DOI:10.7544/issn1000-1239.2018.20180198
55(9):2083-2093,2018

APMSS:一种具有非对称接口的固态存储系统

牛德姣　贺庆建　蔡　涛　王　杰　詹永照　梁　军
(江苏大学计算机科学与通信工程学院　江苏镇江　212013)
(djniu@ujs.edu.cn)

APMSS: The New Solid Storage System with Asymmetric Interface

Niu Dejiao, He Qingjian, Cai Tao, Wang Jie, Zhan Yongzhao, and Liang Jun
(*School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang, Jiangsu* 212013)

Abstract The solid storage system is an important way to solve the problem of computer memory wall. But the existing block-based storage management strategy can't use the advantages of byte-addressable characteristics in solid storage system and causes write amplification, which seriously reduces the I/O performance and lifetime of NVM devices. In term of this problem, the new solid storage system with asymmetric interface named APMSS is presented, and the management of read and write access request are separated by the analysis of two type access request. The read access request is still managed by block unit to avoid increasing the overhead of I/O stack and keep high performance by cache. The minimized direct write strategy is designed and the write access request is managed by dynamic granularity to reduce the communication and written data to solid storage system. At the same time, the multi-granularity mapping algorithm is designed to avoid exchanging amplification between memory and solid storage system and improve the I/O performance. Then the write amplification of solid storage system could be avoided and the write performance of NVM devices could be improved. The prototype is implemented based on PMBD which is the open source solid storage system simulator. Fio and Filebench are used to test the read and write performance of Ext2 on PMBD, Ext4 on PMBD and Ext4 on APMSS. The test results show that Ext4 on APMSS can improve sequential write performance by 9.6%～29.8% compared with Ext2 and Ext4 on PMBD.

Key words solid storage system; write amplification; interface of storage system; read/write strategy; storage system software stack

摘　要 固态存储系统是解决计算机系统存储墙问题的重要手段,但当前所常用的基于数据块访问方式有很大的局限,存在写放大和无法利用内部存储器件支持字节读写特性等问题,严重影响了固态存储系统的I/O性能和使用寿命.设计了具有非对称接口的固态存储系统(APMSS).在分解文件系统层所提交访问请求的基础上,针对读操作的特性,设计了基于块的读机制,避免改变读粒度增加I/O软件栈开销,并能发挥读缓存的作用;设计了最小化直接写机制,包含通用块层的多粒度固态存储系统映射算法和驱动层的动态粒度写算法,仅将实际需写入数据和位置等信息发送给固态存储系统,提高固态存储系

收稿日期:2018-03-26;**修回日期**:2018-06-13

基金项目:江苏省自然科学基金项目(BK20140570);江苏省重点研发计划产业前瞻与共性关键技术项目(BE2015137);中国博士后科学基金项目(2016M601737)

This work was supported by the Natural Science Foundation of Jiangsu Province of China (BK20140570), the Jiangsu Provincial Key Research and Development Plan Industry Foresight and Common Key Technology Projects (BE2015137), and the China Postdoctoral Science Foundation (2016M601737).

通信作者:蔡涛(caitao@ujs.edu.cn)

统的写性能,并减少对固态存储系统使用寿命的影响.在开源的块接口固态存储系统 PMBD 的基础上实现了 APMSS 的原型,使用存储系统的通用测试工具 Fio 和 Filebench 进行测试,结果表明 APMSS 上的 Ext4 相比 PMBD 上的 Ext2 和 Ext4 能提高 9.6%～29.8%的写性能.

关键词 固态存储系统;写放大;存储系统接口;读写策略;存储系统软件栈

中图法分类号 TP316

计算机系统中各部件的发展具有很大的不均衡,当前存储部件的读写速度远低于计算部件的处理能力,这导致了严重的存储墙问题[1].由于存在机械部件,传统的磁盘很难有效提高读写速度.基于 Flash 的固态存储设备具有较高的 I/O 性能,但存在写寿命短和仅支持以块为单位的读写操作等问题.当前出现了一系列 NVM 存储器件,如 PCM[2],STT-RAM[3]和 RRAM[4]等,具有支持以字节为单位的读写、较长的写寿命、低功耗和接近 DRAM 的读写速度等优势,成为解决存储墙问题的重要手段;同时 NVM 存储器件的写寿命和读写速度也还在不断提高.此外 3D XPoint 等技术的出现,使得用 NVM 改造现有基于 Flash 的固态存储设备、构建新型的高速固态存储系统成为当前研究的热点.

但现有 I/O 系统软件栈是面向低速外存系统所设计,难以适应高速固态存储系统读写性能较高的特性,相关研究表明在新型固态存储系统中 I/O 系统软件的开销占总开销的 63%以上[5],因此如何提高 I/O 系统软件的效率是固态存储系统中需要解决的重要问题.PCIe 固态存储设备是当前构建高速固态存储系统的重要基础,但其具有较大的局限性;为了获得较高的传输效率 PCIe 接口以支持块访问方式为主,虽然可以改变每次传输的大小,但效率和灵活性还较低.当前操作系统还是使用块接口访问 PCIe 固态存储设备,无法利用内部 NVM 存储器件支持字节粒度读写的特点;在执行写操作时,以数据块为单位的方式存在写放大的问题,严重影响了固态存储系统的读写性能和使用寿命;在执行读操作时,以数据块为单位读出的数据能用于基于局部性原理构建的读缓存,从而提高访问存储设备时的读性能,同时读操作对固态存储系统的寿命也没有影响.因此有必要针对分析读写操作的不同特性,研究管理粒度和方式不同的读写操作接口,用于构建高效的固态存储系统.

我们设计具有非对称接口的固态存储系统,以提高 I/O 性能和延长其使用寿命.本文的主要贡献有 4 个方面:

1) 针对读写操作的不同特性,分离文件系统所提交的读写访问请求,有效利用固态存储系统内部支持字节读写的特性.

2) 设计了多粒度的固态存储系统映射算法,修改通用块层的结构,为解决写放大问题提供支撑.

3) 设计了动态粒度写算法,避免写放大问题,提高固态存储系统的写性能和延长其使用寿命.

4) 实现了一个具有非对称接口固态存储系统的原型(APMSS),使用 Fio 和 Filebench 进行了测试,验证了 APMSS 具有更高的写性能.

1 相关工作

当前的研究主要集中在使用 NVM 提高存储系统的性能和针对固态存储系统的新型文件系统方面.

1.1 新型 NVM 文件系统方面的研究

BPFS[6]和 PMFS[7]是针对字节寻址的 NVM 存储设备的新型文件系统,提高 I/O 性能,降低读写延迟.BPFS 使用短周期的影子分页法实现 8 b 的原子写操作和以及更细粒度的更新操作,并实现了硬件上写操作的原子性和顺序性.PMFS 分离了元数据和数据的一致性保护方法,使用细粒度日志保护元数据的一致性;同时使用 CoW 策略实现数据写操作,保护文件系统的可靠性和一致性.文献[8]设计了 SCMFS,使用操作系统的 MMU 管理 NVM 存储设备中的数据块,同时使用连续虚拟内存空间管理单位文件简化读写操作;并使用 clflush/mfence 机制保障执行文件访问操作的顺序,但没有给出如何保障文件系统的一致性.文献[9]设计了 Aerie,通过直接让用户态程序访问 NVM 存储设备中的数据,避免现有 I/O 系统软件栈需要进行内核和用户态切换的时间开销,同时也提供了 POSIX 接口以支持现有应用;同时使用 Mnemosyne[10]中的 tornbit RAWL 策略,保护文件系统的一致性.PMFS 分离了元数据和数据的一致性保护方法,使用细粒度日

志保护元数据的一致性;同时使用 CoW 策略实现数据写操作,保护文件系统的可靠性和一致性. 文献[11]针对现有以数据块为单位的写操作机制中,存在写少量数据需先完成冗余读操作的问题,修改虚拟文件系统设计了非数据块粒度的写机制,应用于磁盘能提高 7～45.5 倍的写性能,应用于基于 Flash 的 SSD 能提高 2.1～4.2 倍的写性能. NOVA 是针对 DRAM 和 NVM 混合情况的日志文件系统[12],通过为每个 inode 节点维护一个日志提高并发性和用原子更新实现日志的追加等,在保护文件系统的一致性和操作原子性的同时,能相比现有具有一致性保护的文件系统能提高 3.1～13.5 倍的性能. FCFS 是针对 NVM 的新型文件系统[13],设计了多层次的混合粒度日志,针对元数据和数据分别使用 redo 和 undo 策略,针对应用的选择性并发检查点机制减少了需保存的数据量,使得上层应用的性能提高了近 1 倍. HMVFS 是针对 DRAM 和 NVM 混合情况设计的多版本文件系统[14],通过轻量级的快照技术保护文件系统的一致性,使用内存中的 Stratified File System Tree (SFST)保护多个快照之间的一致性,相比 BTRFS 和 NILFS2 能有效减少快照所需的开销. HINFS 是面向 NVM 设计的高性能文件系统[15],给出了针对 NVM 主存的写缓存机制、以及 DRAM 中索引与缓存行中位图相结合的读一致性机制,并设计了基于缓存贡献模型的 NVM 主存写机制. 文献[16]针对事务内存设计了模糊持久性策略,通过基于日志的执行和易失性的检查点机制,减少了实现事务机制的开销,能提高 56.3%～143.7%的文件系统 I/O 性能. SIMFS[17]是一种能利用虚拟内存管理机制的内存文件系统,每个被访问的文件都拥有一个独立的连续虚拟地址空间,数据地址空间的连续性可以支持高速的顺序访问,而且独立空间的划分方式可以避免文件访问中的地址冲突问题,同时 SIMFS 还使用层级结构的文件页表(file page table)技术来组织文件数据,从而保证了 SIMFS 可以使内存带宽接近饱和. 文献[18]在 NOVA 的基础上设计了 NOVA-Fortis,提高 NVM 文件系统容忍存储器件错误和软件系统 bug 的能力,相比现有没有容错功能的 DAX 文件系统能提高 1.5 倍的读写性能,相比基于块的文件系统能提高 3 倍的读写性能. DevFS 通过将文件系统嵌入到 NVM 存储设备中,能提高 NVM 存储设备 2 倍的吞吐率[19].

1.2 减少 NVM 系统软件栈开销的研究

文献[20]分析了使用不同控制方式访问 PCM 存储设备时的效率,发现使用轮询和同步读写策略、相比使用中断和异步读写策略能有效减少访问 PCM 存储设备时的 I/O 软件栈开销. 文献[21]验证了在读 PCM 存储设备中的数据时,使用轮询能减少接口数据在用户态与内核态之间切换的开销,提高读性能. 文献[22]和文献[23]设计了基于硬件的元数据和数据访问路径分离的方式,在用户态直接访问 NVM 存储设备中的数据,减少元数据的修改次数,提高访问 NVM 存储设备的 I/O 性能. BPFS[6]抛弃块设备接口,在用户态使用 load 和 store 指令直接访问 DIMM 接口的 NVM 存储设备,构建了一个全新的存储系统 I/O 软件栈. Mnemosyne 是针对 DIMM 接口 NVM 存储设备设计的轻量级访问接口[10],实现了在用户态空间中管理 NVM 存储空间和保护文件系统一致性等问题. NVTM 是一个面向 NVM 的事务访问接口[24],将非易失存储器直接映射到应用程序的地址空间,允许易失和非易失数据结构在程序中的无缝交互,从而在读写操作中避免操作系统的介入,提高数据访问性能. NV-heaps 是针对 NVM 存储设备的一个轻量级的高性能对象存储系统[25],减少存储系统软件栈的开销,相比 BerkeleyDB 和 Stasis 能提高 32 倍和 244 倍的执行速度. 文献[26]针对使用 NVM 存储设备构建 SCM 时,块访问接口无法附加优化存储系统性能信息的问题,设计了基于对象的 SCM;文献[27]针对 SCM,设计了一种新型的控制器和相应的函数库,通过操作之间的解耦合,减少存储系统软件栈的开销,并支持并发的原子操作;文献[28]在分析不同事务之间关联性的基础上,设计了 DCT 减少事务提交顺序之间的影响,并提高事务的执行效率;文献[29]在 ISA 的基础上,从程序执行路径中移除日志操作,使用动态标签减少移动日志数据的开销,借助硬件减少日志管理的开销,从而高效的实现访问 NVM 时操作的原子性;文献[30]设计了 Stampede 开发套件,能提高应用访问硬事务内存(HTM)和软事务内存(STM)时的性能,同时简化了 STM 的设计,使得 Blue Gene/Q 上 64 线程和 Intel Westmere 上 32 线程的性能分别提高了 17.7 倍和 13.2 倍. Path hashing 能避免对 NVM 的额外写操作,提高执行速度,减少所使用的 NVM 存储空间[31];文献[32]针对 PCM 设计了基于页的高效管理方式,适应上层应用的访问方式,首

先使用双向链表管理 PCM 中的页,再使用 DRAM 构建了 PCM 页的缓存并设计了基于进入时间的淘汰算法,最后综合页迁移和交换信息优化了 PCM 中页的分配;文献[33]混合使用 SSD 和磁盘设计了 I-CASH,利用 SSD 提供高速随机读性能,利用处理器中多个计算核心的强大计算能力和磁盘构建 SSD 中数据的日志,减少对 SSD 的随机写操作,在延长 SSD 使用寿命的同时,提高随机写的性能.

本文设计了具有非对称接口的固态存储系统,将读写访问请求进行分离,并针对读写访问请求的不同特性设计了相应的管理算法和多粒度的固态存储系统映射算法,从而能降低固态存储系统写放大问题,有效利用固态存储系统内部支持字节读写的特性,提高固态存储系统的写性能和延长其使用寿命.

2 具有非对称接口固态存储系统的结构

数据块是传统磁盘的数据组织和管理单位,现有的存储系统 I/O 软件栈在访问存储设备时也是以数据块为基本单位. 这使得在访问块接口固态存储系统时无法利用 NVM 存储器件支持字节读写的特性,也没有利用 PCIe 接口能动态调整数据传输粒度的优势. 在执行写操作时,存在严重写放大问题,同时还会严重影响固态存储系统的使用寿命,降低了块接口固态存储系统的 I/O 性能. 而读操作不会影响固态存储系统的寿命,较大粒度的读操作也有利于实现数据的预取,提高读操作的性能;同时较大的读粒度也能减少读操作的数量,降低存储系统软件栈的开销.

我们给出具有非对称接口固态存储系统的结构,如图 1 所示. 包括位于驱动层中的动态粒度写模块和面向缓存的读模块,以及位于通用块层中的多粒度映射模块等主要部分,同时去掉了写缓存,将存储设备和文件系统之间的缓存仅仅作为读缓存. 多粒度映射模块用于在文件系统写固态存储系统前获得实际写数据的地址和大小等信息,从而动态改变 PCIe 接口的传输量;访问请求分析器实现对文件系统访问请求的分析,将读和写访问请求分解到对应模块执行;动态粒度写模块依据实际写数据的大小,在固态存储系统内部以字节为最小粒度执行写操作,避免写放大问题;面向缓存读模块负责以数据块为单位读取 NVM 存储器件中保存的数据,并反馈给文件系统.

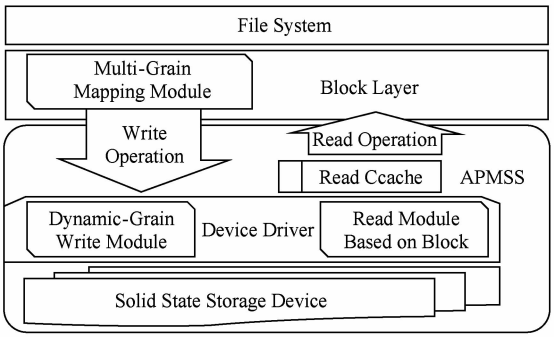


Fig. 1 Structure of new solid storage system with asymmetric interface

图 1 具有非对称接口固态存储系统的结构

3 最小化直接写机制

固态存储系统的写操作会消耗存储器件的使用寿命,因此如何减少写入固体存储系统的数据量是一个重要问题.

在现有的存储系统写机制中,文件系统首先使用块粒度组织需要写入固态存储系统的数据,再通过以块为单位的映射表查找写入存储系统的地址,最后以块为单位向存储系统传输并写入数据. 当仅需要向固态存储系统写入少量数据时,存在较严重的写放大问题,在影响固体存储系统使用寿命的同时,还会因为无效写操作影响写性能.

我们从改变写算法和映射算法两方面入手,设计固态存储系统的最小化直接写机制.

3.1 动态粒度固态存储系统写算法

对文件系统的写访问请求,我们以其实际写数据量为基础,动态调整写入固态存储系统的数据量,避免写放大的问题.

定义 $write\_pos \in N$ 表示写入数据的起始地址,当 $write\_pos = -1$ 时表示不启用动态粒度写算法.

定义 $write\_len \in N$ 表示写入数据的长度.

如图 2 所示,当写入数据量小于系统数据块大小时,首先在文件系统层中获取写入数据块中的 $write\_pos$ 和 $write\_len$,根据 $write\_len$ 设置 PCIe 传输的数据量,仅将写入数据、$write\_pos$ 和 $write\_len$ 传输给固态存储系统,最后根据 $write\_pos$ 和 $write\_len$ 将数据包写入 NVM 存储器件的相应位置.

如图 3 所示,当写入数据量大于系统数据块大小时,同样首先在文件系统层中获取写入数据块的地址和大小,接着按照最大为系统数据块大小将写入数据分成若干个数据包,并以实际大小构建最后

一个数据包;若数据包的大小等于数据块,则设置 $write\_pos=-1$,采用现有块为单位传输方式,并由固态存储系统将数据包作为一个数据块写入设备;若数据包小于数据块的大小,则从该数据包中获取 $write\_pos$ 和 $write\_len$ 的信息,设置 PCIe 接口的传输数据量、并发送数据包、$write\_pos$ 和 $write\_len$,最后固态存储系统根据 $write\_pos$ 和 $write\_len$ 将数据包写入相应位置.

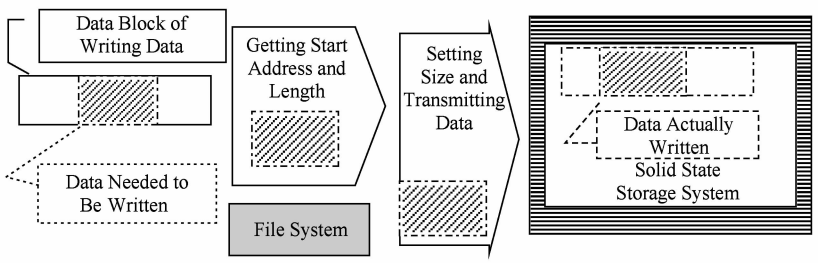


Fig. 2 The write process with actually written data less than size of data block

图 2 写入数据量小于系统数据块大小时的写流程

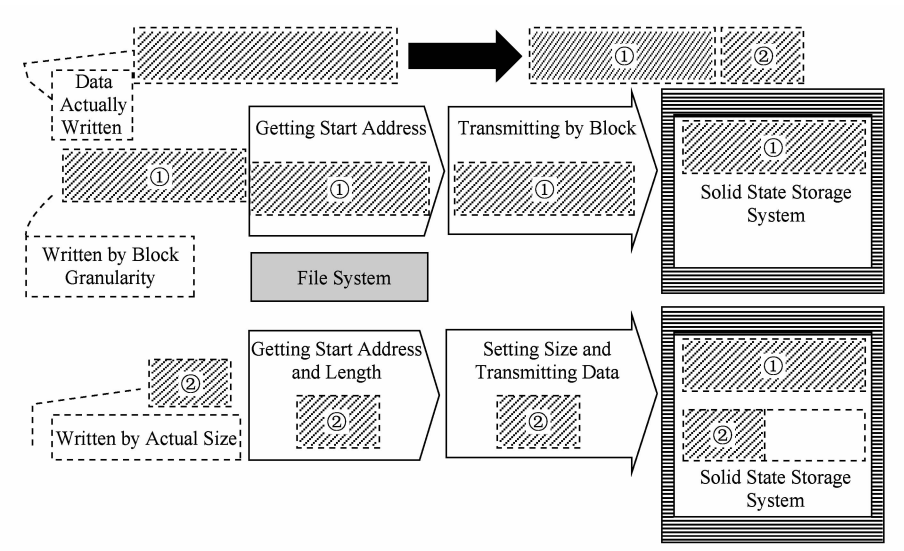


Fig. 3 The write process with actually written data larger than the size of data block

图 3 写入数据量大于系统数据块大小时的写流程

3.2 多粒度的固态存储系统映射算法

文件系统首先将写入固态存储系统的数据首先保存在内存中,通过内存与存储系统之间的映射表,确定写入存储系统的地址和长度后再写入存储系统.现有的存储系统映射表均是以数据块为单位进行组织和管理,这使得内外存之间的数据交换只用以数据块为单位.

我们设计多粒度的固态存储系统映射表,在每个映射项内增加 $dirty\_pos\in N$ 和 $dirty\_len\in N$,保存内外存之间多粒度的映射信息.其中 $dirty\_pos$ 表示在该数据块中需要更新到存储系统数据区的起始地址,$dirty\_len$ 表示该数据块中需要更新到存储系统数据区的长度.当 $dirty\_pos=-1$ 时,表示该数据块不采用多粒度映射算法.

使用 $S\in N$ 和 $L\in N$ 分别表示要写入数据起始逻辑地址与长度,针对每次写操作使用 3 步骤修改固态存储系统映射表,实现多粒度的内外存映射算法.

步骤 1. 依据 S 查找固态存储系统映射表中所对应的映射项,如 S 与该映射项的起始地址相同且 L 的值与数据块大小相同,则将该映射项的 $dirty\_pos=-1$,否则将 S 对应的物理地址保存到 $dirty\_pos$ 中.

步骤 2. 比较 L 的值是否超出了该映射项对应逻辑块的长度,如未超出,则使用 L 设置 $dirty\_len$ 的值,并将 L 的值清零;否则依据数据块大小和 $dirty\_pos$ 计算出 $dirty\_len$ 的值,并更新 S 和 L.

步骤 3. 如果 $L=0$ 则结束整个操作,否则回到步骤 1 继续做.

图 4 给出了一个涉及 3 个映射项的写操作中计算每个映射项 $dirty\_pos$ 和 $dirty\_len$ 的情况.

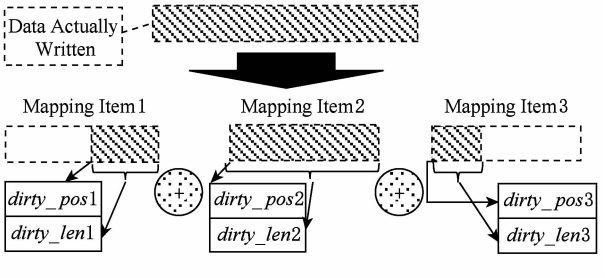


Fig. 4 Calculating *dirty_pos* and *dirty_len* in mapping table between memory and PCIe NVM Device

图 4 固态存储系统映射表中计算 *dirty_pos* 和 *dirty_len*

通过多粒度固态存储系统映射算法,在每个固态存储系统映射项中标识实际需要写入数据的起始地址和长度,使得动态粒度固态存储系统写算法能获得实际需要写入的数据块信息,避免了文件系统与固态存储系统之间只能以数据块为单位交换数据的局限,为解决固态存储系统的写放大问题提供了支撑,从而提高固态存储系统的使用寿命和写性能.

4 基于块的读机制

读操作不会影响固态存储系统的寿命,同时较大的读操作粒度能利用局部性原理实现数据的预取,通过构建读缓存,提高读操作的性能;此外 I/O 操作中的系统软件开销已经成为影响固态存储系统读写性能的重要因素,同时 NVM 存储器件的读操作速度高于写操作,因此在固态存储系统的读算法中减少读次数,能有效降低存储系统软件栈的开销.

我们以数据块为单位执行固态存储系统的读操作,如图 5 所示;首先由文件系统获取读操作所在的数据块信息,并向固态存储系统发出读请求,固态存储系统读出数据后以块为单位传输给文件系统.

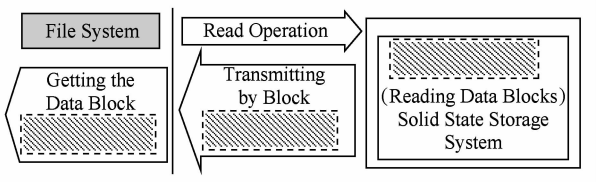


Fig. 5 The process of read strategy based on block

图 5 基于块读机制的流程

以数据块作为读操作的单位,能与现有文件系统和缓存中数据的管理粒度保持一致,避免数据管理粒度转换等额外 I/O 栈软件开销,提高读操作的效率;同时文件系统与固态存储系统之间,每次的传输单位均一致,避免了频繁修改 PCIe 接口的参数,能减少了传输中的额外软件开销,提高读操作的性能.

5 原型测试与分析

我们首先实现具有非对称接口固态存储系统的原型,再使用通用的测试工具进行测试,并加载不同的文件系统进行分析与比较.

5.1 原型系统的实现

我们在块接口固态存储系统 PMBD[34] 的基础上实现 APMSS 的原型. 在模拟 NVM 存储器件时,使用 PMBD 的缺省配置,在 DRAM 的基础上增加 85 ns 的读延迟和 500 ns 的写延迟模拟 NVM 介质;同时修改 Linux 内核,在内核地址的尾部预留 10 GB 内核空间作为 PMBD 的存储地址空间.

此外我们修改 Linux 内核中通用设备层管理内外存映射结构 *buffer_head*,增加 *dirty_pos* 和 *dirty_len* 两个指针用于保存需要实际需要写回存储系统的数据位置和长度信息,并修改内外存映射的源代码实现多粒度的固态存储系统映射算法;再修改 Linux 内核中与设备驱动交互的 bio 结构,增加 *write_pos* 和 *write_len* 保存从 *buffer_head* 结构获得的 *dirty_pos* 和 *dirty_len* 信息;最后修改 PMBD 的源代码,将实际写回数据的位置和长度信息传递到给固态存储设备,增加动态粒度写算法、基于数据块的读算法和读写请求区分器,从而在执行写操作时利用 NVM 存储器件字节寻址特性,仅写入实际修改的数据;从而构建具有非对称接口固态存储系统的原型系统 APMSS.

使用存储系统通用测试工具 Fio 和 Filebench 测试 APMSS 原型的写性能,测试环境的配置如表 1 所示. 在测试时,所有的 Ext4 均使用 DAX 方式.

Table 1 The Configuration of Testing Environment

表 1 测试环境的配置

Testing Item	Configuration
CPU	Intel® Xeon® E5606 @ 2.13 GHz
Memory Size/GB	30
SSD Disk Size/GB	60
OS/GB	Red Hat Enterprise 6.1

5.2 读写不同大小文件的性能

首先使用 Ext4 格式化 APMSS,再应用 Fio 中 Linux 异步 I/O 引擎 libaio,调整文件的大小分别为 8 KB,32 KB,128 KB 和 512 KB,测试 APMSS 顺序读写 3 万个文件时的 I/O 性能,访问块大小设置为

4 KB;并在 PMBD 上 Ext4 和 Ext2 执行相同测试,进行对比和分析.测试结果如图 6 和图 7 所示.

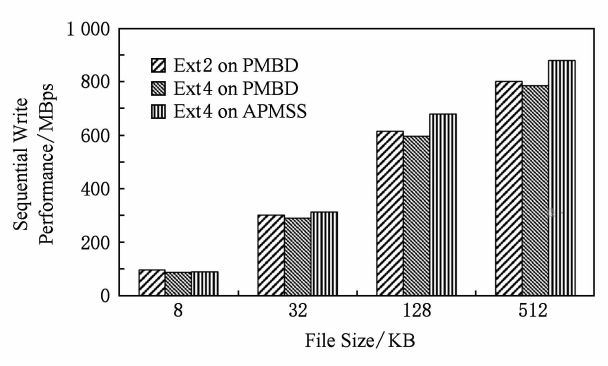


Fig. 6 Sequential write performance with different size files

图 6 顺序写性能的测试

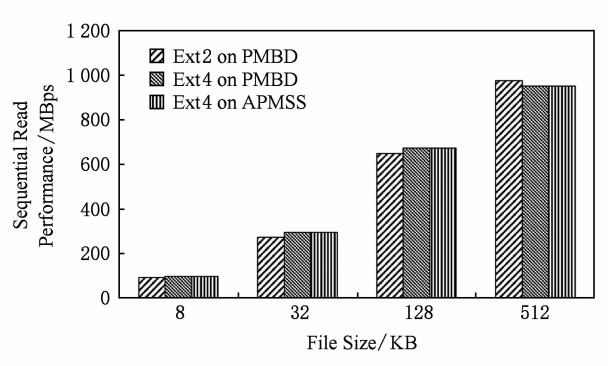


Fig. 7 Sequential read performance with different size files

图 7 顺序读性能的测试

图 6 是顺序写性能的测试结果,从图 6 中可以看出:

1) 采用 APMSS 能有效提高 Ext4 的写性能.当文件大小为 128 KB 时,相比 PMBD 写性能提高了 14%;同时还改变了 Ext4 写性能低于 Ext2 的情况,相比 Ext2 on PMBD 写性能提高了 9.6%.这说明 APMSS 能针对固态存储系统的特性,使用动态粒度写算法通过仅仅写入实际修改的数据,避免写放大问题,提高写性能.

2) 当单个文件大小增加后,写相同数据量所需访问文件的次数不断下降,减少了读写文件元数据所需的时间开销,使得写速度不断提高;在单个文件从 8 KB 增加到 512 KB 后,APMSS 的写性能提高了 9 倍多,而 PMBD 上的 Ext4 和 Ext2 仅增加了 8.2 倍和 7.4 倍,这表明 APMSS 相比现有块接口固态存储系统具有写性能的优势,同时能利用 APMSS 最小化写机制和内部支持字节写的特性减少读写文件元数据所需的时间开销.

3) 在改变所写文件大小时,APMSS 上的 Ext4 的写性能始终高于 PMBD 上的 Ext4,同时所提高的写性能绝对值随着文件大小的增加而不断提高,这和 2)中类似,是由于减少了管理文件元数据的时间开销,这也进一步表明 APMSS 最小化写机制的有效性.在所写文件的大小从 8 KB 增加到 512 KB 时,写性能提高的速度呈现先增加后下降的趋势,在文件大小为 128 KB 时达到最大值,这是由于文件元数据读写速度不断提高并逐步接近文件系统处理文件元数据速度,消弱了文件写性能的提高幅度,这也表明 APMSS 能减轻存储系统所导致的计算机系统性能瓶颈.

图 7 给出了顺序读的测试结果,从图 7 中可以看出 APMSS 和 PMBD 上的 Ext4 具有相同的读性能,这主要由于 APMSS 能利用非对称接口区分读写操作,使用以数据块为粒度的方法处理读操作,避免在提高读写操作灵活性的同时对读性能产生影响,验证了 APMSS 具有良好的适应能力;同时可以发现对读写访问请求进行区分、采用不同的管理机制所可能增加的额外管理开销非常小,没有发现对 APMSS 的读性能照成影响.

5.3 改变读写文件大小时的访问请求处理效率

使用 3.2 节中相同的测试工具和设置,测试 APMSS,PMBD 上 Ext4 和 Ext2 处理访问请求的速度,测试结果如表 2 和表 3 所示.

Table 2 The IOPS of Sequential Write

表 2 顺序写时的访问请求处理速度(IOPS)

File Size/KB	Ext2 on PMBD	Ext4 on PMBD	Ext4 on APMSS
8	24 291	21 905	22 345
32	77 071	74 188	74 428
128	167 247	152 793	192 983
512	212 601	201 068	241 186

Table 3 The IOPS of Sequential Read

表 3 顺序读时的访问请求处理速度(IOPS)

File Size/KB	Ext2 on PMBD	Ext4 on PMBD	Ext4 on APMSS
8	24 499	24 793	24 801
32	70 032	75 376	75 376
128	166 089	172 104	172 009
512	250 309	243 871	243 865

从表 2 和表 3 的测试结果中可以看出:

1) 读写访问请求处理速度的测试结果与 I/O 性能的测试结果类似.

2) 在执行写操作时,APMSS 上 Ext4 处理写访问请求的速度高于 PMBD 上 Ext4 和 Ext2,特别是针对 PMBD 上 Ext4 具有较大的优势,随着文件大小的增加,APMSS 上 Ext4 相比 PMBD 上 Ext4 所提高的 IOPS 越来越显著,这验证了 APMSS 最小化直接写机制能有效避免写放大的问题.

3) 在执行读操作时,APMSS 和 PMBD 上 Ext4 处理访问请求的速度相同,这验证了 APMSS 具有良好的适应能力,能区分读写操作,使用基于数据块的方法完成读访问请求,有利于发挥固态存储系统中读缓存的作用;同时也表明区分读写访问请求分别使用不同的管理策略可能增加的额外管理开销非常小,对 APMSS 的执行读操作时的 IOPS 没有影响;因此这些结果表明,APMSS 在提高写访问请求处理速度的同时保持读访问请求的处理效率.

5.4 改变访问块大小时的 I/O 性能

同样使用 Ext4 格式化 APMSS,应用 Fio 中 Linux 异步 I/O 引擎 libaio,改变文件系统中访问块的大小分别为 4 KB,8 KB,16 KB 和 32 KB,测试 APMSS 顺序读写 3 万个 512 KB 文件时的 I/O 性能;并在 PMBD 上 Ext4 和 Ext2 执行相同测试,进行对比和分析.测试结果如图 8 和图 9 所示.

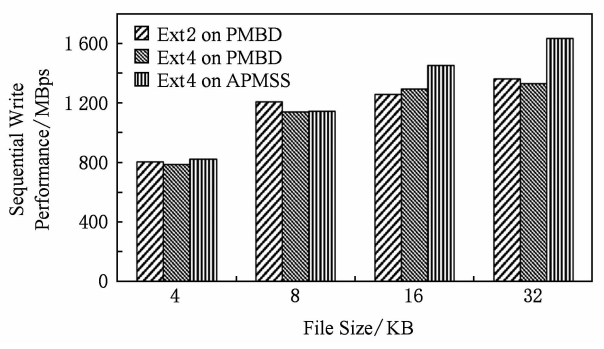


Fig. 8 Sequential write performance with different block sizes

图 8 设置不同大小访问块时的顺序写性能

从图 8 中可以发现:

1) 随着访问块的增加,APMSS 相比 PMBD 提高写性能的效果越来越明显.

2) 当访问块大小为 4 KB 时,APMSS 上 Ext4 相比 PMBD 上 Ext4 和 Ext2 能提高 2%~4%的写性能.

3) 当访问块大小为 32 KB 时,APMSS 上 Ext4 的写性能相比 PMBD 上 Ext4 提高了 22%、相比 PMBD 上 Ext2 提高了 20%.

4) 综合 2)和 3)可以发现,当访问块增大后,单个访问块内实际需要写入固态存储系统数据所占的比重也随之下降,使得 PMBD 上的写放大问题越来越严重,对写性能的影响也就越明显;但同时 APMSS 中最小化直接写机制的效果也就越显著,使得 APMSS 在访问块较大时写性能的优势更加显著.

5) 随着访问块从 4 KB 增加到 32 KB,写 3 万个 512 KB 文件所需写访问请求的数量逐渐下降,减少了 I/O 软件栈的开销,使得 APMSS 的写性能提高了约 1 倍;而同时 PMBD 上的 Ext4 和 Ext2 仅提高了 70%;这进一步说明 APMSS 相比现有的块接口固态存储系统具有写性能的优势.

此外使用相同的配置,改变文件系统中访问块的大小,测试顺序读 3 万个 512 KB 文件的性能,结果如图 9 所示:

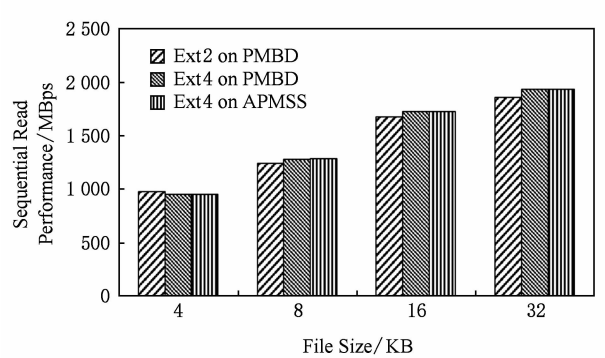


Fig. 9 Sequential read performance with different block sizes

图 9 设置不同大小访问块时的顺序读性能

从图 9 中的结果可知,APMSS 和 PMBD 上的 Ext4 具有相同的读性能,这验证了 APMSS 在提高写性能的同时,不会因为需要区分读写操作而影响其读性能.

5.5 应用综合应用负载时的测试

同样使用 Ext4 格式化 APMSS,应用 Filebench 来模拟应用服务器的访问情况,选择 Copyfile, Webserver 和 Fileserver 三种类型负载测试 APMSS 的 I/O 性能,设置文件大小为 32 KB、文件数量为 5 万个、访问块大小为 4 KB,与 PMBD 上的 Ext4 进行比较,测试结果如图 10 所示.

Copyfile 负载是模拟用户复制文件目录树的行为,主要测试系统持续读写性能.在应用 Copyfile 负载时,从测试结果可以发现 APMSS 上的 Ext4 能提高 6.37%的 I/O 性能.这主要因为 APMSS 优化了固态存储系统的写操作,按照实际数据量大小以字节为单位写入固态存储系统,避免了不必要的写操作,提高了其上 Ext4 文件系统的写性能.

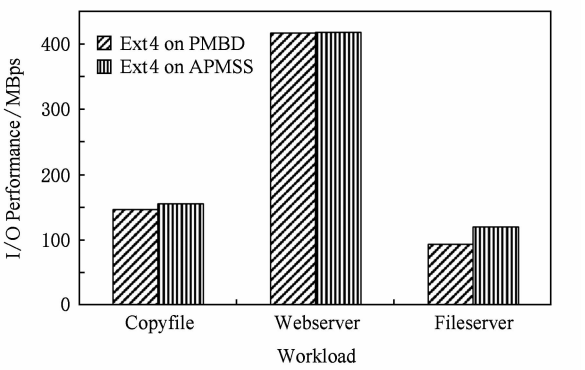


Fig. 10 The I/O performance with different real workload

图 10 不同真实负载下的 I/O 性能

Webserver 负载是模拟用户访问 Web 服务器的负载,主要是文件的读操作.在应用 Webserver 负载时,APMSS 与 PMBD 上的 Ext4 的 I/O 性能基本相同.这主要由于 APMSS 能利用非对称接口区分读写操作,在使用最小化直接写机制的同时,仍然使用基于数据块的方式完成读访问请求,从而保持了较高的读性能,验证了 APMSS 具有良好的适应性.

Fileserver 负载是模拟文件服务器中文件的共享、读写操作等情况,在应用 Fileserver 负载时,APMSS 上 Ext4 的 I/O 性能相比 PMBD 上的 Ext4 提高了 28.4%.这主要是由于 Fileserver 负载中的每个访问请求均包含一系列文件的 open,write,append,read,close 等操作,使得 APMSS 能发挥最小化直接写机制在处理数据量较小的文件元数据读写时的优势,有效解决写放大问题,提高了应用的性能.

6 总 结

使用 NVM 存储器件改造现有 PCIe 固态存储设备是构建新型高速固态存储系统的重要手段.固态存储系统现有块访问方式有很大的局限,存在写放大和无法利用 NVM 存储器件支持字节读写特性等问题,但同时也有利于获得较高的态存储设备读性能.我们针对固态存储系统设计了最小化直接写机制和基于块的读机制,区分读、写访问操作;在处理写访问操作时,仅将修改的数据写入固态存储系统,避免写放大问题,提高写性能;同时仍然以块为单位完成读访问操作,利用读缓存获得较高的读性能.最后我们在开源的 PMBD 的基础上,实现了具有非对称接口新型固态存储设系统的原型 APMSS,使用存储系统的通用测试工具 Fio 和 Filebench 进行测试,并于现有基于 NVM 的块接口存储系统 PMBD 上的不同文件系统进行比较,结果表明 APMSS 上的 Ext4 相比 PMBD 上的 Ext2 和 Ext4 能提高 9.6%~29.8%的写性能,从而验证了所设计的算法能有效提高固态存储系统的 I/O 性能.

当前我们还未利用固态存储系统内部的并行性优化读写操作的执行效率,下一步我们将针对固态存储系统的内部特性,进一步优化读写操作的性能.

参 考 文 献

[1] Carvalho C. The gap between processor and memory speeds [C] //Proc of IEEE Int Conf on Control and Automation. Piscataway, NJ: IEEE, 2002: 51-58

[2] Athmanathan A, Stanisavljevic M, Papandreou N, et al. Multilevel-cell phase-change memory: A viable technology [J]. IEEE Journal on Emerging and Selected Topics in Circuits and Systems, 2016, 6(1): 87-100

[3] Charlie R. Everspin releases fastest and most reliable non-volatile storage class memory [OL]. [2018-03-01]. https://www.everspin.com/news/everspin-releases-fastest-and-most-reliable-non-volatile-storage-class-memory

[4] Jo S H, Kumar T, Narayanan S, et al. 3D-stackable crossbar resistive memory based on field assisted superlinear threshold (FAST) selector [C] //Proc of 2014 IEEE Int Electron Devices Meeting. Piscataway, NJ: IEEE, 2014: 671-674

[5] Swanson S, Adrian M. Refactor, reduce, recycle: Restructuring the I/O stack for the future of storage [J]. Computer, 2013, 46(8): 52-59

[6] Condit J, Edmund B, Frost C, et al. Better I/O through byte-addressable, persistent memory [C]//Proc of the 22nd ACM SIGOPS Symp on Operating Systems Principles (SOSP 2009). New York: ACM, 2009: 133-146

[7] Dulloor S R, Kumar S K, Keshavamurthy A, et al. System software for persistent memory [C] //Proc of the 9th European Conf on Computer Systems (EuroSys 2014). New York: ACM, 2014

[8] Wu Xiaojian, Narasimha R A L. SCMFS: A file system for storage class memory [C] //Proc of 2011 Int Conf for High Performance Computing, Networking, Storage and Analysis. New York: ACM, 2011

[9] Volos H, Nalli S, Panneerselvam S, et al. Aerie: Flexible file-system interfaces to storage-class memory [C] //Proc of the 9th European Conf on Computer Systems (EuroSys 2014). New York: ACM, 2014

[10] Volos H, Tack A, Swift M. Mnemosyne: Lightweight persistent memory [C] //Proc of the 16th Int Conf on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011). New York: ACM, 2011: 91-104

[11] Campello D, Lopez H, Useche L, et al. Non-blocking writes to files [C]//Proc of the 13th USENIX Conf on File and Storage Technologies (FAST 2015). Berkeley, CA: USENIX Association, 2015: 151-165

[12] Xu J, Swanson S. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories [C] //Proc of the 14th USENIX Conf on File and Storage Technologies (FAST 2016). Berkeley, CA: USENIX Association, 2016: 22-25

[13] Ou Jiaxin, Shu Jiwu. Fast and failure-consistent updates of application data in non-volatile main memory file system [C] //Proc of the 32nd Int Conf on Massive Storage Systems and Technology (MSST 2016). Piscataway, NJ: IEEE, 2016

[14] Zheng Shengan, Huang Linpeng, LiuHao, et al. HMVFS: A hybrid memory versioning file system [C] //Proc of the 32nd Int Conf on Massive Storage Systems and Technology (MSST 2016). Piscataway, NJ: IEEE, 2016: 16-29

[15] Ou Jiaxin, Shu Jiwu, Lu Youyou. A high performance file system for non-volatile main memory [C] //Proc of the 17th European Conf on Computer Systems (EuroSys 2016). New York: ACM, 2016: 18-21

[16] Lu Youyou, Shu Jiwu, Sun Long. Blurred persistence in transactional persistent memory [C] //Proc of the 31st Symp on Mass Storage Systems and Technologies (MSST 2015). Piscataway, NJ: IEEE, 2015: 1-13

[17] Sha E H M, Chen Xianzhang, Zhuge Qingfeng, et al. A new design of in-memory file system based on file virtual address framework [J]. IEEE Trans on Computers, 2016, 65(10): 2959-2972

[18] Xu Jian, Zhang Lu, Memaripour A, et al. NOVA-Fortis: A fault-tolerant non-volatile main memory file system [C] // Proc of the 26th ACM Symp on Operating Systems Principles (SOSP 2017). New York: ACM, 2017: 478-496

[19] Kannan S, Andrea C, Remzi H, et al. Designing a true direct-access file system with DevFS [C] //Proc of the 16th USENIX Conf on File and Storage Technologies (FAST 2018). Piscataway, NJ: IEEE, 2018: 12-15

[20] Yang J, Minturn D B, Hady F. When poll is better than interrupt [C] //Proc of the 10th USENIX Conf on File and Storage Technologies (FAST 2012). Berkeley, CA: USENIX Association, 2012: 14-17

[21] Vučinić D, Wang Qingbo, Guyot C, et al. DC express: Shortest latency protocol for reading phase change memory over PCI express [C] //Proc of the 12th USENIX Conf on File and Storage Technologies (FAST 2014). Piscataway, NJ: IEEE, 2014: 17-20

[22] Caulfield A, Mollov T, Eisner L, et al. Providing safe, user space access to fast, solid state disks [J]. ACM SIGARCH Computer Architecture News, 2012, 40(1): 387-400

[23] Eisner L, Mollov T, Swanson S. Quill: Exploiting fast non-volatile memory by transparently bypassing the file system, TR-CS2013-0991 [R/OL]. San Diego: Department of Computer Science and Engineering, University of California, 2013 [2018-03-01]. http://cseweb. ucsd. edu/~ swanson/papers/TR-cs2013-0991-Quil. pdf

[24] Coburn J, Caulfield A, Grupp L, et al. NVTM: A transactional interface for next-generation non-volatile memories, CS2009-0948 [R/OL]. San Diego: Department of Computer Science and Engineering, University of California, 2009 [2018-03-01]. http://csetechrep. ucsd. edu/Dienst/UI/2. 0/Describe/ncstrl. ucsd_cse/CS2009-0948

[25] Coburn J, Adrian M, Akel A, et al. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories [C] //Proc of the 16th Int Conf on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011). New York: ACM, 2011: 105-118

[26] Kang Yangwook, Yang Jingpei, Ethan L. Object-based SCM: An efficient interface for storage class memories [C] // Proc of the 2011 IEEE 27th Symp on Mass Storage Systems and Technologies (MSST 2011). Piscataway, NJ: IEEE, 2011

[27] Doshi K, Giles E, Varman P. Atomic persistence for SCM with a non-intrusive backend controller [C] //Proc of the 22nd IEEE Symp on High Performance Computer Architecture (HPCA 2016). Piscataway, NJ: IEEE, 2016: 12-16

[28] Kolli A, Pelley S, Said A, et al. High-performance transactions for persistent memories [C] Proc of the 21st Int Conf on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2016). New York: ACM, 2016: 399-411

[29] Joshi A, Nagarajan V, Viglas S, et al. ATOM: Atomic durability in non-volatile memory through hardware logging [C] //Proc of the 23rd IEEE Symp on High Performance Computer Architecture (HPCA 2017). Piscataway, NJ: IEEE, 2017: 4-8

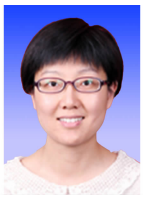
[30] Nguyen D, Pingali K. What scalable programs need from transactional memory [C] //Proc of the 22nd Int Conf on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2017). New York: ACM, 2017: 105-118

[31] Zuo Pengfei, Hua Yu. A write-friendly and cache-optimized hashing scheme for non-volatile memory systems [J]. IEEE Trans on Parallel & Distributed Systems, 2018, 29(5): 985-998

[32] Jin Peiquan, Wu Zhangling, Wang Xiaoliang, et al. A page-based storage framework for phase change memory [C] // Proc of the 33rd Int Conf on Massive Storage Systems and Technology (MSST 2017). Piscataway, NJ: IEEE, 2017: 1-12

[33] Yang Qing, Ren Jian. I-CASH: Intelligently coupled array of SSD and HDD [C] //Proc of IEEE Int Symp on High Performance Computer Architecture. Piscataway, NJ: IEEE, 2011: 278-289

[34] Chen Feng, Mesnier M P, Hahn S. A protected block device for persistent memory [C] //Proc of the 30th Symp on Mass Storage Systems and Technologies (MSST 2014). Piscataway, NJ: IEEE, 2014: 1-12



Niu Dejiao, born in 1978. PhD candidate. Her main research interests include network storage system and NVM.

He Qingjian, born in 1991. Master candidate. His main research interests include network storage system and NVM.

Cai Tao, born in 1976. PhD, associate professor. Member of CCF. His main research interests include network storage system and NVM.

Wang Jie, born in 1994. Master candidate. His main research interests include network storage system and NVM.

Zhan Yongzhao, born in 1962. PhD, professor. Member of CCF. His main research interests include multimediaand big data.

Liang Jun, born in 1976. PhD, professor. Member of CCF. His main research interest is big data.