**Light NVM: The Linux Open-Channel**

**SSD Subsystem**

Matias Bjørling, *CNEX Labs, Inc. and IT University of Copenhagen;* Javier Gonzalez,

*CNEX Labs, Inc.;* Philippe Bonnet, *IT University of Copenhagen*

<https://www.usenix.org/conference/fast17/technical-sessions/presentation/bjorling>

This paper is included in the Proceedings of

the 15th USENIX Conference on

File and Storage Technologies (FAST ’17).

February 27–March 2, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-36-2

Open access to the Proceedings of

the 15th USENIX Conference on

File and Storage Technologies

is sponsored by USENIX.

海量存储课程

专业：信息安全 学号：2016272110001 姓名：Anikkone dalavong

尊敬的 老师，你好

Light NVM: The Linux Open-Channel SSD Subsystem

Matias Bjørling† Javier González† Philippe Bonnet\*

†CNEX Labs, Inc. \*IT University of Copenhagen

**Abstract**

随着固态驱动器（SSD）在数据中心和存储阵列中的普及，对可预测的延迟的需求不断增长。 提供块I / O的传统SSD无法满足此需求。 它们以不可预知的性能和不理想的资源利用率为代价提供高级别的抽象。 我们建议固态硬盘管理的权衡应该通过开放通道固态盘来处理，固态盘是一种新型的固态硬盘，可以让主机控制其内部。我们介绍了我们构建Linux Open-Channel SSD子系统LightNVM的经验。 我们引入了一个新的物理页面地址I / O接口，它显示了SSD的并行性和存储介质的特性。 LightNVM集成到传统的存储堆栈中，同时也使存储引擎能够利用新的I / O接口。 我们的实验结果表明，LightNVM具有适度的主机开销，它可以被调整来限制读取延迟变化，并且可以被定制以实现可预测的I / O延迟。

1 Introduction

固态硬盘（SSD）预计在未来几年成为二级存储的主要形式[18,19,31]。 尽管由于性能优越而取得了成功，但固态硬盘存在以下明显缺陷：登录日志[37,57]，大型尾部延迟[15,23]，不可预知的I / O延迟[12,28,30]和 资源利用不足[1,11]。 这些缺点不是由于硬件的限制：固态硬盘核心的非易失性存储器芯片以受限制的操作和有限的耐用性/可靠性为代价提供了可预测的高性能。 这就是在一个SSD内如何管理数十个非易失性存储器芯片，提供与磁盘相同的块I / O接口，这造成了这些缺点[5,52]。

市场上出现了一类新型固态硬盘，名为Open-Channel SSD。 它们是解决SSD缺点和管理与吞吐量，延迟，功耗和容量相关的折衷的极好平台。 事实上，开放通道固态硬盘揭露了其内部，使主机可以控制数据放置和物理I / O调度。使用open-channel SSD时，SSD管理的责任由主机和SSD共享。 一段时间以来，开放式通道SSD已经被一级云提供商所使用。 例如，百度使用开放式固态盘来精简键值存储的存储堆栈[55]。 另外，Fusion-IO [27]和Violin Memory [54]都实现了一个主机侧存储堆栈来管理NAND介质并提供一个块I / O接口。 但是，在所有这些情况下，将开放式通道SSD整合到存储基础架构中的做法已被限制在设计空间中的一个单一点，并进行固定的权衡。

管理SSD设计权衡可能允许用户重新配置他们的存储软件堆栈，以便针对期望块I / O接口（例如，关系数据库系统，文件系统）的应用程序或针对直接利用开放信道SSD的应用程序进行定制[55]。 这里有两个问题：（1）在open channel固态硬盘之上实现的块设备抽象应该提供高性能，（2）应该清楚地标识设计选择和权衡机会。 这些是我们在本文中讨论的问题。 请注意，演示特定于应用程序的SSD管理的优点超出了本文的范围。

我们描述了我们在Linux内核中构建LightNVM（Open Channel SSD子系统）的经验。 LightNVM是第一个用于开放式通道SSD和基于主机的SSD管理的开放通用子系统。 我们做了四个贡献。 首先描述开放式SSD管理的特点。 我们确定与揭露SSD内部相关的限制因素，讨论存储行业的相关权衡和经验教训。其次，我们介绍物理页面地址（PPA）I / O接口，这是开放式通道SSD的接口，它定义了分层地址空间以及控制和向量数据命令。

第三，我们介绍LightNVM，这是我们为开放式通道SSD管理而设计和实施的Linux子系统。 它提供了一个接口，可以实现特定于应用程序的抽象（称为目标）。 我们提供了一个基于主机的Flash翻译层，名为pblk，将开放式通道SSD作为传统的块I / O设备公开。

最后，我们将展示Light-NVM在第一代开放式SSD上的有效性。 我们的结果是揭示物理页面地址I / O接口的开放通道SSD的第一次测量。 我们将比较最先进的块I / O SSD，并评估运行合成，文件系统和基于数据库系统的工作负载时的性能开销。 我们的研究结果表明，LightNVM实现了高性能，可以调整以控制I / O延迟变化。

2 Open-Channel SSD Management

SSD由数十个存储芯片组成，通过所谓的通道与控制器并联。 随着开放通道固态硬盘，渠道和存储芯片暴露给主机。 主机负责及时使用SSD资源（I / O调度）和空间（数据放置）。 在本节中，我们将重点放在NAND闪存的开放式通道固态盘上，因为管理NAND今天既具有相关性又具有挑战性。 我们回顾了NAND闪存带来的限制，介绍了SSD管理面临的主要挑战，讨论了我们从系统早期采用者那里吸取的教训，并介绍了不同的开放式SSD架构。

2.1 NAND Flash Characteristics

NAND闪存依赖浮栅晶体管阵列（所谓的单元）来存储位。 晶体管尺寸的缩小使闪存容量增加。 SLC闪存每个单元存储一位数据。 MLC和TLC闪存分别存储每个单元2或3位，并且QLC闪存中每个单元有四位。 对于3D NAND而言，增加的容量不再是缩小单元尺寸，而是闪存阵列分层。

**Media Architecture.** NAND闪存提供读/写/擦除接口。 在NAND封装中，存储介质被组织成裸片，平面，块和页面的层次结构。 一个模块允许一次执行一个I / O命令。 一个物理封装内可能有一个或几个管芯。 一个平面允许类似的闪存命令在芯片内并行执行。

在每个平面内，NAND按块和页面组织。 每个平面包含相同数量的块，每个块包含相同数量的页面。 页面是读写的最小单位，而擦除单位是块。 每个页面被进一步分解成固定大小的扇区和一个附加的出站区域，例如，一个16KB的页面包含4个4KB的扇区加上经常用于ECC和用户特定数据的出界区域。

关于内部时序，NAND闪存在读取和写入/擦除延迟之间表现出一个数量级的差异。 读取通常需要几百微秒，而写入和擦除操作需要几个毫秒。 但是，如果直接在写或擦除操作之后安排读操作，读延迟会出现尖峰，从而导致延迟数量级的增加。

**写约束。** 有三个基本的编程约束适用于NAND [41]：（i）写入命令必须包含足够的数据来编程一个（或几个）全闪存页面，（ii）写入必须在一个块内顺序进行， 和（iii）在块内的页面可以被重写之前必须执行擦除。 编程/擦除（PE）周期的数量是有限的。 限制取决于闪光灯的类型：TLC / QLC闪光灯102，MLC 103或SLC 105。

对于不同类型的NAND闪存，必须考虑额外的限制。 例如，在多级单元存储器中，存储在同一单元中的位属于不同的写入页面，称为下/上页面。 在较低页面可被成功读取之前，必须写入较高页面。 下页面和上页面通常不是连续的，为了防止写邻居干扰，必须写入页面之间的任何页面[10]。 此外，NAND供应商可能会引入任何类型的特殊约束，这些约束没有公开披露。 这对于跨供应商，基于主机的SSD管理的设计来说是一个明显的挑战。

**Failure Modes.**  NAND Flash可能以各种方式失败 [7,40,42,49]：

* 位错误。 缩小单元大小的缺点是存储位时出现错误。 对于SLC来说，每KB的误码率是常见的，但对于MLC来说，这个速率增加了四到八倍。
* 读写干扰。 随着位的写入或读取，介质易于泄漏到附近的单元。 这导致了上面描述的一些写约束。
* 数据保留。 随着单元耗尽，数据保留能力下降。 随着时间的推移，数据必须被重写多次。
* 写/擦除错误。 在写入或擦除期间，由于块级别的不可恢复的错误，可能会发生故障。 在这种情况下，块应该被取消，并且已经写入的数据应该被重写到另一个块。
* 死亡。 存储的逻辑单元（即，NAND芯片上的管芯）由于缺陷而可能随时间停止运行。 在这种情况下，所有的数据都将丢失。

**2.2 Managing NAND**

管理NAND强加的限制是任何基于闪存的SSD的核心要求。 对于开放式SSD，这个责任是在主机上运行的软件组件（在我们的例子中是一个Linux设备驱动程序和在其上构建的层）和设备控制器之间共享的。 在本节中，我们介绍与NAND管理相关的两个关键挑战：写缓冲和错误处理。

**Write Buffering.** 当在主机侧（在Linux设备驱动器中）定义的扇区的大小小于NAND闪存页面大小（例如，在16KB闪存页的顶部上定义的4KB扇区大小）时，写入缓冲是必要的。 为了处理这种不匹配，传统的解决方案是使用缓存：缓存扇区写入，直到收集足够的数据以填充闪存页面。 如果在填充缓存之前数据必须被持久化（例如，由于应用程序刷新），则添加填充以填充闪存页面。 读取被定向到缓存，直到数据被保存到媒体。 如果缓存驻留在主机上，则两个优点是（1）写入全部由主机生成，从而避免了主机和设备之间的干扰，并且（2）写入在被命中时被确认。 缺点是在发生电源故障时，缓存内容可能会丢失。

写缓存也可以放在设备端。 主机可以将扇区写入设备，并让设备管理对介质的写入（当有足够的数据累积到闪存页面时），或者主机明确控制对介质的写入并保持设备的持久性。 使用前一种方法，设备控制器可能会在工作负载中引入不可预测性，因为它可能会发生干扰主机发布的写入。 采用后一种方法，主机可以完全访问设备端缓存。在NVMe中，这可以通过控制器内存缓冲区（CMB）来完成 [43]。 因此，主机可以将（i）设备侧高速缓存上的数据分段与（ii）通过显式清空命令写入媒体分离。 这种方法避免了控制器生成的写入，并使主机完全控制媒体操作。 这两种方法都要求设备固件具有电源失效技术，以在掉电情况下将写入缓冲器存储到介质上。 高速缓存的大小受到SSD上可用的功率电容器的限制。

**Error Handling.** 错误处理涉及读取，写入和擦除。 所有恢复扇区级数据的方法都已经用尽时，读取失败：ECC，阈值调整和可能的基于奇偶校验的保护机制（RAID / RAIN）[13,20]。

为了补偿误码，有必要引入纠错码（ECC），例如BCH [53]或LDPC [16]。 通常，ECC编码的单位是一个扇区，通常小于一个页面。 通常将ECC奇偶处理为与页面关联的元数据，并存储在页面的带外区域内。

可以为每个块估计误码率（BER）。 为了将BER维持在一个给定的阈值以下，一些供应商可以调节NAND门限电压[7,8]。 应该重写写入冷和热读写的块，其BER高于给定的阈值，[47]。 可能还需要执行读取清理，即时间表读取操作，其唯一目的是估计写冷和读冷的块的BER [9]。

鉴于手动阈值调整会导致在页面上执行多次读取，因此添加RAID技术可以更快地恢复数据，同时也可以使SSD从故障中恢复。

请注意，不同的工作负载可能需要不同的RAID配置。 通常，高读取工作负载需要较少的冗余，因为它们发出较少的PE周期。 这是基于主机的RAID实施的一个参数。 相反，对于高写入工作负载，RAID是可能由硬件加速（即，基于硬件的XOR引擎[14,48]）补偿的开销的来源。

在写入失败的情况下，由于过度充电或固有故障[51]，在块级恢复是必要的。 当写入失败时，块的一部分可能已经被写入，应该被读取以执行恢复。 早期的NAND闪存芯片允许在部分写入的块上进行读取，但是多级NAND [10]要求在读取数据之前写入一组页面（下/上），从而防止在一般情况下读取部分写入的块。 在这里，应该有足够的缓冲区空间来恢复部分写入的块的内容。

如果擦除发生故障，则不会重试或恢复。 该块被简单地标记为坏。

**2.3 Lessons Learned**

开放式通道SSD为SSD管理开辟了一个巨大的设计空间。 基于行业趋势和早期LightNVM采用者的反馈，这里对设计空间进行了一些限制。

1. 通过物理访问提供设备保修。 最终用户的保证在大批量市场中非常重要。 传统的固态硬盘往往需要三年或五年的运行。 在其生命周期中，必须有足够好的闪存介质来执行写入。 与旋转硬盘相反，NAND介质的使用寿命在很大程度上取决于对介质的写入次数。 因此，基于闪存的SSD通常有两种类型的保证：年保修和每日驱动器写入（DWPD）保修。 DWPD保证驱动器每天可以支持X驱动器写入。 为NAND闪存介质提供数千个PE周期，每天的写入次数通常限制在10次以内，在消费级驱动器中次数更少。

如果在主机上管理PE周期，则不能对明渠SSD进行保修。 实际上，SSD供应商无法评估设备是否合法有资格进行更换，或者由于过度使用而导致闪存完全无法使用。 为了提供保修，PE周期必须在设备上进行管理。 参见图1的插图。

1. **将媒体特征暴露给主机是低效的并且限制媒体抽象。** 传统SSD供应商使用 NAND供应商进行媒体表征，以使其嵌入式闪存转换层适应给定NAND芯片的特性。 这种内部NAND特性在IP下受到保护。 让应用程序和系统开发人员与特定的NAND芯片的内部细节，特别是阈值调整或ECC进行斗争既不可取也不可行。 这些必须在设备上进行管理。 这极大地简化了主机中的逻辑，并让开放式通道SSD供应商区分其控制器实施。
2. 写入缓冲应根据用例在主机或设备上进行处理。 如果主机处理写入缓冲，则设备上不需要DRAM，因为如果需要的话，维护保证和物理媒体信息所需的小型数据结构可以存储在设备SRAM或持久性媒体中。 功耗因此可以大大降低。 通过CMB管理器件上的写入缓冲区可以有效地支持小写操作，但需要额外的器件端逻辑以及功率电容器或类似功能来保证耐用性。 这两种选择应该可供开放式SSD厂商使用。
3. 与应用程序无关的磨损均衡是强制性的。 随着NAND的老化，访问时间变长。 实际上，电压阈值变得更宽，并且需要更多的时间来精细地调整适当的电压以读取或写入数据。 NAND规范通常报告典型的访问延迟和最大延迟。 为了确保延迟不会因访问的块的年龄而波动，

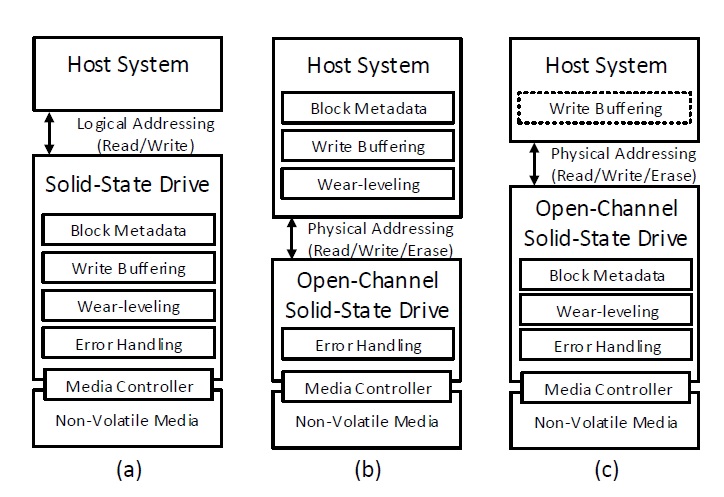


图1：（a）传统Block I / O SSD，（b）本文中考虑的开放通道SSD级别，以及（c）未来开放通道SSD的核心SSD管理模块。

即使引入了开销，也可以独立于应用程序负载形成平均磨损。

对于主机或控制器来说，必须能够以（i）隐藏坏块，（ii）通过在分配时考虑P / E循环计数来实现动态耗损均衡 一个块，并且可能（iii）通过将冷数据复制到热块来实现静态耗损均衡。 这样的决策应该基于设备上收集和维护的元数据：每块的P / E周期，每页读取的计数和坏块。 管理块元数据以及逻辑块和物理块地址之间的间接级别会导致严重的延迟开销，并可能产生可能干扰应用程序I / O的内部I / O（用于存储映射表或由于静态损耗均衡）。 这是磨损平衡的代价[6]。

**2.4 Architectures**

根据主机和SSD之间共享SSD管理的职责的方式，可以定义不同类型的开放式通道SSD。 图1比较了（a）传统块I / O SSD与（b）本文中考虑的开放通道SSD的类别，其中在主机上管理PE周期和写缓冲，以及（c）未来的开放通道SSD将提供 担保，从而支持设备上的PE循环管理和磨损平衡。 PPA I / O接口的定义和LightNVM的架构涵盖了所有类型的开放式通道SSD。

**3 Physical Page Address I/O Interface**

我们提出了一个基于分层地址空间的开放式通道SSD接口，物理页面地址（PPA）I / O接口。 它定义了管理命令来公开设备几何图形，让主机控制SSD管理，数据命令有效地存储和检索数据。 该接口独立于嵌入在开放式通道SSD上的非易失性媒体芯片的类型。

我们的接口是作为针对NVM Express 1.2.1规范[43]的供应商专用扩展来实现的，该规范定义了PCIe接合SSD的优化接口。

3.1 Address Space

我们依靠两个不变量来定义PPA地址空间：

1. SSD架构。 开放式通道SSD向主机提供一组通道，每个通道包含一组并行单元（PU），也称为LUN。 我们将PU定义为设备上的并行单元。 PU可以覆盖一个或多个物理模具，并且模具可以仅仅是一个PU的成员。 每个PU一次处理一个I / O请求。
2. 媒体架构。 无论媒体如何，每个PU上的存储空间都是量化的。 NAND闪存芯片被分解成块，页面（传输的最小单位）和扇区（ECC的最小单位）。 字节寻址存储器可以被组织为扇区的平坦空间。

控制器可以选择PU的物理表示。 通过这种方式，控制器可以在PU级别上公开反映底层存储介质性能的性能模型。 如果控制器为PU选择逻辑定义（例如，通过RAID访问的多个NAND管芯），则PU的性能模型必须基于存储介质特征和控制器功能（例如XOR引擎加速）来构建。 逻辑表示可能对于字节可寻址存储器是有利的，其中多个裸片被组合在一起以形成单个扇区。 在本文的其余部分，我们假设PU对应于单个物理NAND芯片。 有了这样的物理PU定义，控制器就可以提供一个简单易懂的媒体性能模型。

PPA被组织为反映SSD和媒体体系结构的分解层次结构。 例如，NAND闪存可以被组织为平面，块，页面和扇区的分层结构，而诸如PCM的字节可寻址存储器是扇区的集合。 尽管SSD架构，通道和PU的组件都存在于所有PPA地址中，但媒体架构组件可以被抽象化。 这一点如图2所示。

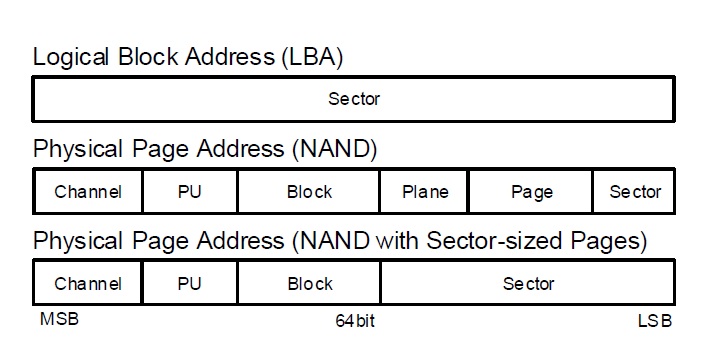
每个器件在64位地址的上下文中定义PPA地址的位阵列命名。 换句话说，PPA格式并没有对其进行限制

图2：与NAND闪存的物理页面地址相比较的逻辑块地址。

每个设备的最大通道数或每个PU的最大块数。 由每个设备来定义这些限制，并可能忽略一些媒体特定的组件。 这种灵活性是PPA格式与为早期硬盘驱动器引入的分级CHS（柱面 - 扇区）格式之间的主要区别。

将SSD和媒体体系结构编码为物理地址，可以定义嵌入在开放通道SSD中的硬件单元，将输入I / O映射到适当的物理位置。 PPA格式也非常适用于标识符是作为二的幂的独立变量的表示。 这样，对名字空间的操作（例如，在下一个通道上获得下一页，或者在不同的PU上获取下一页）通过移位来有效地实现。

PPA地址空间可以逻辑地组织起来，作为一个传统的逻辑块地址（LBA）, 例如通过使用“块，页面，平面和扇区” [34] 来安排NAND闪存。 这使得PPA地址空间可以通过传统的读/写/修剪命令暴露出来。 与传统的块I / O相比，I / O必须遵循一定的规则。 写入必须在一个块内顺序发出。 可以对整个块进行修剪，以便设备将该命令解释为擦除。 单个读取可能跨越单个I / O中的多个块是实现特定的。

与传统的线性LBA空间相比，PPA地址空间可能包含无效的地址，I / O不被接受。 考虑例如每个PU有1067个可用块，那么它将被表示为11个比特。 块0-1066将是有效的，而块1067-2047将是无效的。 在这种情况下，由控制器返回一个错误。 如果层级中的每个级别的媒体配置不是2的幂，则在地址空间中将存在这样的空洞。

**3.2 Geometry and Management**

为了让主机控制SSD管理，open channel SSD必须展现四个特征：

1. 它的几何形状，即PPA地址空间的尺寸。 多少个频道？ 频道中有多少个PU？ 每PU多少架飞机？ 每架飞机有多少块？ 每块有多少页？ 每页有多少个扇区？ 每个页面的出站区域有多大？ 我们假设给定地址空间的PPA维度是统一的。 如果SSD包含不同类型的存储芯片，则SSD必须将存储器暴露为单独的地址空间，每个地址空间都基于类似的芯片。
2. 其性能，即捕获数据命令性能，信道容量和控制器开销的统计数据。 该规范的当前版本捕获页面读取，页面写入和擦除命令的典型和最大延迟，以及针对通道内单独PU的飞行中命令的最大数量。
3. 媒体特定的元数据。 例如，NAND闪存特定元数据包括设备上的NAND闪存的类型，不管是支持多平面操作，用户可访问的出站区域的大小，还是MLC和TLC芯片的页面配对信息。 随着媒体的发展和变得越来越复杂，让SSD处理这种复杂性可能是有利的。
4. 控制器功能。 正如我们在第2节中看到的，控制器可能支持写缓冲，失败处理或供应。 这些功能中的每一个都可能被配置（例如，跨越PU的RAID）。 如果控制器支持写入缓冲，则flush命令使主机能够强制控制器将其缓冲区的内容写入存储介质。

**3.3 Read/Write/Erase**

数据命令直接反映了NAND闪存单元的读，写，擦除接口。 对于不支持的介质，擦除命令将被忽略。

Vectored I/Os.

数据命令扩展传统的LBA访问。 读取或写入命令不再由启动LBA，一些要访问的扇区和一个数据缓冲区来定义。 而是将读或写应用于地址矢量，以利用SSD的固有并行性。 例如，让我们考虑64KB的应用程序数据。 假设页面大小为4KB，则可以通过将写入命令同时应用于16个扇区来分割该数据，从而有效地支持分散的访问。

具体而言，每个I / O都表示为NVMe I / O读取/写入命令。 我们用一个PPA地址或一个指向地址列表的指针替换起始LBA（SLBA）字段，表示为PPA列表。 PPA列表包含要访问的每个扇区的LBA。 同样，我们使用NVMe I / O元数据字段来执行带外元数据。 元数据字段通常用于端到端数据一致性（T10-PI / DIF / DIX [25,43]）。 如何优雅地结合端到端和PPA元数据是未来工作的一个主题。

当数据命令完成时，PPA接口为每个地址返回一个单独的完成状态。 这样，主机就可以区分并从不同地址的故障中恢复。 对于规范的第一次迭代，NVMe I / O命令完成条目的前64位用于表示完成状态。 这将PPA列表中的地址数量限制为64。

我们考虑了PPA清单的替代方案。 事实上，我们评估了三种方法：（i）NVMe I / O命令，（ii）分组I / O，和（iii）向量I / O。 NVMe I / O命令串行发布命令。 当一个完整的页面缓冲区组成时，它被刷新到媒体上。 每个命令响起控制器的门铃通知新的提交。 对于分组的I / O，几页构成一个提交，门铃只有一个响铃，但由控制器维护每个提交的状态。 对于矢量I / O，需要额外的DMA来传递PPA列表。 我们选择第三个选项，因为额外DMA映射的成本通过简化的控制器设计来补偿。

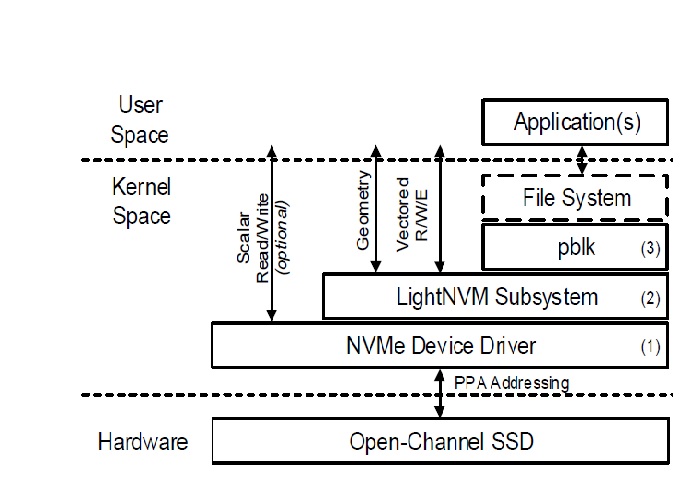
**Media specific.** 每个I / O命令都提供了特定的媒体提示，包括平面操作模式（单平面，双平面或四平面），擦除/程序暂停[56]以及有限的重试。 平面操作模式定义应该一次编程多少架飞机。 控制器可以使用平面操作模式提示来并行有效地编程平面，因为它默认顺序地访问PU。 类似地，擦除暂停允许读取暂停有效的写入或程序，从而改善其访问延迟，代价是较长的写入和擦除时间。 有限的重试允许主机让控制器知道它不应该耗尽所有的选项来读取或写入数据，但是如果数据已经在其他地方已经可用的话，则快速失败以提供更好的服务质量。

**4 LightNVM**

LightNVM是Linux中的开放式通道SSD子系统。 在本节中，我们将对其体系结构进行概述，并详细介绍pblk目标。

* 1. **Architecture**

LightNVM分为三层（见图3），每个层为开放式通道SSD提供一个抽象级别：

图3：LightNVM子系统体系结构

1. NVMe设备驱动程序。 支持LightNVM的NVMe设备驱动程序通过PPA I / O接口为内核模块提供开放式通道SSD的访问权限。 设备驱动程序将设备作为传统Linux设备暴露给用户空间，从而允许应用程序通过ioctl与设备进行交互。 如果PPA接口通过LBA暴露，则也可能相应地发出I / O。

2. LightNVM子系统。 子系统的一个实例在支持PPA I / O的块设备之上初始化。 该实例使内核能够通过内部nvm\_dev数据结构和sysfs公开设备的几何结构。 这样，FTL和用户空间应用程序可以在使用之前了解设备几何。 它还使用blk-mq [4]设备驱动程序私有I / O接口公开了矢量接口，通过设备驱动程序可以高效地发送矢量I / O。

3. **高级I / O接口。** 一个目标通过一个高级I / O接口，或者像pblk提供的块I / O接口这样的标准接口（见4.2节），或者一个基于应用程序的接口来访问开放通道的SSD， 自定义目标提供的特定界面。

4.2 pblk: Physical Block Device

物理块设备（pblk）是一个LightNVM目标，它实现了一个完全关联的基于主机的FTL，该FTL公开了一个传统的块I / O接口。 实质上，pblk的主要职责是（i）处理控制器和媒体特定的约束（例如，缓存必要数量的数据以编程flash页面），（ii）将逻辑地址映射到物理地址（4KB粒度）和 保证关联映射表（L2P）的完整性和最终恢复，（iii）处理错误，以及（iv）实施垃圾收集（GC）。 由于典型的Flash页面大小大于4KB，因此pblk也必须（v）处理刷新。 冲洗强制pblk的飞行数据在完成前存储在设备上。 它可能是文件系统或应用程序（即fsync）所要求的。

**4.2.1 Write Buffering**

pblk目标是基于2.4节描述的体系结构，写缓冲在主机上进行管理。 写入缓冲区被管理为一个圆形的环形缓冲区。 它在内部被分成两个缓冲区：一个存储4KB用户数据项的数据缓冲区（4KB对应一个扇区的大小）和一个存储Perentry元数据的上下文缓冲区。 缓冲区的大小是Flash页面大小（FPSZ），要写入的Flash页面数量（下/上页面）和PU的数量（N）的乘积。 例如，如果FPSZ = 64KB; PP = 8; N = 128，则写缓冲区为64MB。

写入缓冲区由几个生产者和一个消费者访问：

**Producers**. pblk用户和pblk自己的垃圾收集器都将I / O作为条目插入到写入缓冲区中。 当写入新的条目时，L2P表用入口行更新，写入被确认。 如果缓冲区已满，则重新编写写入。 如果传入的逻辑地址已经存在映射，则旧条目无效。

Consumer. 当有足够的数据填充闪存页面或发出刷新命令时，单个线程将消耗缓冲的条目。 如果使用多平面编程，则还必须考虑平面的数量（例如，具有四平面编程的16KB页面需要64KB块用于单次写入）。 在这一点上，逻辑地址映射到物理地址。 默认情况下，pblk的映射策略针对跨页面粒度的通道和PU的吞吐量和条带数据。 其他数据放置策略可以使用。 映射发生后，向量写入命令形成并发送到设备。 请注意，在刷新的情况下，如果没有足够的数据填充闪存页面，则在写入命令发送到设备之前，pblk在写入命令中添加填充（即未映射的数据）。

为了尊重较低/较高页面对（第2.1节），L2P表格在页面映射时不会被修改。 这样，读取直接写入缓冲区，直到所有的页面对都被保持。 发生这种情况时，L2P表会更新为物理地址。 L2P恢复在4.2.2节中讨论。

可以在运行时调整用于映射传入I / O的通道和PU的数量。 我们称它们为活跃的PU。 例如，让我们考虑在具有4个通道和每个通道8个PU的开放通道SSD上的4个活动PU。 首先，PU0，PU8，PU16和PU24有效。 页面以循环方式写在这些PU上。 当一个块填满PU0时，那么PU变为不活动，并且PU1接管为活动PU。 在任何时间点上，只有4个PU处于活动状态，但数据仍然以页面间隔分布在所有可用的PU上。

当应用程序或文件系统发出刷新时，pblk确保将所有未完成的数据写入介质。 消费者线程清空写入缓冲区，并根据需要使用填充填充最后一个闪存页。 在数据持久化的情况下，最后的写入命令会保存一个额外的注释，指示在刷新成功之前它必须完成。

4.2.2 Mapping Table Recovery

L2P映射表对于块设备中的数据一致性至关重要。因此，我们坚持三种形式的映射表的冗余版本：首先，作为一个快照，存储（i）在断电时，以L2P的完整副本的形式，和（ii）周期性地作为检查点形式的FTL日志，坚持块（分配和擦除）的操作。其次，作为块级元数据，在每个块的第一页和最后一页。当一个块被打开时，第一页被用来存储一个块序列号和一个对前一个块的引用。当一个块被完全写入时，最后一页用来存储（1）一个由L2P映射表中对应于该块中的数据的部分组成的FTL日志，（2）与第一页中相同的序号为了避免在恢复期间的额外读取，以及（3）指向下一个块的指针。所需页数取决于块的大小。这个策略允许我们有序地恢复FTL，并防止旧的映射覆盖新的映射。最后，将映射表的一部分保存在（iii）写入设备的每个闪存页的OOB区域内。在这里，对于每个持久闪存页面，我们将与页面上的物理地址对应的逻辑地址与一个表示页面有效的位一起存储。由于块被垃圾收集后可以重复使用，所有的元数据都与CRC和相关的计数器一起被保存，以保证一致性。

设备的任何初始化都将触发完全恢复。 如果L2P映射表快照可用（例如由于正常关机），则映射表直接从磁盘检索并加载到内存中。 在非正常关机的情况下，必须恢复映射表。 我们设计了一个两阶段恢复过程。

首先，我们扫描所有可用块的最后一页，并将其分为自由，部分书写和完整书写。 我们可以通过查看序列号来减少扫描，只能恢复写入的块。 在第一阶段，使用序列号对完整写入的块进行排序。 然后用每个最后一页上存储的地图部分更新L2P表。 同样，在第二阶段，部分书面的块被排序。 之后，线性扫描块，直到到达OOB区域中具有无效位的页面。 每个有效的映射都会触发L2P表中的更新。 为了确保数据的正确性，在发布读取之前填充半页下页/上页是非常重要的。 如果控制器在超级电容器上计数，则可以在非正常关机时在设备中进行填充。 否则，必须在恢复的第二阶段执行填充，因为部分写入的块被恢复。

4.2.3 Error Handling

与处理读取，写入和擦除失败的传统FTL不同，pblk仅处理写入和擦除错误。 如第2.2节所述，ECC和阈值调整由设备执行。 如果读取失败，那么从设备的角度来看，数据是无法恢复的。 恢复必须由系统的上层管理，高于pblk。

写入失败时，pblk启动两个恢复机制。 首先，使用在命令完成条目中编码的每个扇区完成位来识别与其上写入失败的扇区对应的块。 这些失败的扇区被重新映射，并直接重新提交给设备。 由于pblk提供的刷新保证，它们不会被插入到写入缓冲区中。 如果刷新被附加到失败的命令，后续的写入将停止，直到指向的I / O完成。 必须坚持前面的刷新写入，才能进行前进。 第二种机制在对应于失败扇区的块被标记为坏时开始。 在这里，剩余的页面被填充，块被发送给GC。

在擦除失败的情况下，块被直接标记为坏。 由于此时没有发出写入，所以没有要恢复的数据。

4.2.4 Garbage Collection

与任何日志结构的FTL一样，pblk必须实现垃圾回收。 通过垃圾收集任何有效的页面并返回新的写入块来重新使用块。 假定在设备上或在LightNVM内核中发生磨损平衡（2.3节）。 因此，pblk只是为每个块维护一个有效页数，并选择有效扇区数最少的块进行回收。

逻辑与物理映射表反向逻辑不存储在主机内存中。 要找到反向映射，我们利用一个块在完全写入时首先被回收的事实。 因此，我们可以使用存储在块的最后几页上的部分L2P表进行恢复。 如果该块中的页面仍然有效，则将其排队等待重写。 当所有页面被安全地重写时，原始块被回收。

为了防止用户I / O干扰垃圾回收，pblk实现了PID控制的限速器，其反馈回路基于总数

可用的空闲块。 当空闲块的数量低于可配置的阈值时，GC启动。 请注意，GC也可以从sysfs进行管理。 在开始时，GC和用户I / O同时竞争写入缓冲区。 但是随着可用块的数量的减少，GC被优先化，以保证已经存在的数据的一致性。 反馈回路确保传入的I / O和GC I / O移向稳定状态，在此状态下，由于用户I / O工作量的限制，应用程序会执行足够的垃圾回收。 速率限制器使用写入缓冲区条目作为控制输入I / O的一种自然方式; 作为反馈回路的函数保留条目。 如果器件达到其容量，用户I / O将被完全禁用，直到有足够的空闲块可用。

5 Experimental Evaluation

我们的实验评估的目的是三重的。 首先，我们验证LightNVM堆栈的正确性，然后评估它引入的开销。 其次，我们在第一代开放通道SSD（OCSSD）之上描述了pblk，并在吞吐量，延迟和CPU利用率方面与现有的NVMe SSD进行比较。 我们依靠fio [2]和应用程序工作量进行这项研究。 最后，我们将展示如何使用显式的PU写入配置来优化I / O调度并实现可预测的延迟。

我们的实验装置包括一台配备了Intel Xeon E5-2620v3，32 GB DDR4 RAM的服务器，一个带有2TB NAND MLC闪存的Open-Channel SSD（CNEX Labs Westlake SDK），在本节的其余部分用OCSSD表示， 具有400GB存储的NVMe SSD（Intel P3700）表示为NVMe SSD。 两款SSD都是使用MLC NAND的数据中心/企业SSD，这使得它们在硬件原始性能方面具有可比性。 OCSSD上的每个运行使用一个新的pblk实例; NVMe SSD格式化为4K扇区大小，并在每次运行之前进行低级格式化。 主机运行带有Linux Kernel 4.8-rc4和pblk补丁的Ubuntu 15.04。

整个LightNVM堆叠大约10K LOC; pblk负责近70％的代码。

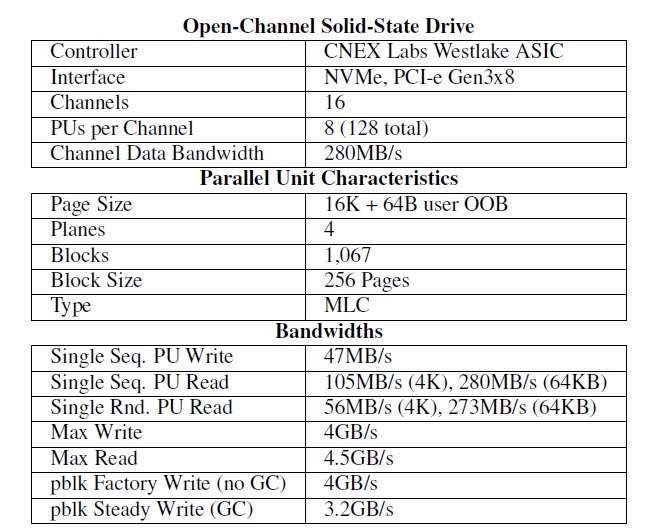


表1：固态驱动器特性

5.1 Sanity Check

表1包含了评估的开放通道SSD的一般特性.Per-PU顺序读取和写入带宽通过使用PPA I / O接口的fio的修改版本[3]通过实验收集，并直接发布向量I / O 装置。 pblk工厂状态和稳定状态（垃圾收集处于活动状态）通过pblk上的标准fio进行实验性测量。 请注意，我们将pblk的详细特性留给未来的工作，只能证明实现如预期那样工作。 除非另有规定，否则每个实验都在出厂状态下进行，并禁用pblk的限速器。

就CPU利用率而言，pblk引入的读取开销少于1％，额外延迟0.4ms（2.32ms，没有1.97ms，相差18％）。 写操作时，会增加4％的CPU开销，额外的0.9ms延迟（2.9ms和2ms，相差45％）。 读取路径上的开销是由于对L2P表的额外查找，并且写入路径上的开销是由于缓冲区和设备写入I / O请求管理造成的。 CPU开销是通过比较在空块设备[4]之上使用和不使用pblk所花费的时间来测量的，并且不包括设备I / O时序。

5.2 Uniform Workloads

图4捕获在100GB数据上使用fio发出的连续读取和随机读取的吞吐量和延迟。 已经使用设备的全部带宽（128个PU）使用pblk执行测试的准备。 这意味着顺序读取在控制器内部更容易并行化，因为顺序逻辑地址在每个页面上跨越通道和PU物理地分条。

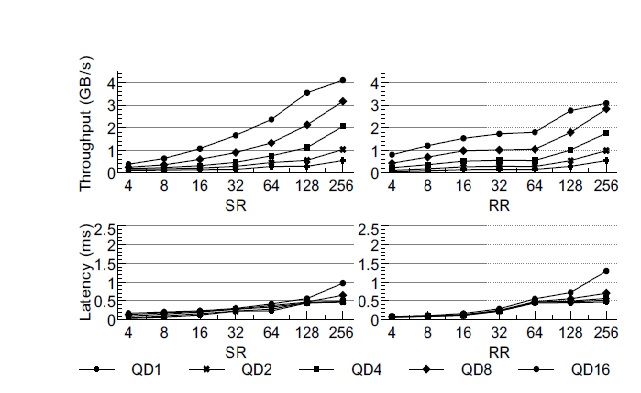


图4：作为队列深度（QD）和块大小（x轴，以KB为单位）的函数，顺序读取和随机读取工作负载的吞吐量和相应的延迟。

我们看到，使用顺序读取的OCSSD能够达到4GB / s，使用256KB的请求大小和16的队列深度，平均等待时间为970ms。第一百九十个百分点的等待时间报告在1,200ms。 同样，我们使用1的队列深度测量4KB读取的吞吐量和延迟。最大吞吐量为105MB / s，平均访问延迟为40ms，在400ms时为第99百分位。 平均访问延迟低于单个Flash页面访问，因为控制器会在内部缓存Flash页面。 因此，位于同一闪存页面上的所有扇区将从控制器缓冲区提供，而不是发出新的闪存页面读取。 此外，读取吞吐量受Flash页面访问时间的限制，因为我们一次只执行一次读取。

纯读写工作负载可用于校准队列深度以达到全带宽。 他们展示了最佳的情况，读写不会彼此阻塞。 现在让我们来讨论混合工作负载，这对SSD来说更具挑战性。

5.3 Mixed Workloads

对于混合工作负载，我们使用与之前实验相同的写入准备（跨100GB数据集的所有PU）。然后，我们继续写100GB的偏移量，而我们从第一个100GB读取。 我们重复这个实验，为新写入改变条带大小（活动写入PU的数量）。 这个假设是，随着条带大小的减小，读取性能的可预测性应该会随着写入被写入的概率降低而增加。

图5描述了混合读写时pblk的行为。 在图5（a）中，我们显示了写入和随机读取的吞吐量以及它们的参考值，分别由100％写入（4GB / s-200MB / s）和100％随机读取（3GB / s）表示。 图5（b）描述了其延迟。 该实验由在队列深度的大量顺序256KB写入组成

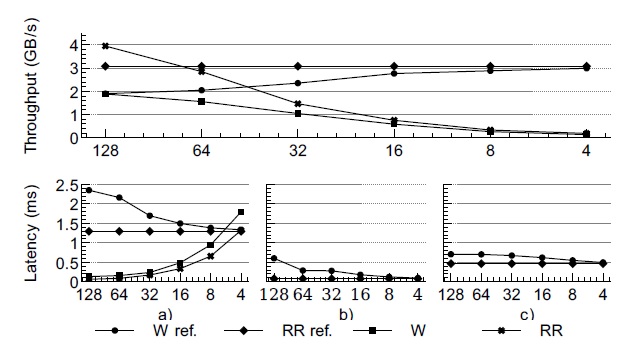


图5：作为活动写入PU配置函数的读/写吞吐量和延迟。 上图和a：吞吐量和相应的延迟（写入：256KB，QD1;读取：256KB，QD16）; b：读取延迟（写入：256KB，QD1;读取：4KB，QD1）; c：读取延迟（写入：256KB，QD1，速率限制在200MB / s;读取：256KB，QD1）。

1和在队列深度16处的256KB随机读取。写入队列深度是1，因为足以满足完全写入带宽（由pblk写入缓冲器的容量定义）。 请注意，读取是在队列深度16发出的，因此控制器可以利用足够的并行性。 这使我们能够更好地观察最坏的延迟和更少写入PU的效果。

我们观察到，当所有128个PU上的新写入被分段时，读取和写入吞吐量与参考值相比减半，而读取的平均延迟加倍（最大延迟增加4？）。写入延迟几乎不受影响，因为它们是缓冲的。这代表了传统SSD上的典型情况：读取堆叠在写入后面，从而影响读取性能;主机和控制器队列充满读取请求，从而影响写入性能。但是，一旦我们开始限制活动写入PU的数量，我们就会观察读取如何快速恢复。对于这个实验，我们在切换到不同的PU之前将一个块配置为完全写入活动PU。所有128个PU中的写入仍然是条带化的，但是不是以页面粒度条带化，而是以块粒度条带化。这降低了对新写入发出相同PU的读取概率（因为读取和写入以不同粒度分条）。如果我们将写入活动PU的数量降低到4，我们可以看到读取与参考读取工作量非常接近，同时仍然以200MB / s的速度写入。

图5（c）显示了队列深度为1的4K读取的延迟。在这里，我们强调读取被写入阻塞的影响。 如图5（b）所示，随着我们减少活动写入PU的数量，延迟变化减小。 使用4个主动写入PU，第99个百分位的随机读取的最大延迟仅比平均情况高2ms。

图5（d）显示了与a）和b）中相同的实验，不同之处在于写速率受限于

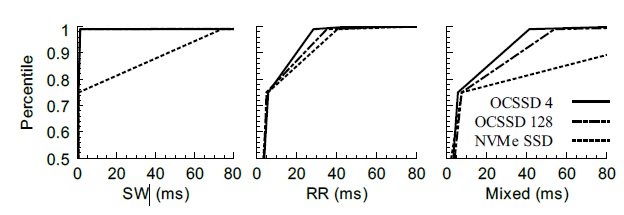


图6：OCSSD和NVMe SSD上的RocksDB顺序写入，随机读取和混合工作负载的延迟

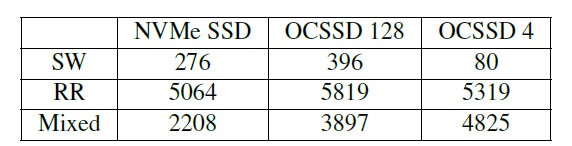


表2：OCSSD和NVMe SSD上RocksDB顺序写入，随机读取和混合工作负载的吞吐量（MB / s）

200MB/ S。 这个实验的动机是期望在下一代SSD中一致的写入。 请注意，目前的SSD已经在三年内定义了最大的持续写入带宽。 例子是重写（例如，Intel DC P3608,1.6GB，5DDPD）和重读（例如，Samsung 960 Pro，2TB，2.2DWPD）SSD，其中极限分别是95MB / s和26MB / s。 这个实验的有趣输出是，即使在写入被评估时，读取的变化仍然受到活动写入PU的数量的很大影响。

更一般地说，混合工作负载的实验表明，可以利用基于应用程序实际工作负载的明智决策来优化传统的块设备接口，而不需要特定于应用程序的FTL。

5.4 Application Workloads

我们使用NoSQL数据库评估pblk，使用OLTP和OLAP工作负载评估MySQL。 NoSQL数据库依赖于一个LSM树进行存储，并倾向于更少的冲刷（启用同步以保证数据的完整性），而MySQL在将事务数据保存到磁盘上有严格的限制。 我们使用NVMe SSD和OCSSD评估使用128和4个活动的写入PU。

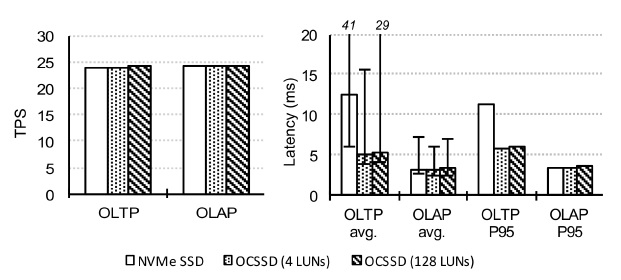
NoSQL的。 在这个实验中，我们在Ext4文件系统上运行了RocksDB [17]，并利用RocksDB的db\_bench来执行三个工作负载：顺序写入，随机读取和混合（RocksDB在线读写测试）。 图6显示了三个工作负载的用户吞吐量和延迟。 我们显示延迟分布的第95,99和99.9百分位的延迟。 请注意，在内部RocksDB执行自己的垃圾收集（即sstable压缩）。 这会消耗设备带宽，而不会由db\_bench报告。

图7：NVMe SSD和OCSSD上的OLTP和OLAP的每秒事务数和延迟。

NVMe SSD的用户写入吞吐量为276MB / s，128个活动PU的OCSSD为396MB / s，4个活动PU的吞吐量为88MB / s。 活动PU越少，清楚地表明写入性能是有限的。 随机读取工作量的性能对于两个SSD都是可比较的。 涉及写入时有显着差异。 首先，两个固态硬盘在顺序工作负载下都具有相同的行为，直到达到第99.9百分位，即OCSSD提供的延迟时间减少一半。 其次，对于混合工作负载，OCSSD在第99个百分点上已经提供了低得多的延迟（大约是三倍）。 这是因为在NVMe SSD写入之后，读取操作往往会卡住，并且OCSSD具有更多内部并行性，可以通过写入操作来实现。

OLTP和OLAP。 图7显示了在MySQL数据库系统和Ext4文件系统之上的Sysbench的[32] OLTP和OLAP工作负载。 延迟误差界限也显示工作负载的最小/最大值。

这两个工作负载目前都是CPU绑定的，因此对于所有的SSD都是相似的。 但是，在编写时，OLTP工作负载会显着增加刷新开销。 对于10GB的写入，发送了44,000次刷新，大约有2GB数据填充。 对于OLAP（RocksDB），只发送了400次冲刷，只有16MB额外的填充。 因此，对于事务性工作负载，设备端缓冲器将显着减少所需的填充量。

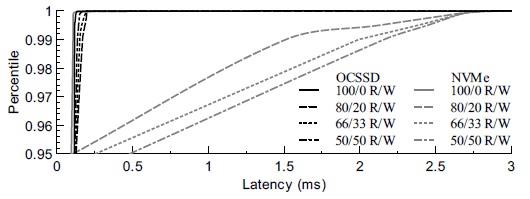
延迟结果显示与RocksDB相同的趋势。 在第95百分位数上，与传统SSD相比，传统SSD上写入OLTP的延迟增加很多，而对于开放式SSD来说，延迟并不明显。 对于OLAP，由于工作负载是CPU密集型且大多数是只读的，因此吞吐量和延迟方面的结果是相似的。 因此，在读取和写入/擦除之间不存在干扰。 调优SQL数据库以获得开放式通道SSD上的性能是未来工作（可能通过基于KV的存储引擎[39]）的一个有趣的话题。

图8：OCSSD和NVMe SSD的延迟比较显示了写入如何影响读取延迟。 请注意，y轴不是从0开始，而是从0.95开始。

5.5 Predictable Latency

这个实验说明了特定应用FTL的潜在好处。 我们再次使用fio的修改版本直接向设备运行两个并行的向量I / O流。 一个线程在队列深度1发出4KB随机读取，而另一个线程在相同队列深度发出64K写入。 用于OCSSD的流被隔离以分离PU，而NVMe SSD混合读取和写入。 我们在五秒内测量工作负载，并报告图8中的延迟百分比。我们将延迟粒度报告为100 = 0; 80 = 20; 66 = 33; 和50 = 50。 随着写入增加，OCSSD的性能保持稳定。

尽管NVMe SSD没有办法将读取与写入分开，并且即使轻负载（20％写入）也会引入更高的读取延迟。

我们的观点是，PPA I / O接口使应用程序开发人员能够明确地管理SSD中每个独立PU的队列，从而实现可预测的I / O延迟。 表征具有开放通道SSD的应用特定FTL的潜力是未来工作的一个主题。

6 Related Work

由于固态硬盘的缺点越来越明显[12,15,30,37]，研究集中在组织主机和固态硬盘之间的合作。 一种形式的合作包括从主机传递提示到嵌入式FTL。 多流的工作属于这一类[28]。 其他形式的合作包括绕过FTL [24,35,50]，或围绕给定FTL的属性设计系统的上层[29,33,36,38,46]。 最后，基于主机的FTL让主机控制数据放置和I / O调度。 这是我们采用LightNVM的方法。

主机端FTL已经被FusionIO DFS [27]和Violin Memory [54]实现，每个FTL都将FTL移入主机，以暴露块I / O SSD。 同样，欧阳等人 [45]提出了软件定义Flash，允许键值存储与底层存储介质集成。 另外，Lu等人 [37]在基于主机的基于对象的FTL（OFTL）上定义了与百度的开放式固态硬盘相对应的原始闪存设备。 相比之下，我们为openchannel SSD指定了一个跨供应商接口，并展示了可调节块I / O目标如何减少读取延迟变化。

Lee等人人。 [34]提出了一个新的SSD接口，兼容传统的块设备接口，通过读/写/修剪操作暴露无错误的仅追加段。这项工作是基于自上而下的方法，该方法显示了如何在建议的仅附加界面之上实现最先进的文件系统（F2FS）。相比之下，我们的论文描述了一种自下而上的方法，其中PPA接口反映了独立于系统上层的SSD特性。首先，我们的PPA接口允许写入和擦除错误传播到主机，以提高I / O可预测性。我们还定义了一个向量I / O接口，允许主机利用设备并行性。最后，我们明确地向主机公开介质的读写粒度，以便可以在主机或设备上放置写缓冲区。利用LightNVM展示特定应用SSD管理的好处是未来研究的一个主题。与RocksDB [22]或多租户I / O隔离[21,26]的初步结果是有希望的。

**7 Conclusion**

LightNVM是Linux内核中的开放式通道SSD子系统。 它通过PPA I / O接口向主机公开任何开放式通道SSD。 LightNVM还提供分区管理器和可调的块I / O接口。 我们的实验结果表明，与传统NVMe SSD相比，Light-NVM提供了（i）低开销和显着的灵活性，（ii）降低了读取的可变性，以及（iii）获得可预测的延迟的可能性。 未来的工作包括描述各种开放通道SSD模型（在撰写本文时已经宣布了三家供应商的产品）的性能，为关系数据库系统设计调优策略，并为关键值商店设计特定于应用的FTL。

8 Acknowledgments

我们感谢匿名审稿人和我们的牧羊人Erez Zadok，他们的建议有助于改进本文。 在撰写本文期间，我们也感谢Alberto Lerner，Mark Callaghan，Laura Caulfield，Stephen Bates，Carla Villegas Pasco，BjörnÞórJónsson和Ken McConlogue的宝贵意见。 我们感谢Linux内核社区，包括Jens Axboe和Christoph Hellwig提供关于上游补丁的反馈，并改进了架构。