分　数：	
评卷人：	

華中科技大學

研究生（数据中心技术）课程论文（报告）

题　目：NVMe 设备 I/O 路径优化

学　　号　M202173480

姓　　名　陈祺汇

专　　业　计算机技术

课程指导教师　施展 童薇

院（系、所）　武汉光电国家研究中心

2022 年 1 月 6 日

目录

NVMe 设备 I/O 路径优化

陈祺汇 [1)]

[1)](华中科技大学武汉光电国家研究中心 湖北 武汉 430074)

摘 要 从磁盘时代开始已经衍生出了一套完整的存储协议，例如 SCSI，SAS，SATA，FC 和其他协议。作为一种有效的存储协议，NVMe 已得到许多闪存、服务器和存储供应商的支持，并且生态系统正在逐步成熟。针对近年来 NVMe 设备 I/O 路径优化的研究列出了三篇相关文献。首先介绍了降低写依赖开销的相关研究，多队列驱动器的写相关性问题进行了研究，提出屏障转换将排序元数据与相关写入分离，以强制执行正确的写入顺序，推出了一个名为 Horae 的新 IO 堆栈，以解开单个物理设备和存储阵列的写依赖性。针对尽快完成对延迟敏感的请求与延迟中断来优化吞吐量的如何权衡的问题，研究发现缺乏语义上下文来推断请求者的意图:请求是延迟敏感的，还是一系列异步请求的一部分。将这一重要信息发送到设备可以使设备能够在适当的时候中断请求者，以此提出校准中断的技术。针对 NVMe 设备的崩溃一致性问题，研究提出 ccNVMe，它是 NVMe 的一种新的扩展，定义了主机软件如何通过 PCI Express 总线与非易失性内存(例如固态驱动器)通信，同时兼顾崩溃一致性和性能效率。

关键词 NVMe；Linux IO 栈；写依赖性；中断；崩溃一致性；

A Survey on SSD Reliability

Ren ZheXuan[1)]

[1)](Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, Hubei 430074)

Abstract A complete set of storage protocols such as SCSI, SAS, SATA, FC and others have been derived from the disk era. As an effective storage protocol, NVMe is supported by many flash, server and storage vendors, and the ecosystem is maturing. Three related papers are listed for recent research on I/O path optimisation for NVMe devices. Firstly, research related to reducing write dependency overhead is presented, the write dependency problem of multi-queued drives is investigated, barrier transformations are proposed to separate sorting metadata from correlated writes to enforce proper write order, and a new IO stack called Horae is introduced to unlock the write dependency of individual physical devices and storage arrays. Addressing the question of how to trade-off between completing latency-sensitive requests as quickly as possible and delaying interrupts to optimise throughput, the study found a lack of semantic context to infer the requester's intent: whether the request is latency-sensitive or part of a series of asynchronous requests. Sending this important information to the device could enable the device to interrupt the requester at the appropriate time, thus suggesting techniques for calibrating interrupts. In response to the crash consistency problem of NVMe devices, the research proposes ccNVMe, a new extension to NVMe that defines how host software can communicate with non-volatile memory (e.g. solid state drives) over the PCI Express bus, while balancing crash consistency and performance efficiency.

Key words NVMe；Linux IO stack；Write Dependency；Interrupt；Crash Consistency；

1.引言

从磁盘时代开始已经衍生出了一套完整的存储协议，例如 SCSI，SAS，SATA，FC 和其他协议，并且这些协议指定的存储系统和数担中心其他组件的通信规范可确保整个存储系统高效可靠地运行。随着存储技术的发展由固态电子存储芯片阵列制成的固态驱动器（SSD）在当前以作为数据存储的常见方式，且大部分 SSD 以 SATA 接口的方式与主机接口作出对接通过执行协议的方式完成传输。SATR3.1 在理论上的最优传输速度可达 60OMB/ s(实际速度为 560MB / s)。

为解决传输速率限制问题,NVM Express 自 2009 年开始致力于 NVMe 研究,并于 2011 年 3 月发布 NVMe1.0 版,至今已更新至 1.4 版本。作为一种有效的存储协议,NVMe 已得到许多闪存,服务器和存储供应商的支持,并且生态系统正在逐步成熟。NVMe 通过提供一种通过 PCIe 访问 SSD 的标准方法,从而提供了安全的端到端数据保护支持,从而减少了延迟并简化了指令集,从而显着提高了性能。

NVMe 是现阶段对非易失性固态存储架构开展优化的具体标准协议,在协议中对寄存器接口、指令集、功能机作出定义,同时全面结合存储介质特征与数据实际传输效应,对路径与方法作出优化,进而提升数据传输质量,使得数据访问延迟现象得以明显减少。

1.1.1 PCIe 接口

NVMe 物理层基于 PCIe 通道进行数据传输,PCIe 是具有带宽可扩展性的全双工串行总线,总线版本与吞吐量之间的关系如表 1.1 所示。

在 NVMe 标准协议中,PCIe 总线规范可以视作为最为重要的通信接口,对应层次结构可以包括事务层、数据链路层、物理层三部分,其中事务层为数据交互层。控制器生成 TLP(事务层数据包)数据包,并将数据包传输至数据链路层。携带设备端数据信息与相关命令的 TLP 包则发送至主设备,由此可见 NVMe 命令执行时,单次读写均需要进行多次数据传递。

1.1.2 NVMe 队列

NVMe 协议主要基于提交队列 SQ(Submit Queue)和完成队列 CQ(Complete Queue)。每个主机 CPU 核都有自己独立的提交队列(SQ)、完成队列(CQ)和相关的门铃(SQ DB 和 CQ DB)。提交队列（SQ）是具有固定大小的循环缓冲区，主机软件使用该缓冲区提交命令以供控制器执行。完成队列（CQ）也是一个循环缓冲区，具有固定大小，用于发布已完成命令的状态。

表 1.1 总线版本与吞吐量之间的关系

PCIe 版本	编码方式	Lane-吞吐量(MB/s)		
		×1	×4	×16
1.0	8b/10b	250	1 000	4 000
2.0	8b/10b	500	2 000	8 000
3.0	128b/130b	985	1 969	15 754
4.0	128b/130b	1 969	3 938	31 508

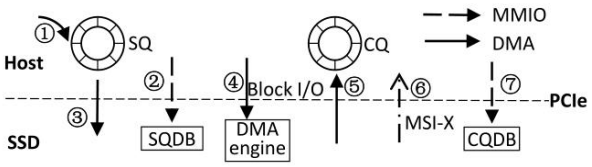

图 1.1 NVMe 命令处理

完成的命令由关联的 SQ 标识符和由主机软件分配的命令标识符的组合唯一地标识。SQ DB 存储 SQ 的尾值，而 CQ DB 存储 CQ 的头值。作者用图 1.1 来简要介绍它的数据传输机制。

主机首先将输入/输出命令放在空闲的 SQ 插槽中(1)，然后用新的尾值更新 SQ DB，以通知固态硬盘传入的命令。然后固态硬盘获取命令(3)并从主机传输数据(4)。命令执行完毕后，固态硬盘会在 CQ (5)的空闲插槽中放置一个 CQ 条目，然后向主机(6)生成一个中断。主机取走新的 CQ 条目，然后用新的头值写入 CQ DB，以指示 CQ 条目已被使用(7)。如作者所见，一个输入/输出请求至少需要 2 个 MMIO、2 个队列 DMA、1 个块 I/O 和 1 个中断请求(例如，MSI-X)。

1.1.3 崩溃一致性

崩溃一致性指的是在突然系统崩溃(如停电)的情况下持续更新持久数据结构。它是存储系统面临的一个基本和具有挑战性的问题。提供崩溃一致性会导致昂贵的性能开销，并进一步阻止系统软件充分利用快速存储设备。目前的研究存在一个关键问题:来自硬件和软件边界(即，NVMe 驱动程序)的低效率阻止了软件栈进一步提供更高的性能。

1.1.4 写依赖性

写依赖性指示了要在存储介质中持久存储的数据块的特定顺序，并且它进一步支持各种技术(例如，日志记录、数据库事务)来在输入输出堆栈中提供排序保证。写顺序是通过一种昂贵的方法实现的，称为独占 IO 处理。在独占

输入输出处理中，直到前一个输入输出请求通过 PCIe 总线传输，然后由设备控制器处理，最后返回完成响应，才能处理后面的输入输出请求。

2. 相关研究

本节主要是对三篇关于NVMe设备I/O路径优化的相关文献[1][2][3]进行简单讲述与介绍，并列出文献中作者观测得到的结论以及实验后的研究成果。

2.1 降低写依赖开销实现优化

2.1.1 介绍

本论文出自 OSDI 20[1]，主要是提出了一种称为 HORAE 的新 IO 堆栈，以减轻高性能驱动器的写依赖开销。写依赖指示了要在存储介质中持久存储的数据块的特定顺序，并且它进一步支持各种技术(例如，日志记录、数据库事务)来在输入输出堆栈中提供排序保证。然而，写顺序是通过一种昂贵的方法实现的，在本文中称为独占 IO 处理。在独占 IO 处理中，直到前一个输入输出的请求通过 PCIe 总线传输，然后由设备控制器处理，最后返回完成响应，才能处理后面的输入输出请求。

这种一次一个输入输出的处理方式与 NVMe 堆栈的高并行性相冲突，并进一步抵消了多个设备之间的并发潜力:首先，它连接了 NVMe 固态硬盘的多个硬件队列，从而消除了主机端和设备端内核的并发处理。此外，它将对物理上独立的驱动器的访问序列化，阻止应用程序享受聚合设备的好处。在本文动机研究中，作者观察到随着硬件队列和设备的扩展，写相关性带来的性能损失可能高达 87%。相反，没有依赖性的无顺序写入很容易使 NVMe 固态硬盘的高带宽饱和。

因此，为了在保留依赖性的同时利用 NVMe 固态硬盘的高带宽，作者提出了屏障转换（Barrier Translation），将有序写入转换为无序数据块和描述写依赖性的有序元数据。屏障转换的关键思想是在正常执行和崩溃恢复期间将写相关性维护转移到排序元数据，同时并发调度无顺序数据块。

作者通过使用 Horae 来重新构建现代 IO 栈以具现这个想法。一方面，Horae 将传统的输入输出路径分为两个专用路径，即有序控制路径和无序数据路径。在控制路径中，Horae 使用内存映射 IO (MMIO)将排序元数据直接刷新到设备的持久控制器内存缓冲区(CMB)中，这是 NVMe 固态硬盘控制器上的通用读/写内存区域。另一方面，香然重用经典的IO堆栈(即块层到设备驱动程序到设备)来保持无顺序写入。这种设计也有利于扩展，因为无顺序数据块可以异步(对单个器件)和流水线(对多个器件)方式处理。

现在，有了分开的 IO 路径，作者进一步开发了一系列技术来确保香然的高性能和一致性:1.设计了紧凑的排序元数据，并在 CMB 中有效地组织它们；2.Horae 的联合刷新在相关设备上执行并行刷新命令，且 Horae 使用写重定向来打破依赖循环，以强大的一致性保证并行化就地更新；3.对于崩溃恢复，香然重新加载排序元数据，并进一步仅提交有效数据块，但丢弃违反写入顺序的无效数据块。

作者构建了一个名为 HoraeFS 的内核文件系统来测试依赖 POSIX 接口的应用程序，并为分布式存储构建了一个名为HoraeStore的用户空间对象存储。将HoraeFS与ext4和 BarrierFS 进行了比较，在文件系统和应用程序(例如 MySQL）级别分别获得了高达 2.5 倍和 1.8 倍的速度提升。此外还针对 BlueStore 评估了 horestore，显示事务处理性能提高了 2.1 倍。

2.1.2 相关研究

（1）传统基本排序保证方法

现代 IO 堆栈是软件（即块层、设备驱动程序）和硬件（即存储设备）层的组合。每一层可以对写请求重新排序，以获得更好的性能或公平性。由于这种设计，文件系统必须显式地强制执行存储顺序。传统上，文件系统依赖于两个重要步骤：同步传输和缓存隔离（cache barrier）。首先，同步传输要求文件系统处理和传输依赖的数据块，通过每一层将其串行传输到存储接口。然后，为了进一步避免控制器在存储设备层中的写重新排序，文件系统发出缓存条命令（例如，FLUSH），将易失性嵌入缓冲区（volatile embedded buffer）中的数据块排空到持久性存储器。之后，文件系统在下一个请求中重复此处理。通过使用独占 IO 处理交错相关请求，文件系统确保写入请求以正确的顺序持久化。保证存储顺序的基本方法是昂贵的。

（2）加速的排序保证方法

许多技术通过减少同步传输和 cache barrier 的开销，改进了基本方法。

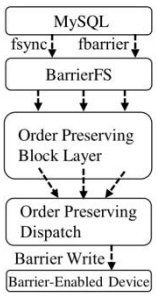

图 2.1 屏障启用 IO 栈

屏障启用 IO 栈（BarrierIO）：BarrierIO 通过在整个 IO 栈中保存排序来减少存储排序开销。具体地说，BarrierIO 主要使用两种技术来强制执行写依赖关系：用于加速同步传输的排序保存和用于改进缓存屏障的 barrier write 命令。图 2.1 中 order-preserving block layer 确保 IO 调度程序遵循文件系统指定的写入顺序。此外，order-preserving dispatch 维护（单个）硬件队列中排队的请求的写入顺序。作为合作者，存储控制器还以序列化方式获取和服务请求。其次，BarrierIO 用一个轻量级的 barrier write 命令取代了昂贵的刷新，以使存储控制器保持写入顺序。

文件系统为应用程序提供 fsync( )调用，以对其写入请求进行排序。然而，通过 fsync( )保持顺序仍然太昂贵。因此，与 OptFS 一样，BarrierIO 将顺序与耐久性分开，并进一步引入了文件系统接口 fbarrier( )，以保证顺序。fbarrier( )按顺序写入关联的数据块，但返回时没有持久性保证。

2.1.2 Horae 设计

（1）基本思想

作者将文件系统或应用程序发出的一系列有序写入请求称为写入流。通常，一个写流可以有多组数据块，和作为两组数据块之间的排序点的中间屏障。作者将设备 a 的一组数据块单调集 ID $_x$ 标记为 A_x，并将屏障（写依赖）称为≤。因此，对于写入流，应确保 A_x 的数据块在(<) B_{x+1} 前必须持久化或与 B_{x+1} 同时（=）持久化。

接下来，作者继续删除两个写请求之间的依赖关系（即≤）。作者使用{}对一组可以并发处理的独立写请求进行分组。作者的关键问题是将 Ax≤Bx+1 转化为{Ax，Bx+1}。作者将写流的索引与其数据内容分离。索引（即，排序元数据）保留了一组最小的信息，以保留依赖关系。作者将排序元数据记为 $\widetilde{A}_x$，数据内容记为 $\overline{A}_x$，$A_x = \widetilde{A}_x \cup \overline{A}_x$。具体来说，如果 A_x 从逻辑块地址(lba) m 到设备 A 有 n 个长度的连续数据块，则 A_x = {A，m，n}。

给定写入流 $A_x \le B_{x+1}$，屏障转换将其转换为 $\widetilde{A}_x \le \widetilde{B}_{x+1} \le \{\overline{A}_x, \overline{B}_{x+1}\}$。具体来说，转换后的写流分两步保证顺序:（1）$\{\widetilde{A}_x, \widetilde{B}_{x+1}\} \le \{\overline{A}_x, \overline{B}_{x+1}\}$；（2）$\widetilde{A}_x \le \widetilde{B}_{x+1}$。首先，作者必须确保排序元数据的持久化不晚于数据内容。然后，提取原始写入流的写依赖性作为排序元数据。

（2）具体设计

Horae 扩展了通用 IO 堆栈，使用一个针对有序写的排序层。此外，Horae 将排序控制从数据流中分离出来:排序层将有序的数据写(①)转换为有序的元数据(②)加上无序的数据写(③)。排序元数据通过一个专用的控制路径(MMIO)。如往常一样，无顺序数据写入流经原始块层和设备驱动程序(块 IO)。作为分离的结果，数据路径不再受写依赖关系的限制，从而允许文件系统将数据块分派给专用硬件队列或存储设备，不占用硬件资源。

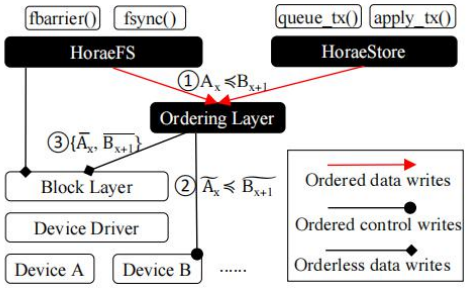

图 2. 2 Horae IO 栈架构

Horae 的关键是排序层。Horae 使用排序队列结构将排序元数据存储在存储设备的持久控制器内存缓冲区中。Horae 利用 epoch 对一组没有内部依赖关系的写操作进行分组，并进一步使用排序队列结构本身来反映具有相互依赖关系的每个 epoch 的顺序。此外，这种设计使 Horae 能够执行并行刷新，尽管多个设备之间存在依赖性。

然而当多个就地更新(IPU)在同一个地址上操作时，就会出现依赖循环。由于 Horae 的数据路径是完全无顺序的，多个有序的进行中(由文件系统发出，但未返回完成响应)的 IPUs 可以共存并自由重新排序。如果处理不当，依赖循环会引入数据版本问题(例如，前一个请求覆盖后一个请求)。Horae 通过写重定向打破了依赖循环，将 IPUs 视为版本化写入，串行存储它们的排序元数据，并同时将它们重定向到预先保留的磁盘位置。在后台，Horae 将重定向的数据块写回其原始目的地。通过这种方式，Horae 在保持秩序的同时，平行于 IPUs。

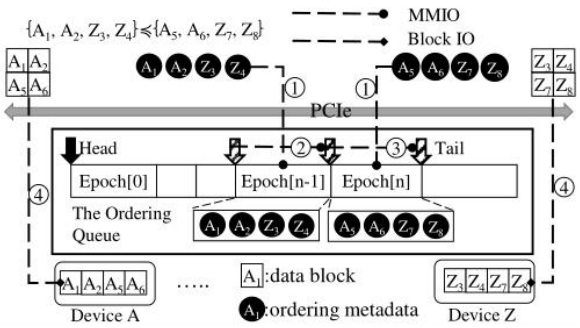

图 2.3 循环排序队列组织

如图 2.3 所示，通过 PCIe 基址寄存器，Horae 使用持久控制器内存缓冲区(CMB)作为持久循环排序队列。排序队列由头指针和尾指针界定，并存储一个写流的排序元数据。对于传入的有序写入请求，Horae 首先通过 MMIO 将其排序元数据附加到队列中。通过控制路径存储排序元数据是具有字节寻址能力的中央处理器到设备的传输；与基于中断的内存到设备的 DMA 传输不同，它不传输完整的块或开关上下文。因此，通过 MMIO 持久化紧凑排序元数据是有效的。Horae 利用 epoch 对一组独立的写操作进行分组。Horae 利用 epoch 对一组独立的写操作进行分组。而且，Horae 只强制执行以 epoch 为单位的写依赖，用 etag 来表示 epoch 的边界。

2.1.3 总结

本文中，作者对单个和多个设备设置中的多队列驱动器的写相关性问题进行了研究。结果表明，在确保写相关性方面有相当大的开销。

本文建议使用屏障转换将排序元数据与相关写入分离，以强制执行正确的写入顺序。作者推出了一个名为 Horae 的新 IO 堆栈，以解开单个物理设备和存储阵列的写依赖性。它引入了专用路径来控制写入顺序，并使用联合刷新和写入重定向来确保高性能和一致性。

作者将一个内核文件系统和一个用户空间对象存储应用于 Horae，并进行了各种各样的实验，显示出显著的性能改进。

2.2 利用校准中断优化存储性能

2.2.1 介绍

本论文出自 OSDI 21[2]，作者提出利用校准中断在尽快完成对延迟敏感的请求与延迟中断来优化吞吐量中做出权衡。

首先，请求完成后，输入/输出设备必须决定要么通过立即触发中断来最小化延迟，要么通过延迟中断来优化吞吐量，预计很快会有更多的请求完成，并帮助分摊中断成本。设备采用自适应中断合并试探法，试图在这些相反的目标之间取得平衡。由于设备缺乏关于哪些输入/输出请求对延迟敏感的语义信息，这些启发式方法有时会导致灾难性的结果。

最初作者为 NVMe 实现了自适应合并，这是一种动态的、仅设备端的方法，它试图根据工作负载调整批处理，但发现它仍然会给请求增加不必要的延迟。因此作者指出设备端启发式方法，如作者的自适应合并方案，无法实现最佳延迟，因为设备缺乏语义上下文来推断请求者的意图:请求是延迟敏感的，还是请求者并行完成的一系列异步请求的一部分？将这一重要信息发送到设备可以弥合语义鸿沟，并使设备能够在适当的时候中断请求者。作者称这种技术为校准中断(或简称为 cinterrupts)，通过向发送到设备的请求添加两位来实现。然后，设备“校准”其中断，以完成对延迟敏感的请求。研究表明在内核和应用程序中表达这些语义只需要对设备接口进行适度的两位更改。校准后的中断可将吞吐量提高 35%，将 CPU 消耗降低 30%，并将中断合并时的延迟降低 37%。

2.2.2 研究背景

NVMe 规范对存储设备的中断合并的思想进行了标准化，其中只有当完成队列中有足够的项目阈值时或在超时后才会触发中断。NVMe 中断合并有两个关键问题。首先，NVMe 只允许聚合时间以 100 秒的增量设置，而设备的延迟接近 10 秒。这种高延迟放大的风险使得超时在工作负载未知的通用部署中不可用。其次，即使 NVMe 聚合粒度更合理，阈值和超时都是静态配置的(当前的 NVMe 标准及其实现没有自适应合并)。这意味着在工作负载发生微小变化后，中断合并很容易中断。

2.2.3 Cinterrupt 设计

cinterrupts 的核心观点:由于请求者和设备之间存在语义鸿沟，设备级的合并启发式算法总是会失败，因为设备会看到一个请求流，无法确定哪些请求需要中断才能解除对应用程序的阻塞。为了弥合语义鸿沟，发出输入/输出请求的应用程序应该总是在设备希望被中断时通知它。

（1）自适应合并

cinterrupts 中的自适应合并策略观察到，一个设备应该为一个突发生成一个中断，或者为一个间隔时间在某个界限内的请求序列生成一个中断。

图 2.4 显示了自适应策略。突发检测发生在第六行，每次新的完成到达时，超时都会被推出。为了限制请求延迟，自适应策略使用的 thr 是它将合并成一个中断的最大请求数(第 14-15 行)。

自适应策略虽然增加了Δ延迟来确定突发的结束，但仍能准确检测突发;如果没有额外的信息，这种延迟是不可避免的。

Algorithm 1: Adaptive coalescing strategy in cinterrupts

```
1  Parameters: Δ, thr
2  coalesced = 0, timeout = now + Δ;
3  while true do
4      while now < timeout do
5          while new completion arrival do
               /* burst detection, update timeout */
6              timeout = now + Δ;
7              if ++coalesced ≥ thr then
8                  fire IRQ and reset;
       /* end of quiescent period */
9      if coalesced > 0 then
10         fire IRQ and reset;
11     timeout = now + Δ;
```

图 2.4 自适应策略

（2）Urgent 标志

Urgent 用于为单个请求请求中断:设备将为任何带 Urgent 注释的请求生成一个立即中断。Urgent 的主要用途是使设备能够校准延迟敏感请求的中断。

为了演示 Urgent 的有效性，作者使用 fio 运行了一个合成的混合工作负载，其中有两个线程:一个通过 libaio (iodepth=16)提交 4kb 的读请求，另一个通过 read 提交 4kb 的读请求，这个线程会阻塞，直到系统调用完成。在 cinterrupts 中，对延迟敏感的读请求被标注为 Urgent，它被嵌入到发送到设备的 NVMe 请求中。

没有 cinterrupts，来自任何线程的请求对于设备来说都是不可区分的。默认(不合并)策略通过为每个请求生成一个中断来解决这个问题，导致的中断比 cinterrupt 多 2.7 倍。

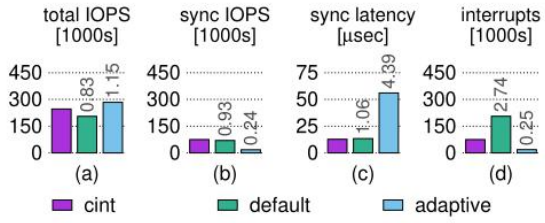

图 2.5 Urgent 标志效果。两个线程运行混合工作负载的合成工作负载:一个线程通过 read 提交同步请求，一个线程通过 libaio 提交异步请求。

（3）Barrier 标志

cinterrupts 使用 Barrier 标记了一批请求的结束,指示设备在所有 Barrier 之前的请求完成后，立即产生一个中断。Urgent 和 Barrier 在语义上的区别是，当 Urgent 请求完成时，会立即产生 Urgent 中断，而 Barrier 中断可能需要等待，在请求的顺序被打乱的情况下。

为了证明 Barrier 的有效性，作者在同一个内核上运行了一个不同数量线程的实验，其中每个线程通过 libaio 进行 4 KB 的随机读取，提交的批处理大小固定为 4。这个技巧是在没有额外开销的情况下确定批处理的结束，这只可能在 cinterrupt 中实现:作者修改 fio，用 Barrier 标记每个批处理中的最后一个请求。图 2.6 显示了吞吐量、延迟、CPU 利用率和中断速率。

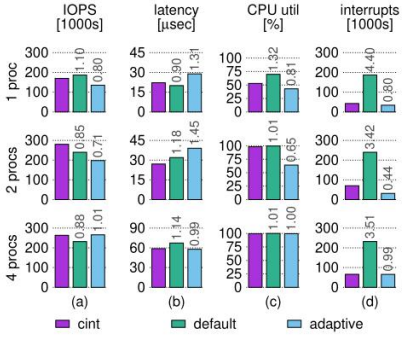

图 2.6 Barrier 标志效果。每次提交 4 个请求的批次，在前一批请求完成后提交一个新的批次。

（4）无序 Urgent（out-of-order Urgent）

全中断的中断生成策略如图 2.7 所示。请求要么没有标记，要么被标记为紧急或障碍。未标记的请求由底层的自适应算法处理，当然也会附带由 Urgent 或 Barrier 产生的中断。紧急请求有时会与其他请求一起完成，这增加了它们的延迟，因为中断处理程序直到接收到中断上下文中的所有请求时才返回。为了解决这个问题，cinterrupts 实现了无序(OOO)处理，这是一种针对紧急请求的驱动程序级优化。

未标记的请求将不会被获取，直到完成批次仅包含那些请求，驱动也不写 CQ DB，直到它完成一系列连续的条目。thr 确保最终获得非紧急请求。

OOO 处理的代价是产生的中断数量增加。图 2.8 报告了运行与图 2.5 相同的混合工作负载的性能指标。OOO 处理产生 2.4 倍数量的中断，以便将同步请求的延迟减少近一半。额外中断的影响显而易见，异步 IOPS 的数量减少了。

Algorithm 2: cinterrupts coalescing strategy

```
1  Parameters: Δ, thr
2  coalesced = 0, timeout = now + Δ;
3  while true do
4      while now < timeout do
5          while new completion arrival do
6              timeout = now + Δ;
7              if completion type == Urgent then
8                  if ooo processing is enabled then
                       /* only urgent requests */
9                      fire urgent IRQ;
10                 else
                       /* process all requests */
11                     fire IRQ and reset coalesced;
12             if completion type == Barrier then
13                 fire IRQ and reset coalesced;
14             else
15                 if ++coalesced ≥ thr then
16                     fire IRQ and reset coalesced;
       /* end of quiescent period */
17     if coalesced > 0 then
18         fire IRQ and reset coalesced;
19     timeout = now + Δ;
```

图 2.7 Cinterrupt 合并策略算法

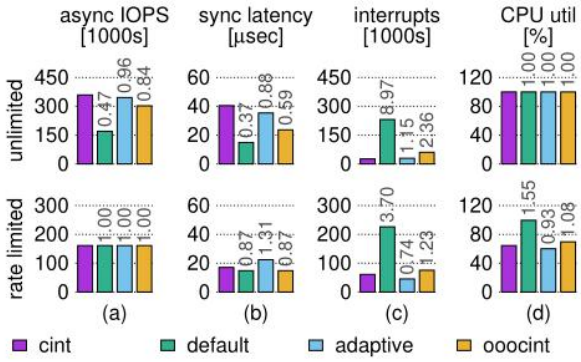

图 2.8 运行与图 2.5 相同的混合工作负载时的性能指标。OOO 处理产生 2.4 倍的中断数量，以便将同步请求的延迟减少近一半

2.2.4 总结

本文表明现有的 NVMe 中断合并 API 对实际的合并造成了严重的限制。除了为 NVMe 设计自适应合并策略，作者的主要见解是软件指令是设备产生中断的最佳方式。Cinterrupts 结合了 Urgent、Barrier 和自适应突发检测策略，能够在工作负载需要时准确生成中断，使工作负载即使在动态环境中也能获得到更好的性能。因此 cinterrupts 使软件栈能够充分利用现有和未来的低延迟存储设备。

2.3 崩溃一致性的 NVMe 优化

2.3.1 介绍

本论文出自 SOSP 21[3]，研究方向是崩溃一致性的 NVMe 优化。本文介绍了崩溃一致性非易失性内存快速(ccNVMe)，它是 NVMe 的一种新的扩展，定义了主机软件如何通过 PCI Express 总线与非易失性内存(例如固态驱动器)通信，同时兼顾崩溃一致性和性能效率。

现有的存储系统在崩溃一致性上付出了巨大的代价，因此无法充分利用 NVMe 接口的多队列并行性和低延迟。ccNVMe 通过将崩溃一致性结合到数据传输来缓解这一主要瓶颈。与使用复杂更新协议和重量级块 I/ o 的传统系统不同，这种新思想允许存储系统通过 NVMe 的数据传输机制，只使用两个轻量级内存映射 I/ o (MMIO)来实现崩溃一致性。

ccNVMe 引入了事务感知的 MMIO 和门铃，以减少 PCIe 流量，并提供原子性。作者还提出了如何在 ccNVMe 之上构建一个高性能和与崩溃一致的文件系统，即 MQFS。实验表明，与最先进的文件系统和没有日志记录的 Ext4 相比，MQFS 使 RocksDB 的 IOPS 分别提高了 36%和 28%。

2.3.2 研究背景

在现代 NVMe ssd 上的崩溃一致性研究中，日志记录是许多文件系统用来解决崩溃一致性问题的流行解决方案，包括 Linux Ext4、IBM 的 JFS、SGI 的 XFS 和 Windows 的 NTFS。在 Ext4- nj 设置中，本文禁用了 Ext4 的日志记录，并假定它是现代 NVMe ssd 上 Ext4 的理想上限。使用 FIO 工具，作者启动多达 24 个线程，每个线程对其私有文件执行 4 KB 的追加写操作，然后分别执行 fsync。图 2 显示了总体结果。Ext4- nj 和 Ext4(或 HoraeFS)之间的差距量化了崩溃一致性开销。

随着 NVMe ssd 的发展，崩溃一致性开销变得非常重要，并且趋于严重，如图 2.9 (b)-(c)所示。值得注意的是，在图 2.9 (c)的 24 核情况下，崩溃一致性开销(即(Ext4-NJ - HoraeFS)与 HoraeFS 的比率)接近 66%。除了 Ext-NJ 之外，所有文件系统都无法充分利用可用带宽。进一步的分析表明，低效率是由软件开销和 PCIe 流量造成的。

软件开销:许多基于 ext4 的文件系统，包括 HoraeFS 和 BarrierFS，都使用单独的线程来分派日志块，以实现排序和一致性。单个 CPU 核心的计算能力对于旧的驱动器是足够的，但是对于新的快速驱动器是不够的。此外，应用程序和日志记录线程之间的上下文切换引入了不可忽略的 CPU 开销。

PCIe 流量:为了实现 n4 KB 数据块的原子性，日志为单个事务生成两个额外的块(即日志描述和提交块)。这种方法需要 2×(N+2) mmio, 2×(N+2) dma 从/到队列条目，(N+2)块 I/ o 和(N+2)中断请求(如果块合并被禁用的话)。当 SSD 完全由具有足够 CPU 核的软件栈驱动时，应用程序

的性能反而会受到软件和硬件边界的限制。由于设备驱动程序消耗了大量的带宽，因此提供给文件系统的可用带宽是有限的。

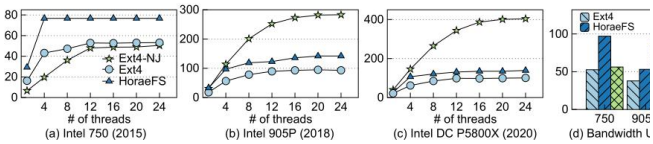

图 2.9 动机测试结果

PMR(持久存储区)是 NVMe 的一个新特性。它是固态硬盘的通用读/写永久内存区域。固态硬盘可以通过暴露一部分可由中央处理器加载和存储指令访问的永久存储器(例如，电容支持的动态随机存取存储器或光平面存储器)来启用该功能。

2.3.2 ccNVMe 设计

ccNVMe 的关键思想是将崩溃一致性与数据传播结合起来。最初的 NVMe 已经将请求记录在提交队列中，并将它们的状态记录在门铃中；按提交队列门铃(SQDB)表示请求即将被提交给 SSD，同时按下完成队列门铃(CQDB)提示这些请求已经完成。这两个门铃(状态)自然代表原子性的“0”(无)和“1”(全)状态。ccNVMe 使提交队列在突然崩溃的情况下保持持久，并以事务而不是请求为单位按门铃，让事务的请求达到相同的状态(例如，全部或没有)，从而实现原子性。

图 2.10 展示了 ccNVMe 的概述。在图的最左边，ccNVMe 是一个位于块层和存储设备之间的设备驱动程序。ccNVMe 在硬件和软件层的边界上提供原子性保证。这种设计有两个主要优点。首先，ccNVMe 允许原子性免费使用快速 NVMe 队列和门铃操作，从而加速了崩溃一致性的保证。其次，ccNVMe 提供了通用的原子原语，它可以使上层系统不需要实现复杂的更新协议，也不需要考虑正确性和一致性。例如，应用程序可以直接发出原子操作，或者使用经典的文件系统 api(例如，write)，再加上 fsync 或者一个新的文件系统原语 fatomic 来确保失败的原子性。在图 3 的右侧，ccNVMe 保持了原始 NVMe 的多队列设计；如果底层的 SSD 提供了足够的硬件资源，每个 CPU 核心都有自己独立的 I/O 队列(比如 SQs 和 CQ)，门铃和硬件上下文(比如中断处理器)。

ccNVMe 在 NVMe SSD 的持久内存区域(PMR)中创建持久提交队列(P-SQ)和相应的门铃(P-SQDB)。ccNVMe 在接收原子操作时，会向 P-SQ 下发 ccNVMe I/O 命令，并对 P-SQDB 进行环行。在通常情况下，原子性仅由两个 mmio 实现。作者利用 NVMe 通用命令的保留字段来设计 ccNVMe I/O 命令;这使得 ccNVMe 与 NVMe 兼容。因此，存储设备可以直接从 P-SQ 获取 I/O 命令，而不需要进行任何逻辑更改。

（1）事务:基本操作单元

SQ 的每个条目表示对连续的逻辑块地址范围的请求。ccNVMe 通过一个特殊的 attributeREQ_TX 来区分原子请求和非原子请求。带有 req_tx_commit 的特殊原子请求作为事务的提交点。因此，如果 SSD 中存在不稳定的写缓存，提交请求会隐式地刷新设备以确保持久性，首先发出一个 flush 命令，然后在 I/O 命令中设置 FUA 位。ccNVMe 将这些属性嵌入到 I/O 命令中的一个保留字段中，并根据类别对这些原子请求进行不同的处理。

ccNVMe 将一组请求分组为一个事务，并为每个事务分配一个唯一的事务 ID。这个 ID 用于事务的唯一标识，以及决定跨多个提交队列的持久性顺序。

（2）事务感知的 MMIO 和门铃

ccNVMe 使用持久化的 MMIO 写操作向 P-SQ 插入原子请求并环化 P-SQDB，这与 NVMe 使用非持久化的 MMIO 写操作不同。MMIO 写入是由 CPU 存储指令直接执行的。在这里，由于 P-SQ 结构是在一个循环日志中组织的，所以 ccNVMe 利用 CPU 的 WC 缓冲模式将连续的写合并成一个更大的写突发，从而提高内存和 PCIe 访问的效率。为了确保持久性，ccNVMe 通过两个步骤使用 MMIO 刷新。首先，clflush 和 mfence 被用来刷新到 PCIe 根节点的 MMIO 写。其次，利用 PCIe 的顺序，即读请求不能通过一个已发布的请求，ccNVMe 发出一个额外的零字节长度的 MMIO 读请求，以确保 MMIO 写最终到达 PMR。

原始 NVMe 使用非持久 mmio，可以将提交队列放置在主机内存中。因此，它更新提交队列并以一种相对积极的方式按门铃:每当一个请求插入到 NVMe 提交队列时，它就立即按门铃。然而，在需要持久 mmio 并以事务为单

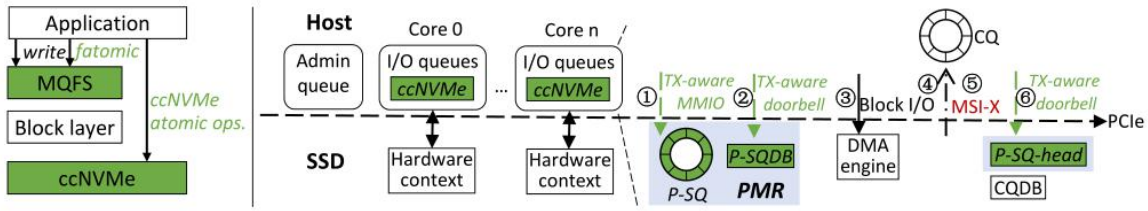

图 2.10 ccNVMe 设计

位进行操作的 ccNVMe 中，按每个请求的门铃会导致相当大的开销。ccNVMe 引入了事务感知的 MMIO 和门铃，用于发送请求和按门铃。这里的关键思想是推迟 MMIO 刷新(即图 4(a)中的 2 和 3)和门铃，直到事务被提交。图 4(b)描述了一个示例。假设一个事务由两个请求 Wx−1 和 Wx−2 组成。在步骤 1 中，Wx−1 是一个普通的原子请求，REQ_TX 首先出现，ccNVMe 使用 CPU 存储指令存储它。当收到 REQ_TX_COMMIT 提交请求时，ccNVMe 触发 MMIO 刷新。在步骤 2 中，ccNVMe 使用缓存行刷新和 PCIe 读取将队列条目持久化为 P-SQ。最后，在第 3 步中，ccNVMe 通过设置 P-SQ 的尾部指针来环绕 P-SQDB。

（3）正确性和崩溃恢复

正常执行。在 ccNVMe 中，请求在事务单元中提交和完成。这是通过感知事务的门铃机制实现的。特别是，ccNVMe 持有相同的假设，即按门铃(即向一个 4b 地址写入一个值)本身是一个原子操作，就像在原始 NVMe 中一样。ccNVMe 在更新 P-SQ 和 CQ 条目后自动按门铃(即图 3 的步骤 2 和步骤 6)。因此，事务以原子的方式提交和完成。

崩溃恢复。在崩溃恢复期间，ccNVMe 会找到未完成的事务，并将特定的恢复算法(例如，回滚)留给上层系统。特别是在突然断电的情况下，PMR 的 P-SQ、P-SQDB、P-SQ-head 等数据保存在 SSD 的持久化介质(如 flash memory)的备份区域。当恢复供电时，数据被加载回 PMR。然后，ccNVMe 在 NVMe 探测期间执行崩溃恢复;它为上层系统提供了用于恢复的未完成事务。具体来说，从 P-SQ-head 到 P-SQDB 的 P-SQ 事务都是未完成的。ccNVMe 在内存中复制这些未完成的事务;因此，上层系统可以使用该副本进行恢复逻辑(例如，重放已完成的事务并丢弃未完成的事务)。由于 ccNVMe 总是以原子的方式按顺序完成事务，因此它在崩溃恢复后保持正确的持久性顺序。ccNVMe 不保证任何跨多个硬件队列的全局顺序;它仅通过提供持久事务 ID 字段来帮助上层系统处理全局顺序。上层系统可以在该字段中嵌入全局顺序，以决定恢复时的持久化顺序。

4. 总结

现有的 NVMe 中断合并 API 对实际的合并造成了严重的限制，而软件指令是设备产生中断的最佳方式。但设备缺乏语义上下文来推断请求者的意图:请求是延迟敏感的，还是请求者并行完成的一系列异步请求的一部分。因此提出了 ccNVMe，这是一种在存储系统中同时实现高性能和崩溃一致性的新方法。通过将崩溃一致性与数据传播相结合，并将原子性与持久性相分离，ccNVMe 只需两个轻量级 MMIOs 即可确保原子性保证，从而提高性能。作者引入了 MQFS 来充分利用 ccNVMe，表明 MQFS 成功地用更少的 CPU 内核饱和了固态硬盘的带宽，并优于最先进的文件系统。

5. 参考文献

[1] Liao X, Lu Y, Yang Z, et al. Crash Consistent Non-Volatile Memory Express[C]//Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. 2021: 132-146.

[2] Tai A, Smolyar I, Wei M, et al. Optimizing Storage Performance with Calibrated Interrupts[C]//15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21). 2021: 129-145.

[3] Liao X, Lu Y, Yang Z, et al. Crash Consistent Non-Volatile Memory Express[C]//Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. 2021: 132-146.