VMFS: 一种持久性内存统一管理系统

张佳辰[1,2],胡泽瑞[1,2],赵 盛[1,2],施文杰[1,2],王 刚[1,2],刘晓光[1,2]
(1. 南开大学计算机学院,天津300350;2. 天津市网络与数据安全技术重点实验室,天津300350)

摘 要: 针对当前持久性内存(PM,Persistent Memory)资源管理方案无法兼顾持久化特性和可字节寻址特性的问题,提出了一种融合Linux系统内核虚拟内存系统和文件系统的持久性内存统一管理系统VMFS(Virtual Memory File System).VMFS中的单个PM分区可同时提供内存分配和文件存储服务,并利用内外存统一管理的特性可实现内存到文件的重映射机制,避免了不必要的数据拷贝,提升了文件读写性能,且维持了原生编程接口.实验结果表明,对比内外存分别使用PM的方案,VMFS有效提升了文件读写性能.在两种实际工作负载下,VMFS相对于使用DRAM(Dynamic Random Access Memory)和PM分别作为内存和存储的方案具有成本优势和一定程度的持久化性能提升.

关键词: 持久性内存; 非易失内存; 文件系统; 虚拟内存; 数据存储; 操作系统
中图分类号: TP316 **文献标识码:** A **文章编号:** 0372-2112(2021)12-2299-08
电子学报URL: http://www.ejournal.org.cn **DOI:** 10.12263/DZXB.20201239

VMFS: A Unified Persistent Memory Management System

ZHANG Jia-chen[1,2], HU Ze-rui[1,2], ZHAO Sheng[1,2], SHI Wen-jie[1,2], WANG Gang[1,2], LIU Xiao-guang[1,2]
(1. *College of Computer Science, Nankai University, Tianjin* 300350, *China*;
2. *Tianjin Key Laboratory of Network and Data Security Technology, Tianjin* 300350, *China*)

Abstract: Current resources management schemes of persistent memory cannot take advantage of data persistence and byte-addressability of PM(Persistent Memory) at the same time. We propose VMFS(Virtual Memory File System), a unified management system based on the virtual memory subsystem and a file system of OS kernel. VMFS provides memory allocation and file storage services for applications using a single PM partition. Based on the unified management of virtual memory and file storage, VMFS supports the data re-mapping between memory and files. As the re-mapping mechanism reduces the number of data copying times, file read and write latency performance is improved. The test results show that compared with the system using PM as memory and storage sperately, the proposed method can accelerate file reading and writing while the native system call interface was maintained. Under two realistic workloads, VMFS also shows cost and performance advantages over the scheme using DRAM(Dynamic Random Access Memory) as memory and PM as storage.

Key words: persistent memory; non-volatile memory; file system; virtual memory; data storage; operating system

1 引言

随着PCM[1]、ReRAM[2]、STT-MRAM[3]和3D-XPoint[4]等新型非易失存储器技术(NVM, Non-Volatile Memory)的发展,NVM和DRAM(Dynamic Random Access Memory)之间的速度越来越接近[5],基于NVM的内存设备也陆续在市场中[6]出现,这种设备被称为持久性内存(PM, Persistent Memory). 由于PM基于NVM介质,并直接连接到内存总线,因此兼具内存的可字节寻址特性和外存的数据持久性特性.

不同于固态硬盘(SSD, Solid State Drive)主要针对外存储性能的改进[7],PM的出现还给操作系统的存储层次抽象带来了新的挑战[8]. 近年来的相关工作,大多是针对PM单纯作为内存或单纯作为外存开展研究的[5, 9~20],解决面临的相应问题. SoupFS[10]、NOVA[11]和Ext4-DAX[21]都是Linux内核中实现的PM感知文件系统,它们考虑到PM细粒度持久化的特点,利用了不同方法来解决崩溃一致性问题,并优化了文件操作的性能. SplitFS[12]、ZoFS[13]和FLEX[5]是实现于用户态的PM感知文件系统,它们都减少了内核态和用户态之间切换. PM感知文件系统需要应用程序主动利用文件映射

收稿日期:2020-11-02;修回日期:2021-05-06;责任编辑:覃怀银
基金项目:国家自然科学基金项目(No.U1833114,No.61872201);天津市科技发展计划(No.18ZXZNGX00140,No.18ZXZNGX00200,No.20JCZDJC00610)

的方式操作文件,并且由于PM分区被格式化为文件系统,应用程序无法再从该分区分配内存使用.另一方面,一些工作在现有内存管理系统的基础上针对PM加入对数据持久化的支持.Makalu[14]和LSNVMM[15]优化了PM内存空间分配,降低了分配延迟,并将降低碎片率作为研究目标.NV-Heaps[16]、Mnemosyne[17]、Hea-po[18]、SoftWrAP[19]和PMDK[20]等则提出了新的内存分配和持久化库.这些PM内存管理系统的接口大多与现存的内存分配接口、文件操作接口不兼容,增加了现有应用程序或基础软件库的移植和维护难度.

为了兼顾PM的内外存特性,CHEN等提出了在PM设备上内存与文件系统统一管理系统UnistorFS[22],采用Linux内核的内存管理机制实现了一种基于内存的文件系统(memory-based file system),消除了页缓存(page cache)所带来的数据搬移,一定程度上提升了文件读写性能.

综上所述,操作系统领域现有PM虚拟化的相关工作大多从PM用作内存或PM用作存储两个方向着手,解决各自所面临的问题,如图1所示.

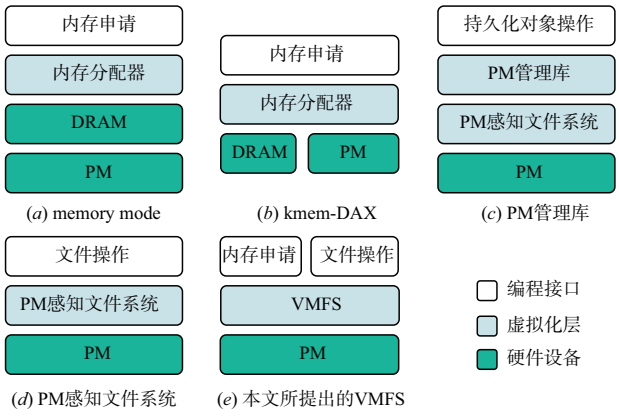


图1 操作系统PM进行管理的几种形式对比

但是,以PM作内存会牺牲其持久性或是需要应用适配新的编程接口,以PM作存储又会导致PM设备无法用于进程的内存分配.本文提出VMFS系统,尝试解决这一问题.VMFS同时支持原生的文件操作接口和内存分配接口,并且利用内存和文件系统统一管理的优势,通过减少数据拷贝来提升文件读写性能.相较于UnistorFS,VMFS不仅消除了系统缓存操作(DAX特性),还减少了用户进程与文件间的数据搬移(重映射机制).这些设计策略在大数据场景下可降低内存资源占用,还可优化流式处理应用的性能.

本文设计了VMFS系统,支持以单个PM分区为进程同时提供文件操作和内存分配的服务;利用内存和文件空间统一管理的优势,减少数据拷贝来优化文件读写性能;并对其进行了微基准测试和大数据应用负载性能测试.

2 总体设计

VMFS基于操作系统的PM感知文件系统和虚拟内存系统进行设计,整体架构如图2所示.从图中可以看到,VMFS同时对上层应用提供内存申请和文件操作接口,在系统内部以不同文件类型分别管理内存和文件,并且与PM的DAX驱动以及内存缺页处理模块进行交互来实现内存和文件两种方式的访问.

2.1 文件组织

VMFS将它所管理的文件分为普通文件(DAXFILE)和内存文件(DAXMEM).如图2所示,VMFS中DAX-FILE文件同其他PM感知文件系统所管理的文件类似,支持所有POSIX文件操作.而DAXMEM文件较为特殊,它不支持读写操作,只支持被映射到进程的虚拟内存地址空间,DAXMEM文件所管理的文件块被用户进程用作易失内存使用.DAXMEM文件块和DAXFILE文件块的分配和释放都依赖Ext4-DAX文件系统的块分配器.

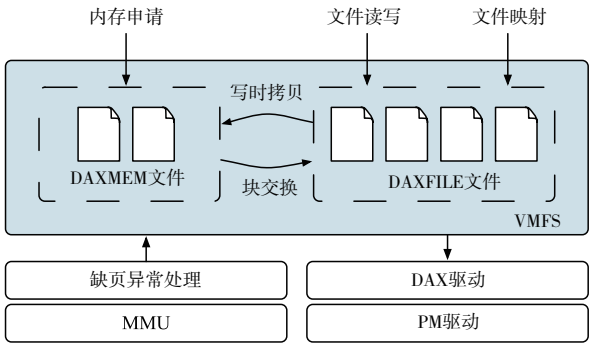


图2 VMFS系统整体架构

2.2 内存分配

应用程序可通过调用用户态内存分配器或者直接发起系统调用两种方式向内核申请内存,这两种方式最终都会在内核中发起对内存页的匿名映射.VMFS将对匿名内存的映射修改为对PM设备数据块文件的映射,以实现内外存空间实际分配方式的统一.

VMFS会在进程启动时为其创建一个DAXMEM文件,进程对内存的申请和释放会导致DAXMEM对文件系统块的申请和释放,所占用的文件块数也随之增加或者减小.在用户进程通过mmap系统调用发起内存申请时,VMFS会从进程对应的DAXMEM文件末端进行扩展,并将新扩展的部分映射到用户地址空间.另一方面,系统提供对DAXFILE文件的支持,这意味着应用仍然可以在VMFS所挂载目录中进行文件创建、删除和读写等操作.

这种设计实现了PM分别用作内外存的两种模式在单个PM分区中的共存.

2.3 内存分配优化

DAXMEM文件预扩展 DAXMEM文件是以追加的方式来服务内存分配的,由于用于文件扩展的开销

较大,若每次VMFS从DAXMEM文件申请内存时都对它进行扩展,会对内存分配的整体开销影响较大. 为了降低VMFS分配内存的延迟,我们采用了一种将文件扩展提前的预分配策略. 其思路是在进程向VMFS申请内存时,以次数更少的、较大粒度的扩展代替每次分配时在线、按需的扩展.

异步块回收　VMFS在多个流程中利用Ext4-DAX的文件打洞操作实现了DAXMEM文件块的回收,比如内存释放时、读写重映射时或者fork导致的写时拷贝时. 由于文件打洞涉及文件元数据的修改,频繁的调用会对VMFS的整体性能造成损失. 因此,在VMFS中,我们采用了一种异步块回收的策略:利用Linux内核现有的workqueue这一延后执行机制,为VMFS创建了一个用于DAXMEM文件块回收的后台任务队列.

2.4　零拷贝文件读写

VMFS使用了一种称为"读写重映射"的机制,以消除不必要的拷贝. 在应用发起read族系统调用时,VMFS将PM上文件块到用户缓冲区的拷贝操作替换为从用户缓冲区的虚拟地址到目标文件块物理地址的重映射操作. 在应用发起write族系统调用时,由于用户缓冲区所对应的PM块被VMFS的DAXMEM文件管理,我们可以在逻辑上将DAXMEM中相应的块交换到目标文件中. 这两种重映射操作都消除了原来所必需的拷贝操作,实现了文件读写的"零拷贝". 但这种"零拷贝"只减少了非必要的数据拷贝,一些必要的拷贝只是通过写时拷贝机制被延后了:被重映射的目标缓冲区被设置成只读权限,当这些缓冲区被修改时,写时拷贝过程将被触发.

2.5　编程语义的维持

文件的写时拷贝　为了达到VMFS维持标准系统调用的编程接口和语义的设计目标,我们需要在必要的时候进行写时拷贝. 这是因为,将缓冲区对应的虚拟地址重新映射到文件对应的文件块物理地址时,相当于这些PM物理块同时被文件和进程虚拟内存两者所共享,也就是说,相同的存储区域(数据)被以两种方式暴露给应用程序.

由于文件所隐含的持久化语义,文件数据只能通过文件共享映射或者write族调用被修改,而不通过"零拷贝"优化产生的重映射缓冲区被修改. 因此,VMFS应对读写导致的重映射缓冲区加入只读等权限,来保证文件读写系统调用的原始语义. 与此同时,重映射缓冲区所共享的文件区域被正常修改时,也需要在文件被写入前进行特殊处理保证不被改变. 因此,当重映射导致的共享区域被通过缓冲区虚拟地址进行修改或者被通过文件系统write族调用修改时,VMFS会进行写时拷贝和二次映射,将原本共享的文件块分为两份,来保证数据的完整性.

内存的写时拷贝　进程内存的写时拷贝机制被引入操作系统的进程创建机制. Fork系统调用在创建进程时,子进程和父进程会以只读的方式共享已分配内存,当任何一个进程试图修改共享内存时,内核会在相关区域被修改前先进行拷贝,来打破这一区域在多个进程之间的共享,以保证进程间各自内存的私有性. 类似地,在VMFS中,进程的内存由DAXMEM文件所管理,因此在进程被fork时,类似于原本的内存拷贝机制,VMFS也需要进行DAXMEM的文件拷贝和对新文件的映射.

2.6　读写流程示例

图3展示了VMFS结合零拷贝读写和写时拷贝来维持编程语义的示意图,每个子图中,从上到下分别表示一个进程进行读写操作时的目标DAXFILE文件、进程虚拟地址空间和VMFS中服务该进程内存分配的DAXMEM文件.

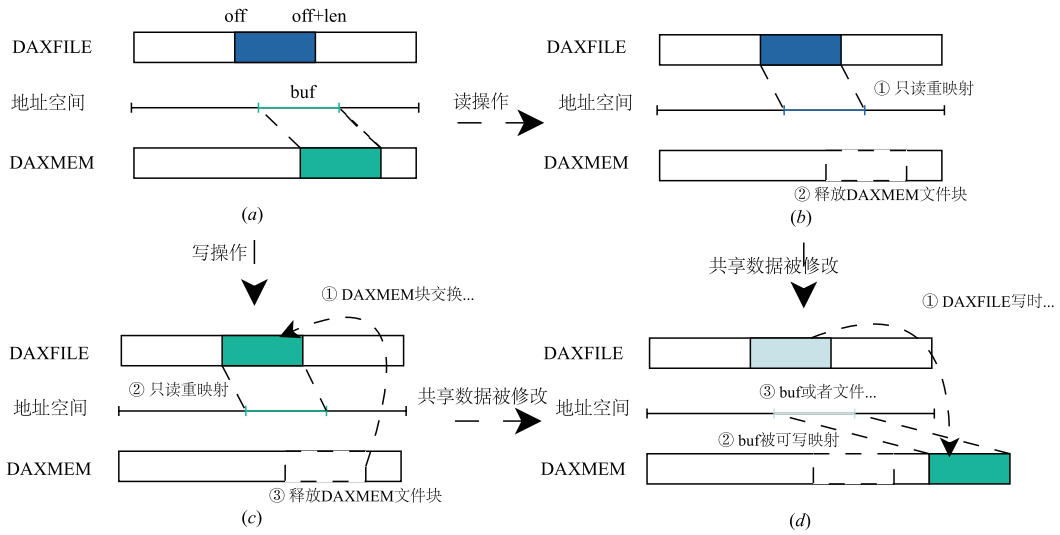


图3　VMFS进行零拷贝读写和写时拷贝的图例

图3(*a*)展示了VMFS一个操作例子的最初状态,进程地址空间buf区域的内存已被VMFS从DAXMEM文件中分配完成,并且进程未来将使用buf对VMFS中一个DAXFILE文件的[off, off+len)区域进行读写. 这里简单假设buf长度也为len.

图3(*a*)到图3(*b*)展示了零拷贝读的流程. 当进程调用read族系统调用从DAXFILE读时,系统并不进行拷贝,而是先以只读的权限将buf地址重映射到DAXFILE的[off, off+len)区域(①),然后将buf之前所映射的DAXMEM文件块释放(②).

图3(*a*)到图3(*c*)展示了零拷贝写的流程. 当进程调用write族系统调用从buf向DAXFILE的[off, off+len)区域写时,VMFS将先将buf对应的DAXMEM文件块和DAXFILE目标文件块进行交换(①),随后将buf地址重映射到被交换后的DAXFILE块(②),最后将buf之前所映射的DAXMEM文件块释放(③).

图3(*c*)到图3(*d*)和图3(*b*)到图3(*d*)分别展示了被只读保护的buf或其对应的DAXFILE区域被改写时将会进行的写时拷贝操作. 首先,这个受保护的DAXFILE区域会被拷贝到DAXMEM文件的末尾(①),之后buf地址被重新映射到新的DAXMEM区域(②),这样,文件块的共享状态便被拷贝所打破. 最后对buf或者DAXFILE的修改可以被正常进行(③).

3 实验和分析

3.1 实验设置

本文对VMFS进行性能测试均在虚拟机中进行,虚拟PM设备是对Intel的Optane DC系列的物理PM设备的直接映射. 在3.2节到3.4节的实验中,每个虚拟PM设备创建于物理地址交错映射的4条物理PM设备上. 在3.5节的实验中,虚拟PM设备创建于单条物理PM设备上. 实验平台详细配置如表1所示. 我们分别测试了VMFS的读写性能、内存分配性能和文件压缩/Kafka消息队列这两种实际应用负载的性能.

表1 实验平台配置

处理器型号	Intel Xeon Gold 5220 CPU @ 2.20GHz
处理器核数	18 ×2
DRAM容量	128 GB
PM设备	Intel Optane DC Persistent Memory 128GB ×4
宿主机OS	CentOS 7.0, kernel version 4.16.0
客户机OS	CentOS 7.0, kernel version 5.4.0
虚拟机管理器	QEMU version 4.1.0
文件系统	Ext4-DAX 及 VMFS

3.2 读写重映射性能

首先,本文将对比VMFS和Ext4-DAX的文件读写延迟. 测试读性能时,会分配某一固定单元大小的缓冲区,之后VMFS或Ext4-DAX上的文件将以固定单元大小被读入缓冲区. 测试写时,我们会首先分配一个较大的缓冲区并在其中填入要写到文件的数据,之后将这个缓冲区以某一固定单元大小写入VMFS或者Ext4-DAX上的文件. 对文件写性能的测试又分为覆盖写和追加写,两者都是从文件起始处开始顺序写,区别是覆盖写的目标是一个事先创建的、足够大的文件. 实验结果如图4所示.

图4(*a*)展示了两种系统的文件读性能,可以发现由于重映射机制,VMFS在读入时避免了内存拷贝,因此延迟并不像Ext4-DAX一样随着读单元的增加而增加. 由于避免了对读入后缓冲区的实际读写操作,该测试结果也说明了VMFS所能达到的最好情况.

图4(*b*)和图4(*c*)分别展示了文件覆盖写和追加写的性能. 首先可以发现,与读性能测试的结果类似,

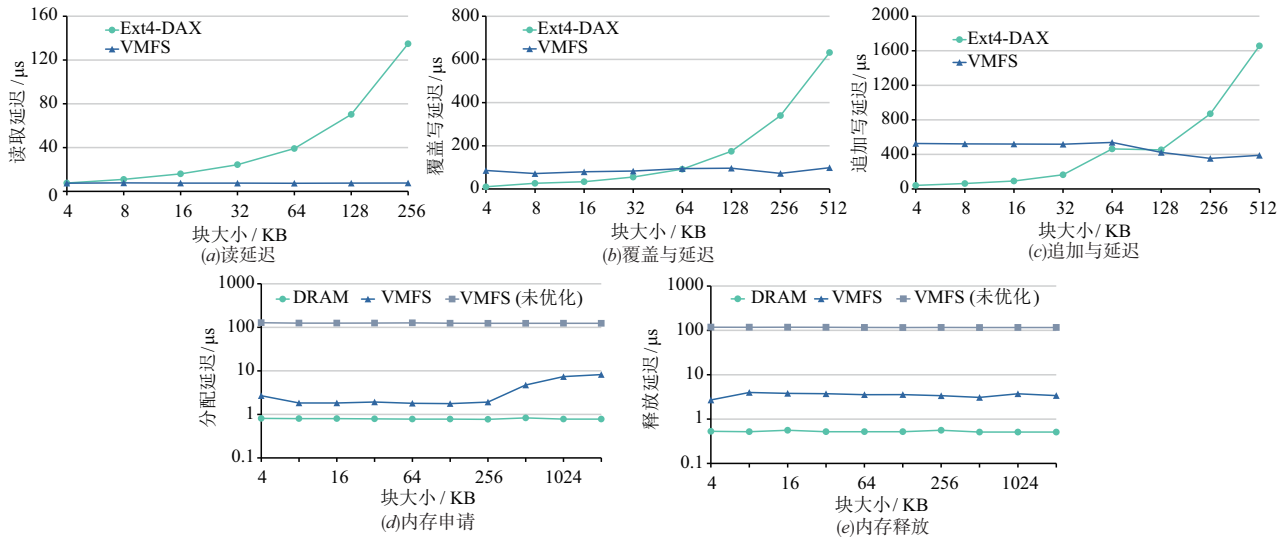


图4 VMFS读写延迟与内存申请性能

VMFS由于其重映射机制,以文件块交换代替了内存拷贝,所以在两种写负载下,延迟都没有随块大小的增加而增加,而且远好于Ext4-DAX的性能. 这两组写性能测试并未触发写时拷贝,因此也体现了VMFS相对于普通PM感知文件系统的最好情况.

3.3 内存分配性能

本小节中,我们对申请DRAM内存、从VMFS申请PM内存的以及从没有进行文件预扩展优化过的VMFS申请PM内存三种情况的内存分配、释放性能进行测试. 测试时,我们在这三种情况下分别以mmap和munmap对相应大小的内存连续地进行申请和释放操作100次,并将延迟取平均值作为测试结果.

图4(*d*)和图4(*e*)表明,虽然VMFS统一了文件操作和内存分配的接口,但是VMFS并未达到原生内存分配器的分配、释放效率:VMFS的未优化版本相对DRAM内存分配和释放性能差100倍以上;经过预分配优化后的版本将内存分配的差距减小到2~3倍左右,内存释放的差距约为6倍.

导致这种性能差异的原因是多方面的. 首先,VMFS是对PM进行分配而非DRAM,存在硬件上的性能差距;其次,由于VMFS内存分配是基于文件块的分配,由于设计目标并非为针对内存,因此不能达到最佳的分配性能;最后,由于读写重映射等功能的需求,相对传统内存管理,VMFS利用了DAXMEM这类文件进行内存组织,因此引入了额外的元数据分配、更新和释放的开销.

虽然基于VMFS的进程内存分配性能与DRAM间存在一定差距,但在内存分配、释放频率不高的应用中,VMFS依然会体现其节省DRAM内存和降低读写延迟的优势. 受限于静态设定的预分配块容量,在频繁申请内存的场景中,由于块分配所引入的开销仍不容忽视. 后续工作中,可将实际的块分配操作延后到初次访问该段内存时,使得这部分延迟在内存申请阶段被隐藏. 另一方面,可以设计针对VMFS的用户态内存分配器以及适合PM的文件块分配器,从而提高VMFS内存分配的效率.

3.4 实际工作负载——文件压缩

我们还测试了文件压缩工作负载在VMFS、P+P、D+P三种系统中的表现. 其中,"P+P"表示"内存"和"外存"都使用PM设备,而"D+P"表示"内存"和"外存"分别使用DRAM和PM设备,两种模式都采用传统的内存、外存管理. 测试时,我们分别从不同压缩比的文件中取样64 MB,并用LZ4压缩算法[23]进行压缩或解压缩.

表2描述了我们选用的三种不同压缩比文件的详情. 其中压缩比的值等于压缩前的数据大小除以压缩后的数据大小. "全零文件"指文件中的全部数据为零,压缩比最高,达252.1;"源文件"由Linux 5.4版本的内核源码打包而来,内容以代码和文本为主,压缩比较高;"压缩文件"是在"源文件"的基础上以gzip进行压缩后的压缩包文件,由于已经被压缩过,这种文件最不适合压缩.

表2　本小节测试文件压缩所采用的文件类型

负载	压缩比	压缩前大小	压缩后大小
全零文件	252.1	64 MB	0.254 MB
源文件	3.104	64 MB	20.6 MB
压缩文件	0.999	64 MB	63.9 MB

我们分别测试了三种文件在前述三种系统中文件压缩任务的执行时间. 执行文件压缩任务所需的时间主要包括三部分:读取待处理文件、压缩或解压缩数据和将处理后的数据写入一个新文件. 由图5可见,与硬件条件相同的P+P系统相比,在三种文件负载下VMFS的压缩耗时分别减少了66.7%、44.5%和62.6%,解压耗时分别减少了52.8%、34.4%和50.6%,这是由于VMFS系统的读写重映射机制,数据压缩任务的读写文件部分时间已缩短至几乎可忽略不计. 相比于硬件性能更优的D+P系统(内存使用DRAM),VMFS的压缩性能也有一定的优势,压缩耗时分别减少了22.6%、12.1%和12.6%. VMFS的数据解压缩部分的时间较D+P系统存在明显差距,这是由于在VMFS中,解压缩是直接在PM上而非DRAM上进行. 因此虽然VMFS仍然大幅度减少了文件读写时间,但已不足以抵消解压缩时间上的劣势,总体解压缩性能略差于D+P系统. 但这种差距在18%以内,而且对于计算量较小的全零文件,VMFS的解压时间较之D+P系统仍有细微优势.

本小节的实验表明,VMFS由于使用PM代替DRAM作为应用内存,节省了DRAM资源的消耗,可将珍贵的DRAM资源用于其他更需要的应用. 这当然可能带来性能损失,但是由于VMFS可以利用其内存和文件存储统一管理的优势,显著减少文件读写的时间,在文件压缩这种流式负载下,性能与使用DRAM的方案相比差距不大,甚至在某些情况下更优.

3.5 实际工作负载——Kafka消息队列

在大数据场景下,也有很多处理流式任务的框架,例如Kafka[24]、MapReduce[25]、Spark[26]等. 这类流处理框架有共同的特点:在工作时需要频繁分配内存进行数据的传输以及计算,同时也需要大量的外存空间来进行数据的持久化存储,这恰是VMFS的优势所在. Kafka是一种流式处理的大数据消息队列框架,图6显示了其基本架构.

延迟是衡量Kafka性能的一个很重要的指标,其中Broker的处理效率对延迟有很大影响[27]. 因此,我们评估采用VMFS对Broker处理延迟的影响,也就是从消息到达Broker直到持久化完毕这个过程的延迟.

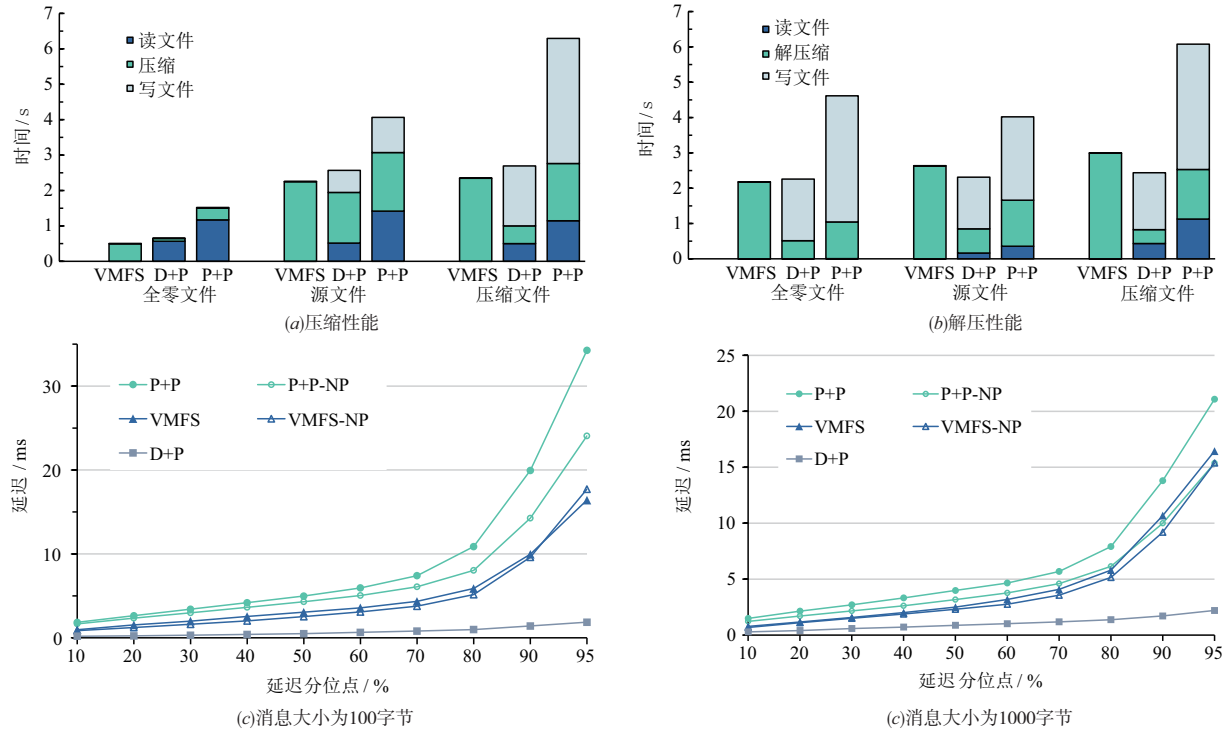


图5 实际负载测试:文件压缩性能与Kafka消息延迟

测试时,我们使用了Kafka自带的kafka-producer-perf-test基准测试.该测试会启动一个Kafka生产者,该生产者会根据事先设定好的消息数量以及其他参数,不断地调用Send方法,向Kafka Broker集群一个接一个地发送消息.如上文所述,我们统计从消息到达Broker直到持久化完毕这个过程的延迟.与上一节的实验一样,我们还是对比了VMFS与P+P、D+P两种方案.实验中分别测试了消息长度为100字节和10000字节(对应短消息场景或长消息场景)时,单Broker节点(仍架设在虚拟机上)上三种方案的延迟.实验中消息总数据量保持一致,因此消息大小为100字节时,消息数为10000时的100倍.

图5(*c*)和图5(*d*)分别显示了消息大小为100字节和10000字节时,三种方案的延迟(后缀-NP表示无持久化,对应实际应用中为去掉或延迟持久化的方案),横坐标表示百分位.可以看到,当消息大小为100字节时,较之P+P方案,VMFS的延迟在所有分位点均要少40%到50%,当消息大小为10000字节时要少20%到50%,这清楚地体现出我们的设计中减少数据拷贝、减少频繁内存分配对性能优化的效果.还可以看到,未持久化方案的延迟显著低于持久化方案,这是很自然的.但同时也注意到,无论消息大小是100字节还是10000字节,VMFS-NP与VMFS的差距很小,P+P-NP和P+P的差距则比较明显.这是因为VMFS中的持久化只是块交换,相比VMFS-NP只是多了一些文件系统元数据操作的开销,并未有额外的数据拷贝开销.在消息大小为100字节或10000字节时,D+P方案的延迟均只有P+P和VMFS的10%到25%.这是因为Kafka在内存中对消息进行了解析、校验、计算offset等一系列处理,频繁的内存访问令DRAM相比PM访问延迟低的优势凸显.但VMFS方案能够节省大量的DRAM资源:通常Kafka单节点配置6~8GB的JVM堆内存即可,剩下的绝大部分机器内存用作page cache来缓存数据,总的来说一台机器上80%~90%的内存都要用于Kafka.对于一个DRAM容量为256GB的节点,Kafka要占用至少200GB.因此,在云计算场景中资源紧张的情况下,VMFS大幅度节省内存的特性是有价值的.与此同时,未来通过优化VMFS的内存分配机制,以及异步处理等机制可望显著缩短VMFS与D+P的性能差距.

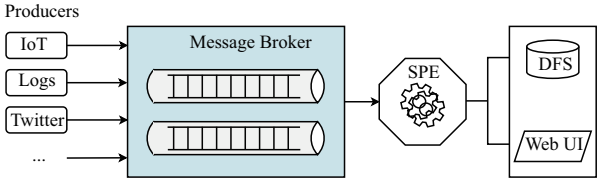


图6 Kafka流处理架构[31]

4 总结

操作系统领域中现有的与PM虚拟化相关的工作大多仅将PM作内存使用或仅将PM作存储使用.但是,以PM作内存会牺牲其持久性或是需要应用适配新

的编程接口,以PM作存储又会导致PM设备无法用于进程的内存分配. 为了解决这一问题,本文设计了一种将内存和文件统一管理的系统VMFS. VMFS实现了PM同时作内存和存储时,两种使用方式共享同一PM空间,且利用了统一管理的优势,减少了文件读写操作的数据拷贝. 本文通过文件读写、内存分配等微基准测试验证VMFS这些设计带来的性能优势. 由于以PM服务内存分配,VMFS可以被用于DRAM资源紧张的场景来节省DRAM资源占用. 对于大数据场景中常见的流式处理应用,还可以利用VMFS的零拷贝读写优化来提升整体性能. 本文以文件压缩和Kafka消息队列这两个大数据场景下的常见应用为例,验证了VMFS对实际应用的性能优化效果.

参考文献

[1] RAOUX S, BURR G W, BREITWISCH M J, et al. Phase-change random access memory: A scalable technology[J]. IBM Journal of Research and Development, 2008, 52(4.5): 465 – 479.

[2] XU C, NIU D, MURALIMANOHAR N, et al. Overcoming the challenges of crossbar resistive memory architectures [A]. PATTERSON D A. Proceedings of the 21st International Symposium on High Performance Computer Architecture[C]. Washington, United States: IEEE, 2015. 476 – 488.

[3] HUAI Y, et al. Spin-transfer torque MRAM(STT-MRAM): challenges and prospects[J]. AAPPS Bulletin, 2008, 18(6): 33 – 40.

[4] HANDY J. Understanding the Intel/Micron 3D XPoint memory[A]. Storage Developer Conference[C]. Santa Clara, United States: SNIA, 2015.

[5] XU J, KIM J, MEMARIPOUR A, et al. Finding and fixing performance pathologies in persistent memory software stacks[A]. BAHAR I. The Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems[C]. New York, United States: ACM, 2019. 427 – 439.

[6] SPELMAN L. Reimagining the Data Center Memory and Storage Hierarchy [EB/OL]. https://newsroom.intel.com/editorials/re-architecting-data-center-memory-storage-hierarchy,2021-05-05.

[7] BJØRLING M. Operating System Support for High-Performance Solid State Drives[D]. Copenhagen, Denmark: IT University of Copenhagen, 2016.

[8] DULLOOR S R, KUMAR S, KESHAVAMURTHY A, et al. System software for persistent memory[A]. BULTERMANN D. European Conference on Computer Systems[C]. New York, United States: ACM, 2014. 1 – 15.

[9] BEAUCHAMP B, et al. NVM Programming Model[EB/OL]. https://www.snia.org/sites/default/files/technical_work/final/NVMProgrammingModel_v1.2.pdf, 2021-05-05.

[10] DONG M, CHEN H. Soft updates made simple and fast on non-volatile memory[A]. SILVA DD. Proceedings of the 2017 Annual Technical Conference[C]. Santa Clara, United States: USENIX Association, 2017. 719 – 731.

[11] XU J, SWANSON S. NOVA: a log-structured file system for hybrid volatile/non-volatile main memories[A]. BROWN AD. Proceedings of the 14th Conference on File and Storage Technologies[C]. Santa Clara, United States: USENIX Association, 2016. 323 – 338.

[12] KADEKODI R, LEE S K, KASHYAP S, et al. SplitFS: reducing software overhead in file systems for persistent memory[A]. BRECHT T. Proceedings of the 27th Symposium on Operating Systems Principles[C]. New York, United States: ACM, 2019. 494 – 508.

[13] DONG M, BU H, YI J, et al. Performance and protection in the ZoFS user-space NVM file system[A]. BRECHT T. Proceedings of the 27th Symposium on Operating Systems Principles[C]. New York, United States: ACM, 2019. 478 – 493.

[14] BHANDARI K, CHAKRABARTI D R, BOEHM H J. Makalu: fast recoverable allocation of non-volatile memory [A]. VISSER E. Proceedings of the 2016 SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications[C]. New York, United States: ACM, 2016. 677 – 694.

[15] HU Q, REN J, BADAM A, et al. Log-structured non-volatile main memory[A]. SILVA DD. Proceedings of the 2017 Annual Technical Conference[C]. Santa Clara, United States: USENIX Association, 2017. 703 – 717.

[16] COBURN J, CAULFIELD A M, AKEL A, et al. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories[A]. GUPTA R. Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems[C]. New York, United States: ACM, 2011. 105 – 118.

[17] VOLOS H, TACK A J, SWIFT M M. Mnemosyne: lightweight persistent memory[A]. GUPTA R. Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems[C]. New York, United States: ACM, 2011. 91 – 104.

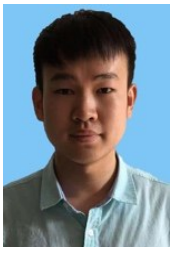
[18] HWANG T, JUNG J, WON Y. HEAPO: Heap-based persistent object store[J]. ACM Transactions on Storage,

2015, 11(1): 1 – 21.
[19] GILES E R, DOSHI K, VARMAN P. SoftWrAP: a lightweight framework for transactional support of storage class memory[A]. COLEMAN S. Proceedings of the 31st Symposium on Mass Storage Systems and Technologies [C]. Santa Clara, United States: IEEE, 2015. 1 – 14.
[20] RUDOFF A, et al. Perisistent Memory Development Kit [EB/OL]. https://pmem.io, 2021-05-05.
[21] LINUX KERNEL ORGANIZATION, INC. DAX – Direct Access for Files[EB/OL]. https://www.kernel.org/doc/Documentation/filesystems/dax.txt,2021-05-05.
[22] CHEN S H, CHEN T Y, CHANG Y H, et al. UnistorFS: a union storage file system design for resource sharing between memory and storage on persistent RAM-based systems[J]. ACM Transactions on Storage, 2018, 14(1): 1 – 22.
[23] COLLET Y. LZ4: Extremely Fast Compression Algorithm [EB/OL]. https://github.com/lz4/lz4, 2021-05-05.
[24] APACHE. Kafka 2.8 Documentation[EB/OL]. http://kafka.apache.org/, 2021-05-05.
[25] APACHE. MapReduce Tutorial[EB/OL]. https://hadoop.apache. org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial. html, 2021-05-05.
[26] APACHE. Spark: Lightning-Fast Unified Analytics Engine [EB/OL]. http://spark.apache.org, 2021-05-05.
[27] HASEED JM, XIAOYI L, DHABALESWAR K. P, et al. Cutting the tail: designing high performance message brokers to reduce tail latencies in stream processing[A]. NIKOLOPOULOS D. Proceedings of the 20th International Conference on Cluster Computing[C]. Belfast, United Kingdom: IEEE, 2018. 223 – 233.

作者简介

张佳辰 男,1994年生于河北唐山.南开大学计算机学院硕士研究生.主要研究方向为存储虚拟化和持久性内存.
E-mail:jczhang@nbjl.nankai.edu.cn

施文杰 男,1994年生于山西大同.南开大学计算机学院硕士研究生.主要研究方向为大数据技术.
E-mail:shiwj@nbjl.nankai.edu.cn



胡泽瑞 男,1996年生于安徽池州.南开大学计算机学院硕士研究生.主要研究方向为大数据存储和分布式系统.
E-mail:huzr@nbjl.nankai.edu.cn

王 刚(通信作者) 男,1974年生于天津.南开大学计算机学院教授,博士生导师.主要研究方向为海量信息存储和并行计算.
E-mail:wgzwp@nbjl.nankai.edu.cn

赵 盛 男,1998年生于安徽芜湖.南开大学计算机学院硕士研究生.主要研究方向为存储虚拟化和持久性内存.
E-mail:zhaos@nbjl.nankai.edu.cn

刘晓光 男,1974年生于河北安国.南开大学计算机学院教授,博士生导师.主要研究方向为搜索引擎和区块链系统.
E-mail:liuxg@nbjl.nankai.edu.cn