题目

摘要

现代 SSD 固件不断优化，以获得更高的并行性，以匹配不断增长的前端 PCIe 带宽和更多的后端闪存通道。虽然通常采用多核微处理器来并发处理来自多个 NVMe 队列的独立 NVMe 请求，但现有的一对多线程请求映射模型 (每个线程服务一个或多个传入 I/O 请求) 由于严重的锁争用问题，特别是在缓存管理方面，可扩展性很差。本文首先在一个开放通道 NVMe SSD 上进行了初步实验，展示了一对多线程-请求映射模型中多个线程竞争同一 cacheline 时的锁争用问题，它占用了长尾请求延迟的 50% 以上。

为了缓解这个问题，我们提出了 PipeSSD，这是一种无锁的基于管道的 SSD 固件设计，具有多对一线程请求映射模型，该模型以管道的方式分配多个线程来服务每个 I/O 请求的不同阶段。在 PipeSSD 中引入了三个新颖的设计:1) 一个无循环的请求处理管道，延迟缓存操作阶段，以减少缓存竞争;2) 无锁缓存管理方案，保证缓存行为的一致性;3) 基于 fifo 的阶段间请求流，用于正确的请求依赖关系。我们在真实硬件上实现了 PipeSSD，并在多核 NVMe SSD 原型上评估了其性能。评估结果表明，与最先进的多核 SSD 固件相比，PipeSSD 具有显著的吞吐量改进。

关键词：SSD 固件，闪存转换层，多核架构，PipeSSD

1 引言

具有多个请求队列的 NVMe 协议充分利用了高速 PCIe 接口 [5] 的潜力。为了满足不断增长的前端 PCIe 带宽，NVMe ssd 结合了多个后端闪存通道 [3, 4] 以实现更高的吞吐量。然而，现有的 SSD 固件在多核 SSD 控制器中采用一对多的线程请求映射模型，无法有效利用宝贵的计算资源，而多核 SSD 控制器是弥补前端 PCIe 带宽不断增长和后端闪存通道数量不断增加之间差距的关键组件。

由于每个线程处理一个或多个传入 I/O 请求，SSD 控制器可以很容易地合并额外的内核，以增强线程级别的并行性。尽管多个 NVMe 请求应该通过多核 SSD 控制器并发处理，但我们观察到由于缓存竞争导致的锁争用问题很严重。这个问题导致了超过 50% 的计算资源浪费，并阻碍了 SSD 固件的可扩展性。

为了缓解锁争用，我们提出了 PipeSSD，这是一种基于多核 SSD 控制器的无锁管道 SSD 固件。与传统的一对多线程请求映射模型不同，我们引入了多对一线程请求映射模型，将处理每个 NVMe 请求的过程划分为四个阶段 (请求获取 → 缓存管理 → Flash 翻译层 (FTL) → Flash 接口层 (FIL))，并将它们分布在不同的核心上。尽管这种方法允许以流水线方式处理这四个阶段，但它仍然面临几个具有挑战性的问题。

第一个挑战是如何构建一个无循环的管道。简单地遵循上述请求处理序列仍然会遇到管道内的循环，这是由于缓存更新造成的。具体来说，当一个请求触发缓存行缺失时，需要一个锁来确保数据一致性，直到相关的缓存行被正确的数据更新，阻塞所有需要访问该缓存行的后续请求。如果是读缺失，则在完成 FIL 阶段 (即从闪存芯片读取数据) 后需要重新进入缓存管理阶段，从而在管道中引入循环。为了解决这个问题，我们通过将缓存管理阶段推迟到最后一步来重新组织管道阶段。通过这样的请求处理顺序 (请求获取 → FTL → FIL → 缓存管理)，所有的缓存修改，包括新数据的写入和缓存的替换，都被推迟到最后阶段，从根本上消除了管道内部的循环。如果没有管道循环，我们可以安全地删除缓存行锁，因为所有缓存行修改都只来自前一个管道阶段，可以按顺序处理。

与缓存一致性问题相关的第二个挑战出现在延迟缓存管理阶段之后。一旦请求到达并检查缓存，结果 (命中/未命中) 可能会改变，因为之前的请求必须比这个请求更早到达缓存管理阶段。这可能会修改目标缓存，并为该请求引入错误的缓存状态。为了解决这个问题，我们在开始时分配了一个缓存管理阶段的试点，以收集缓存线的状态，并为每个请求提供一个路簿。roadbook 记录目标缓存线路的关键信息，如缓存命中情况、Tag 等。每个请求都带着它的路簿继续执行管道。在超光速阶段，准备并记录回脏缓存所需的物理页码，以及需要从闪存通道读取的缺失页码。在 FIL 阶段，每个请求都可以将 roadbook 中的缓存行标识与实际的缓存行标识进行比较，以确保写回正确的页面。在试点的帮助下，我们可以实现具有一致缓存行为的无锁缓存管理。

第三个挑战是确保正确的请求依赖关系。我们通过在不同的内核之间采用几个 FIFO 队列来实现这一点。因此，四个管道阶段之间的通信本质上遵循一种顺序的方式，这简化了依赖项的维护。我们在真正的硬件平台上使用 4 核 ARM CPU、4GB DDR4 DRAM 和 512GB NAND 闪存芯片实现 PipeSSD。实验结果表明，PipeSSD 优于最先进的多核 SSD 固件，比单线程多请求模型的吞吐量高出 46%。

2 相关工作

近年来，3D NAND Flash 因其高密度、低成本等优点，已广泛应用于各种数字存储系统中，比如用于构建大容量 SSD 存储等。随着技术的快速发展，3D 堆叠层的数量在过去的十年中显著增加，从早期的 24 层到最先进的 220+ 层。此外，每个单元的比特数也从 1 比特/单元增加到最近的 4 比特/单元。然而，在实现芯片密度显著提升和成本显著降低的同时，4 位或更多位的 3D NAND Flash 也面临着不可忽视的可靠性和性能下降问题，这日益成为进一步提升存储性能的主要设计障碍。

2.1 传统 FTL 设计

FTL 是 SSD 中逻辑地址与物理地址转换的关键结构。传统 FTL 中的映射表是缓存在动态随机存取内存（Dynamic Random-Access Memory，DRAM）中，这导致随着 SSD 容量的增加，有限的内建 DRAM 逐渐难以满足不断增大的地址映射表的存储需求。为了降低对 DRAM 的占用，当前主流的 FTL 构建方案只在 DRAM 中缓存部分活跃的映射表项，而将完整映射表项存储在闪存中。但是，传统的映射表项回收策略存在缓存命中率低的问题，这导致 SSD 的访问性能差。

2.2 基于热聚类映射表的 FTL 设计

华侨大学 Yubiao Pan 等基于有限的缓存空间提出一种新的缓存映射表（Cached Mapping Table，CMT）结构——热聚类映射表（hot-cluster FTL，HCFT）。该结构通过引入了两种不相同的高效索引结构来加快动态翻译页中映射项的搜索速度，并将从 CMT 中删除的映射项缓存到另一个额外分配的辅助缓存中。随后计算这些映射项逻辑页码的最大偏差以及生成与该偏差最小的动态页，从而提高动态翻译页的命中率。实验结果显示，所提出的 HCFTL 相比于传统 FTL 方案最多能够提升 41.1% 的 CMT 命中率以及减少 33.3% 的系统响应时间。

2.3 ReCA-FTL 设计

美国俄勒冈州立大学 Youngbin Jin 等提出一种称为 ReCA-FTL 的方案，通过整体管理 FTL 功能（如页面分配、垃圾收集、磨损均衡和维护负载平衡的调度）来提高 SSD 的性能。这缓解了资源竞争，使 SSD 更有效地利用所有资源。最后通过详细模拟研究表明，FTL 方案在各种工作负载下比选定的基线平均提供高 1.75 倍的吞吐量，最高可达到 2.04 倍。

2.4 基于数据驱动的自适应超级块 FTL 设计

中国科学院张伟东等提出一种基于数据驱动的自适应超级块闪存转换层算法。该算法的主要优势在于采用将超级块映射表 SMT 作为一级映射，页地址映射表 PMT 作为二级映射的自适应超级块的分级地址映射方案，进而有效提高系统的响应速度。此外，该算法将超级块作为闪存地址管理的最小单元，从而减少了存储系统对主机的依赖。为了平衡闪存芯片内各物理块的磨损程度并延长使用寿命，还引入了动态块回收权重作为超级块分组和目标回收块选择的标准。张伟东等通过实验结果说明，所提出的算法在主机占用率和系统响应速度方面都比传统星载 FTL 算法有明显的优势。

2.5 HDFTL 设计

针对单层单元（SLC）+ 多层单元（MLC）闪存构成的混合固态硬盘，杭州电子科技大学 Yingbiao Yao 等提出了一种新的按需 FTL（On-demand FTL，HDFTL）。该算法将 SLC 划分为数据块和转换块，而所有的 MLC 均视为数据块，从而实现了 SLC 和 MLC 区域地址映射方法的统一，简化了混合 SSD 的 FTL 设计。此外，HDFTL 还将缓存的映射表拆分为一个热写映射表和一个普通映射表，利用时间局部性解决混合 SSD 的热写识别问题。热写映射表和普通映射表的大小根据 SLC 和 MLC 区域的相对磨损率自适应调整，进一步提高了系统性能。实验结果表明，HDFTL 相对于现有的混合 SSD FTL 算法具有更好的性能表现。

2.6 基于多核的存储平台 DeepFlash

NVMe 旨在将闪存从传统的存储总线中解放出来，允许主机使用多个线程来实现更高的带宽。尽管 NVMe 使用户可以充分利用现代 SSD 提供的所有级别的并行性，但由于有限的计算能力以及不同硬件之间的竞争，袋子现有的固件设计存在可扩展性不足的问题，难以并行处理大量 I/O 请求。为此，韩国韩国科学技术院 Jie Zhang 等提出了一种新颖的基于多核的存储平台 DeepFlash，可以在每秒内处理超过 100 万个 I/O 请求，同时能够降低内部闪存

媒体造成的长延迟。DeepFlash 可以通过跨设备内多个核心并发执行多个固件组件来提取底层闪存复合物的最大性能。最终综合评估表明，DeepFlash 可以提供约 4.5 GB/s 的速度，同时最大限度地降低微基准测试和真实服务器工作负载的 CPU 需求。

3 本文方法

3.1 本文方法概述

采用一对多线程请求映射模型的传统 SSD 固件会受到护航效应的影响，因为每个线程都是全功能的，并且可以独占共享资源。另一个线程请求映射模型是多对一的，它将一个请求的处理划分为几个连续的阶段，每个线程执行一个阶段。理想情况下，每个阶段执行只需要访问其本地状态，并且请求通过 FIFO 队列在阶段之间传输。管道执行可以实现多对一映射模型的并行性，并且管道的并行性仅受阶段间依赖关系的限制。我们的目标是设计一个具有最小阶段间依赖的管道，即没有循环和锁同步。最具挑战性的问题是解决数据缓存存在多个写入器的问题，确保数据缓存的状态仅被一个阶段更新。

3.2 无锁的 pipeline

数据缓存中存储了两个数据源: 一个来自主机，另一个来自 flash 通道。如果我们将 cacheline 的更新阶段设置在 FIL 阶段之前，那么对于来自 flash 通道的数据，必须有一个从 FIL 阶段到更新阶段的循环。这种简单的设计提出了两个问题。首先，管道中的循环意味着阶段间的依赖，这将破坏并行性。其次，更新 cacheline 阻塞后续请求，特别是阻止它们进入 FIL 阶段，这限制了多通道架构中并行性的利用。

我们提出了 PipeSSD，通过将数据缓存后置到管道的最后阶段来实现我们的目标，这是一种反直觉但有效的管道设计，可以减少锁争用。具体来说，FTL 阶段和 FIL 阶段无法获得请求的缓存状态来确定它们的行为，例如，只有缓存丢失请求需要地址转换和 flash 事务。然而，放置后的数据缓存对于丢弃管道内的循环至关重要。此外，我们的技术可以克服上述挑战，包括数据缓存的试点和数据一致性调度。

PipeSSD 的工作流程如图1所示。我们将分层固件分为 4 个阶段。各阶段的功能简述如下:

- 第一个阶段是 NVMe Fetch 阶段，它从主机的 SQs 中获取 NVMe 请求，将每个请求分段为一系列页面级子请求，并从数据缓存试点中获取每个子请求的缓存状态。
- 第二阶段是 FTL 阶段，它进行必要的地址转换，然后为每个子请求提交 flash 事务。
- 第三个阶段是 FIL 阶段，它调度多个闪存通道来完成闪存事务。
- 最后一个阶段是 NVMe Post 阶段，完成请求所需的数据已准备就绪。它更新 cacheline 并发送响应。

下一部分介绍了管道设计中的技术。通过这些技术，我们的管道确保了数据的一致性，这取决于锁定机制，并始终保持并行性，而不会遇到阻塞问题。

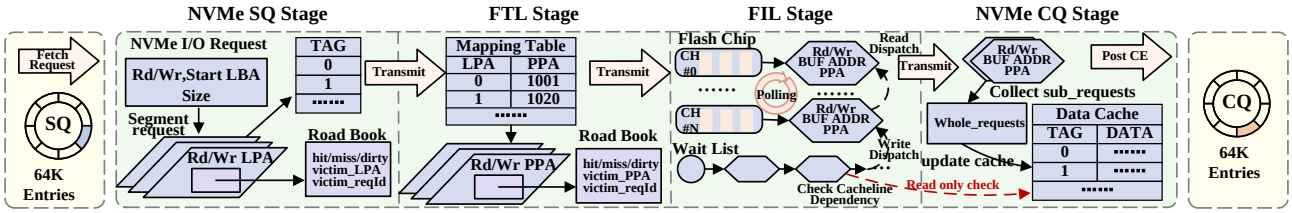

图 1. PipeSSD 的设计细节

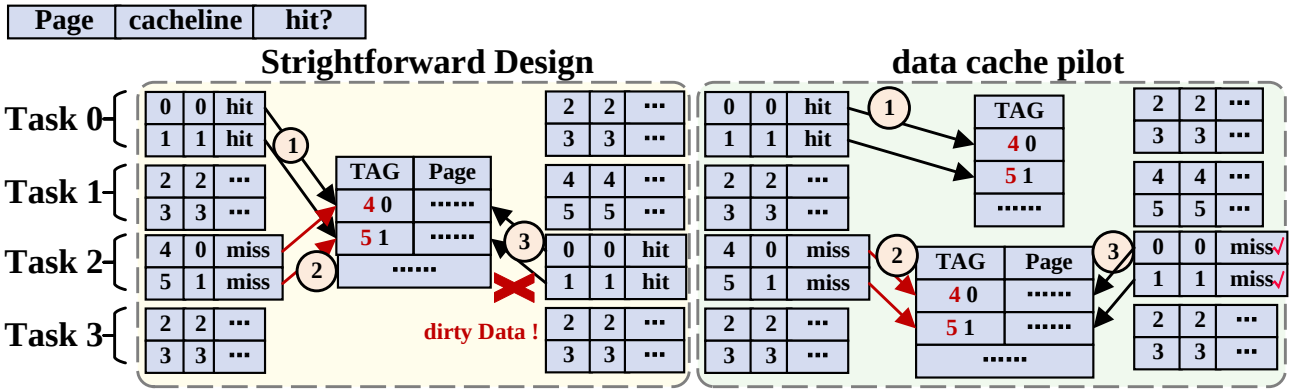

图 2. pilot 表的例子

3.3 基于无锁架构下的数据一致性问题

数据一致性由两个维度来保证，一个是控制路径的正确性，另一个是数据路径的顺序性。我们首先解决控制路径上的一致性问题。控制路径主要由 FTL 管理，FTL 将 LPA 转换为 PPA，以弥合两个地址空间之间的差距。具有不同缓存状态的请求采用不同的控制路径。

物理地址仅用于缓存丢失请求。对于读请求，所请求数据的物理地址是必需的，而对于脏缓存，准备受害者数据页的物理地址对于不合适的更新是必不可少的。因此，在地址转换之前，我们需要确认请求的缓存状态。

在我们的设计中，我们在 NVMe Fetch 阶段在超光速阶段之前获得缓存的状态。如图2所示，对后放置的数据缓存标签进行直接只读检查将破坏数据一致性，因为之前的请求可能会修改目标缓存，并为该请求引入错误的缓存状态。例如，

Task0 (NVMe SQ) 检查两个请求的 cacheline 标签。当检查完成时，但在这些请求到达 Task2(FIL) 之前，Task2 完成两个闪存 I/O 事务，然后修改缓存中的数据。因此，当这两个请求到达 FIL 阶段时，它们的缓存状态应该被忽略。但是，它们沿着 hit 状态的控制路径最终会从缓存中获取脏数据。

后放置的数据缓存只能反映最后阶段的本地信息，中间阶段内的请求尚未在相应缓存的跟踪上生成占用空间。对于每个请求，如果在访问数据缓存之前，它可以在相应的缓存线上看到前一个请求的内存占用，并且保证它按照前一个请求的内存占用进入缓存线上，那么就可以保持数据一致性。

3.3.1 pilot 表格

我们在第一阶段设置了一个数据缓存的试点，以跟踪每个缓存的占用空间。在将 NVMe 请求分割为页面级子请求后，试点为每个子请求分配一个 roadbook，该 roadbook 记录缓存

状态 (例如命中/未命中/脏) 和与相应缓存相关联的前一个请求 ID(即占用空间)。为了记录缓存的状态，pilot 使用的数据结构和算法与缓存的标签一致。对于给定的子请求，试点遵循缓存映射方法获得其目标缓存，然后将子请求的 LPA 与缓存的 LPA 进行比较。如果两个 lpa 匹配，则子请求的缓存状态被分类为命中; 否则，将错过状态。缓存状态将被记录在 roadbook 中，以便与子请求一起通过管道。此外，道路手册还存储了一份试点线路的副本。一旦建立了路簿，我们就通过将当前子请求的 LPA、ID 和脏位 (来自主机的数据) 存储到目标导航线来更新导航线的信息。

FTL 阶段和 FIL 阶段参考缓存状态来确定如何处理每个子请求。如图1所示，FTL 阶段检查地址映射表，将请求的 LPA 转换为读取缺失子请求的 PPA，并为脏受害者缓存分配新的 PPA。受害者 cacheline 的 LPA 可以在 pilot line 的副本中找到。如果有必要，FIL 阶段根据受害者的 PPA 将子请求发送到 flash 通道以回写脏页。然后，可以根据其 PPA 将子请求分派到 flash 通道以读取所请求的页面。

3.3.2 数据调度的一致性

在数据路径上，影响数据一致性的两个关键操作是回写操作和缓存更新。我们集成了数据一致性调度，保证数据在数据路径上的一致性。

在 FIL 阶段，回写操作的目的是清除脏缓存，需要准确的输入以确保持久性。同时，在 NVMe Post 阶段，cacheline 更新操作涉及用新到达的数据覆盖 cacheline。为了保证被覆盖的数据在重写之前是无效的，我们在 FIL 阶段引入了一个等待列表，以确保数据操作遵守上述约束。

在整个管道中，数据流遵循先进先出 (FIFO) 模式，除了 FIL 阶段，它通过多个闪存通道并发处理多个数据流。在我们的设计中，来自超光速阶段的请求不会直接发送到 flash 通道，而是首先进入等待列表。在调度 flash 命令时，我们定期检查等待列表中的请求是否满足缓存依赖的约束。每个请求通过 Pilot 在同一 cacheline 上获得前一个请求的 ID(即受害者 ID)。只有当路径簿中记录的受害者 ID 已经存在于当前缓存中时，请求才会退出等待列表。这种机制确保了对缓存的顺序访问。自然地，如果请求不需要 flash I/O，它将在离开等待列表后直接进入下一阶段。

基于等待列表的数据一致性调度不强制执行全局 FIFO 顺序，而是维护每个缓存的 FIFO 顺序。当一个缓存上的后续请求在队列中等待时，它不会影响其他缓存上请求的处理。这种方法确保尽可能多地保持多个闪存通道的并行性。

4 复现细节

4.1 与已有开源代码对比

该文献没有参考任何相关源代码

4.2 实验环境搭建

4.2.1 实验平台

我们在基于 fpga 的平台上构建了 NVMe SSD 原型。FPGA 使用 DMA 控制器、NAND Flash 控制器 (每个闪存通道一个) 和特定的 PCIe IP 核构建数据路径，以连接内存、闪存芯片和 PCIe。所有的关键功能都是由软件定义的，包括 NVMe 连接、地址转换和闪存通道调度。根据硬件平台采集的数据，我们构建了一个基于多线程的双迹驱动多核多闪存通道仿真框架。硬件原型与其孪生仿真平台最显著的区别在于主机接口层。块 I/O 跟踪被预加载到内存中，因此获取指令的开销比受 DMA 速率限制的硬件原型要小得多。在这个仿真平台上，我们可以用高水平的连通性 (即 NVMe 队列的请求率) 来评估我们的设计的性能，这在硬件平台上很难实现。除了性能评估，我们还在硬件原型上验证了 PipeSSD 的有效性。

4.2.2 配置信息

硬件原型配备 8 个闪存通道 (每个通道一个闪存芯片)，我们目前使用的是 mlc 型 NAND 闪存芯片 (每个芯片 64GB)。我们收集到的 flash 芯片的读写性能数据如表1所示。在我们的仿真平台上，flash 操作的延迟与硬件原型是一致的。我们的 PipeSSD 与 DeepFlash 进行了比较，后者是多核系统上最先进的 SSD 框架 [6]，以及传统的一对多线程建模 (Tradition-N, N 是线程数)。典型的 SSD 内存大小是存储空间的千分之一，我们遵循这个模式来配置我们的平台。

表 1. 基于 MLC 的闪存芯片参数设置

	地址设置	请求执行	数据迁移
读	3us	40us	60us
写	5us	400us	60us

在主流的高性能商用 NVMe 固态硬盘上，主控制器通常是基于 arm 的多核处理器。例如，三星 980pro 和 990pro 都是 5 核 32 位 ARM 处理器，支持 8 个闪存通道。我们的 FPGA 核心板采用四核 64 位 ARM 处理器。我们使用 4 核作为标准，在默认配置中，PipeSSD 的四个阶段分别固定在一个核上。DeepFlash 支持可扩展性，我们也将其缩小到 4 核。传统的方案从 4 个线程开始。

4.2.3 工作负载

我们使用来自 UMassTraceRepository [1] 的 5 块 I/O 跟踪来评估我们的设计。Financial1 (F1) 和 Financial2 (F2) 是从 OLTP 应用程序 I/O 收集的，WebSearch1 (W1)、WebSearch2 (W2) 和 WebSearch3 (W3) 是由流行的搜索引擎生成的。我们可以参考表2，很明显，第一组是高度本地化的，而第二组则完全不是。我们的设计重点是解决高速缓存的车队效应和利用多个闪存通道的并行性。这两种工作负载在缓存上的性能差别很大，一方面可以测试缓存系统的设计，另一方面可以测试缓存背后的闪存级的并行性。

为了评估我们的硬件原型，主机发出 NVMe 请求来显示用于读写的带宽。

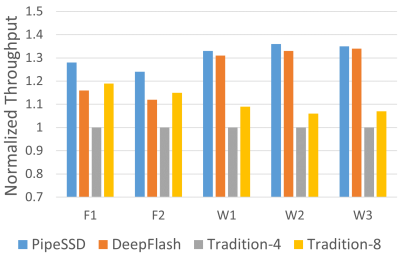

图 3. 4 个闪存通道

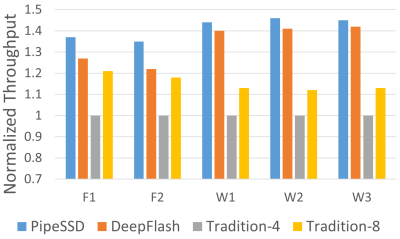

图 4. 8 个闪存通道

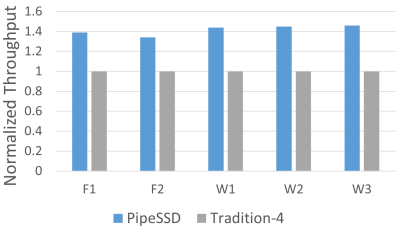

图 5. 4 个闪存通道 & 一半的数据缓存

4.3 创新点

在本文中，我们提出了 PipeSSD，一种基于多核 SSD 控制器的无锁流水线 SSD 固件。为了充分挖掘 NVMe SSD 的内部并行性，PipeSSD 通过后期放置数据缓存来调整 NVMe 请求工作流，使管道无阻塞和无循环。为了保证数据一致性，我们将管道与缓存试点和数据一致性调度集成在一起。我们在硬件原型上验证了 PipeSSD 的功能正确性。评估结果表明，PipeSSD 优于最先进的多核 SSD 固件。

5 实验结果分析

5.1 吞吐量比较

我们首先用 4 个闪存通道评估吞吐量，然后将闪存通道扩展到 8 个。

在评估中，以 Tradition-4 为基准，将不同方法的吞吐量归一化为 Tradition-4 的吞吐量进行公平比较。从评价结果中，我们可以得到以下观察结果。首先，我们的 PipeSSD 优于所有其他方法，PipeSSD 的吞吐量平均比基线高 31.2%，平均比 DeepFlash 高 5%。其次，当面对本地化程度较差的工作负载 (例如 W1、W2 和 W3) 时，PipeSSD 的优势更加明显，比基线高出 36%。第三，传统的一对多线程模型即使将工作线程数量增加一倍 (traditional -8) 也不能显著提高吞吐量。

为了验证可扩展性，我们将闪存通道的数量增加到 8 个。随着闪光通道数量的增加，非阻

表 2. 数据缓存命中率

	F1	F2	W1	W2	W3
PipeSSD	50.19%	51.03%	1.08%	0.03%	0.06%
DeepFlash	44.22%	46.17%	1.07%	0.03%	0.06%
Trandition	50.19%	51.03%	1.08%	0.03%	0.06%

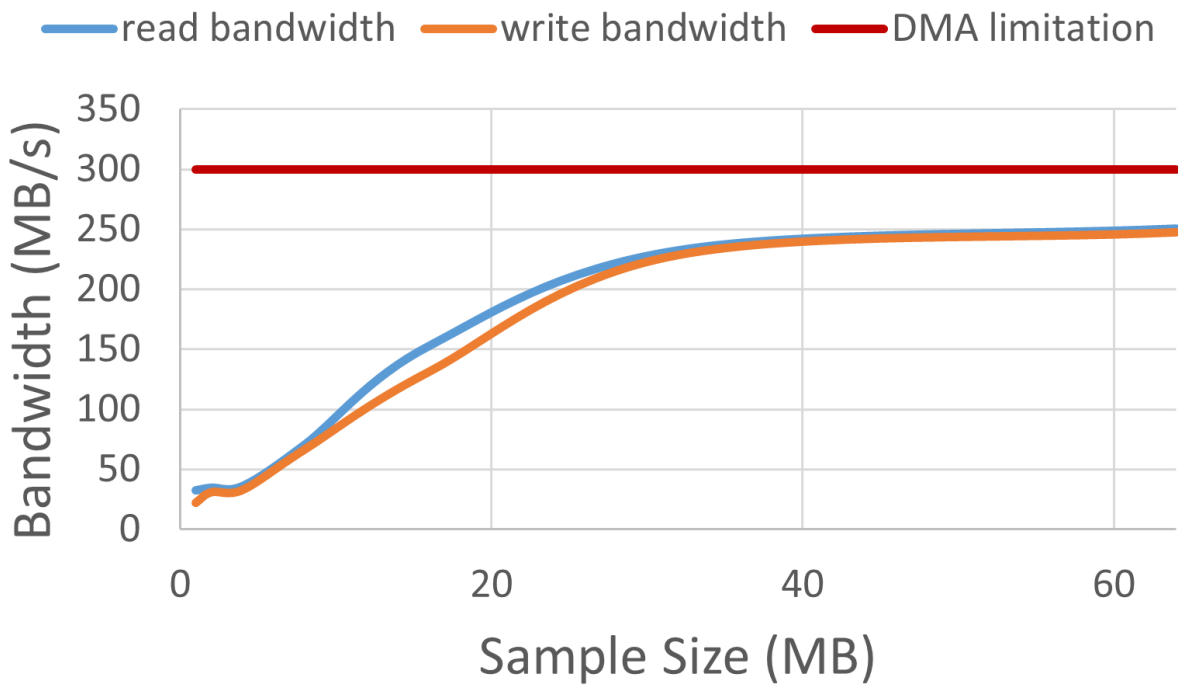

图 6. 硬件原型的带宽

塞设计的优势变得更加明显。在基于管道的设计中，数据流在 FIL 阶段之外不会被阻塞。当缓存被占用时，DeepFlash 会绕过缓存，以避免阻塞后续请求。我们的 PipeSSD 既确保连续的数据流进入闪存通道，又通过数据一致性调度保持缓存的一致性。如表2所示，PipeSSD 的缓存命中率没有下降。对于高局部性的工作负载，缓存命中可以直接减少长时延的 flash I/O 操作，这对提高 SSD 性能非常重要。PipeSSD 还具有在整个管道中最大限度地减少阶段间依赖的优点，没有回环和单向数据流。综合考虑所有这些点，PipeSSD 在这组实验中的吞吐量比基线高出约 40%，比 DeepFlash 高出 5.2%。

除了常规配置外，我们还进行了一组具有内存限制设置的附加实验。cacheline 的竞争更加激烈，非阻塞管道比基于锁的多线程方法更适合这种情况。如图5所示，在这种情况下，PipeSSD 的平均吞吐量比基线高 42%。

5.2 硬件平台验证

最后，在我们的硬件原型上验证 PipeSSD。基于 ARM SoC 支持的内存模型，没有锁保护的只读检查在唯一写入器的情况下是安全的。根据无锁队列设计 [2]，消息可以通过无锁队列安全地从单个写入器传输到单个读取器。因此，阶段之间的通信也是基于无锁队列的。为了提高内存效率，带有数据字段的请求是一个大的有效负载，内存空间是静态保存的，并且

这个大的有效负载的引用通过队列传递。

最后，在我们的硬件原型上验证 PipeSSD。基于 ARM SoC 支持的内存模型，没有锁保护的只读检查在唯一写入器的情况下是安全的。根据无锁队列设计 [2]，消息可以通过无锁队列安全地从单个写入器传输到单个读取器。因此，阶段之间的通信也是基于无锁队列的。为了提高内存效率，带有数据字段的请求是一个大的有效负载，内存空间是静态保存的，并且这个大的有效负载的引用通过队列传递。

6 总结与展望

在本文中，我们提出了 PipeSSD，一种基于多核 SSD 控制器的无锁流水线 SSD 固件。为了充分挖掘 NVMe SSD 的内部并行性，PipeSSD 通过后期放置数据缓存来调整 NVMe 请求工作流，使管道无阻塞和无循环。为了保证数据一致性，我们将管道与缓存试点和数据一致性调度集成在一起。我们在硬件原型上验证了 PipeSSD 的功能正确性。评估结果表明，PipeSSD 优于最先进的多核 SSD 固件。

参考文献

[1] Umass traces repository. http://traces.cs.umass.edu/, 2023.

[2] John Giacomoni, Tipp Moseley, and Manish Vachharajani. Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 43–52, 2008.

[3] Jen-Wei Hsieh, Han-Yi Lin, and Dong-Lin Yang. Multi-channel architecture-based ftl for reliable and high-performance ssd. *IEEE Transactions on Computers*, 63(12):3079–3091, 2013.

[4] Renping Liu, Xianzhang Chen, Yujuan Tan, Runyu Zhang, Liang Liang, and Duo Liu. Ssdkeeper: Self-adapting channel allocation to improve the performance of ssd devices. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 966–975. IEEE, 2020.

[5] Jiwon Woo, Minwoo Ahn, Gyusun Lee, and Jinkyu Jeong. {D2FQ}:{Device-Direct} fair queueing for {NVMe}{SSDs}. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 403–415, 2021.

[6] Jie Zhang, Miryeong Kwon, Michael Swift, and Myoungsoo Jung. Scalable parallel flash firmware for many-core architectures. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 121–136, 2020.