

**海量存储技术**

**结课报告**

**姓名：李浩南**

**学号：2017286190084**

**KAML：一种灵活的高性能Key-Value SSD**

**摘要**

现代固态硬盘（SSD）不必要地将主机程序限制在传统的块I / O接口，导致性能不佳和资源利用率不足。 最近尝试用面向键值的接口替代或扩展这个接口和/或对事务处理的内置支持提供了一些改进，但是它们的实现细节使得它们与许多应用程序匹配不佳。

本文介绍了键可寻址的多日志SSD（KAML），这是一种具有键值接口的SSD，采用新型多日志架构，将数据存储为可变大小的记录而不是固定大小的扇区。 暴露键值接口允许应用程序去除在应用程序级键（例如，数据库记录ID或文件inode号）和存储在SSD中的数据之间的间接层。KAML还提供本地事务支持，以支持细粒度锁定，与以前需要页面级锁定的设计相比，实现了更高的性能。最后，KAML包含一个类似于传统页面缓存的缓存层，利用主机DRAM提高性能并提供额外的事务型特点。

我们在商用SSD原型平台上实现了KAML原型，结果表明，与现有的键值存储相比，KAML将在线事务处理（OLTP）工作负载的性能提高了1.1倍--4.0倍，NoSQL键-值存储应用提高了1.1倍 - 3.0倍。

**1 引言**

闪存正在取代旋转磁盘作为许多企业存储系统的首选存储介质。固态硬盘（SSD）提供了更低的延迟，更大的带宽和更高的并发性，所有这些使得它们适用于现代数据密集型应用程序。但是，SSD用来存储数据的闪存与主导存储系统的基于块的，以磁盘为中心的接口不匹配。键值接口是一个有吸引力的选择，因为SSD必须在逻辑存储地址（或键）和物理存储位置之间保持一个间接层。这允许更简单的实现并且使得可以利用间接层来提供诸如快照或多部分原子操作之类的附加服务。

然而，目前关于SSD键值接口的建议在很多方面都不尽人意。首先，它们为SSD中的所有键值对提供单一的共享映射，迫使SSD使用统一策略来管理所有应用程序中的键值映射。其次，提议在SSD上停止，忽略了其余的存储堆栈。 特别是，他们不提供通用的缓存设施来提高键值访问的性能。

大多数基于闪存SSD的多部分原子写入提议也受到限制。 最值得注意的是，它们只支持事务的页面级原子更新。尽管大多数SSD使用的页面级映射使得这是一个方便的选择，对于实现者来说，这对于许多应用程序所需要的是不匹配的。 提供页面级别的原子写入操作需要应用程序执行页面级别的锁定。对于处理较小数据单位（例如数据库记录）的应用程序来说，页面级别的锁定已经显示出比精细化的方法提供更低的性能。

为了解决这些限制，本文介绍了键可寻址的多日志SSD（KAML）。 KAML扩展了基于键值固态硬盘的现有建议，并提供细粒度的多部分原子写入接口和主机端缓存层来加速访问它包含的数据。缓存层建立在SSD的事务接口上，并实现一个精细的锁定协议，最大限度地减少事务中止并最大限度地提高并发性。

KAML允许应用程序创建多个键值名称空间，这些名称空间可以根据应用程序的需求来表示文件，数据库表或任意的对象集合。

这个新的接口有三个好处。首先，KAML负责将键映射到值的物理地址，允许应用程序绕过不必要的间接寻址。如果没有KAML，应用程序必须维护自己的索引以将键映射到文件偏移量，并依靠文件系统将文件偏移量转换为逻辑块地址（LBAs）。

其次，应用程序可以使用KAML接口自动创建或更新多个键值对。如果没有这个接口，应用程序必须使用writeahead日志记录（WAL）或其他应用程序级别的技术来提供原子更新，消耗更多空间并导致昂贵的文件系统操作（如fsync）。 最后，KAML的细粒度键值接口与粗粒度的对应接口相比，提高了系统的并发性，因为它允许细粒度的锁定。

KAML管理其内部闪存的方法反映了其接口的要求。现代固态硬盘具有多个并行的半独立闪存通道，并可在这些通道中并行执行闪存操作。KAML将这些通道映射到记录更新到特定名称空间的日志，并允许应用程序调整名称空间和闪存通道之间的映射，以优化性能并提高服务质量。

**2 动机和背景**

KAML利用闪存的特性和要求为应用程序提供丰富的基于键值的事务处理接口。以前有关SSD中事务支持的建议为页面级操作提供了事务处理，因为这些粗糙的事务处理方式非常适合传统SSD管理其所包含的闪存。然而，许多应用程序（例如，MySQL的InnoDB [2]，Oracle数据库[3]和Shore-MT[1]）依靠细粒度的事务来实现高并发性。KAML还提供了一种新颖的闪存管理方法，支持细粒度的交易，同时仍充分利用现代SSD硬件。

**2.1 闪存和SSD**

闪存有三个重要特征KAML需要适应。首先，闪存读取，编程和擦除操作在不同的粒度上工作，并且具有非常不同的延迟。读取和编程操作4-8 KB页面。 一旦写入，页面就变得不可变，除非包含它的块被擦除。块通常包含64到512个页面，块内的页面必须按顺序写入。读取页面一般不会超过100μs，而编程一个页面则需要100μs到2000μs，具体取决于底层闪存设备[4]。擦除操作需要几毫秒才能完成。最后，每个块在变得不可靠之前只能经受有限数量的擦除操作。

为了解决闪存的局限性，提高数据访问的性能和可靠性，SSD提供了逻辑块地址（LBA）映射到每个访问的物理页号（PPN）的FTL。

由于页面在写入之前是不可变的（直到擦除），就地更新是不可能的。 相反，传统的FTL将传入的写入指向已擦除的页面，并更新LBA和PPN之间的映射。 FTL允许SSD向主机公开一个通用的块设备抽象，并隐藏损耗水平和垃圾收集（GC）的内部细节。

现代SSD通常具有多个闪存通道，可提供丰富的内部I / O带宽。SSD包含嵌入式处理器来执行FTL的固件，磨损水平和GC。SSD的内部数据结构（例如，地址映射表，块元数据和快照）驻留在板载DRAM上。

NVM Express（NVMe）是通过PCIe连接到主机系统的高端SSD的新标准接口[5]。NVMe是面向块的，但它允许供应商特定的扩展。

**2.2 FTL的实现**

现代FTL共享的两个核心功能是逻辑到物理地址映射表和GC设施。常规的FTL使用映射表来跟踪从逻辑到物理地址的映射。FTL以日志结构的方式管理闪存 ---- 它不会将数据写回原始的物理位置，并更新地图，以便找到最新的值。 这些不在场的更新导致数据的陈旧副本（即“垃圾”），并且SSD必须回收（或“收集”）垃圾以为新写入提供路线。

为了维护映射，促进垃圾收集和存储纠错信息，FTL还将每页元数据存储在每个闪存页的所谓的“带外”（OOB）区域中。OOB区域通常占用128到256个字节。

**2.3 SSDs中的事务**

许多应用程序需要某种事务支持[5]，并且有几个小组已经提出了向SSDs添加事务支持[6][7][8][9]。KAML（以及这些提议）为原子性和持久性提供了原生支持，这是事务性ACID支持的两个关键方面，但是却使应用程序具有隔离性和一致性。

利用传统的FTL设计来支持原子性是诱人的，因为FTL已经依赖于原子写入拷贝（COW）。然而，大多数常规的FTL是基于页面的，并且许多现有的用于FTL中的原子写入的建议[9][6][7]使交易能够以页为单位更新数据。另一个建议[10]支持更细粒度的锁定，但依赖于更奇特的字节寻址非易失性存储器而不是闪存。

当应用程序允许同时访问同一页上的记录时，基于页面的原子写入接口是有问题的[11]。如果一个事务成功提交页面，而在同一页面上更新的另一个事务中止，则不存在反映正确状态的页面副本。因此，使用基于页面的原子写入的事务必须获取粗粒度页面锁以确保正确性，这对系统性能是有害的。相反，大多数数据库存储引擎例如 InnoDB [2]，Oracle数据库[3]，Shore-MT [1]等允许事务锁定单个记录。因此，他们不能从基于页面的原子写入接口中受益。

**3 系统概况**

KAML的核心是使用新型FTL结构来管理闪存的固态硬盘。这个FTL提供了一个事务性的键值界面。该系统包括一个可定制的SSD和三个不同的软件。首先，新的FTL运行在工业SSD参考平台上。其接口允许主机软件（例如数据库，文件系统或其他用户空间应用程序）使用细粒度事务来创建，管理和访问多个键值存储。该固件与在主机上运行的内核驱动程序和用户空间库协同工作。 最后，主机缓存层使用主机DRAM加速访问，并提供额外的事务处理功能。

图1说明了这些组件之间的关系。本节介绍KAML的系统组件，下一节将更详细地介绍其实现。

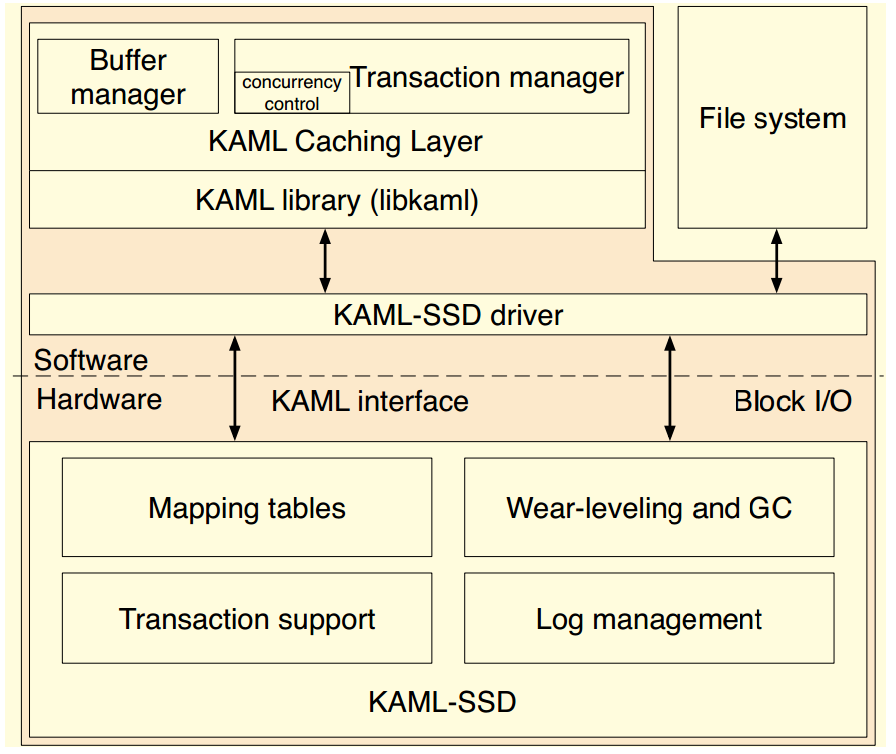


图1：系统结构，阴影部分表示包含四个组件的KAML系统：KAML-SSD，驱动程序，用户库（libkaml）和KAML缓存层。

KAML提供了一个支持细粒度事务的键值接口。该接口允许应用程序创建和销毁表示逻辑上相关的键值对的键值名称空间。命名空间允许多个独立的应用程序共享一个KAML SSD。表1总结了该界面中包含的命令。

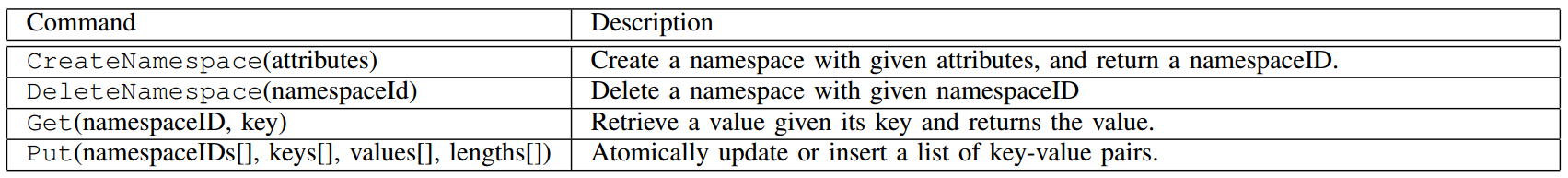


表1：KAML命令，这些命令允许应用程序使用事务语义和键值寻址方案来访问SSD上的数据。

**3.1 键，值，命名空间**

KAML中的键是64位的，但值的大小可能有所不同。本地支持可变大小的值可让应用程序在KAML中存储各种各样的对象。例如，传统的基于页面的文件系统可以将键作为块地址，并将4 KB页面作为值存储在名称空间中。或者，数据库可以将各个元组存储为值，并使用元组的键或记录ID将其映射到SSD中。

暴露键值接口允许应用程序消除冗余间接层。在传统系统中，应用程序可能会将键映射到文件中的位置，底层文件系统会将该文件位置映射到逻辑块地址（LBA），FTL会将LBA映射到物理页码（PPN）。KAML将其转换为从PPN到PPN的单一映射，并消除了与其他映射关联的应用程序和文件系统开销。

KAML还避免了在文件系统中使用日志结构技术的需要，消除了“日志堆栈”[13]可能导致的性能问题。该接口允许KAML整合SSD中的垃圾回收操作，其中固件具有关于闪存的性能特征和数据布局的最完整信息。

**3.2 原子性和耐久性**

KAML的Put操作可以原子地插入/更新多个键值对，提供原子性和耐久性（ACID中的“A”和“D”）。KAML缓存层可以提供隔离。我们保持应用程序的一致性，因为尽管原子性和持久性要求在各个应用程序中是相似的，但一致性要求（例如数据库级联，触发器和约束）差异很大。同样，如果一个应用程序需要一个自定义锁定协议，它可以提供它自己的，而不是依靠缓存层的设施。

**3.3 细粒度锁定**

尽管KAML SSD没有实现并发控制，但是允许高效的并发控制是KAML SSD接口的核心目标，而高速缓存层则广泛使用它。由于KAML事务操作的是可变大小的键值对而不是固定大小的页面，应用程序可以使用细粒度的基于记录的锁定协议，而不是基于页面的粗粒度协议。

细粒度的锁定对性能至关重要。没有它，修改页面中的记录的事务必须持有整个页面的锁定，直到事务提交，阻止需要访问同一页面上的另一个记录的任何其他事务。我们量化了第五节中粗粒锁定的性能影响。

**3.4 KAML缓存层**

KAML可以通过在DRAM中缓存数据来提高应用性能，就像传统系统缓存基于页面的SSD或硬盘数据一样。具有常规SSD的系统将数据缓存在操作系统的页面缓存或应用程序特定的缓存层（例如数据库缓冲池）中。KAML需要不同的缓存体系结构，因为它的接口基于键值对而不是页面或块。

KAML的缓存层提供了主机基于DRAM的缓存和更丰富的事务接口，除了提供KAML SSD提供的耐用性和原子性外，还提供隔离。

**Caching**缓存层允许KAML提供对缓存数据的快速访问，并通过Get和Put命令与SSD进行交互。缓存层不同于传统的页面缓存，因为它缓存了可变长度的键值对，而不是固定大小的页面（或文件系统术语中的块）。

缓存层使用名称空间ID和键来组成一个复合键并探测一个哈希表。如果哈希表中有一个与键匹配的条目，则键值对已经在缓存中，缓存层将键值对返回给应用程序。

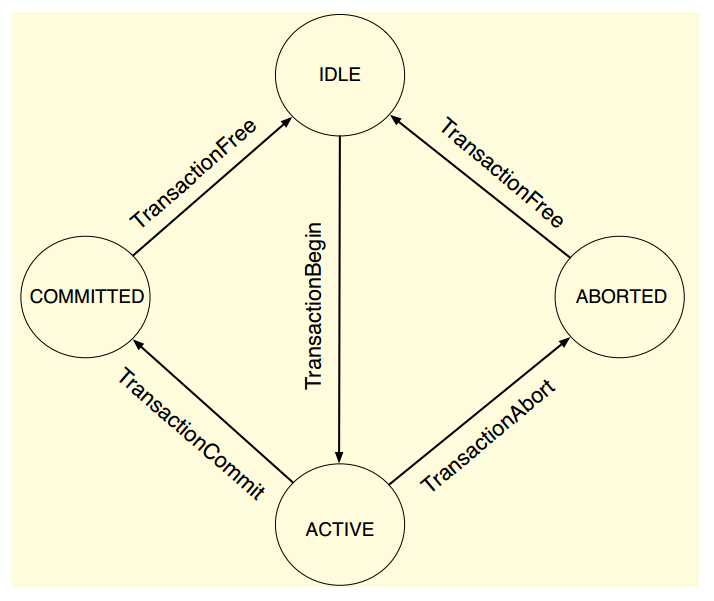
如果哈希表没有匹配的条目，则缓存层向SSD发出Get请求，并将生成的键值对插入到哈希表中。

当事务提交时（或缓存层空间不足时），缓存层使用Put将键值对写回SSD。

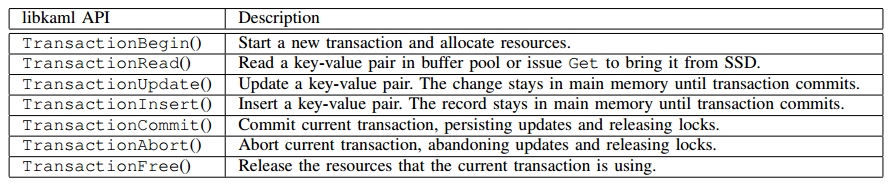
一旦命令完成，KAML可以重新使用缓存空间的值占用或保存在缓存中的键值对，并将其标记为干净。

**Transactions**缓存层的事务管理器使用主机上强制严格的两阶段锁定（SS2PL）[13]提供对隔离的支持，并通过缓存层访问存储在KAML上的键值对。

事务管理器维护一组事务控制块（XCB）。事务开始时，事务管理器为事务分配一个XCB。XCB允许事务处于四种状态之一：空闲（即，事务还没有开始），激活（即事务正在进行中），中止和提交。图2显示了这些状态之间可能的转换，表2总结了导致状态转换的操作以及事务管理器的API。



图二 事务的状态转换图，任何时候事务都处于这些状态之一，并通过调用KAML API转换到另一个状态



表二 KAML缓存层API，缓存层提供了一个事务接口

活动事务在访问它们之前获取键值对上的锁，并创建键值对的私有副本以在事务中使用。事务提交时，事务管理器用其私有副本替换缓冲池中的键值对，然后发出Put命令来刷新对KAML的更改。在放弃时，事务只是放弃它的私人副本并释放其锁。

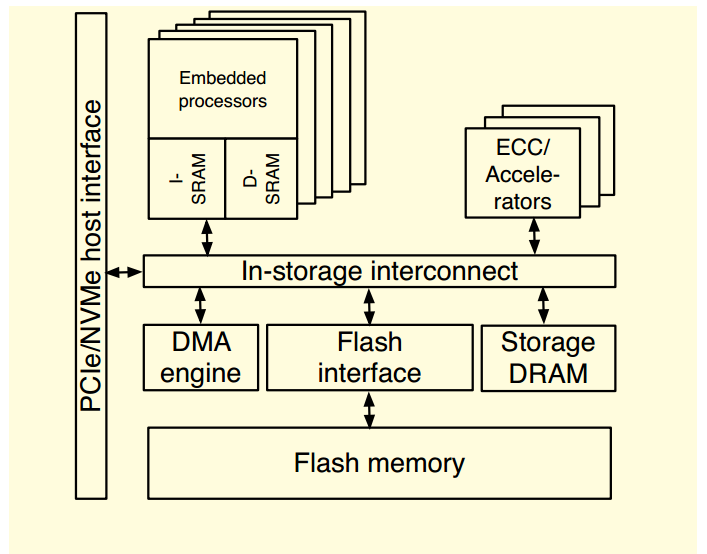
**4 KAML**

KAML使用定制的SSD，定制驱动程序和轻量级用户空间库来实现KAML接口，比以前提供键值接口的SSD更高效。固件管理存储器中的日志，维护映射表，执行损耗均衡，并实施垃圾回收（GC）。下面，我们介绍我们使用的SSD原型硬件和KAML固件和系统软件的关键方面。

**4.1 硬件架构**

我们已经在一个商用的NVMe SSD开发板上实现了KAML。 像其他现代固态硬盘一样，它包含一排16“通道”的闪存芯片（总共375 GB），连接到一个通过PCIe与主机系统进行通信的多核控制器。 控制器上运行的固件定义了SSD的接口，并实施旨在提供高性能和最大化闪存寿命的管理策略。

图3描述了闪存控制器的内部组织。 控制器架构的主要特性是内核本身，闪存接口和板上DRAM。



图三 KAML体系结构KAML通过PCIe / NVMe接口向主机提供内部计算能力

内核具有64 KB专用指令和数据存储器。他们可以使用片上网络互相通信，DRAM和闪存接口。为了访问闪存，内核向闪存控制器发出命令，闪存控制器从DRAM中的缓冲区读取数据或向DRAM中的缓冲区写入数据。为了访问DRAM的内容，内核明确地在DRAM和它们的私有数据存储器之间进行复制。

每个闪存通道包含4个共享控制和数据线的闪存芯片，并连接到SSD控制器上的通用闪存控制器。闪存芯片可以并行执行读取，编程和擦除操作，但是一次只能有一个芯片传输数据到控制器或从控制器传输数据。

对于这项工作，我们假设DRAM是持久的。在实践中，它可能是电池或电容式的，或者可以用Intel的3D-XPoint存储器等非易失性技术来替代。

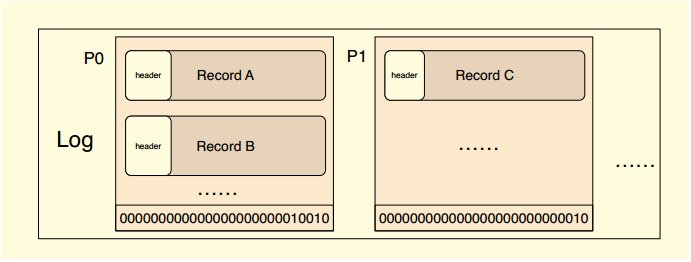
**4.2 命名空间管理**

KAML将每个闪存目标中的闪存管理为日志。当Put操作到达时，KAML将键值对写入日志的尾部，并更新存储与每个键相对应的键值对的位置的索引结构。

SSD的体系结构决定SSD维护的日志数量，KAML将每个键值名称空间分配给多个日志。这改善了局部性，并导致更有效的垃圾收集（见下文）。但是，名称空间和日志之间的对应关系并不固定：随着工作负载的变化，SSD可以将更多或更少的日志分配给单个名称空间，以匹配名称空间正在经历的工作负载。 如果名字空间特别冷，它可能与其他名字空间共享一个日志。默认情况下，所有SSD的日志都可用于所有名称空间。多个日志不会产生额外的硬件开销，因为现代SSD已经具有内部并行性的多通道架构。

限制名称空间和日志之间的映射作为局部性优化允许SSD控制资源（特别是带宽）到名称空间的分配。但是，由于映射是灵活的，因此KAML的GC和闪存管理算法不会假定特定名称空间的数据驻留在其分配的日志中。

为了缩小应用程序需求和闪存特性之间的差距，KAML将可变大小的键值存储在固定大小的闪存页面中。我们SSD中的物理Flash页面大小为8 KB + 256字节。每个页面被分成64个固定大小的块，每个记录可以占用可变数量的块。 固件使用8字节的OOB数据来存储位图，该位图记录页面上每个记录的开始和结束块，如图4所示。



图四 存储内日志和记录日志由闪存页面组成，每个闪存页面都在OOB区域中存储描述页面中可变大小记录的位图。记录A占据页面P0的块0和1，记录B占据P0的块2,3,4，记录C占据P1的块0和1

如果记录的最后一个块是第i个块，则位图的第i位被设置。第一条记录始终从页面的块0开始，并且在同一页上的两条记录之间没有未使用的块。固件的其他组件（如GC）可以使用此位图来解析页面的内容（请参阅第4.5节）。

Flash仅支持整页程序操作，但即使不填写页面，它也必须提交单独的Put操作。为了克服这个问题，KAML使用一个非易失性的DRAM类缓冲区来缓冲多个Put操作，直到一页数值的数据准备写入或内部定时器超时为止。现有的固态硬盘使用电容或电池备份的存储器，但是3D XPoint等新兴的非易失性字节寻址存储器也是合适的。

**4.3 映射表**

KAML使用每个名称空间索引来定位与每个键相关联的键值对。KAML指数与传统SSD中的FTL有两种不同的地图。首先，传统的FTL将LBA映射到PPN，并且LBA形成连续的块地址空间。这意味着LBA的含义是固定的（即，它对应于SSD逻辑存储空间中的特定逻辑地址），所以它不能包含有用的应用程序级数据（例如，它们不能表示数据库表中的记录ID）。这也意味着LBA对应于固定大小的数据块。

其次，在传统的FTL中，只有一个映射表，而不是很多。这进一步限制了应用程序如何使用LBA的灵活性，因为LBA位于全局共享名称空间中。它还可以防止SSD根据访问模式或应用程序需要对索引进行不同的管理。

例如，KAML可以限制名称空间的映射表的大小，甚至可以使用不同的数据结构（例如树，而不是KAML使用的哈希表）来存储映射表。

当名称空间正在使用时，我们将索引存储在SSD的DRAM中。像其他现代高端固态硬盘一样，我们使用几GB的DRAM来存储KAML指数。由于KAML指数可以更精细，它们可能比传统的LBAto-PPN图大。例如，一个1亿个键值对的哈希表和75％的负载率需要大约2 GB的内存DRAM。

为了支持更大的数据容量，SSD提供的DRAM数量可能需要增加，或者固件可能需要在闪存和DRAM之间“交换”索引信息。KAML当前在应用程序访问时加载名称空间的索引，但不交换索引的部分内容。KAML采用简单的策略将未使用的映射表交换为闪存，以便在应用程序启动之前为正在使用的用户腾出空间。

**4.4 交易支持**

KAML的事务机制支持高并发性的细粒度数据访问。这个设计决定将KAML与以前大多数关于SSD中多部分原子写入支持的提议区分开来[9][8]。具体而言，KAML-SSD为细粒度的多记录原子写入提供本地支持，而KAML缓存层则为数据缓冲和锁定提供支持。KAML的本地原子写入提供了持久性和原子性，并且缓存层实现了隔离。

Put命令提供事务性接口。它接受名称空间ID，键，值和值大小的数组，指定一组更新/插入来应用，SSD保证所有的更新/插入都以原子方式进行。

Put分三个阶段执行。在第一阶段，固件会收到要更新或创建的键值对的列表。KAML固件将数据传输到SSD的非易失性RAM。然后，固件检查每个键是否已经存在于命名空间的索引中。如果是，则锁定该条目。如果密钥不存在，则固件会保留并锁定索引中的空条目。在固件获得锁之后，操作已经逻辑提交，并且固件通知主机。

在第二阶段，固件启动闪存写入操作，将键值对写入闪存。闪存写入完成后，固件会记录其物理地址。如果在所有闪存写入操作成功之前发生故障，固件将使用非易失性缓冲区中的数据进行恢复。

一旦所有闪存写入操作完成，固件就具有所有新的闪存地址的信息。 固件使用闪存中新近写入的键值的物理地址更新索引。一旦固件完成将更改应用到映射表，它释放锁，释放缓冲区并将该命令标记为成功。如果发生故障，则使用第二阶段记录的闪存地址恢复固件。

**4.5 Wear-leveling and GC**

KAML的GC算法在可变大小的键值对上运行，而不是固定大小的页面，但基本操作与其他SSD GC方案类似。GC子系统维护每个日志的擦除闪存块列表，并在日志末尾的块满时选择新的闪存块。在块中写入所有页面之后，KAML固件将块移动到按擦除计数和有效数据量（即未被相同记录的较新版本替换的字节）对块进行排序的排序列表。

当空闲块数量低于阈值时，KAML的GC算法开始回收空间。它选择块清理，擦除次数少，有效数据量小。这样做可以在块之间均匀分布擦除，并最大限度地减少将有效数据复制到新块所需的工作量。

一旦选择了一个闪存块进行清理，固件会将闪存中的页面读入存储器DRAM中，并使用第4.2节中描述的每页位图来提取所有的键值对。对于每个键值对，固件在索引中搜索键。如果搜索结果与正在扫描的当前键值对的物理地址相匹配，则键值对有效，固件会将其写回新位置。如果固件无法找到键值对的条目，或者搜索结果指向了不同的物理位置，则键值对无效，固件会将其丢弃。检查完一个块中的所有页面之后，固件擦除该块并将其添加到空闲块列表中。

**4.6 Key-Value stores in SSDs**

研究人员已经构建了针对SSD进行优化的键值存储。FlashStore [16]和SkimpyStash [17]将键值对存储在SSD上，并应用各种优化来提高性能和内存消耗，但是它们依赖于传统的SSD导致堆栈日志以及这些日志所带来的低效率。 同样，使用Bw-Tree [18]作为键值数据库也会导致类似的问题。相比之下，KAML在SSD中实现了一个键值接口，消除了堆栈日志，降低了复杂性，同时通过缓存层提供快速访问。

SILT [19]是单个节点上的高性能，高效内存的存储系统。它有三个关键价值商店，其内存效率和写入性能都有不同的优化目标。KAML通过将密钥直接映射到物理地址，将应用索引与原始FTL结合在一起。新的映射表位于SSD内的电池支持的DRAM中，因此减少了主机内存使用量。

NVMKV [14]以VSL [15]为基础来实现键值存储。其映射表驻留在主机内存中，不同存储区中的密钥均匀分布在整个LBA空间中。而且，NVMKV依赖于Level-DB的读缓存。相比之下，KAML为不同的键值存储和通用缓存层提供了单独的名称空间。

**5 总结**

通过提供键值语义来实现SSD接口现代化的现有方法提供了一个不灵活的关键空间，而针对事务性SSD的建议则要求低效率的粗粒度锁定。KAML通过支持键值对的多个独立名称空间来解决这些问题，并允许应用程序调整每个名称空间的性能以满足应用程序的要求。它也通过将键值对作为事务操作的基本单位来实现细粒度锁定。最后，KAML提供了一个类似于传统页面缓存的缓存层来提高性能。通过这些更改，KAML可以在各种工作负载下超越传统设计。

**参考文献**

[1] Shore-MT, http://research.cs.wisc.edu/shore-mt/.

[2] InnoDB, http://dev.mysql.com/doc/refman/5.7/en/innodb-storage-engine.html.

[3] Oracle Database, https://www.oracle.com.

[4] L. Grupp, A. Caulfield, J. Coburn, S. Swanson,E. Yaakobi, P. Siegel, and J. Wolf, “Characterizing FlashMemory: Anomalies, Observations, And Applications,”inMICRO-42, 2009, pp. 24 –33.

[5] J. Gray et al., “The transaction concept: Virtues and limitations,” in VLDB, 1981, pp. 144–154.

[6] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou, “Transactional Flash,” in OSDI, 2008, pp. 147–160.

[7] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, andD. Panda, “Beyond Block I/O: Rethinking TraditionalStorage Primitives,” in HPCA, 2011, pp. 301–311.

[8] W.-H. Kang, S.-W. Lee, B. Moon, G.-H. Oh, andC. Min, “X-FTL: Transactional FTL for SQLiteDatabases,” in SIGMOD, 2013, pp. 97–108.

[9] W. Shi, D. Wang, Z. Wang, and D. Ju, “Mobius: A HighPerformance Transactional SSD with Rich Primitives,”in MSST, 2014, pp. 1–11.

[10] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, andS. Swanson, “From ARIES to MARS: Transaction Support for Next-generation, Solid-state Drives,” in SOSP,2013, pp. 197–212.

[11] G. Graefe, “Write-optimized B-trees,” in VLDB, 2004,pp. 672–683.

[12] J. Yang, N. Plasson, G. Gillis, N. Talagala, and S. Sundararaman, “Don’t Stack Your Log on My Log,” inINFLOW, 2014.

[13] P. A. Bernstein, V. Hadzilacos, and N. Goodman, Concurrency Control And Recovery in Database Systems.Addison-wesley New York, 1987, vol. 370.

[14] L. Marmol, S. Sundararaman, N. Talagala, and R. Rangaswami, “NVMKV: A Scalable, Lightweight, FTLaware Key-value Store,” in USENIX ATC, 2015, pp.207–219.

[15] VSL, http://www.fusionio.com/products/vsl.

[16] B. Debnath, S. Sengupta, and J. Li, “FlashStore: HighThroughput Persistent Key-value Store,” VLDB, vol. 3,no. 1-2, pp. 1414–1425, 2010.

[17] “SkimpyStash: RAM Space Skimpy Key-valueStore on Flash-based Storage,” in SIGMOD, 2011, pp.25–36.

[18] J. J. Levandoski, D. B. Lomet, and S. Sengupta, “TheBw-Tree: A B-Tree for New Hardware Platforms,” in ICDE, 2013, pp. 302–313.

[19] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky,“SILT: A Memory-efficient, High-performance Keyvalue Store,” in SOSP, 2011, pp. 1–13.