高技术通讯 2020 年 第30卷 第10期:983 ~997

doi:10.3772/j.issn.1002-0470.2020.10.001

面向存储网络的融合 I/O 模拟器设计与实现①

魏 征②[*,**,***] 黎斐南[**,***] 邢 晶[**,***] 霍志刚[**,***] 孙凝晖[**,***]

([*]中国科学院大学 北京 100089)

([**]计算机体系结构国家重点实验室 北京 100089)

([***]中国科学院计算技术研究所 北京 100089)

摘 要 在现有计算机系统结构中,要实现跨节点的数据存储操作,数据需要先经过本地网卡、远程网卡、远程内存才能到达远程存储设备。这一过程通常还需要远程节点操作系统和应用软件的参与才能完成。随着硬件技术的发展,存储设备的访问延迟大幅下降。为了进一步降低节点数据传输延迟,充分发挥存储设备的性能优势,本文设计了面向网络与存储的融合 I/O 设备模型 UIO,通过让多个功能模块共享存储设备中的内存和数据通路,以简化跨节点数据存储的传输通路优化。并加入辅助计算功能实现数据处理操作,通过与可编程硬件的结合,可以让用户定制辅助计算模块,提高 UIO 设备的处理效率,扩展应用场景。设计并实现了面向 key-value 存储系统的 UIO 设备模拟器,使用 4 kB 长度 value的情况下,与传统数据通路对比,远程 put 延迟可以降低 31%,远程 get 延迟减少 20%。综合指令延迟和指令使用频率对整体操作延迟进行分析,UIO 设备在目标场景中预期可以达到比传统数据通路更低的延迟。

关键词 设备融合;模拟器;key-value 存储;可编程硬件

0 引 言

存储设备在过去很长一段时间都被视为高延迟的慢速设备,随着新材料和设备的研制,存储设备的性能有了很大改善,I/O 延迟大大下降。然而目前的大部分软硬件的设计还是建立在慢速设备的假设上,这种设计无法充分发挥新式存储设备的性能,因而仍存在读写延迟高的问题。

机械硬盘(hard disk drive, HDD)[1]作为主要的廉价存储设备,从最初的 3.75 MB 到现在的 14 TB,HDD 的容量增加了 373 万倍。但是,HDD 的访问延迟从 600 ms 下降到了 2 ms,只减少到了原来的 1/300。HDD 数据存储在盘片上,受机械结构的限制,碟片的旋转速度不能无限提高。HDD 的平均延迟基本都是毫秒级的。固态硬盘(solid state drive, SSD)使用集成电路作为存储介质,受益于半导体工艺的发展,延迟可以达到微秒的量级[2]。现在的 NVMe SSD 可以达到几十微秒的延迟。基于 3D XPoint 材料的存储设备拥有更低的延迟,填补了 SSD 和内存之间的差距,其延迟达到微秒级[2]。

传统的跨节点数据通路中,数据传输都要经过内存。数据一开始在远程主机的内存中,中央处理器(central processing unit, CPU)设置网络接口控制器(network interface controller, NIC)的直接存储器访问(direct memory access, DMA),由 DMA 把数据发送到网络上。本地的 NIC 从网络收到数据,数据经过 DMA 的搬运,保存到内存中。然后 CPU 再控制存储设备的 DMA,把数据写入本地的存储设备。

① 国家重点研发计划(2018YFC0809300,2018YFB0204400),国家自然科学基金(61502454)和 CCF-百度松果基金资助项目。

② 男,1988 年生,博士生,工程师;研究方向:分布式文件系统;联系人,E-mail:weizheng@ncic.ac.cn

(收稿日期:2019-10-17)

随着存储技术的发展,存储设备的延迟已经从原来的毫秒级变为了现在的微秒级,数据在数据通路上延迟在整个传输延迟中的占比会大大增加,原来的传输路径可能会导致额外的传输开销。

NoR 技术(NVMe over RDMA)[3]通过改变传统的数据传输通路优化延迟。本地网卡从远程主机接收到的数据不再写入内存,直接通过 PCIe[4]总线写入本地的 PCIe SSD。通信路径的改变使得传输延迟从 12 μs 缩减到了 7 μs。缩短后的通信路径绕过了 CPU,但也失去了对数据的处理能力。数据直接写入存储只能满足应用,很多其他应用都需要对数据进行一些处理,NoR 技术即使是很简单的处理都无法完成。而且,只有存储设备是非易失性随机访问存储器(non-volatile random access memory, NVRAM)或随机存取存储器(random access memory, RAM)时,远程直接数据存取(remote direct memory access, RDMA)网卡才能直接给存储设备写入数据。NVRAM 存储设备虽然拥有接近 RAM 的性能,同时造价成本也很高,而性价比较高的且真正大量使用的 SSD 采用的是 NAND Flash 介质,属于块设备,无法通过 RDMA 协议直接写入数据。如果希望能写入块设备,那么数据就需要经过闪存转换层(flash translation layer, FTL)的处理。

由于机械硬盘的高性价比,目前依然作为主流的存储设备使用。然而机械硬盘的随机延迟很高,为了充分发挥机械硬盘的性能,将随机读写组织成顺序读写作为主要优化方式。但是大数据的数据形式多样,数据的组织有可能很复杂,难以组织成顺序的读写。RAMCloud[5](内存云存储)把大数据应用所需求的数据全部存放在内存中,完全可以满足大数据分析的需求。但是 RAMCloud 的成本高昂,且由于持久化问题,还需要额外的成本和机制用于保障可靠性。机械硬盘和 RAMCloud 之间的性能和成本落差太大,SSD 具有介于内存和机械硬盘之间的性能和成本。随着 3D XPoint 技术的推出,SSD 拥有更高的存储密度和对随机高并发的支持。如果 SSD 的性能可以满足大数据分析的要求,可以有远小于 RAMCloud 的成本以及集群规模。

NoSQL 数据库在大数据应用中有着很重要的地位,根据用户需求,弱化了 SQL 数据库的一些要求以满足大数据应用的场景。key-value 存储是 NoSQL 数据库的一种典型案例,只保存最简单的 key 和 value 之间的映射。Redis[6]和 Amazon 的 Dynamo[7]都是 key-value 存储系统的著名实例。本文的另一个重点就是尝试将 SSD 应用于大数据应用,为大数据应用提供一种存储解决方案。

本文设计了面向网络与存储的融合设备模型用户级融合 I/O(unity I/O in user space, UIO),并加入辅助计算功能扩展其适用场景。本文的主要工作和贡献包括:

(1) 提出了一种融合存储和网络功能的 UIO 设备模型。UIO 设备在同一种设备中融合了存储和网络功能,通过让多个功能模块共享设备中的内存和数据通路以简化数据的传输路径,降低数据的传输延迟。

(2) 设计了基于 UIO 设备模型的辅助计算模块。通过在设备中加入辅助计算功能实现数据处理,并把辅助计算模块分为处理设备间数据分布和存储介质格式转换 2 个部分。通过与可编程硬件的结合,可以让用户定制辅助计算模块,提高 UIO 设备的处理效率,扩展其应用场景。

(3) 实现了与 key-value 存储系统相结合的 UIO 设备模拟器。参照 key-value 存储系统设计了 UTL 和 GME 等各个模块,让用户可以通过 put、get 指令操作设备。在最终测试中与传统数据通路相比,远程 put 减少 8.2% 延迟,远程 get 减少 11% 延迟,本地 put 增加 0.9% 延迟,本地 get 增加 2.7% 延迟的效果。

1 相关研究

1.1 相关硬件技术

随着硬件技术的发展,已经有了高速低延迟的存储设备 NVMe SSD。这种具有低延迟特性的存储设备的出现,使得传统的硬件结构不再适合所有场景,反而成为了系统性能进步提升的瓶颈。

1.1.1 SSD

SSD 使用半导体作为存储介质,包括 DRAM、

NOR Flash、NAND Flash 等。现在比较常用的是 NAND Flash 作为存储介质的 SSD。NAND Flash 的数据由浮栅单元存储,每个单元根据存储介质的不同(SLC, MLC, TLC)可以存储 1、2 或 3 个 bits。由浮栅单元组成逻辑页,逻辑页组成逻辑块。一个页由 4 kB 数据存储空间和 128BECC 校验数据的存储空间组成,64 个页又组成了一个块。页是 NAND Flash 中最小的读写寻址单元,块是 NANDFlash 中最小的可擦除单元。可以随意读取页中的数据或者向空白的页写入数据,但是不能改写已经写入页中的数据,如果改写需要将整个块的数据擦除,然后才能重新写入新的数据。

NAND Flash[8] 的这种读写机制导致其读写接口不能使用常见的读(Read)和写(Write)接口,在读、写之外还需要加入擦除(Erase)功能。在 Flash 和传统接口之间需要添加一个转换层 FTL[9,10],位于 SSD 的 Host Interface 和 NAND Interface 之间,负责 2 种接口之间的转换。FTL 主要负责的功能有地址映射、磨损均衡、垃圾回收、坏块管理等。地址映射为主机使用逻辑地址定位数据,由 FTL 完成从逻辑地址到物理地址的映射。当数据修改时,把数据写到一个新的页上并修改 FTL 中记录的地址映射,以避免多余的数据读写。磨损均衡为 Flash 中每个存储单元的可写入次数是有限的,为了延长整个设备的使用寿命,尽量保证每个单元的写入次数均匀。FTL 记录各个块的剩余寿命,尽量使用寿命长的块。同时,也会把一些存储在剩余寿命比较长的块中但是不经常改动的数据搬移到剩余寿命较短的块中。垃圾回收是在 NAND Flash 中,以块为单位擦除,失效的页空间不会立刻释放。而垃圾回收负责释放失效的页,保证空间利用率。它会把一个块中还有效的页复制到其他块中,然后擦除整个块。坏块管理会记录下已经无法使用的块,在写入数据时避开。

1.1.2　NVMe 协议

SSD 中除了存储介质和控制器还有一个关键的组成部分就是主机接口。以 Intel 为代表的厂商联合推出了专为 PCIe SSD 设计的协议 NVMe[11]。NVMe 协议专为 SSD 设计,最大支持 64 k 个队列,每个队列可以容纳 64 k 条指令,使用多队列和 doorbell 的方式读写指令,不需要对队列加锁,可以充分发挥高性能 SSD 设备的性能。PCIe 接口解决了 SSD 主机接口的硬件通路瓶颈问题,NVMe 协议解决了 SSD 主机接口的软件协议瓶颈问题。

NVMe 协议中的队列分为 Admin Queue 和 I/O Queue。每类队列又分为提交队列(submission queue,SQ)和完成队列(completion queue, CQ)。Admin Queue 负责对设备的管理,完成创建删除队列、设置参数、查询日志等工作,I/O Queue 负责具体的传输指令,完成 Read、Write、刷回(Flush)等操作。SQ 用于主机向设备发送指令,CQ 用于设备向主机发送指令的返回结果。一个 CQ 可以匹配多个 SQ,通过 SQ 和 CQ 的配合完成对设备的操作。

1.1.3　RDMA 网卡

以太网网卡在接收数据时,数据必须经过系统的处理才能提供给需要这个数据的应用。这种处理增加数据传输延迟的同时也增加了 CPU 的负担,在网卡的性能越来越高的同时,对 CPU 的性能也提出了挑战。

基于解决数据在内存和设备之间的数据传输,并减轻 CPU 负担的 DMA 机制,RDMA 被提出并用于解决网络传输的问题[12-17]。RDMA 通过网络直接把数据传入主机的用户空间,不再需要内核参与,提高了网络传输性能,降低了 CPU 的处理负担。在 RDMA 的 Infiniband 网卡中,具有和 NVMe 协议中类似的队列结构,其中有 Send Queue, Receive Queue 和 Completion Queue。Infiniband 网卡的设计目的就是为了提高网络性能降低传输延迟,使用的场景通常也是对网络有着很高要求的应用,同时也需要高性能的队列结构避免在处理 I/O 请求时成为瓶颈。

1.2　设备融合

数据在不同设备之间传输所消耗的时间一直是研究者们尽力减少的,如果一种设备可以同时完成 2 种工作,那么数据只要在设备内部流动,不需要跨设备的传输。针对不同的设备类型和不同的使用场景,已经有了一些通过设备融合优化系统性能的研究。

1.2.1　存储和计算的融合

XSD(accelerator SSD)[18] 通过在 SSD 中加入嵌

入式图形处理器(graphics processing unit, GPU)以提供计算功能,并为用户提供了易用的 Map-Reduce[19]接口。XSD 在 SSD 中插入了嵌入式 GPU,和 SSD 原有的嵌入式 CPU 共享内存空间。XSD 通过流水化来避免数据拷贝时间的浪费。由于 Flash 控制器和 GPU 是共享内存的,Flash 向动态随机存取存储器(dynamic random access memory, DRAM)写入数据的同时 GPU 也在从 DRAM 中复制数据进行计算再把数据写回 DRAM。XSD 通过流水化避免了 GPU 对数据的等待问题,充分利用了 GPU。XSD 通过在 SSD 内使用嵌入式 GPU 加速计算,通过计算的流水化充分利用了 SSD 和 GPU 的性能,通过采用 Map-Reduce 编程框架提供简单高效的设备使用接口。虽然在 SSD 中加入计算设备可以有效减少设备等待数据传输的开销,但是因为嵌入式设备自身性能问题,其性能提升能力还是有所不足的。

1.2.2 存储、网络和计算设备的融合

BlueDBM[20]是具有 Flash 存储系统、存储端计算引擎和网络功能的 PCIe 设备。其中,Flash 存储系统直接开放原始的 flash 接口(也就是不使用 FTL 层),由系统层管理数据。BlueDBM 通过提供多种软件接口(文件系统接口、块设备接口和加速器接口)以方便用户对设备的使用。BlueDBM 在图遍历测试的性能(ISP-F)虽然比使用 DRAM 的方案(H-DRAM)要差,但是当数据只能部分装入内存时(50% F,30% F),BlueDBM 的性能已经足以替代 DRAM。BlueDBM 通过对 Flash 存储介质的直接控制和存储端的计算能力在多个测试中取得了优秀的结果。但是 BlueDBM 采用了使用固定配置文件的方式,不具有扩展能力。虽然提供了多种 Flash 存储的操作接口,但没有提供操作设备的存储、网络和计算功能的统一接口,用户需要十分了解 BlueDBM 的结构才能使用。

1.2.3 存储和网络的融合

NVMe over Fabric[3]就是一种在共享 NVMe SSD 资源的同时尽量减少额外开销的方案。资源池化,可以让多台计算机共享存储资源,提升扩展性和资源的利用率,降低成本。NVMe over Fabric 分离了 NVMe 设备的前后端,前后端之间通过 RDMA 网络传输。前端可以把 internet 小型计算机接口(internet small computer interface, iSCSI)协议转换为 NVMe 的协议,让系统统一可以使用 NVMe over Fabric。通过网络可以很容易地扩展存储,想要更大存储容量时只要添加存储设备,不需要扩展机群规模。同时,因为 RDMA 网络有着低延迟,NVMe over Fabric 对数据传输延迟的影响很小。测试表明,通过 NVMe over Fabirc 使用远程 NVMe 设备时,读写的 IOPS(input/output operations per second)几乎不受影响,传输延迟的增加也低于 8 μs。但是 NVMe over Fabric 只是对 NVMe 存储的扩展,缺少计算能力。NVMe over Fabric 虽然可以改善主机读写远程存储时的延迟,但是在数据需要处理时仍然无法改善数据路径冗余导致的浪费。

2 UIO 设备模型

在传统计算机硬件中,存储和网络设备往往是独立的 2 种设备,而且数据无法直接在这 2 种设备之间传输,必须经过 CPU 和内存。随着硬件技术的发展,存储设备和网络设备的延迟越来越低,数据无法在设备之间直接传输导致的额外延迟在传输延迟中的占比也越来越高,这导致新式 I/O 设备无法充分发挥其性能。通过设备融合,让同一设备具有存储和网络功能,设备内部的存储和网络模块可以直接通信,避免数据在通路上的性能损失,减少通信的延迟开销,改善系统整体性能。

2.1 UIO 设备结构

如图1所示,UIO设备是插于PCIe卡槽的I/O

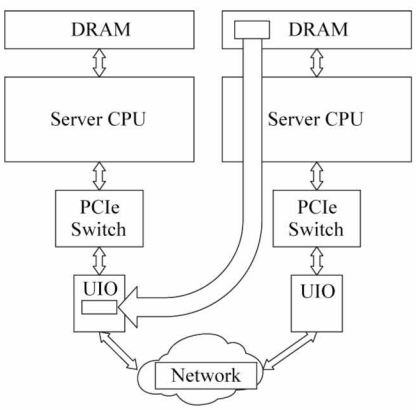

图 1 UIO 设备在系统中的位置

设备,同时具有存储设备和网络设备的功能,为了提供一定的数据处理能力,UIO 设备还具有辅助计算功能。与 SSD 和网卡一样,UIO 设备需要系统中的驱动程序支持,由驱动程序负责与设备的通信。

如图 2 所示,UIO 设备中具有指令队列、指令处理、DMA、DRAM、存储模块、辅助计算模块和网络模块。

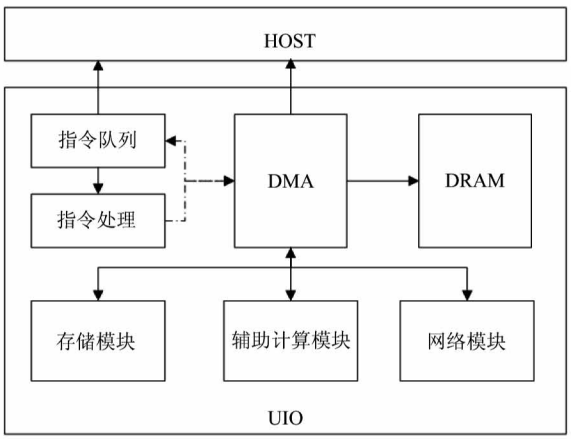

图 2　UIO 设备模块设计

指令队列结构　NVMe SSD 和 RDMA 网卡都是具有高吞吐量和低延迟的高性能 I/O 设备,这 2 种设备都使用了高性能的 I/O 指令队列作为主机接口。UIO 设备的设计目标是解决高性能 I/O 设备的传输延迟问题,为了支撑设备的高性能,采用了队列结构。基于并发性考虑,管理队列为每个应用建立独立的指令队列。管理队列为不同的程序创建完队列并分配好权限后,应用有读写需求时只需要向自己的队列中写入指令即可。设备会从指令队列的空间读取指令,因为 I/O 队列的权限已经由管理队列设置好,可以由设备检查指令是否符合该指令队列的权限,而不需要内核的处理。这为用户级 I/O 和虚拟化提供了方便。

指令处理　指令处理要完成 2 部分工作,一个是指令仲裁,一个是指令解析。UIO 设备具有多对指令队列,不同进程使用设备时不必给队列加锁,每个进程都可以自由地给设备发送指令。设备上有相应的模块来处理多个指令队列之间的协调调度问题,这个功能就是指令仲裁。指令仲裁会决定设备执行的下一条指令是什么。执行的顺序通常由队列的优先级决定。指令解析会把指令翻译成设备内部模块的控制信息,而设备的其余模块才是真正完成数据传输工作的部分。

DMA　设备在解析完指令后,执行对应的数据传输,即主机和设备之间的数据传输。UIO 中 DMA 主要负责把主机中数据传输到 DRAM 中,而数据从设备到主机的传输会根据实际情况由不同的模块完成。

DRAM　UIO 中的 DRAM 有 2 个作用,一个是保存设备运行时必要的信息,一个是作为数据传输的缓存和缓冲。一般的存储设备,读写速度和延迟都是不一致的,Flash 介质的存储设备还存在写放大的问题,FTL 正在执行的一些任务(例如静态磨损均衡和垃圾回收)也会导致读写时间出现较大波动。为了提供更好的设备性能,设备中的 DRAM 可以作为读写的缓存。UIO 中网络传输到设备中的数据,也需要一些内存空间进行缓冲,平衡存储模块和网络模块读写速度的差异。

2.2　可编程模块

在 UIO 设备中,需要具备存储、网络和辅助计算 3 种功能,如何使用这 3 种模块是提供高性能的关键问题。

2.2.1　模型分析

基于独立设备接口的指令接口　用户需要控制各个模块对数据的处理,以及数据在各个模块之间的传输。用户使用设备的自由度高,通过精心设计的指令可以充分发挥设备的性能。但是,这种方案在实际使用中会有很大的局限。首先,处理器的性能受功耗限制,在 I/O 设备中嵌入的处理器性能有限。如果向用户提供全功能的通用计算处理器,处理器的性能可能无法匹配高速 I/O 设备的性能,反而成为性能瓶颈,带来过多的性能损失。其次是 I/O设备的接口复杂性。对 Flash 的读写不能使用传统的 Read 和 Write 原语,而要使用编程(Program)、Read 和 Erase 的原语进行读写。SSD 中通常用 FTL 层进行这 2 种接口的转换。如果向用户开放 Flash 的原始接口,那么 FTL 负责的功能都需要用户来完成。这会极大提高用户的使用难度,用户除了要熟悉 Flash 的接口还要进行复杂的性能优化,难以发挥高性能存储的能力[21-27]。

基于需求抽象分析的统一指令接口　UIO 设备采用的方案是抽象出用户需求，根据用户需求设计专用设备，向用户提供统一的指令接口，用户通过使用统一的指令就可以完成对设备全部功能的操作。这种方案的缺点显而易见，设备的功能受限，只能在特定场景下使用。但是这一缺点可以通过抽象用户需求弥补。可以通过分析用户使用较多的场景，把某一类用户的需求提取出来，然后制作设备专门用于满足这类用户的需求。虽然无法满足所有用户的需求，但是通过抽象出多种用户的需求也足以覆盖大部分用户。

统一的用户指令使用简单，通过抽象用户需求，可以向用户提供具有高级语义的接口，完全屏蔽设备实现的具体细节。甚至可以根据为用户提供的功能深度定制底层模块的逻辑。如图 3 所示，移除了原有设计中的辅助计算模块，改为统一转换层（unity translation layer，UTL）模块和全局信息管理引擎（global management engine，GME）模块。UTL 模块和 GME 模块都是可编程模块，可以根据用户需求写入不同的处理逻辑。其中，UTL 模块负责读写存储介质，GME 负责分离本地和远程的传输。

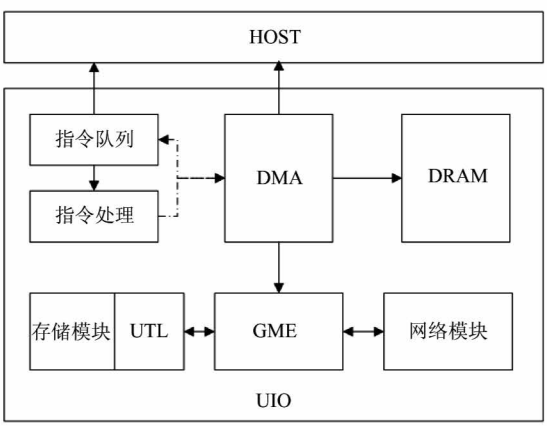

图 3　UIO 模块设计专用方案

2.2.2　UTL 模块

UTL 模块用于取代 SSD 中的 FTL 模块，负责实际的存储介质的读写。UTL 把用户对设备的操作接口转换为对 Flash 介质的 Program、Read 和 Erase 接口，UTL 提供可编程的接口。UIO 针对抽象出来的用户需求，完成特定类型的操作，针对 Flash 存储介质的特点，完成用户定义接口到 flash 介质接口的高效转换。

2.2.3　GME 模块

GME 模块负责多个设备之间的通信，通过增加设备的数量进行扩展，通过多个 UIO 设备的协同工作，共享存储空间，扩展各 UIO 设备的性能。

如图 4 所示，把多个设备映射到一个全局空间中。用户通过本地设备的接口就可以透明地操作全局空间，而不必感知到其他设备的存在。当系统中增加设备时，用户看到的是全局空间的扩展。GME 控制数据的分布，并根据数据的分布规则把数据发送到相应的设备。如果是本地的数据就交给 UTL 模块处理，由 UTL 模块写入本地的存储介质；如果是远程的数据就通过网络模块发送到目标设备，由目标设备写入存储。

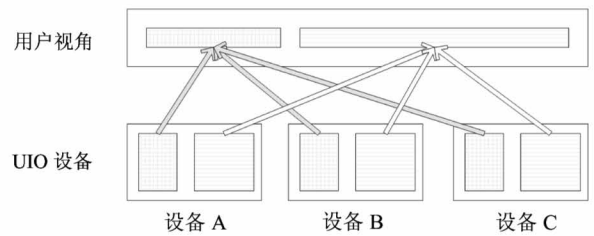

图 4　UIO 设备多机模型

2.3　基于 UIO 的应用示例

2.3.1　key-value 存储

在 key-value 数据库中，哈希存储方式把 KV 对组成的记录顺序写入存储介质，在内存中建立从 key 到记录地址的映射。在读取时就根据映射地址读取；修改时写入一条新的记录，然后修改内存中映射的地址；删除时并不删除实际的记录，而是写入一条代表已删除的记录。为了支持更好的扩展性和并发性，在 UIO 中，可以进行 2 次哈希存储，一次由 GME 进行，把数据分布到不同的设备上，一次由 UTL 快速定位数据在存储介质中的位置。哈希存储虽然实现简单，查找效率高，但是只能查找单条记录，如果想查找一个范围内的数据效果比较差，此时需要考虑顺序分布的存储方式。

如图 5 所示，顺序分布把 key 分为多个范围，不同范围的数据组成不同的子表，Root 表记录每个子

表存储在哪个节点中。在 UIO 设备中,可以由 GME 记录每个子表的数据范围和节点信息,在查找范围时由 GME 把大的范围转换为对多个子表的查询,UTL 则负责一个子表,根据查询返回结果。

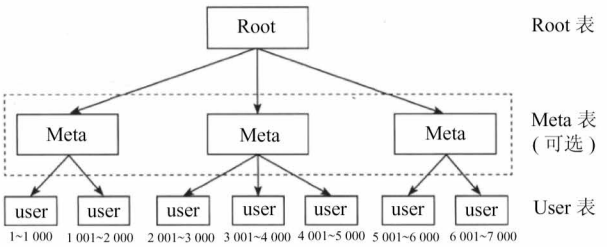

图 5　顺序分布

2.3.2　图片检索

图片检索功能就是向 UIO 设备发送图片后,UIO 设备从存储中检索出相似的图片并发送给主机。其中,GME 模块负责把图片分发给本地 UTL 模块和其他远程设备。所有 UIO 设备中的 UTL 模块收到图片后就会读出存储中的图片,并进行比对,并把相似的图片发送给源设备。源设备收到图片后会发送到主机。与传统方案中数据需要在存储设备和 CPU 或 GPU 通信相比,由专用的硬件模块处理图像可以取得更高的运算效率。在存储设备中的比对使得大部分数据都不需要从设备传输到主机,只需要传输比对后相似度高的图片,设备和主机间的带宽可以用来传输远程设备传输过来的图片,避免了带宽的浪费。

2.3.3　kNN 算法

最邻近结点(k-nearest neighbor,kNN)算法是一种常用的机器学习算法,通过计算未知数据和已知数据的“距离”来预测未知数据的类别。kNN 算法在预测时会取出已知数据中和待预测数据最接近的前 k 项数据作为预测的样本,这也是 kNN 算法中 k 的含义。UIO 在处理 kNN 算法时,存储介质中存储所有的样本。在主机向设备发送待预测样本后,GME 模块把样本发送给所有设备。所有设备的 UTL 模块接收到样本后,会读出存储介质中的样本并计算距离值,在计算的过程中保存最接近的前 k 个值。全部的值比对完成后设备会把前 k 个最接近的结果发送给源设备,源设备的 GME 模块在收集到所有设备的返回结果后会进行排序并选出其中的前 k 个值预测分类,再把分类结果发送给主机。

3　UIO 模拟器实现

为了验证 UIO 设备的设计是否有效,并为将来 UIO 设备的相关研究提供验证平台,本文将实现 UIO 设备的模拟器。

3.1　整体架构

如图 6 所示,UIO 设备模拟器的结构依照 UIO 设备的基本结构而设计。由指令队列和控制寄存器共同组成设备接口。其中指令队列分为 Admin Queue 和 I/O Queue,每种 Queue 又由 SQ 和 CQ 组

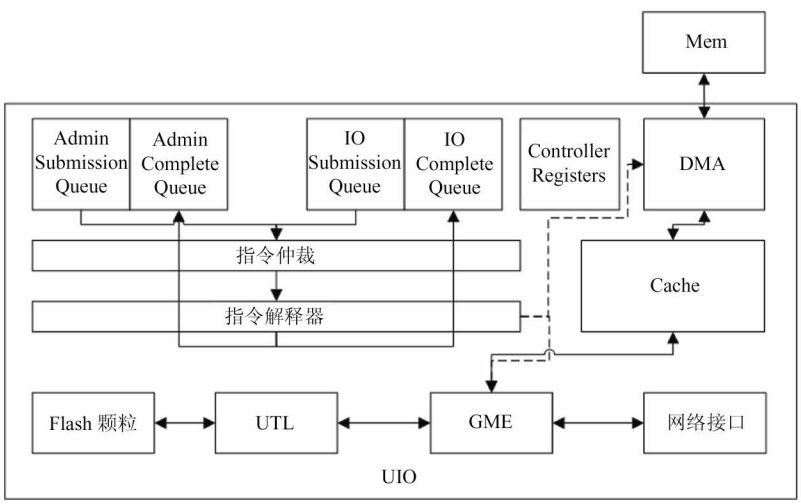

图 6　UIO 设备结构图

成。除了指令队列,主机还需要一些寄存器为设备进行初始化以及保持状态信息。在实际 UIO 设备模拟器的实现中,本文选择的目标是实现采用 key-value 接口,使用 put/get 指令,GME 模块和 FTL 模块都采用哈希分布。

UIO 设备模拟器是用户态功能级的模拟器,目标是验证 UIO 设备的模型是否有效。在 UIO 模拟器的设计中,首先进行初始化工作,然后创建 2 个线程,一个模拟主机端,一个模拟设备端。主机端会接收来自控制台的命令,并把用户的命令转换为相应指令写入设备端的控制结构。设备端则是根据控制结构中的指令执行实际的操作,从主机端复制数据或者向主机端写入数据并根据指令执行情况提供返回结果。设备端启动后会创建出指令处理、DMA、GME、UTL 和网络通信(network communication, NET)一系列线程用于模拟 UIO 设备中的各个模块。考虑各个模块的运行都是并行的,采用多线程来尽量接近真实的设备特性。

3.2 UIO 设备接口设计与实现

UIO 的设备结构主要借鉴了 NVMe 的设备接口设计,并根据 UIO 设备的特性进行了修改。

3.2.1 指令设计

在 UIO 中,本文设计了 2 种指令,Admin Command 和 IO Command。出于简化设计突出原理的目的,只设计了实现 UIO 最核心功能的指令。Admin Queue 的 4 种操作就是创建和删除 SQ 和 CQ。需要注意的是,在创建 SQ 前应该先创建 SQ 所对应的 CQ。如果先创建了 SQ 又向队列中写入了指令,设备在处理完指令后找不到应该写入返回结果的 CQ 就会导致未知的结果。IO Command 只有 put 和 get 2 条。put 提供 key 和 value,表示把 key 的值更新为 value。get 提供 key,表示读出 key 存储的值。

如表 1 所示,每一条指令有 64 个字节组成。其中,opcode 用于指明要执行的是哪条指令;cid 是指令的表示,使用队列的 id 加 cid 可以确定唯一的一条指令;nsid 用于指明该条指令用于操作哪个 namespace,为不同用户和应用的操作系统硬件级的隔离(在改版 UIO 中尚未实现,只是预留接口);其余部分会根据指令的不同有不同含义或者是保留字段。

表 1 指令格式

字节	描述			
07:00	opcode	res1	cid	nsid
15:08	adr1			
23:16	len1			
31:24	adr2			
39:32	len2			
47:40	cdw10		cdw11	
55:48	cdw12		cdw13	
63:56	cdw14		cdw15	

例如在 Create IO Submission Queue 的指令中,adr2 用于保存队列的起始地址;cdw10 用于保存队列大小和队列 id;cdw11 中保存 SQ 对应的用于放置返回结果的 CQ 的队列 id 以及该 SQ 的优先级。

而在 put 和 get 指令中,adr1 用于存放 key 的地址,len1 存放 key 的长度,adr2 用于存放 value 的地址,len2 用于存放 value 的长度(get 指令中实际上是主机为 get 的返回值创建的缓冲区的地址以及缓冲区长度)。

表 2 所示是 CQ 中返回项的格式。Command Specific 是指令的返回结果,根据指令的不同也有所不同;SQ Identifier 是发送该指令的 SQ 的 id,SQ Head Pointer 指明返回该指令时,设备中维护的 SQ 的 Head Doorbell 指向的位置(在主机读取到时设备中该值可能已被更新);Status Field 返回指令执行的基本信息,例如是否成功,是否可以继续尝试;P 是 Phase Tag,Phase Tag 只有 1 位,用于指明该返回项是否已经填写完成;Command Identifier 就是该返回项对应的指令的 id,前面提到队列 id 加指令 id 的

表 2 CQ 返回项格式

字节	描述		
03:00	Command Specific		
07:04	Reserved		
11:08	SQ Identifier		SQ Head Pointer
15:12	Status Field	P	Command Identifier

组合可以确定一条指令,也就是说,主机根据这 2 个 id 就可以知道完成的是哪条指令了。

3.2.2　BAR 寄存器

基地址寄存器(base address register, BAR)解决的是最基本的主机与设备通信的问题。主机所看到的 BAR 寄存器是内存中的一段空间,主机通过读写其中的数据实现对设备的控制。UIO 设备的 BAR 寄存器参照了 NVMe 的设计,主要使用了其中关于 Admin Queue 的 AQA、ASQ、ACQ 以及 Doorbell 部分的寄存器,如表 3 所示。

虽然主机和 UIO 设备的主要通信方式是通过指令队列,但是这就需要知道指令队列的地址。IO Queue 的地址空间是由主机在内存中申请并通过 Admin Queue 指令把地址提供给设备,由设备自行从主机内存中获取指令。这就需要主机和设备都统一预先知道 Admin Queue 的地址。而这个地址就是由 BAR 寄存器提供的。BAR 寄存器中的 ASQ 和 ACQ 分别提供了 Admin Submission Queue 和 Admin Completion Queue 的起始地址。AQA 则提供了 Admin Queue 的大小。之后的 CQ0TDBL 和 CQ0HDBL 等寄存器则分别提供了 SQ 的 Tail Doorbell 和 CQ 的 Head Doorbell。Head 和 Tail Doorbell 是 2 个指针,分别指出了指令队列中存放有数据部分的起始和结束。

基于 Queue 的起始地址(Admin Queue 的在 BAR 寄存器指出,IO Queue 的由主机和设备分别自己维护)和 Doorbell,主机就可以完成对设备的控制,设备就可以获得主机的指令,完成自己的工作了。

表 3　Bar 寄存器定义

寄存器	描述
AQA	Admin Queue Attributes
ASQ	Admin Submission Queue Base Address
ACQ	Admin Completion Queue Base Address
SQ0TDBL	Submission Queue 0 Tail Doorbell (Admin)
CQ0HDBL	Completion Queue 0 Head Doorbell (Admin)
SQ1TDBL	Submission Queue 1 Tail Doorbell
CQ1HDBL	Completion Queue 1 Head Doorbell
……	……

3.3　UIO 设备模拟器控制结构的实现

3.3.1　指令仲裁

由于指令优先级和 UIO 模拟器要验证的原理关联不大,UIO 模拟器中的指令队列只使用了 1 级优先级,指令仲裁使用轮询的方式从队列中取指。

UIO 中会为所有队列建立一个链表,维护每个队列的 Head 和 Tail,每当队列创建时就加入该链表,队列删除时就从中释放。指令仲裁通过检查 Head 和 Tail 判断队列是否为空,如果队列不为空就从队列中取出 Head 所指向的指令并交给指令解释器执行。在取出一定条数的指令后,或者队列为空时,就通过 next 指针找到下一个队列。

3.3.2　指令解释器

指令解释器首先根据取指的队列 id 判断是 Admin Command 还是 IO Command。然后从指令中取出 opcode 字段,根据 opcode 判断是哪条指令并根据该指令的字段解析其中有用的信息,并把这些信息发送给数据处理模块。

在处理 Admin Command 的时候,因为不涉及数据的传输,由指令解释器执行具体的指令。在创建队列时,指令解释器会创建图 7 中的队列轮询控制结构,并指向指令中给出的地址;而在删除队列时,指令解释器会释放队列的控制结构。

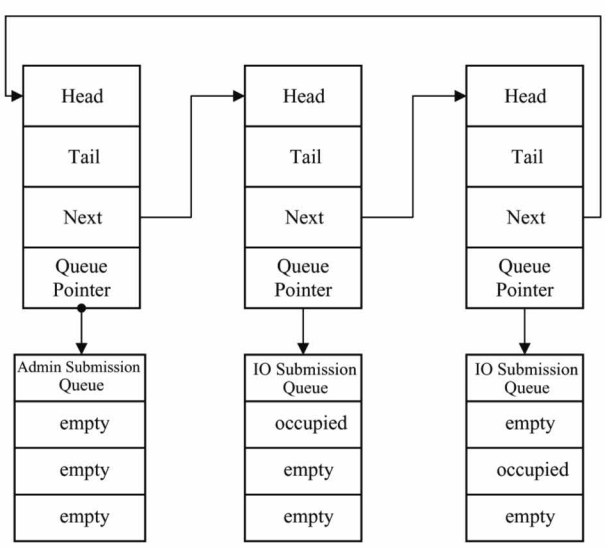

图 7　UIO 队列轮询结构

3.4　数据处理模块

UIO 模拟器采用消息队列模拟模块间的通信,

包括指令解释器模块向数据处理模块发送的请求和数据处理模块之间发送的请求。模块间消息队列只有请求项不设返回值,当模块处理指令成功或者失败时,会根据处理情况向 Completion Queue 填写返回信息以完成对指令的处理。

3.4.1　DMA 模块

在 UIO 模拟器中,DMA 负责把数据从主机内存中复制到设备的内存中,以供后续其他模块处理。当指令是 put 时,DMA 会复制 key 和 value;当指令是 get 时,DMA 会复制 key。

DMA 将数据组织成如表 4 所示的记录形式,以方便传输或者写入 Flash 颗粒。校验码是由数据内容计算出来的固定长度的字段。因为 Flash 存储技术自身的缺陷,存储单元中的数据过一段时间后可能会失效或者改变,通过错误检查和纠正(error correcting code, ECC)保证数据可以完整正确地读出。在网络传输和 key-value 数据库中,一般也会加入校验字段来验证数据的有效性。

表 4　记录结构示意图

校验码	key 长度	value 长度	key	value

UIO 模拟器中的 DMA 只负责从主机端读取数据到设备 DRAM 中,从设备中读取数据到主机端的工作则交给其他模块完成。存储介质的写入速度或者向远程设备发送数据的速度都低于写入 DRAM 的速度,用 DRAM 作为缓存提高效率。但是从设备中读取数据时,可以由设备内模块直接向主机内存传输。

3.4.2　GME 模块

GME 负责把数据的读写请求根据 key 把指令分配到不同的设备上。如图 8 所示,GME 模块收到请求后会读出其中的 key,然后计算 key 的哈希值,并在映射表中查找该哈希值对应的目标服务器 id。如果指令目标是本地设备,就交给 UTL 模块处理,如果指令目标是远程设备就交给 NET 模块处理。GME 模块中的映射表可以通过分布式协议获取,也可以通过配置文件写入,UIO 模拟器中是通过函数计算哈希值和目标设备 id 之间的关系。

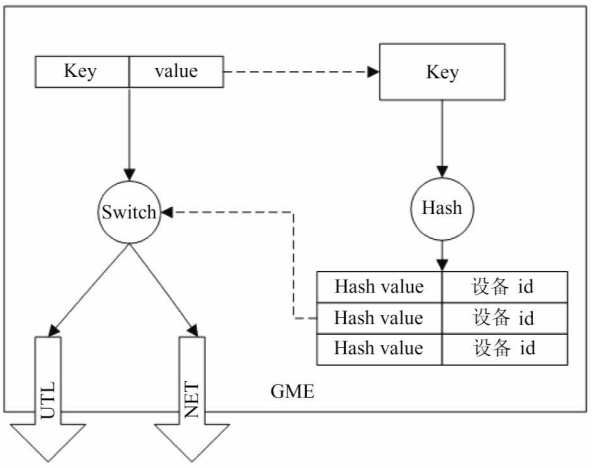

图 8　GME 模块原理示意图

3.4.3　UTL 模块

在 UIO 模拟器中通过建立 Flash 颗粒的块和页 2 级来模拟 Flash 介质的基本性质。Flash 颗粒中,页只能被写入不能改写,修改引起的拷贝复制导致写放大问题。如图 9 所示, UIO 模拟器借鉴了键值存储系统的处理方式,创建 1 条新的记录,顺序写到现在记录的尾端把原有记录视为失效。当 1 个页完整写入后只读取不再改动。块是由页组成的,当 1 个块中所有页都被写入后会成为固定块,不再发生改动。

UIO 提供 key-value 接口,UIO 接收的指令不包含地址,UIO 仿照 FTL 实现 key 到物理地址的映射。UTL 会在 DRAM 中建立哈希映射表,哈希表通过 key 来索引 value 保存的具体位置,其中包括 block id、page id、value 位置和长度。当发生写请求时,UTL 就把记录顺序写入 Flash 中,然后根据写入时的地址更新内存中的哈希表;当发生读请求时,先从哈希表中查询到 value 的地址,然后再从 Flash 中定位并读出数据。当请求是来自本地时,UTL 就可以直接把返回结果写入本地设备的 Completion Queue 并把数据写入主机内存。当请求是来自远程时,UTL 就把数据暂存在设备的内存中,并给 NET 模块发送请求,由 NET 模块处理。

UTL 对块进行定期扫描,进行垃圾回收,提高空间利用率。UTL 从头到尾扫描块,从中读出每一条记录的 key,并把记录地址和该 key 在哈希表中指向的地址进行对比。如果是同一个地址说明该记录是

有效的,UTL 就把该记录写入到活动块中继续保留;如果该记录的地址和哈希表中的不一致,说明该记录已经失效,直接跳过即可,扫描完整个块,该块就可以擦除回收了。

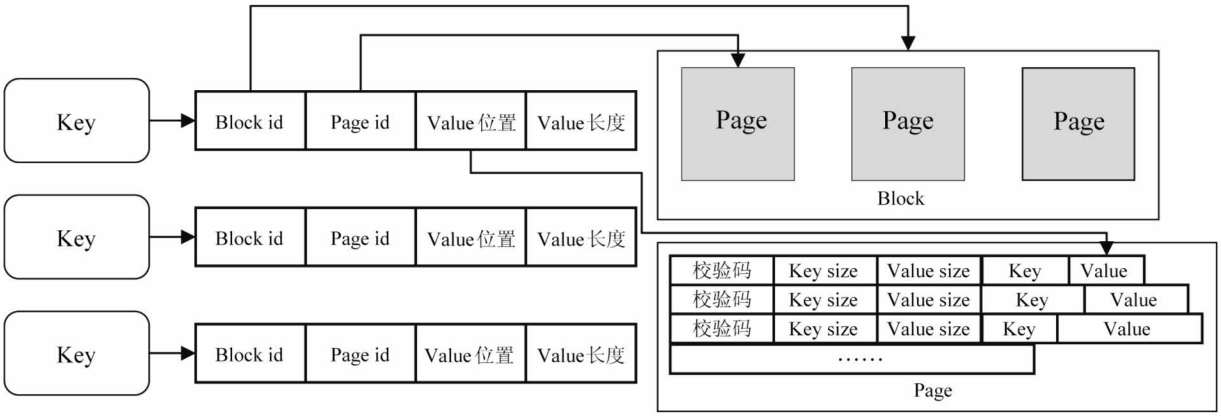

图 9　UTL 哈希表示意图

3.4.4　NET 模块

NET 模块由 2 部分组成,分别负责发送、接收网络请求。Send 流程中会有 4 种情况,远程 put 请求、远程 get 请求、远程 put 回复、远程 get 回复。其中,远程 put 和 get 请求是来自 GME 的指令,GME 根据 key 判断指令目标是远程设备时,会向 NET 模块发送指令并告知目标设备 id,由 NET 模块负责发送。其中,put 请求需要向远程设备发送 key 和 value,get 请求需要发送 key。

远程 put、get 请求的接收,都接收到设备内存中,由 UTL 模块处理。远程设备把 put 和 get 请求处理完后会发送回复,对于 put 的回复只要知道 put 指令执行的状态即可,获得结果后向 CQ 填写返回结果这条指令就完成了。而远程 get 指令是有查询结果的,设备接收到结果后还要复制给主机,等复制结束后就可以填写返回结果了。

3.5　指令整体流程

在主机写入指令后,指令仲裁把指令从指令队列中取出并送入指令解释器,指令解释器解析指令后给 DMA 发送控制信号,然后就由数据通路模块执行。

3.5.1　put 指令

put 指令,主机会提供 key 和 value 的地址和长度,DMA 根据地址和长度分别从主机内存中复制 key 和 value 的值到设备内存中,并存储成记录的格式,然后向 GME 模块发送信号。GME 模块会从设备内存中读出 key,并计算 key 的哈希值,根据哈希值在内存中的映射表查找目标设备的 id。

如果目标设备是本地设备,那么 GME 模块就向 UTL 模块发送请求,UTL 在收到请求后从设备内存中读出记录并写入介质,并把写入到介质中的地址存到设备内存中的映射表。完成写入后,UTL 模块会向 CQ 填写完成信息。如果目标设备是远程设备,则由 NET 模块负责处理。NET 模块收到控制信号后会把内存中的记录分解为多个包并填写元数据以便于传输,然后向目标设备依次发送数据包。目标设备的 NET 模块收到数据包会读取数据包的 opcode,如果判断出是 put 请求就会依次接收包并重新组成记录的格式,然后向 UTL 模块发送信号。UTL 模块接收到信号后会把内存中的记录写入存储介质并修改内存中的映射表,然后向 NET 模块返回完成信息。NET 收到完成信息后会发送到源设备,源设备的 NET 模块收到完成信息后会向 CQ 填写完成信息。

3.5.2　get 指令

get 指令的过程和 put 指令的过程大体相似,这里只介绍其中有区别的部分。DMA 在执行 get 指令时只会从内存中复制 key(因为指令中传递的 value 地址是应用申请好的缓冲区地址和长度)。

UTL 在收到信号后会先从映射表中查找 key 所

对应 value 在存储介质中的地址，根据这个地址找到 value 并复制到主机内存（因为主机内存的带宽和延迟一般都优于存储设备，所以直接复制也不会有性能问题还可以减少开销），然后填写返回信息。NET 模块在向远程设备发送信息时发送的只有 key，远程设备在返回完成信息时还要发送查询到的 value 的值。同样，本地设备的 NET 模块在接收到 value 的值后会直接复制到主机内存中。

4　测评与分析

4.1　测试环境与测试方案

4.1.1　测试环境

UIO 模拟器的测试是在一台服务器上完成。服务器的配置如表 5 所示。UIO 的测试采用在单台服务器上运行多个模拟器程序。由于程序之间通过 Linux 系统提供的消息队列进行通信，使用内存模拟存储介质，硬件只用到了 CPU 和内存。

表 5　测试环境参数表

	部件	型号
硬件	CPU	双 Intel Xeon E5-2650 2.0 GHz
	内存	Samsung DDR3 1333 MHz, 32 GB
软件	OS	CentOS 6.3 64 位
	Kernel	Linux Kernel 2.6.32

4.1.2　测试方案

UIO 的设计目的是减少远程数据传输在设备和内存之间传输导致的额外延迟。为了进行对比测试，在 UIO 模拟器添加了一种模式，让 UIO 模拟器模拟传统的数据通路，在接收到数据后复制到主机内存中，再从主机内存重新读入设备，写入存储介质。除此之外，UIO 模拟器在处理指令时会有一些额外开销。例如 GME 模块会验证指令是发往本地还是远程，传统存储设备中所有指令都是写入本地存储，并没有此项开销。所以用 UIO 模拟器模拟数据无需经过 GME 模块直接读写 UTL 模块的场景，测试 UIO 模拟器在本地模式下的额外开销。

4.2　数据传输延迟测试

在具体的测试中，会使用多种 value 长度（从 4B 到 16 kB）的指令，其中的 key 和 value 都会使用英文字母填充。

4.2.1　UIO 模拟器与传统数据通路对比

在图 10 和图 11 中，可以看出传统通路的指令执行延迟都高于 UIO 设备，根据指令长度的不同会多出 10 ~ 30 μs。延迟的增加主要是因为数据复制和设备间的数据包导致，而传统通路的数据复制次数多于 UIO 设备，所增加的延迟也高于 UIO 设备。put 指令中，value 的长度越长执行延迟越长与 UIO 模拟器的实现方式有关，在使用 put 指令时因为需要向设备写入数据，模拟器在实际运行时需要向系统申请空间，所以越长的指令执行时间也越长。但是这部分时间的增量在 2 种通路都是存在的，并不会影响结论。测试结果表明，UIO 设备路径与传统数据通路相比，远程 put 减少 8.2% 延迟，远程 get 减少 11% 延迟。

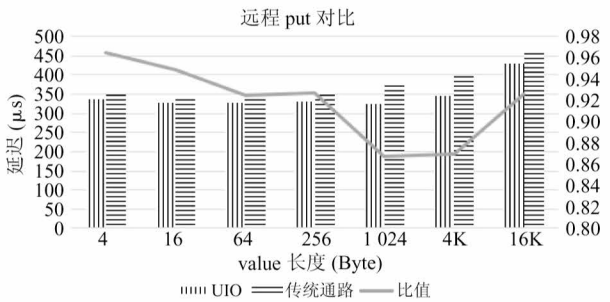

图 10　UIO 设备与传统通路远程 put 指令延迟对比

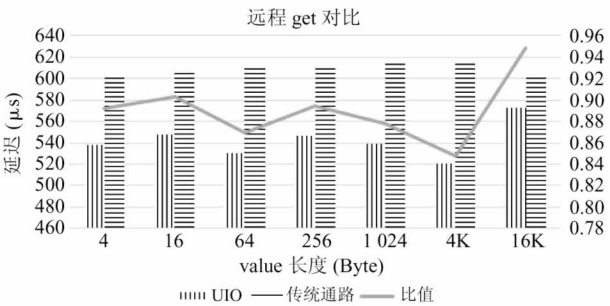

图 11　UIO 设备与传统通路远程 get 指令延迟对比

4.2.2　UIO 模拟器与本地数据通路对比

对于本地 put 指令，可以看到有的指令是 UIO 的延迟更长，有的是 UIO 的延迟更短。大多数的数据差距都在 2 ~ 3 μs 内，差距在于 UIO 设备中需要额外经过 GME 模块计算目标设备 id，对于 3 μs 左右的延迟这种随机波动影响很大，因此 UIO 设备和

传统设备在运行本地 put 指令时开销基本相当。对于本地 get 指令,可以看到 UIO 设备的执行延迟基本超过传统通路,但是超过的幅度很小,基本在10 μs左右。如图 12 和图 13 所示,测试结果表明,UIO 设备路径与传统数据通路相比,本地指令的差距基本不随 value 的长度变化,这是因为 GME 的操作是从设备内存中读取 key,并计算 key 的 Hash 值,没有复制数据的操作。在实际使用中 key 的长度较小,所以 Hash 值的计算开销也不高,开销的变化通常也不大。

4.3　数据分析

从 4.2 节中可以看出,UIO 设备在执行远程指令时延迟小于传统通路,而在执行本地指令时略微高于传统通路。

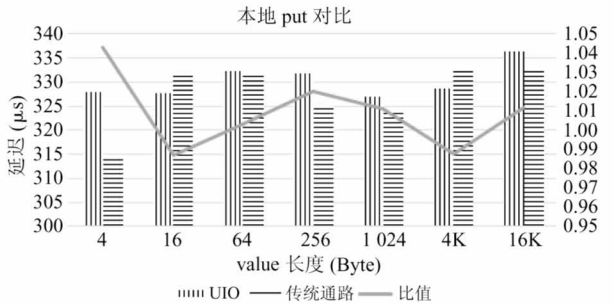

图 12　UIO 设备与传统通路本地 put 指令延迟对比

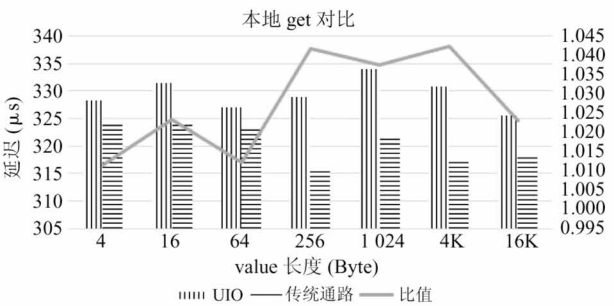

图 13　UIO 设备与传统通路本地 get 指令延迟对比

对于远程指令,传统通路中需要把指令从设备复制到主机中再从主机复制到设备中,相比 UIO 设备各个模块可以共用内存相比,增加了数据复制的开销,这部分开销造成了传统通路的额外延迟。而对于本地指令,UIO 设备相比传统通路增加了 GME 计算指令的目标设备是本地设备还是远程设备的开销,这部分计算主要是使用哈希函数计算哈希值,再根据哈希值查表,都是开销比较小的计算,所以相比传统通路时间延迟增加并不显著。

如果设 UIO 设备减少的远程指令开销比例是 a,增加的本地指令开销比例是 b,UIO 设备有 N 块,设备的能力都是相同的,哈希函数可以把数据均匀分布到 N 块设备上,那么用户的数据会有 $1/N$ 在本地设备,$(N-1)/N$ 在远程设备,UIO 设备相比传统通路的开销比值是

$$u' = \frac{N-1}{N} \times (1-a) + \frac{1}{N} \times (1+b) \qquad (1)$$

设 a 为 10%,b 为 2%,那么 μ 就等于 $0.9 + \frac{0.12}{N}$,N 越大时整体的开销比值就越小。也就是说,在设备数比较多的系统中,设备的请求主要都是设备间发送的远程请求,延迟开销会更接近远程指令的开销。

5　结　论

现有系统中,数据发往远程计算机存储设备的数据通路经过本地网卡、远程网卡、远程内存最终到达远程存储。数据无法直接在网卡和存储设备之间传输,导致了额外的传输延迟。新型高速存储介质的出现,存储设备的访问延迟大幅下降,使得这段延迟在整个传输过程的延迟中占比大大增加。为了降低数据传输延迟,充分利用新式存储设备的性能,本文设计了面向网络与存储的融合设备模型,提出了融合存储、网络和辅助计算功能的融合 I/O 设备。通过在同一设备中融合存储和网络功能,使得设备内的网络模块可以直接与设备内的存储模块通信,简化了数据的传输路径,降低了数据的传输延迟。实现了与 key-value 存储系统相结合的 UIO 设备模拟器。在最终测试中与传统数据通路相比,远程 put 减少 8.2% 延迟,远程 get 减少 11% 延迟,本地 put 增加 0.9% 延迟,本地 get 增加 2.7% 延迟的效果。

参考文献

[1] Oracle. Time Capsule, 1956 Hard Disk[EB/OL]. http://www.oracle.com/technetwork/issue-archive/2014/14-jul/o44timecapsule-2219543.html: Oracle Magazine,

2014

[2] Intel. Product Brief | Intel? Optane? SSD DC P4800X Series [EB/OL]. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-dc-p4800x-p4801x-brief.pdf: Intel Corporation, 2019

[3] Noureddine W. NVMe over Fabrics [EB/OL]. http://www.snia.org/sites/default/files/SDC15_presentations/networking/WaelNoureddine_Implementing_%20NVMe_revision.pdf: Chelsio Communications, 2015

[4] Mayhew D, Krishnan V. PCI express and advanced switching: evolutionary path to building next generation interconnects [C] // The 11th Symposium on High Performance Interconnects, Stanford, USA, 2003: 21-29

[5] Ousterhout J, Agrawal P, Erickson D, et al. The case for RAMClouds: scalable high-performance storage entirely in DRAM [J]. *ACM SIGOPS Operating Systems Review*, 2009, 43(4): 92-105

[6] Sanfilippo, S. Redis [EB/OL]. http://redis.io: Redis Labs, 2009

[7] Decandia G, Hastorun D, Jampani M, et al. Dynamo: Amazon's highly available key-value store [J]. *Operating Systems Review*, 2007, 41(6): 205-220

[8] Kim Y, Tauras B, Gupta A, et al. FlashSim: a simulator for NAND flash-based solid-state drives [C] // International Conference on Advances in System Simulation, Washington, USA, 2009: 125-131

[9] Hardock S, Petrov I, Gottstein R, et al. NoFTL: database systems on FTL-less flash storage [J]. *Proceedings of the VLDB Endowment*, 2014, 6(12): 1278-1281

[10] Gupta A, Kim Y, Urgaonkar B. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings [C] // International Conference on Architectural Support for Programming Language and Operating System, New York, USA, 2009: 229-240

[11] NVM Express Workgroup. NVM Express Specification [M]. Beau Bassin: PlacPublishing, 2013

[12] Seshadri S, Gahagan M, Bhaskaran S, et al. Willow: a user-programmable SSD [C] // Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, Berkeley, USA, 2014: 67-80

[13] Mellanox. RDMA Basic [EB/OL]. https://www.mellanox.com/: Mellanox, 2015

[14] Keeton K, Patterson D A, Hellerstein J M. A case for intelligent disks (IDISKs) [J]. *ACM SIGMOD Record*, 1998, 27(3): 42-52

[15] Mellanox. RDMA Aware Programming user manual [EB/OL]. https://www.mellanox.com/: Mellanox, 2015

[16] 李强. 多核异构的高性能计算机通信关键技术研究 [D]. 北京: 中国科学院大学, 2012

[17] Recio R. An RDMA protocol specification [R]. IETF Inter-draft draft-ietf-raddp-rdmap-03: Santa Clara, 2005

[18] Cho B Y, Jeong W S, Oh D, et al. XSD: accelerating MapReduce by harnessing the GPU inside an SSD [C] // The 46th IEEE/ACM International Symposium on Microarchitecture, Davis, USA, 2013: 1-6

[19] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters [J]. *Communications of the ACM*, 2008, 51(1): 107-113

[20] Ming S J, Sungjin L, et al. BlueDBM: an appliance for big data analytics [J]. *ACM SIGARCH Computer Architecture News*, 2015, 43(3): 1-13

[21] Do J, Kee Y S, Patel J M, et al. Query processing on smart SSDs: opportunities and challenges [C] // ACM SIGMOD International Conference on Management of Data, New York, USA, 2013: 1221-1230

[22] Riedel E, Gibson G A, Faloutsos C. Active storage for large-scale data mining and multimedia [C] // Proceedings of the International Conference on Very Large Data Bases, New York, USA, 1998: 62-73

[23] Cho S, Park C, Oh H, et al. Active disk meets flash: a case for intelligent SSDs [C] // International ACM Conference on International Conference on Supercomputing, New York, USA, 2013: 91-102

[24] Kang Y, Kee Y, Miller E L, et al. Enabling cost-effective data processing with smart SSD [C] // MASS Storage Systems and Technologies, Long Beach, USA, 2013: 1-12

[25] Kang S H, Koo D H, Kang W H, et al. A case for flash memory SSD in Hadoop applications [J]. *International Journal of Control and Automation*, 2013, 6: 1075-1086

[26] Lee S W, Moon B, Park C, et al. A case for flash memory ssd in enterprise database applications [C] // ACM SIGMOD International Conference on Management of Date, Vanconver, Canada, 2008: 1075-1086

[27] Agrawal N, Prabhakaran V, Wobber T, et al. Design

tradeoffs for SSD performance[C]// USENIX Technical Conference, Boston, USA, 2008:57-70

Network and storage oriented converged I/O device simulator

Wei Zheng[*,**,***], Li Feinan[**,***], Xing Jing[**,***], Huo Zhigang[**,***], Sun Ninghui[**,***]

([*]University of Chinese Academy of Sciences, Beijing 100089)

([**]State Key Laboratory of Computer Architecture, Beijing 100089)

([***]Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100089)

Abstract

In the existing computer system architecture, to achieve cross-node data storage operations, data often need to go through the local network card, remote network card and remote memory to reach the remote storage device. This process often requires the participation of remote node operating system and applications. With the development of hardware technology, storage equipment access delay is greatly reduced. In order to further reduce the delay of data transmission across nodes and give full play to the performance advantages of new storage devices, this paper designs a network and storage oriented converged I/O device model UIO to simplify the transmission path of cross-node data storage. The UIO device converges storage and network functions in the same device, reducing the data transmission delay by allowing multiple functional blocks to share memory and data paths in the device to simplify the data transmission path. The UIO device model implements the data processing operations by adding the auxiliary calculation function to the device. Through the combination with the programmable hardware, allowing users to customize the auxiliary calculation modules improves the processing efficiency of UIO devices and expands its application scenarios. This work designs and implements a UIO device simulator for a key-value storage system to verify the effect of the UIO device model. In the simulation test, the remote put delay can be reduced by 31% compared to the traditional data path in the case of 4 kB value, and the remote get delay can be reduced by 20%.

Key words: converged device, device simulator, key-value storage, programmable hardware