DRAM。

如果B_{toput}的未来重用次数大于B_{low}，则要比较将B_{toput}放置在 DRAM 中带来的性能提升和缓存块迁移带来的开销。NVM 和 DRAM 中的相关性能指标具体如下：

B_{toput}表示尝试缓存的缓存块，B_{low}表示 DRAM 缓存中重用次数最少的缓存块（或缓存块集合）。

R_{toput}和R_{low}分别表示B_{toput}和B_{low}的未来重用次数，这两项数据从 DAG 信息中获得。

S_{toput}和S_{low}分别表示B_{toput}和B_{low}的大小，这两项数据可以从缓存块对象中获得。

RPM_n表示 NVM 中每 MB 数据的读时间，RPM_d表示 DRAM 中每 MB 数据的读时间；WPM_n表示 NVM 中每 MB 数据的写时间，WPM_d表示 DRAM 中每 MB 数据的写时间；EPM_d表示 DRAM 中每 MB 数据的清除时间，EPM_n表示 NVM 中每 MB 数据的清除时间。这 6 项数据将从 Spark 运行时计算得出。

基于以上数据，本文首先分析了将B_{low}迁移到 NVM 中，再在 DRAM 中缓存B_{toput}的迁移方案，该方案的总访问时间$T_{\text{Migration}}$如式(1)所示：

$$T_{\text{Migration}}=RPM_d*S_{\text{toput}}*R_{\text{toput}}+RPM_n*S_{\text{low}}*R_{\text{low}}+WPM_n*S_{\text{low}}+EPM_d*S_{\text{low}}+WPM_d*S_{\text{toput}} \tag{1}$$

然后分析了B_{toput}放置在 NVM 中的不迁移方案，该方案的总访问时间$T_{\text{NotMigration}}$如式(2)所示：

$$T_{\text{NotMigration}}=RPM_n*S_{\text{toput}}*R_{\text{toput}}+RPM_d*S_{low}*R_{low}+WPM_n*S_{\text{toput}} \tag{2}$$

最后计算两种方案的总时间差$MigrationBenefit_{\text{toput}}$，也就是放置缓存块时的迁移收益，如式(3)所示：

$$MigrationBenefit_{\text{toput}}=T_{\text{NotMigration}}-T_{\text{Migration}} \tag{3}$$

迁移收益$MigrationBenefit_{\text{toput}}$为正值意味着将$B_{\text{low}}$迁移到 NVM 中，再在 DRAM 中缓存$B_{\text{toput}}$的迁移方案比$B_{\text{toput}}$放置在 NVM 中的不迁移方案节省了更多的数据访问时间，应当采用迁移方案。迁移收益为负值说明不迁移方案的数据访问时间更短，应当采用不迁移方案。

4.2.2　由 NVM 向 DRAM 迁移

随着 Spark 计算的进行，每个 Stage 缓存空间中的缓存块的未来重用次数可能会发生改变。这就可能导致 NVM 中某些缓存块的未来重用次数多于 DRAM 中的某些缓存块。此外，DRAM 中也可能有空闲空间。出现这两种情况时，都应该将 NVM 中未来重用次数多的缓存块迁移到 DRAM 中。

若 DRAM 中无空闲缓存空间，此时比较的对象是读取到的 NVM 缓存块B_{toread}和 DRAM 中未来重用次数最少的缓存块B_{low}。如果B_{toread}的未来重用次数少于B_{low}，那么直接在 NVM 中读取；如果B_{toread}与B_{low}的未来重用次数相等，显然后者的开销远大于前者，此时应当选择在 NVM 中直接读取B_{toread}，将B_{low}维持在 DRAM；同理，如果B_{toread}的未来重用次数多于B_{low}，则要比较将B_{toread}放置在 DRAM 中带来的性能提升和缓存块迁移带来的开销。

下文中 NVM 和 DRAM 的相关性能指标与前一小节大体相似。

B_{toread}表示 NVM 缓存中读取的缓存块，B_{low}表示 NVM 重用次数最低的缓存块。

R_{toread}和R_{low}分别表示B_{toread}和B_{low}的未来重用次数，这两项数据将从 DAG 信息中获得。

S_{toread}和S_{low}分别表示B_{toread}和B_{low}大小，这两项数据可以从缓存块对象中获得。

RPM_n，RPM_d，WPM_n，WPM_d，EPM_d，EPM_d这 6 项数据与前一小节相同。

基于以上数据，本文首先分析了将B_{toread}迁移到 DRAM 中并读取，将B_{low}迁移到 NVM 中的迁移方案，并计算了该方案的总访问时间$T_{\text{Migration}}$：

$$T_{\text{Migration}}=RPM_d*S_{\text{toread}}*R_{\text{toread}}+RPM_n*S_{\text{low}}*R_{\text{low}}+WPM_n*S_{\text{low}}+EPM_d*S_{\text{low}}+WPM_d*S_{\text{toput}}+EPM_n*S_{\text{toread}}+RPM_d*S_{\text{toread}} \tag{4}$$

然后分析了直接在 NVM 中读取B_{toread}的不迁移方案，并计算了该方案的总访问时间$T_{\text{NotMigration}}$：

$$T_{\text{NotMigration}}=RPM_n*S_{\text{toread}}*R_{\text{toread}}+RPM_d*S_{\text{low}}*R_{\text{low}}+RPM_n*S_{\text{toread}} \tag{5}$$

最后计算两种方案的总时间差$MigrationBenefit_{\text{toread}}$，也就是读取缓存块时的迁移收益。

$$MigrationBenefit_{\text{toread}}=T_{\text{NotMigration}}-T_{\text{Migration}} \tag{6}$$

迁移收益$MigrationBenefit_{\text{toread}}$为正值意味着将$B_{\text{toread}}$迁移到 DRAM 中并读取，将$B_{\text{low}}$迁移到 NVM 的迁移方案比在 NVM 读取$B_{\text{toread}}$的不迁移方案节省了更多的数据访问时间，应当采用迁移方案。迁移收益为负值说明不迁移方案的数据访问时间更短，应当采用不迁移方案。

4.2.3　基于未来重用信息和迁移成本的最小重用代价策略

综合前两小节提出的基于 DAG 信息的缓存块的未来重用次数和混合内存中缓存块迁移代价计算，本文提出基于最小重用代价（Least Reuse Cost，LRUC）的混合缓存策略。该策略的原则为：将未来重用次数最多的缓存块放置或保留在 DRAM 缓存中，将未来重用次数少的缓存块迁移或保持在 NVM 缓存中，在迁移时考虑潜在的迁移成本，使缓存块的整体重用代价最小，缓存系统的整体读写性能最优。

4.3　基于最小重用代价的混合缓存管理系统实现

在本文设计的平面混合内存中，对缓存块的操作包括放置、驱逐和迁移，如图 9 所示，本节将分别介绍缓存块获取空间以及缓存块写入和读取过程中涉及的缓存块操作。

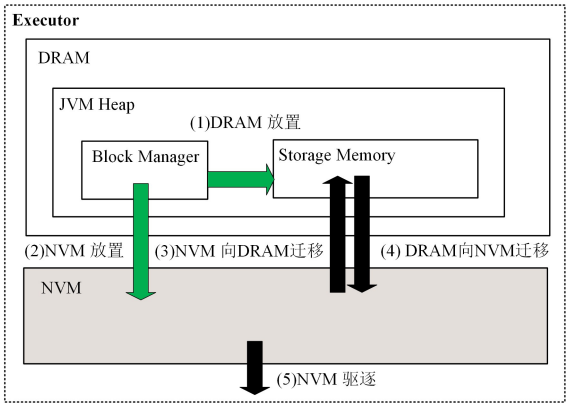


图 9 Executor 的放置和迁移图

Fig. 9 Executor's placement and migration

(1)缓存块获取空间。在放置缓存块或将其迁移到目标缓存空间时,首先要检查目标缓存空间是否有可用空间,如果空间充足则可直接进行放置;如果可用空间不足时,则根据 BlockTreeMap 中的缓存块权重信息(也就是缓存块的未来重用次数)输出可供驱逐的缓存块列表。

(2)缓存块写入过程。写入缓存块时,首先尝试将其放置在 DRAM 缓存中,如果 DRAM 缓存空间充足,则将其写入 DRAM 缓存中,如图 9 中过程(1)所示;在这个过程中如果 DRAM 缓存空间不足,为了给缓存块腾出空间,要对 DRAM 缓存中未来重用次数最少的缓存块进行驱逐,也就是缓存块从 DRAM 缓存向 NVM 缓存迁移的过程,如图 9 中(4)过程所示。如果无法为目标缓存块腾出足够的空间,则将缓存块尝试放置在 NVM 缓存中,如果 NVM 空间充足,则将其写入 NVM 中,如图 9 中过程(2)所示;如果 NVM 空间不足,则在 NVM 缓存中尝试清理空间,如图 9 中过程(5)所示;如果无法腾出目标缓存块所需要的 NVM 空间,则不再缓存目标缓存块。

(3)缓存块读取过程。当读取到 NVM 中某个缓存块的未来重用次数大于 DRAM 缓存中部分缓存块或当前 DRAM 缓存空间有空闲时,可能触发由 NVM 缓存向 DRAM 缓存迁移的操作,如图 9 中过程(3)所示,在此过程中也可能伴随 DRAM 中未来重用次数较少的缓存块向 NVM 迁移的过程,如图 9 中过程(4)所示。

4.3.1 目标缓存块获取可用缓存空间

在放置或迁移缓存块到目标缓存空间之前,要为目标缓存块获取可用缓存空间,如果获取成功则可接着进行放置或迁移;若获取空间失败则输出可驱逐的缓存块列表供后续驱逐操作使用。

本文分别为 DRAM 缓存和 NVM 缓存设计了获取可用空间的函数 *AskMemSpace* 和 *AskNVMSpace*。当缓存块放置或迁移到 DRAM 缓存空间时,首先调用负责为缓存块获取 DRAM 缓存空间的函数 *AskMemSpace*。同样地,当缓存块放置或迁移到 NVM 缓存空间时,要调用负责为缓存块获取 NVM 缓存空间的函数 *AskNVMSpace*。

AskMemSpace 函数的算法如算法 1 所示,*AskNVMSpace* 函数与之类似。首先检查 DRAM 缓存空间中现有的空闲空间 Memory. freeSpace,如果此空间大于目标缓存块的大小则获取空间成功,可以直接进行放置,无需进行驱逐;如果内存中空闲空间小于目标缓存块的大小,则启动待驱逐缓存块的标记过程,从头遍历 BlockTreeSet,将 blockStatus 属性指示为当前缓存层次且将未来重用次数小于目标缓存块的缓存块依次列入待清理列表(CleanList)。当 CleanList 所占用的空间大于目标缓存块的占用空间或者当前缓存块的未来重用次数大于目标缓存块的未来重用次数时停止遍历。需要注意的是,为避免循环淘汰,属于当前 Stage 的 Block 也不会被列入待清理列表 CleanList 中。遍历结束时,输出获取空间失败以及待清理列表 CleanList。

算法 1 *AskMemSpace*

输入:BlockTreeSet 缓存块管理集合,targetBlock 要添加的 Block

输出:enoughSpace 是否有可用空间,CleanList 待清理缓存块列表

```
1. function AskMemSpace(BlockTreeSet,targetBlock)
2.     CleanList ← {}
3.     if targetBlock. size > Memory. freeSpace then
4.         return enoughSpace ← true,CleanList
5.     else
6.         for block ∈ BlockTreeSet do
7.             if block. blockStatus = Memory and targetBlock. reuseC-
               ount > block. reuseCoount then
8.                 if block. rdd∈CurrentStage then
9.                     CleanList. add(block)
10.            end if
11         . end if
12.        end for
13.        return enoughSpace ← false,CleanList
14.    end if
15. end function
```

需要注意的是,获取空间失败时不一定会进行缓存块的驱逐。首先,CleanList 的空间小于目标缓存块大小时,不会进行驱逐;其次,对于 DRAM 缓存中的缓存块向 NVM 缓存迁移的过程,还要进一步计算迁移代价来决定是否迁移。

此外,由于展开内存和存储内存具有动态占用的特点,因此在 Unroll 的过程中也会触发获取缓存空间函数 *AskMemSpace*。Spark 中 RDD 在缓存时,会从迭代器 Iterator 形式转换为连续的物理空间存储,这个过程是在展开内存中完成的。但由于展开内存和存储内存共享内存空间,在进行 Unroll 的过程中要申请展开内存空间,所以在 Unroll 过程之前,系统将调用 *AskMemSpace* 函数检查缓存空间能否占用。若获取缓存空间函数返回结果为 True,则获取空间成功,可以进行 Unroll 操作;否则获取空间失败。在本文设计的系统中,缓存块的缓存不一定要经过 DRAM 缓存空间,因此 Unroll 获取空间失败说明该块重要性(也就是重用代价)小于 DRAM 缓存中的块,该块不应该放置在 DRAM 缓存中,直接尝试将该块放置在 NVM 缓存空间中即可。

4.3.2 缓存块放置过程

本文基于最小重用代价策略设计了缓存块放置机制,当一个缓存块需要缓存时,首先调用获取空间函数,若获取空间成功,则直接放置在 DRAM 缓存中;若获取空间失败,则根据待清理的缓存块列表以及迁移代价决定是否将 DRAM 缓存

中的部分缓存块迁移到 NVM 缓存中来为目标缓存块腾出空间；否则将目标缓存块放置在 NVM 缓存中，NVM 缓存空间不足时直接将未来重用次数最低的缓存块清除。

缓存块放置的具体过程如图 10 所示，当系统接写入目标缓存块的指令时，首先考虑将它放置在 DRAM 缓存中，调用 *AskMemSpace* 函数获取 DRAM 缓存中的可用空间，若获取空间成功，则直接将目标缓存块放置在 DRAM 缓存中；若获取 DRAM 缓存空间失败，则根据 *AskMemSpace* 输出的待清理列表 CleanList 计算可释放的缓存空间。如果可释放的缓存空间小于目标缓存块的大小，则将目标缓存块放置在 NVM 缓存中；如果可释放的缓存空间大于目标缓存块的大小，则进一步计算迁移收益 $MigrationBenefit_{toput}$。如果迁移收益为正，则将 CleanList 中的缓存块驱逐到 NVM 缓存中（该过程也是缓存块从 DRAM 缓存迁移到 NVM 缓存的过程），并将目标缓存块放置在 DRAM 缓存中；否则将尝试将目标缓存块放入 NVM 缓存中。

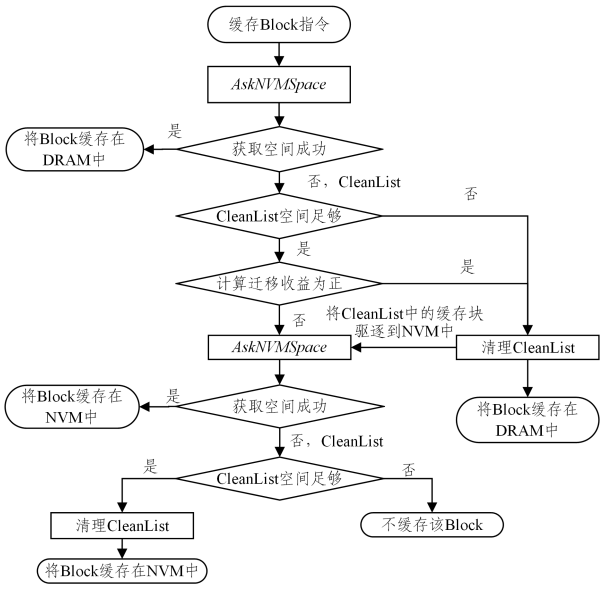


图 10　缓存块放置流程图

Fig. 10　Flow chart of cache block placement

目标缓存块放置到 NVM 缓存中的过程与放入 DRAM 缓存的过程类似。首先调用 AskNVMSpace 获取 NVM 缓存中的可用空间，如果获取到的 NVM 空间大于目标缓存块的大小，直接将它存放在 NVM 中；若获取 NVM 空间失败，则根据 *AskNVMSpace* 输出的待清理列表 CleanList 计算可驱逐的空间。如果该空间大于该目标缓存块的大小，则驱逐 CleanList 中的缓存块，为目标缓存块腾出空间；如果 CleanList 中的占用空间小于目标缓存块的大小，说明目标缓存块的未来重用次数少于 NVM 缓存空间中的缓存块，根据放置策略，不再缓存目标缓存块。

4.3.3　缓存块读取过程

读取目标缓存块时，DRAM 缓存中的缓存块可直接读取；读取 NVM 缓存中的缓存块时，如果此时 DRAM 缓存有可用空间或者目标缓存块的未来重用次数较多，则会触发缓存块从 NVM 缓存向 DRAM 缓存的迁移，同时也可能伴随从 DRAM 缓存中淘汰的缓存块向 NVM 缓存迁移的过程。

如图 11 所示，当系统接收到读取缓存块的指令时，首先尝试在 DRAM 缓存中查找该块，如果找到该块则在 DRAM 缓存中直接读取，否则将在 NVM 缓存中查找；如果在 NVM 缓存中也未找到该块，则开始重新计算。若在 NVM 缓存中找到该缓存块，则调用 *AskMemSpace* 为这个块获取 DRAM 缓存中的可用空间。如果获取空间成功，则直接放置在 DRAM 缓存中并读取；若获取 DRAM 缓存空间失败，则根据 *AskMemSpace* 输出的待清理列表 CleanList 计算可释放的缓存空间。如果可释放的缓存空间小于目标缓存块的大小，则直接从 NVM 缓存中读取；如果可释放的缓存空间大于目标缓存块的大小，则进一步计算迁移收益 $MigrationBenefit_{toread}$。如果迁移收益为正，则将 CleanList 中的缓存块驱逐到 NVM 缓存中（该过程与上一小节相同），并将目标缓存块从 NVM 缓存迁移到 DRAM 缓存中并读取；如果迁移收益为负，则直接从 NVM 缓存中读取目标块。

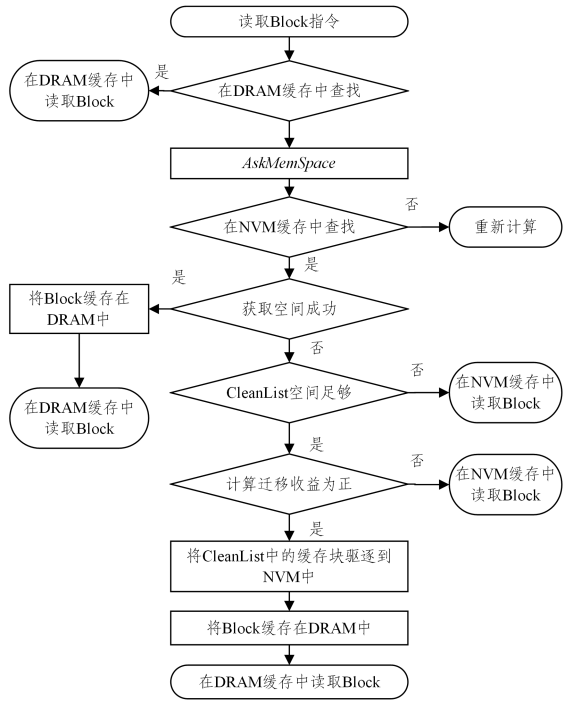


图 11　Block 读取过程

Fig. 11　Block reading process

5　实验设计与分析

5.1　实验环境

本节首先介绍了本文所使用的硬件平台和 Spark 集群设定，然后介绍了 SparkBench 数据集并展示了不同负载的数据访问特征，最后选择了其中 4 种负载作为本文的实验数据集。

5.1.1　实验平台配置

本文实验平台由 5 台服务器组成，每台服务器都配置了一个主频为 3.70 Hz 的 10 核心 20 线程 Intel(R) Core(TM) i9-10900X CPU，128G DDR4 3 200 MHz 内存和 Samsung 860 EVO 1TB SATA SSD，操作系统为 Ubuntu 16.04.1。服务器之间通过全千兆以太网网络连接。由于 NVM 硬件平台暂不可用，NVM 通过 RamDisk 模拟。RamDisk[29] 是 Linux 系统中一种在内存中划出一部分空间作为存储分区使用的技术。

根据 Sehgal 等[30]的工作可知，RamDisk 的物理存储设备是 DRAM，其物理延迟是内存级别的，又由于其使用了块存储的方式，降低了整体 I/O 效率，整体读写延迟与 NVM 接近，因此可用于模拟 NVM 缓存。此外，由于本文中 NVM 读写延迟等参数是在运行负载时动态获得的，因此如果迁移到物理 NVM 设备，也可以适应其性能表现。

本文的 Spark 集群以 Yarn-Client 方式部署，包括一个 Driver 和 4 个 Executor，分布于 5 台服务器上，每个 Executor 使用内存根据实验的不同在 4GB～16GB 之间。JDK 版本为 Oracle JDK 1.8.0_281，Hadoop 版本为 2.7.2，使用 HDFS 用于测试集的输入和输出数据。为了避免执行内存等参数不同对实验结果的影响，本文使用静态内存管理模型，运行的 Spark 版本为 1.5.2，调整 *spark.storage.memoryFraction* 和 *spark.executor.memory* 参数来设置 DRAM 缓存空间的大小，同时确保所有实验的执行内存大小相同。

5.1.2 实验数据集和负载选择

本文实验采用的测试集为 SparkBench[31]，这是一款开源的测试集，主要包括机器学习(Machine Learning)、图计算(Graph Computation)、结构化查询(SQL Queries)、流计算(Streaming Application)等。

输入数据规模如表 1 所列。

表 1 选定负载的输入数据规模

Table 1 Input data size of selected workloads

WorkLoad	Input Data Size
PageRank/GB	1.9
ConnectedComponent/GB	6.1
PregelOperation/GB	4.0
SVD++/GB	1.4

本文首先分析了基于不同硬件和不同缓存管理策略的整体运行表现，通过和基于 SSD 硬盘的缓存系统的对比，验证了在 Spark 缓存系统中引入 NVM 缓存对系统性能的提升；通过和默认策略下的混合缓存系统以及相同容量的纯 DRAM 缓存系统的对比，验证了本文设计的基于最小重用代价的混合缓存策略的有效性。然后对不同缓存系统相应的缓存命中率进行分析，验证了本文设计的系统对于缓存命中率的提升。最后探究了不同 DRAM-NVM 比例对混合缓存系统的性能影响。

5.2 实验结果与分析

5.2.1 不同缓存系统的整体运行表现

为了测试和验证本文设计的少量 DRAM 和大量 NVM 组成的混合缓存系统的性能优势以及本文提出的基于最小重用代价的混合缓存管理策略(LRUC)，本小节测试了 4 种使用不同存储设备和不同管理策略的系统，对于 PageRank，PregelOperation，ConnectedComponent，SVDPlusPlus 这 4 种类型的负载，4 种缓存系统的运行时间表现如图 12 所示。从图 12 中可以看出，与默认缓存策略下的 4GB DRAM＋16GB NVM 混合缓存系统相比，相同容量的 4GB DRAM＋16GB SSD 混合缓存系统，对于 PageRank，PregelOperation，ConnectedComponent，SVDPlusPlus 这 4 种负载，分别有 65.21%，55.56%，46.83%，44.64%的性能提升，性能提升平均为 53.06%。

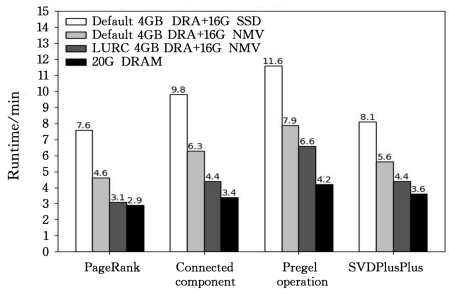


图 12 4 种负载在 4 种缓存系统下的运行时间

Fig. 12 Running time of 4 types of workload under 4 types of cache systems

总的来说，相比 Spark 默认混合缓存系统，使用 DRAM-NVM 混合缓存系统和本文设计的 LRUC 策略的系统性能平均提升了约 88%。另外，使用本文设计的 LRUC 策略的混合系统，只需要使用 1/4 的 DRAM 和 3/4 的 NVM 作为缓存，就能达到全部 DRAM 缓存约 79%的性能表现，再次说明了本文设计的混合缓存系统和 LRUC 缓存策略整体的性能优势。

5.2.2 不同缓存系统的缓存命中表现

为了验证本文设计的缓存管理策略对缓存命中率的提升，本文测试了 PageRank，PregelOperation，ConnectedComponent，SVDPlusPlus 这 4 种负载分别在 5.2.1 节设计的默认 4GB DRAM＋16GB NVM，LRUC 和全部 DRAM 缓存系统上的缓存命中情况，结果如图 13 所示。这 3 种缓存系统的配置分别为：

Default：使用默认缓存策略的 4GB DRAM＋16GB NVM 混合缓存。

LRUC：使用最小重用代价策略的 4GB DRAM＋16GB NVM 混合缓存。

All DRAM：使用默认策略的 20GB DRAM 缓存。

整体命中率表现：从不同负载的缓存命中表现图可以看出，使用 LRUC 策略的 4GB DRAM＋16GB NVM 混合缓存系统在 4 种负载下的整体缓存命中率(包括在 NVM 缓存中命中和在 DRAM 缓存中命中，在图 13 中表示为中间浅绿色和右侧深绿色部分)相比默认策略相同配置的混合缓存系统分别提升了 12.41%，61.4%，1.60%和 4.60%，平均提升了 6.19%，甚至高于只使用 DRAM 的缓存系统 4.75%。这是由于很多缓存块在第一次计算时会被认定为缓存未命中，因此即使是在全 DRAM 的情况下仍有大约 30%的缓存未命中；其次是由于本文设计的基于最小重用代价的缓存策略及时从 DRAM 和 NVM 中驱逐了不会再使用的 RDD 缓存块，因此整体的缓存命中表现也略有提升，而整体的性能提升更依赖缓存策略的合理放置和迁移，使访问频繁的数据块在 DRAM 缓存中访问。

DRAM/NVM 命中率表现：从缓存命中表现图可以看出，相比基于默认策略的混合缓存系统，基于 LRUC 策略的混合缓存系统在 4 种负载下 DRAM 命中率分别提升了 28.13%，19.66%，9.91%，21.57%，平均提升了 19.82%；与纯 DRAM 缓存系统的 DRAM 缓存命中率之间的差距分别为 13.32%，16.54%，12.17%，13.21%，平均差距为

13.81%。这是因为 LRUC 缓存策略将未来访问更频繁的缓存块维持在 DRAM 缓存中。

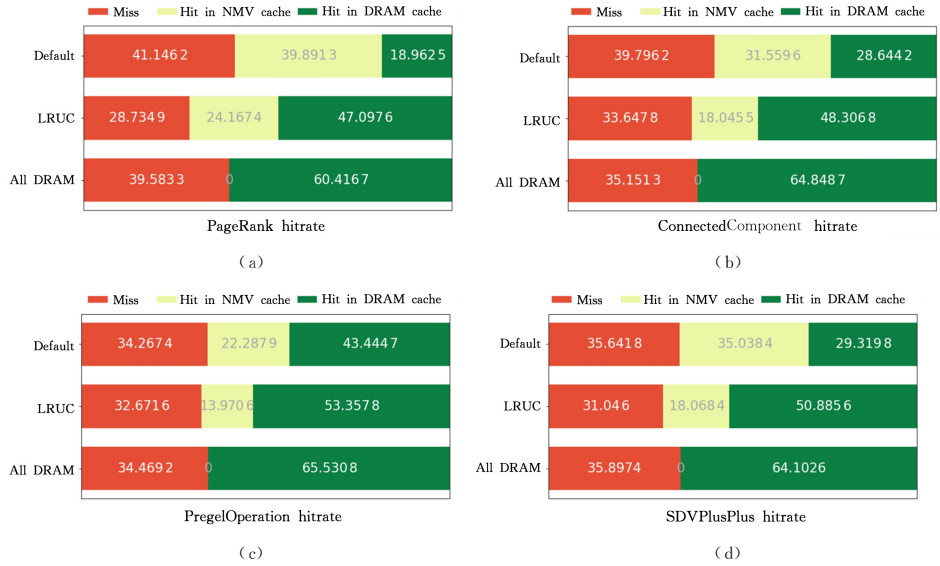
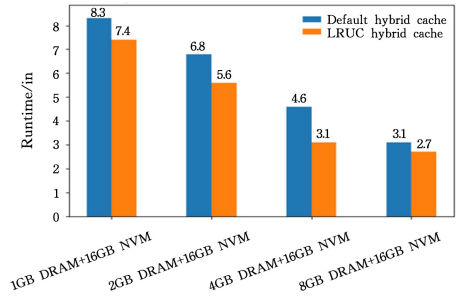


图 13　不同负载的缓存命中表现(电子版为彩图)

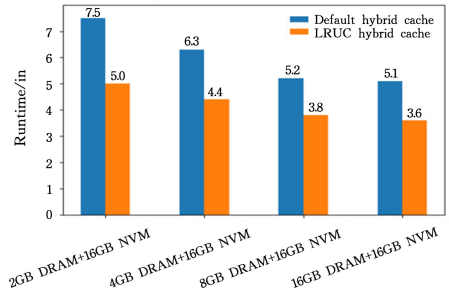
Fig. 13　Cache hit performance with different workloads

5.2.3　不同 DRAM/NVM 比例的混合缓存性能表现

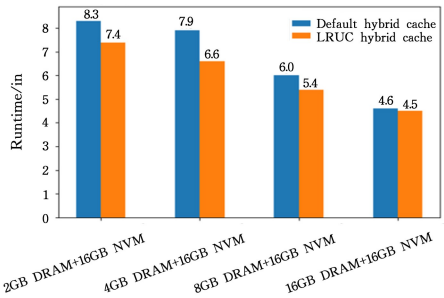
本小节验证了本文设计的缓存管理策略的可扩展性,并探索了 DRAM/NVM 比例不同对混合缓存系统性能的影响,为混合缓存系统调优打下基础。本文比较了默认缓存策略和 LRUC 策略在不同 DRAM/NVM 比例的混合缓存系统上运行 PageRank,PregelOperation,ConnectedComponent,SVDPlusPlus 这 4 种负载的性能表现,如图 14 所示。



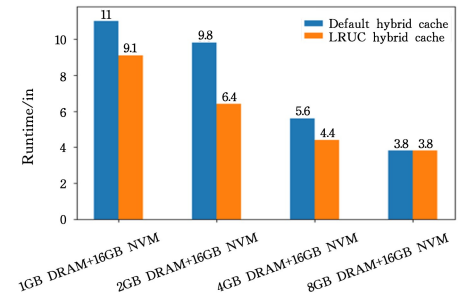
(a)PageRank runtime under different DRAM/NMV ratios



(b)ConnectedComponent runtime under different DRAM/NMV ratios



(c)PregelOperation runtime under different DRAM/NMV ratios



(d)SVDPlusPlus runtime under different DRAM/NMV ratios

图 14　不同 DRAM/NVM 比例的混合缓存性能表现

Fig. 14　Hybrid cache performance with different DRAM/NVM ratios

两种缓存策略的相同之处在于集群中内存缓存的可用性越低,工作负载运行的时间越长。在选定的缓存比例中,本文设计的 LRUC 策略的性能表现均优于 Spark 中的默认缓存管理策略,具体的性能提升幅度因缓存比例以及工作负载而异。例如,对于 PageRank 负载,LRUC 策略性能提升最大的缓存系统是 4GB DRAM+16GB NVM,也就是 DRAM/NVM 比例为 1∶4 时。对于 PregelOperation,ConnectedComponent 和 SVDPlusPlus,最佳 DRAM/NVM 比例分别是 1/8,1/4 和 1/8。

总的来说,在不同测试负载和不同 DRAM-NVM 比例下,最小重用代价策略 LRUC 的性能表现相较于 Spark 默认缓存管理策略都有不同幅度的提升,且每个负载都存在一个提升幅度最大的最佳比例。该项测试也为通过调整 DRAM/NVM 比例来优化混合缓存系统奠定了基础。

结束语 本文设计的混合内存系统有效利用了 NVM 字节可寻址、更高容量的优势以及 DRAM 的低延迟、高带宽的优势，优化了缓存系统的读写性能；缓存管理策略使需要频繁访存的缓存块保留在 DRAM 缓存中，同时极大地减少了缓存块迁移带来的性能开销，提高了缓存效率，提升了应用程序的运行速度。

此外，对于 DRAM-NVM 的比例，本文采取了手动调优的方式，而在大规模集群中，手动调优可能无法完全满足调优需求。因此，未来可能通过机器学习方式实现混合内存比例的自动化调优。

参 考 文 献

[1] SHVACHKO K, KUANG H, RADIA S, et al. The hadoop distributed file system[C]// 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies(MSST). IEEE, 2010: 1-10.

[2] DEAN J, GHEMAWAT S. MapReduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113.

[3] LANKHORST M H R, KETELAARS B W, WOLTERS R A M. Low-cost and nanoscale non-volatile memory concept for future silicon chips[J]. Nature Materials, 2005, 4(4): 347-352.

[4] CHEN A. A review of emerging non-volatilememory(NVM) technologies and applications[J]. Solid-State Electronics, 2016, 125: 25-38.

[5] IZRAELEVITZ J, YANG J, ZHANG L, et al. Basic performance measurements of the intel optane DC persistent memory module [J]. arXiv: 1903. 05714, 2019.

[6] WU X, LI J, ZHANG L, et al. Power and performance of read-write aware hybrid caches with non-volatile memories[C]// 2009 Design, Automation & Test in Europe Conference & Exhibition. IEEE, 2009: 737-742.

[7] MENG H T, YU S P, LIU F, et al. Research on Memory Management and Cache Replacement Policies in Spark[J]. Computer Science, 2017, 44(6): 31-35, 74.

[8] XU L, LI M, ZHANG L, et al. Memtune: Dynamic memory management for in-memory data analytic platforms[C]// 2016 IEEE International Parallel and Distributed Processing Symposium(IPDPS). IEEE, 2016: 383-392.

[9] YU Y, WANG W, ZHANG J, et al. LRC: Dependency-aware cache management for data analytics clusters[C]// IEEE INFOCOM 2017IEEE Conference on Computer Communications. IEEE, 2017: 1-9.

[10] YU Y, WANG W, ZHANG J, et al. LERC: coordinated cache management for data-parallel systems[C]// 2017 IEEE Global Communications Conference(GLOBECOM 2017). IEEE, 2017.

[11] WANG B, TANG J, ZHANG R, et al. LCRC: A dependency-aware cache management policy for Spark[C]// 2018 IEEE International Conference on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications(ISPA/IUCC/BDCloud/SocialCom/SustainCom). IEEE, 2018: 956-963.

[12] PEREZ T B G, ZHOU X, CHENG D. Reference-distance eviction and prefetching for cache management in spark[C]// Proceedings of the 47th International Conference on Parallel Processing. 2018: 1-10.

[13] ZHAO Y, DONG J, LIU H, et al. Performance Improvement of DAG-Aware Task Scheduling Algorithms with Efficient Cache Management in Spark[J]. Electronics, 2021, 10(16): 1874.

[14] SONG Y, YU J, WANG J J, et al. Memory management optimization strategy in Spark framework based on less contention[J]. The Journal of Supercomputing, 2023, 79(2): 1504-1525.

[15] YU B, FENG G, CAO H, et al. Chukonu: a fully-featured high-performance big data framework that integrates a native compute engine into Spark[C]// Proceedings of the VLDB Endowment. 2021: 872-885.

[16] WANG C, CUI H, CAO T, et al. Panthera: Holistic memory management for big data processing over hybrid memories[C]// Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2019: 347-362.

[17] KHAN M M, ALAM M A U, NATH A K, et al. Exploration of memory hybridization for RDD caching in Spark[C]// Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management. 2019: 41-52.

[18] CHEN L, ZHAO J, WANG C, et al. Unified Holistic Memory Management Supporting Multiple Big Data Processing Frameworks over Hybrid Memories[J]. ACM Transactions on Computer Systems(TOCS), 2022, 39(1/2/3/4): 1-38.

[19] QURESHI M K, SRINIVASAN V, RIVERS J A. Scalable high performance main memory system using phase-change memory technology[C]// Proceedings of the 36th Annual International Symposium on Computer Architecture. 2009: 24-33.

[20] RAMOS L E, GORBATOV E, BIANCHINI R. Page placement in hybrid memory systems[C]// Proceedings of the International Conference on Supercomputing. 2011: 85-95.

[21] CHEN Y, PENG I B, PENG Z, et al. Atmem: Adaptive data placement in graph applications on heterogeneous memories [C]// Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization. 2020: 293-304.

[22] DOUDALI T D, BLAGODUROV S, VISHNU A, et al. Kleio: A hybrid memory page scheduler with machine intelligence[C]// Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing. 2019: 37-48.

[23] CHAE S J, CHUNG T S. Dsmm: A dynamic setting for memory management in apache spark[C]// 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, 2019: 143-144.

[24] SENAPATI R K, PATI U C, MAHAPATRA K K. Listless block-tree set partitioning algorithm for very low bit rate embedded image compression[J]. AEU-International Journal of Electronics and Communications, 2012, 66(12): 985-995.

[25] HANKE S. The performance of concurrent red-black tree algorithms[C]// International Workshop on Algorithm Engineering. Berlin: Springer, 1999: 286-300.

[26] GENG Y，SHI X，PEI C，et al. Lcs：an efficient data eviction strategy for spark[J]. International Journal of Parallel Programming，2017，45(6)：1285-1297.

[27] RUAN K. Cache Optimization in Spark[D]. Shanghai：Shanghai Jiaotong University，2020.

[28] GLEICH D F. PageRank beyond the Web[J]. SIAM Review，2015，57(3)：321-363.

[29] WICKBERG T，CAROTHERS C. The RAMDISK storage accelerator：a method of accelerating I/O performance on HPC systems using RAMDISKs[C]// Proceedings of the 2nd International Workshop on Runtime and Operating Systems for Supercomputers. 2012：1-8.

[30] SEHGAL P，BASU S，SRINIVASAN K，et al. An empirical study of file systems on NVM[C]// 2015 31st Symposium on Mass Storage Systems and Technologies(MSST). IEEE，2015：1-14.

[31] LI M，TAN J，WANG Y，et al. Sparkbench：a comprehensive benchmarking suite for in memory data analytic platform spark [C]// Proceedings of the 12th ACM International Conference on Computing Frontiers. 2015：1-8.

WEI Sen，born in 1999，postgraduate. His main research interests include big data systems and distributed systems.

CHENG Dazhao，born in 1984，Ph. D，professor，is a member of China Computer Federation. His main research interests include big data and cloud computing.

（责任编辑：何杨）