Reaping the performance of fast NVM storage with uDepot
借助uDepot获得快速NVM存储的性能

摘要

引言

- 快NVM设备（FNDs）
  - 性能/成本比例位于DRAM和SSD之间，平衡了目前流行体系结构将数据全部存放在主存中的趋势。
  - FNDs将性能瓶颈从存储转移到网络，因此FNDs KV可以提供和DRAM一致的性能。
- 现有KV store无法充分利用FNDs
  - 现有KV将数据全部放在DRAM上需要OS显式的以页的方式使用FNDs，这导致了性能低下。
  - 将数据放在存储设备上的键值存储，即使是那些专门针对传统SSDs设计的存储器也考虑到不同的需求：较慢的设备，较小的容量，是否需要多设备或多核扩展。
- uDepot
  - 一个自底向上设计的利用FNDs性能的键值存储。uDepot是一个嵌入式存储可以被应用当作一个库进行使用。
  - 构建了两个网络服务：一个是使用自定义网络协议的分布式KV存储，另一个是实现Memcache协议的分布式缓存，可以作为memcached的完全替代，Memcached是一个广泛使用的基于DRAM的缓存。
- 优点
  - 1. 实现低延迟
  - 2. 提供每个核心高带宽
  - 3. 在多个核心和设备上扩展其性能
  - 4. 在字节和操作数上，限制端到端IO放大的较低
  - 5. 存储容量的高利用率
- 多核优化
  - 有效利用FNDs。许多现存的KV存储使用同步IO严重降低了性能，因为它依赖于内核调度的线程来处理并发性。uDepot采用异步IO并在可能的情况下直接从用户态访问存储设备。为此，uDepot建立在TRT（task runtime）上，使用用户空间协作调度使得任务用时在微秒级。
  - uDepot使用一个高性能的DRAM索引结构能够匹配FNDs的性能，同时维持其内存指纹较小。（小内存指纹导致高效的容量利用率，由于较小的DRAM需要索引小存储容量。）uDepot的索引结构是可调整大小，为了根据存储项的数目来调整其内存消耗。调整大小不需要任何IO操作，以增量的方式执行使得中断最小化。
- 贡献
  - 1. uDepot，一种传递了FNDs性能的KV存储，提供低延迟，高吞吐，扩展性和高效的CPU、内存、存储利用率。
  - 2. TRT，一个任务运行时间系统，适合微秒级的IO，作为uDepot的基石。TRT为写能够开发快存储的应用提供了编程友好的框架。
  - 3. uDepot的索引数据结构能够满足其性能目标，同时在空间上高效，动态可调整使得满足KV对存储的数量。
  - 4. 实验评估结果证明uDepot符合FNDs的性能，就我们所知，还没有系统能够做到。实际上，uDepot比优化SSD存储的性能好得多，可达14.7倍。还匹配基于DRAM的memcached服务性能，允许将其用作Memcache进行替换以显著降低成本。使用uDepot的云Memcache服务作为实验性提供在公共云上。

背景和动机

- DRAM
  - DRAM尺寸正在接近物理极限，DRAM也越来越贵。
  - 随着容量需求的增大，内存KV存储依赖横向扩展，通过增加更多的服务器来实现所需的存储容量。
  - 由于节点的其余部分（CPU，存储）未得到充分的利用，格外节点也需要按比增加资源如空间，电源，冷却设备。
- FNDs
  - 高效利用FNDs的键值存储提供了具有吸引力的可选项作为DRAM的替代品，特别在普通的网络环境中（10Gbits/s 的因特网下）FNDs将瓶颈从存储转移到了网络，网络环境无法发挥出DRAM键值存储的全部性能。
  - 使用一个多核多设备的系统（20cores&24NVMe drives）与两个典型传统存储引擎对比（RocksDB，Wired Tiger），使用Linux aio和SPDK测试，测试数据为50M4KB的键值对。在对RocksDB和WiredTiger尽力调优后，仍无法超过1Mops/s和120Kops/s。另一方面，存储设备自身使用异步IO可以提供3.89Mops/s，使用用户空间IO可以提供6.87Mops/s的性能。
- 存在的问题
  - 1. 基于慢设备构建的系统使用同步IO在微秒级造成很大问题。
  - 2. 使用LSM或B树索引造成大的IO放大。
  - 3. 在DRAM中缓存数据需要额外的同步，同时也由于内存需求限制了扩展性。
  - 4. 提供了许多额外的功能（如事务）对性能造成不好的影响。

TRT 快速IO的任务运行时系统

- 三种访问存储
  - 1. 同步IO：网络中需要线程一对一处理并发请求，在请求数大于内核数时，上下文的切换会导致无法充分利用性能。
  - 2. 异步IO：Linux AIO允许多个IO在一个线程上批量收发。但仅此并不足以充分利用性能。
  - 3. 用户空间IO：SPDK通过避免上下文切换，数据负责和调度开销来最大化性能。但由于这需要直接访问设备而经常无法使用，而且许多环境无法支持。
- TRT，任务运行时系统
  - 任务被联合调度，而每个任务有自己的栈。TRT生成许多线程（一个核一个），并在每个自己的栈上执行用户空间调度器。
  - 调度器和任务之间的切换是轻量的，包括寄存器的保存和恢复，不涉及内核。任务会通过调用来切换到调度器。也有衍生任务的调度和同步调度：等待和通知。
  - 由于TRT尽量避免内核间通信，所以为同步提供了两个变体：**intra- and inter-** core。Intra-core更有效由于不需要同步区阻止并发访问，只要关键部分不包含切换到调度器的命令。
  - TRT为异步IO提供基础结构。一个典型场景：每个网络连接被不同TRT任务服务。为了启用不同IO后端和工具，每个后端实现一个轮询任务来轮询事件并通知任务处理。每个核心运行自己的轮询任务实例。

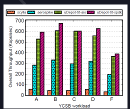
结论

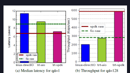
- 贡献
  - 本文提出了uDepot，这是一个KV存储，充分利用了像Intel Optane这样的快速NVM存储设备的性能。实验证明了uDepot获得了它所使用的底层IO设备的可用性能，并且与现有系统相比可以更好地利用这些新设备。此外，本文还展示了uDepot可以使用这些设备来实现一个缓存服务，该服务的性能与基于DRAM实现的缓存类似，但成本要低得多。实际上，已经使用uDepot的Memcache实现作为一个实验性公有云服务[39]的基础。
- 缺陷
  - uDepot有两个主要的限制，计划在未来的工作中加以解决。首先，uDepot并不（高效地）支持一些已经被证明对应用程序有用的一些操作，比如范围查询、事务、检查点、数据结构抽象等[78]。其次，有很多机会可以通过支持多租户[13]来提高效率，而uDepot目前并没有利用到这些机会。

相关工作

- Flash KV存储
- DRAM KV存储
- NVM KV存储
- MemCache
- 基于任务的异步IO

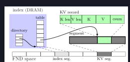
评估

- 索引结构
  - 

  - 1）插入5千万（5·107）个条目，其中没有发生重新调整大小操作；2）插入10亿（109）个条目，其中发生了4次扩容操作。本实验对比了libcuckoo[53，54]，这是一种目前最先进的哈希表实现，通过运行它附带的基准测试工具（universal_benchmark），为5千万和10亿条目测试分别配置了226和230的初始容量。结果如图5所示。对于5千万条目，本文的实现取得了每秒8770万次的lookup和每秒6400万次的insert，分别比libcuckoo的性能快5.8倍和6.9倍。对于10亿条目，由于存在重新调整大小操作，insert速率下降到了23.3Mops/s。
- 嵌入式uDepot
  - 

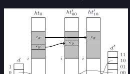
  - 本文考虑了在比较uDepot和设备性能过程中表现较好的GET操作。主要关注单个排队请求（qd = 1）时的延迟和较高队列深度时（qd=128）的吞吐量。使用trt-aio后端获得的延迟是9.5μs，与相之应的使用fio的原始设备延迟是9μs。一个trt-aio后端的初始实现使用了io getevents()系统调用来接收IO完成，这导致了更高的延迟（接近12μs）。通过在用户空间中实现这个功能后提高了性能[17，28，77]。当使用这种技术时（fio的userspace_reap选项），fio的延迟保持不变。
- uDepot Server/YCSB
  - 给出了所有工作负载下的256个客户机线程的吞吐量。uDepot使用trt-spdk后端提高了YCSB的吞吐量，相比Aerospike分别提高1.95倍（工作负载D）和2.1倍（工作负载A），相比ScyllaDB分别提高10.2倍（工作负载A）和14.7倍（工作负载B）。

uDepot

- uDepot
  - log-structed方式。空间被连续分配，GC处理碎片。
  - 

- 存储设备空间管理
  - 1. 在NAND闪存这种特殊存储器上有较好性能。
  - 2. 即使对DRAM这种非特殊存储器，它也比传统方法更高效。
  - 3. 一个重要的用例是缓存，在协同设计GC和缓存时有大量的优化机会。
- 索引数据结构
  - 哈希表
    - 修改一：使用条目数量的二次幂对哈希表进行索引，类似于组相联缓存[38]：使用从key计算而来的指纹的最低有效位（LSB）来计算邻域索引。这允许在重新调整大小期间有效地重建原始指纹，而不需要完全存储指纹或执行IO来获取key并重新计算。
    - 修改二：本文没有维护每个邻域的位图，也没有维护每个邻域的所有条目的链表，而这两者是原始算法所建议使用的[37]。后者将使默认配置的内存需求增加50%（8字节的条目，邻域大小为32，每个条目4字节）。一个链表至少需要两倍的内存（假设使用8字节指针和单链表或双链表）；更不用说增加的复杂性了。本文不使用位图或链表，而是直接对条目的lookup和insert操作执行线性探测。
  - 同步
  - 哈希表条目

```
struct HashEntry {
  uint64_t neigh_off:5; // neighborhood offset
  uint64_t key_fp_tag:8; // fingerprint MSBs
  uint64_t kv_size:11; // KV size (grains)
  uint64_t pba:40; // storage addr. (grains)
};
```

  - 容量利用率
  - 操作
    - 

- 实现说明
  - uDepot采用C++11实现。值得注意的是，uDepot的性能需要许多优化：使用本地CPU核心的slab分配器从数据路径消除堆分配，使用大内存页面（huge pages），与动态多态相比更倾向于静态，避免使用scatter-gather型IO带来的数据复制，在IO缓冲区的适当位置存放网络数据，使用批处理等等。