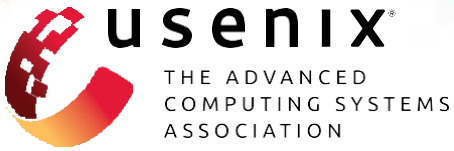


NyxCache:灵活高效的多租户持久性内存缓存

*威斯康星大学麦迪逊分校的*Kan Wu、Kaiwei Tu和Yuvraj Patel；*微软的*Rathijit Sen和Kwanghyun Park；*威斯康星大学麦迪逊分校的*Andrea Arpaci-Dusseau和Remzi Arpaci-Dusseau

https://www.usenix.org/conference/fast22/presentation/wu

这篇论文被收录在《国际学术会议论文集》中。

第20届USENIX文件和存储技术会议

开放访问第20届USENIX文件和存储
技术会议论文集
是由USENIX赞助的。

NyxCache:灵活高效的多租户持久性内存缓存

Kan Wu, Kaiwei Tu, Yuvraj Patel, Rathijit Sen† , Kwanghyun Park† , Andrea C. Arpaci-Dusseau, Remzi H.Arpaci-Dusseau
威斯康星大学麦迪逊分校† Microsoft

摘要。我们提出了NyxCache（Nyx），一个用于多租户持久性内存（PM）缓存的访问调节框架，它支持轻量级访问调节、每个缓存的再源使用估计和缓存间干扰分析。通过这些机制和现有的准入控制和容量分配逻辑，我们建立了重要的共享政策，如资源限制、QoS意识、公平减速和比例共享。Nyx资源限制可以准确地限制每个缓存的PM使用，提供比带宽限制方法好5倍的performance隔离。Nyx QoS可以为延迟关键型缓存提供QoS保证，同时为没有干扰的最佳努力型缓存提供更高的吞吐量（与以前基于DRAM的方法相比，最高可达6）。最后，我们表明Nyx对于现实的工作负载是有用的，可以隔离写峰值，并确保重要的缓存不会因为增加的尽力流量而减慢。

1 简介

基于内存的旁观键值缓存（如memcached[14]）是许多系统和应用的一个重要组成部分[3, 5, 23, 74]。为了提高利用率和简化管理，多个缓存实例经常被合并到一个多用户服务器上。例如，Facebook[54]和Twitter[74]各自维护着数百个专用的缓存服务器，承载着成千上万的缓存实例。然而，多租户服务器有一个额外的挑战，即确保每个客户端缓存满足其性能目标；一系列生产和研究的内存多租户缓存目前提供了不同的共享策略，如强制限制使用的内存容量和带宽[7]，保证服务质量（QoS）的水平[18]，并按比例分配资源[60]。

持久性内存（PM），如英特尔的Optane DC PMM[10]，正在成为这些缓存的一个有吸引力的构建块，因为PM的容量大，每字节成本低，而且性能与DRAM相当。然而，PM的性能在一些方面与DRAM和Flash不同，这降低了目前其他设备的多租户缓存的有效性[34, 62]。特别是，与DRAM不同，Optane DC PMM表现出高度不对称的读与写性能（对于单个DC PMM，最大读带宽为6.6GB/s，而最大写带宽为2.3GB/s）[45]，读与写之间的干扰很小且不公平（以1GB/s的速度写入会导致共同运行的读工作负载的吞吐量和P99延迟减慢，与以8GB/s读取相同）[55]，以及对256B的倍数访问特别有效[73]。

不幸的是，现有的多租户DRAM和存储缓存技术并不容易转化为PM。一些方法只关注客户之间的容量分配[34, 60, 62]；容量分配是必要的，但对于PM共享来说是不够的，因为对PM的请求率也必须被调节。主机级的请求调节已经被广泛地用于闪存设备，使用块层I/O调度[58, 61]，但是考虑到100ns的PM访问，这些软件的过度是不可接受的[24]。器件级的请求调度假设了PM所缺乏的特殊硬件[53, 65, 78, 79]。最后，粗粒度的请求节制是绝大多数DRAM带宽分配技术的基础；然而，这些方法假定硬件计数器和性能特征对PM来说是不成立的（例如，带宽是对利用率的准确估计）。

在本文中，我们介绍了NyxCache（Nyx），这是一个独立的轻量级和灵活的多租户键值缓存的PM访问调节框架，它为今天的PM进行了优化，无需特殊的硬件支持。给定一个PM服务器和共享策略（如QoS），使用现有的负载接纳[36, 37, 52]和容量分配[34, 60, 62]技术对缓存实例进行接纳和分配空间。在运行时，Nyx监控缓存的信息（例如，PM资源利用率），调节每个缓存被允许访问PM的速率，从而执行共享策略的性能目标。Nyx适用于任何遵守memcached接口的内存键值存储[14]；目前的实现包括Twitter的Pelikan[17]的PM优化版本，对于get-heavy工作负载，可以将单缓存性能提高50%以上，对于write-heavy工作负载，可以提高3倍。Nyx的核心贡献是一套为PM设计的软件机制，以提取灵活执行流行的共享政策所需的信息。

Nyx提供了新的机制，以有效地i）调节PM访问，ii）获得客户的PM资源使用情况，iii）分析客户间的干扰，并有两个特别有用和新颖的PM机制。首先，Nyx不仅有效地估计了PM DIMM的总利用率（建立在这一领域的研究工作上[55]），而且还估计了每个缓存实例引起的PM利用率，这是共享政策所需要的；估计PM利用率是具有挑战性的，因为与DRAM不同，传输的字节数不是PM利用率的准确代表。第二，Nyx可以确定哪个缓存实例对另一个缓存的干扰最大；在基于PM的系统中，这些相互作用很难确定，因为与DRAM不同，一个受伤害的客户可能受到低带宽客户的影响比高带宽客户的影响更大。这两种机制都准确地说明了

	资源限制	服务质量	展会放缓	成比例的资源分配
请求监管	3	3	3	3
资源使用情况	3			3
干扰		3		
应用程序速度减慢			3	3

表1：**需要的控制和信息。** 3 *表示政策要求进行控制或提供信息。*表示可选的。*

CPU缓存预取，这对PM上的高性能至关重要。这些新机制使Nyx能够轻松有效地支持共享政策，如资源限制、QoS、公平减速和按比例共享。

Nyx提供的共享策略非常强大。Nyx可以准确地限制每个缓存的PM利用率（类似于谷歌云的memcache[7]），而只测量带宽的方法则不能。Nyx可以为延迟关键型的缓存提供QoS保证，同时为没有干扰的尽力型缓存提供更高的吞吐量（最高可达6）。Nyx可以提供按比例的资源分配，同时将空闲的PM利用率重新分配给不会无意中降低其他客户的速度。最后，正如在Twitter真实的大规模缓存跟踪中所显示的那样，Nyx可以将客户与写入峰值隔离开来，并确保重要的缓存不会因为增加的尽力流量而变慢。

在本文的其余部分，我们评估了以前的多租户缓存及其对PM的限制（§2）；讨论了Nyx的设计（§3）；评估了Nyx机制的开销和其政策的有效性（§4）；讨论了潜在的扩展（§5）；与相关工作进行比较（§6）；并得出结论（§7）。

2 动机和挑战

我们提供了许多内存多用户键值缓存所提供的共享策略的背景，以及实现这些策略所需的机制。我们解释了为什么以前在DRAM或块I/O上提供控制和信息的方法在PM上效果不好。

2.1 多租户缓存的共享策略

内存键值缓存，如memcached[14]、Redis[66]和Pelikan[17]，是许多实时和批量应用的网络基础结构的重要组成部分[3, 74]。在从缓慢的后端存储或计算节点访问数据之前，应用程序首先检查内存中的缓存服务器。在生产环境中，缓存服务器通常是多租户的：许多缓存实例被整合在一台服务器上，以提高利用率并简化管理和扩展[54]。在一个多租户缓存中，请求被路由到相应租户的缓存实例。例如，像Facebook[54]和Twitter[74]这样的大公司拥有数以百计的大内存专用服务器来承载数以千计的缓存实例。较小的公司使用缓存即服务提供者，如ElastiCache[1]、Redis[20]和Memcachier[16]。在本文中，我们专注于管理一个个人多租户缓存服务器。

在多租户缓存中，给竞争的客户，强制执行性能和共享目标是至关重要的。不同的行业和研究多租户系统提供了不同的目标；我们主要关注以下四个方面。

资源限制。当客户为资源付费时，一个常见的目标是保证每个客户不能超过一定的使用量，如带宽、OPS/秒或资源数量[2, 7]。例如，谷歌云的memcache根据定价层限制操作，如"每GB每秒最多10k读或5k写（独占）"[7]。多个资源可以同时被限制，例如，亚马逊ElastiCache[2]对内存和vCPU都收费。

多租户高速缓存执行每个客户的资源限制有两个要求。首先，系统必须准确地确定每个客户正在使用的资源量；我们把这称为*资源使用估算*。其次，如果每个客户的请求超过这个限制，系统必须重新安排或节流，我们称之为*请求调节*。下面（第2.2节），我们描述了以前的多租户缓存是如何提供请求调节和资源使用估算的，以及为什么这些以前的方法对PM来说是不够的。**服务质量。**一个多租户系统可以确保每个客户的性能目标（吞吐量、延迟或尾部延迟）得到满足，而不考虑其他共处的客户，如Twitter[18]和微软[62]中。这个目标对于必须满足服务级别目标（SLO）的延迟关键型客户是很有用的。例如，Twitter的生产缓存提供了一个p999的延迟<5毫秒[18]。

提供QoS需要了解每个客户端在运行时是否满足其目标。当系统观察到一个客户端没有达到它的性能保证时，干扰的客户端会被识别和限制[31, 39, 51]（例如，用*请求调节*）。对于基于DRAM的缓存来说，识别造成最大伤害的客户端通常是直接的，并且是基于简单的带宽[39]，但是对于PM来说不是这样。在PM上需要一种涉及*干扰估计的*新技术来确定工作负载的构成。

除了运行时支持外，保证QoS还需要接纳控制和空间分配。必须对新来的客户进行准入控制，以确保系统有足够的资源，并且新客户不会干扰现有客户[36, 37, 52]。必须对缓存实例进行空间分配，为每个客户提供一个特定的命中率，以确保每个客户都能达到其目标。以前的研究已经关注到了这个挑战。例如，微软[62]分配空间以满足QoS带宽的要求，Robinhood[29]分配空间以最小化尾部延迟。接纳控制和空间分配与PM引入的新挑战大多是正交的，不是我们的重点。

公平的减慢。在更多的合作环境中的多租户系统可以确保所有的客户都以相同的数量放慢速度。从形式上看，这些方法使最大减速与最小减速的比率最小。

缓慢[38,
63]。在网络缓存设置中，应用程序的请求可能呈扇形分布，在这种情况下，具有最长延迟的缓存访问决定了整体延迟[29,
54]；因此，平衡减速有利于整体请求延迟。

执行公平减速需要了解每个缓存在运行时的减速情况。系统必须监控每个缓存在与他人共享服务器时的当前性能，并了解其单独运行时的性能。这就需要一种*慢速估算*的技术。此外，为了平衡不同缓存的慢速，慢速小的缓存应该被进一步限制，慢速大的缓存应该被减少限制（例如，用*请求调节*）。

成比例的资源分配。最后，一个多租户系统可以通过保证*N*个客户中的每一个的性能都在其独立性能的*1/N*以内来激励客户共享资源。这种保证可以被概括为给每个客户一个不同比例的份额。闲置的资源可以在客户之间进行重新分配，这样一些客户就可以获得比其保证更多的资源。例如，FairRide[60]确保按比例分配缓存空间。

为了保证按比例分配，多租户高速缓存必须满足三个要求。系统必须进行*再追求调节*和*资源使用估算*，以保证每个客户的消耗不超过其分配的资源。当把闲置资源分配给客户时，系统必须评估额外的资源使用不会干扰其他客户；因此，系统必须跟踪每个客户的减速情况（即用*减速估算*），并在严重影响某些客户之前停止闲置资源的重新分配。

总之，多租户缓存要提供上述策略，必须控制每个缓存实例的资源使用，并获得有关资源和应用程序性能的信息。表1总结了每个策略所需的控制和信息。

2.2 PM缓存共享的挑战

持久性内存是一个吸引人的键值缓存的构建块。在介绍了持久性内存的背景后，我们描述了将持久性内存用于多租户缓存的挑战。

2.2.1 持久记忆的特点

PM正在成为产品和研究原型的现实。例如，英特尔Optane DC PMM[10]是一种流行的设备；还有一些研究原型[30,49,70]。在本文中，我们用PM来指代Optane DC PMM。PM的性能与DRAM相似，但可以以低成本提供极大的容量[10,
11]。PM的速度明显快于NAND闪存，并且是可字节编址的。PM直接连接到内存总线上，当配置为App Direct模式时，可以使用加载和存储进行访问。对PM的访问存在不同的CPU缓存选项：带有CPU缓存和预取的加载和存储；禁用预取的加载和存储（对PM和DRAM）；完全绕过CPU缓存的非时间性（NT）操作[73]。

表2总结了Optane的带宽和延迟情况。

	公制	负载	无预取	NT-Load	商店	商店+clwb	NT-Store
256B	GB/s	1.59	1.53	0.29	1.12	0.52	3.73
	我们	0.49	0.52	0.84	0.38	0.47	0.08
4KB	GB/s	4.08	2.92	2.24	1.03	1.50	3.44
	我们	1.22	1.69	1.84	4.14	2.71	1.22

表2：**PM加载/存储性能。***该表总结了单线程随机256B和4KB加载/存储操作的吞吐量/延迟（在2个DC PMM上）。无预取：CPU的预取功能被关闭（对于DRAM/PM）；NT：绕过CPU缓存的非时间操作。*

DC PMM的工作负载与键值缓存有关：运行256B和4KB的负载和存储。如图所示，对于负载来说，常规负载表现最好。CPU缓存的预取对于隐藏PM延迟和增加吞吐量是至关重要的。对于随机工作负载的存储，绕过CPU缓存的NT-存储的性能要好得多。因此，我们使用优化的PM内键值缓存来使用常规负载和NT-存储。

PM具有影响多租户缓存的独特特性。例如，正如以前所确定的，PM表现出不对称的读与写的性能[45]，特别是对特定大小（如256B）的有效访问[73]，以及在读和写之间严重和不公平的干扰[55]。正如我们将描述的那样，这些特征深深地影响了执行请求调节和估计资源使用、干扰和应用减速的能力。

2.2.2 请求监管

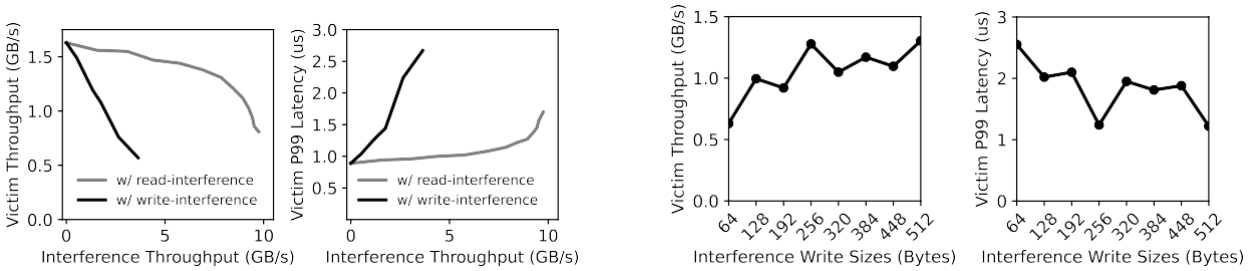
以前的请求调节方法已经为DRAM和块I/O去掉了签名。然而，这些方法都不适合用于PM。

现有的调节内存请求的技术已经调整了专用于一个应用程序的核心数量[39]，使用了时钟调制（DVFS）[57]，以及英特尔内存带宽分配（MBA）[9]。在多租户缓存中，减少核心的数量是不合适的，因为一个缓存的位置通常只分配给一个核心[2]。英特尔MBA管理来自每个核心的最后一级缓存（LLC）失误，以限制内存流量，但是没有区分对PM和DRAM的失误[8]，因此不能在不降低DRAM速度的情况下限制对PM的访问。此外，英特尔MBA无法获得有关资源使用、干扰和应用减速的准确信息，我们将对此进行讨论。同样，调整CPU频率对所有的指令都有影响；Oh等人[55]证明了CPU频率的调整对调节PM流量是无效的。

I/O请求已经通过软件与块层I/O调度[12]进行了调节，这不适合PM，原因有二。首先，块的抽象会给可字节寻址的PM增加大量的读/写放大作用。第二，用合并、重新排序和其他同步化来调度请求，会给本来低延迟的PM访问增加不可接受的开销[24]。

2.2.3 资源使用量估算

以前估计客户的内存或I/O使用时间的技术对PM来说效果并不好。我们描述了以前跟踪I/O的软件方法所存在的问题。



(a) 读与写的　　干扰　　(b)与访问规模有关的干扰

图1：**各种干扰下的PM负载性能。***我们将一个受害者的工作负载（单线程256B负载）与各种干扰放在一起。(a)显示了在不同数量的读和写干扰下，受害人的吞吐量和尾部延迟。(b)显示了与不同访问量的1GB/s存储流量（范围从64B到512B，步长为64B）集中在一起时的受害者性能。*

使用和DRAM的硬件方法。

如上所述，CPU缓存预取是PM提供高带宽和低延迟的必要条件。然而，当估计软件中的块I/O流量时[4, 35, 76]，没有观察到由预取引起的额外PM访问。在运行一个1KB随机负载的实验时，我们发现软件层面的跟踪只占实际内存流量的60%，导致了不准确的资源使用估算。

DRAM上的核算使用硬件计数器来跟踪每个内核的L3高速缓存行的失误。虽然硬件计数器准确地测量了预取，但它们并没有考虑到缓存行大小和PM访问粒度之间的差异，而这正是PM核算所需要的。因为PM有一个256B的最小访问粒度，一个64B的负载（一个L3高速缓存线）与一个256B的负载（四个L3高速缓存线）所使用的PM资源量相同。因此，四个高速缓存线的访问可以导致一到四个PMEM的访问。以前的资源估计系统经常使用带宽消耗作为资源使用的代理[39, 51, 77]，但这并不适合PM，因为操作成本受访问大小的影响，并且对于读和写是不同的。不幸的是，目前PM中的硬件计数器也是不够的；现有的PM计数器是在DIMM介质上的水平，并且不跟踪每个客户或每个核心的使用情况[13, 55]。

2.2.4　干扰估计

在基于内存的方法中，由特定客户引起的干扰被认为与内存带宽有关。例如，Caladan[39]确定了LLC失误次数最多的客户，这直接对应于拥有最高内存带宽的客户。这种简化方法对PM不起作用，因为PM的干扰取决于流量的大小和模式。

具体来说，在PM上，写密集型客户机比相同带宽的读密集型客户机产生更大的干扰，如图1.a所示。例如，在读密集型客户机上，一个竞争性的1GB/s的写与一个竞争性的8GB/s的读造成同样的吞吐量和尾部延迟干扰。如图1.b所示，较小的访问（64B）会比较大的访问（256B）造成更多的干扰。由于PM的最小颗粒度为256B，64B的访问在设备上被放大为256B；因此，在相同的带宽下，64B访问产生的干扰明显多于256B访问。简而言之，与DRAM不同，竞争客户的带宽不能很好地估计PM的干扰。

2.2.5　应用放缓的估计

众多的努力估计了基于DRAM和Flash的系统的减速；然而，所有这些都需要专门的副支持。例如，FST[38]需要在每次内存访问时更新DRAM组内的冲突计数器；MISE[64]和ASM[63]需要DRAM控制器对应用程序请求的优先级进行标识。FLIN[65]改变了闪存控制器来跟踪和重新安排每个闪存交易。尽管在PM上应用程序的速度与DRAM或I/O没有本质上的区别，但以前的方法需要特殊的硬件，而这些硬件在PM上是不可用的。

摘要：多租户PM缓存需要新的方法来调节PM的访问，并提取PM资源的使用情况、干扰信息和应用程序的减速。

3　缓存设计（NyxCache Design

鉴于现有的多租户缓存服务器无法处理PM，我们引入了NyxCache（Nyx）。Nyx提供了控制（如请求节流）和PM信息估计（如资源使用、干扰和应用减速）的机制，并支持一系列的共享策略（如资源限制、服务质量、公平减速和资源使用比例）。我们描述了Nyx的整体结构，提出了我们的设计目标，描述了Nyx如何提供这些机制和政策。

3.1　建筑学

如图2所示，Nyx提供了一个多租户的PM缓存框架。运行Nyx的每个PM服务器可以包含任意数量的缓存实例（例如，memcached、Pelikan、Redis）。成千上万的用户可以向他们相关的缓存实例发送请求（例如，设置/获取）。当缓存空间耗尽时，缓存实例可以使用任何驱逐策略（例如，FIFO、LRU和LFU）。就像其他的旁观者缓存一样，用户明确地将所需的数据写入缓存；Nyx不会在缓存错过时从远程存储中获取数据。

Nyx可以配置不同的共享政策和参数（例如，资源限制、延迟目标或比例）。

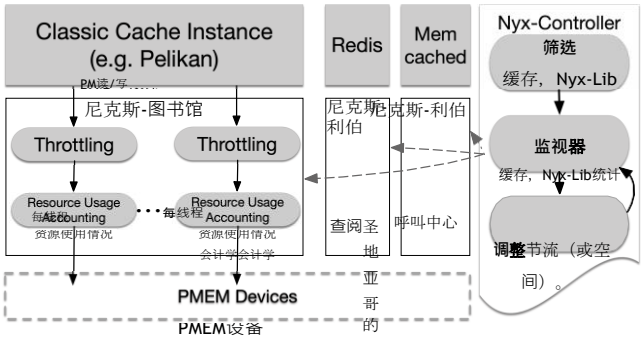


图2：**NyxCache架构。***Nyx为每个缓存实例实现节流和资源使用核算，并执行跨缓存实例的共享策略。Nyx包含两个主要部分。1）每个实例的Nyx库，和2）一个集中的Nyx控制器。*

的重量）。管理员可以使用Nyx提供的控制和信息机制来实现新的政策。在运行时，Nyx执行所需的共享策略。根据Nyx获得的关于每个实例资源使用和性能的信息，Nyx控制器动态地调整分配给实例的节流和空间。

Nyx对高速缓存实例有两个要求。首先，每个缓存实例必须报告应用层面的性能指标，如吞吐量和尾部延迟；大多数系统都有这种能力或可以扩展[15]。第二，这些实例必须与一个可信的Nyx库集成。当一个缓存实例从PM读/写时，它必须使用Nyx库的API（例如，read(dest, src), write(dest, src)）。对于每个PM的访问，Nyx库都会对访问进行节制，跟踪PM的使用，并执行实际的访问。该库使用一个单独的线程与Nyx控制器进行通信。控制器与库相互作用，以查询统计数据和设置配置、空间和节流值。Nyx利用以前多租户内存缓存的技术来实现基本的共享功能，如准入控制和空间分配。到目前为止，Nyx只管理单个NUMA节点上共享PM的缓存实例（而且所有的PM访问都是local）；多个Nyx可以用来管理多个NUMA节点。我们把对NUMA的管理留给未来的工作。

3.2 设计目标

Nyx有以下目标。(i) **轻量级。**性能对PM内的缓存至关重要；因此，相对于访问PM的成本，增加控制和获取信息的成本必须低。(ii) **灵活的共享政策。**管理员可能需要不同的共享策略来处理不同的情况。因此，Nyx可以在一套共同的简单机制的基础上配置多个政策。(iii) **没有特殊的硬件。**以前的工作假设智能再源（如闪存，DRAM）提供可配置的控制和信息[53, 65, 78, 79]。Nyx用现有的硬件接口处理当前的设备。(iv) **最小化假设。**存储设备在不断发展，新一代的设备具有新的性能特征。因此，Nyx并没有为所有的PM设备假设一个特定的性能模型（例如，不同操作的干扰）。

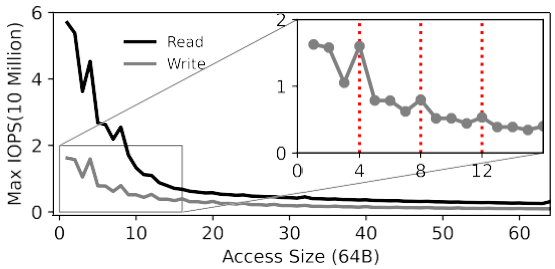


图3：**MaxIOPS简介。***随机读取和写入的最大IOPS为在我们的2→Intel Optane DC PMM系统上有不同的尺寸。*

3.3 尼克斯机制

Nyx包含了低级别的机制，使更高级别的共享策略可以很容易地实现。由于请求调节、资源使用的估计、干扰和应用减速都因PM而发生重大变化，我们详细描述了这些Nyx机制。访问控制和空间分配在很大程度上与PM无关，不是本文的重点；Nyx从以前的系统中借用了这些技术[29, 34, 52, 60, 62]。

PM访问调节。为了最大限度地减少调节PM请求的开销，Nyx坚持以前的DRAM调节技术所使用的基本原则：以粗粒度的方式节制请求，不重新排序或确定优先次序。为了模仿英特尔MBA的行为，Nyx通过在用户层面延迟PM访问来实现简单的节流。我们目前的实现是通过一个简单的基于计算的繁忙循环，以10ns为单位增加延时。在某些情况下，PM操作可能需要无限期地延迟（例如，当达到资源限制时）；在这种情况下，PM操作被停滞，直到Nyx控制器将延迟设置为一个有限的值。**资源使用估算。**Nyx必须确定每个缓存实例正在使用多少PM资源。如第2节所述，对于PM来说，传输的字节数并不是资源使用量的一个很好的估计；在PM上，每个操作类型（如读或写）和访问模式（如请求大小）都会消耗不同的资源量，并有不同的每秒最大操作。因此，Nyx将PM的利用率确定为每个操作类型的当前IOPS相对于该操作类型的最大IOPS的函数。例如，如果模式A的最大IOPS是$MaxIOPS_A$，那么模式A的每个操作的成本是$1/MaxIOPS_A$。如果模式B的最大IOPS是$1/NMaxIOPS_A$，那么每个B操作消耗的PM是A操作的N倍，成本是N倍→。

IOPS成本模型准确地捕捉到，写入的成本更高。昂贵于读取，以及对请求大小的依赖。

Nyx通过剖析确定每个访问模式的最大IOPS，每个PM服务器执行一次。该程序测量64B和4KB之间随机读写操作的IOPS（以64B为单位）。由于预取是在剖析过程中进行的，所以测量的最大IOPS准确地代表了操作本身和任何浪费的预取的成本。剖析集中在随机访问上，因为

多租户键值缓存大多是随机的：首先，由于多个租户同时访问PM（在不同的地址空间），他们的请求是交错的；其次，键往往根据其生存时间和大小被映射到任意的PM位置[14, 75]。剖析器在获得设备最大带宽的4KB的请求大小时停止。

图3显示了读取和写入的最大IOPS与请求大小的关系。如图所示，写的IOPS较低，因此每个操作的成本比读的高。虽然较大的请求通常有较低的IOPS，但与最小的PM访问大小有复杂的关系：例如，64B的随机存储的最大IOPS与256B的最小PM访问大小相似；不与256B对齐的访问有较低的MaxIOPS。

在运行时，Nyx跟踪每个缓冲区的PM使用情况。当一个高速缓存实例访问PM时，Nyx会查找到

该操作和规模的MaxIOPS，并递增一个成本缓存实例的计数器。$\frac{1}{MaxIOPS}$.为了减少同步的归化开销，这些计数器是按线程维护的而只有在需要时才懒洋洋地结合起来（例如，为了响应Nyx控制器的资源使用查询）。

虽然CPU缓存在理论上会给PM成本估算带来误差，但这些误差对于Nyx来说是可以忽略不计的。首先，由于CPU的预取浪费部分取决于空间的局部性，剖析器模拟了几乎没有顺序性的缓存实例的随机访问。第二，给定一个高速缓存实例在使用NT-存储的情况下（如Nyx-Pelikan），CPU缓存有对存储没有影响。最后，尽管PM负载可以在CPU缓存中提供服务，并且永远不会访问PM，但在多租户缓存中，很少有PM负载在CPU缓存中命中：因为每个实例的工作集通常是几十GB[28, 74]（并且有许多实例），在几十MB的CPU缓存中几乎没有时间定位。对于具有空间（如扫描）和时间定位（如突发性重试）的缓存实例，更复杂的成本模型留待未来工作。

干扰分析。当多个缓存实例共处一地时，Nyx会确定哪个实例对另一个实例影响最大。例如，当一个有效的QoS实现观察到一个受影响的客户端W没有达到它的保证时，它将迭代地减慢一个竞争的客户端，这将为W带来最大的利益。在基于PM的系统中，与DRAM不同，这些相互作用很难识别，因为一个受影响的客户端可能受到低带宽客户端比高带宽客户端的影响更大。干扰的数量是由于PM设备内的复杂调度造成的；随着未来几代PM设备的出现，哪些客户端会干扰其他客户端可能会发生变化。因此，Nyx假设没有事先了解这些相互作用。

Nyx通过运行时的微观实验来确定哪个客户对受影响的客户干扰最大。给定受影响的客户W和几个竞争客户，Nyx在测量对客户W的影响的同时，通过X对每个竞争客户进行迭代节流，以达到某些感兴趣的指标。

算法1：资源限制
灰色区域表示用于处理PM问题的独特功能

EpochLen：一个纪元中的刻度（例如100），**TickLen**：（例如10ms） **A.getResCounter()**：查询A的Nyx-Lib的资源使用情况
A.setThrottling(t)：为A的每次访问添加10ns延迟
ResAssigned[1...N]：每个缓存的每个纪元的分配资源

```
while true do
    # 第1步：开始一个纪元并将所有缓存节流设置为0
    foreach cache A do
        A.setThrottling(0)
        InitResCounter[A] = A.getResCounter()
    #
      第2步：监控资源利用率，暂停已用完分配资源的
      客户。
    虽然Epoch没有完成，但做
        SleepFor(TickLen)。
        foreach cache A do
            ResUsed = A.getResCounter() - InitResCounter[A]
            如果ResUsed>ResAssigned[A]，那么
                A.setThrottling(INFINITE) # 暂停
```

识别为对W干扰最大的客户端。X的值是可配置的，指标也是可配置的（例如，吞吐量、平均延迟或尾部延迟）。Nyx使用简单的剪枝技术，只对资源使用量最大的客户进行节流。减少微观试验时间的优化（例如，在不同的试验中专注于不同的客户端子集）留待今后的工作。

慢下来的估计。Nyx确定慢速每个客户在运行时的体验，应通过计算$\frac{T_{share}}{T_{alone}}$慈禧是客户端单独运行时的性能（对于某些感兴趣的指标），而T_{share}是它在共享环境中的当前性能。由于我们没有假设任何特殊的硬件，Nyx使用了一种类似于以前工作的方法[47]。

首先，为了学习T_{alone}，Nyx短暂地暂停所有其他客户端；T_{alone}定期更新（例如，1s）或每当观察到工作负载的变化。第二，使用运行时测量的T_{share}周期性地计算减速。正如我们将表明的那样，以带宽的少量损失和尾部延迟的增加为代价，这个解决方案充分地接近了没有硬件支持的减速情况。对于不经常变化的工作负载，暂停的影响可以减少。

3.4 尼克斯共享政策

Nyx实现了四个流行的共享政策。我们描述这些政策如何利用Nyx的机制来实现PM。**资源限制：**Nyx可以限制每个客户在多租户缓存中使用的PM再源的数量，将客户的性能相互隔离。我们的政策在标准操作方面去限制资源，类似于谷歌云的memcache[7]（例如，1000个1KB随机的每秒读数，或1MB/s随机读数）。

如算法1所示，Nyx为每个客户端逐个纪元提供资源限制，扩展了现有的方法[77]。每个纪元，Nyx监控每个

客户端的资源利用情况；如果一个客户端达到了这个纪元的极限，它对PM的访问将被延迟到下一个纪元。当纪元结束时，每个客户端的节流值被重置为零。

算法2：QoS

灰色区域表示PM的功能。我们省略了当行动违反任何LC任务的目标时回滚节流的代码。

```
ExperimentStep：一个缓存的吞吐量支出，用于支付干扰分析实验。(如500MB/s)
虽然是真的，但做
    # 第1步：监测每个客户的SLO松弛情况
    foreach cache A do
        slack[A] = (A.target - A.latency) / A.target
    S = 具有最小松弛度的缓冲区
    # 第二步：保护违反SLO的客户
    如果slack[S]<0，那么
    如果S是节流的，那么
            节流 S
        否则
            # 步骤2.1:挑选候选人进行节流
            如果有BE缓存，那么
                候选人=资源使用量最大的3个BE
            否则
                候选人=前3名资源使用量大的LC，松弛>0.2
                如果所有的LC都没有什么松弛，那么
                    候选人=拥有最多松弛的LC
            # 步骤2.2:找到干扰最大的客户 I = getLargestInterference(S, candidates)
            节流 I
    否则，如果slack[S] > 0.2，那么
        # 所有的缓存都有松弛-> 放松节流
        节制每一个缓冲区

函数getLargestInterference(S, Candidates)。
    #
      找到能以相同的费用（吞吐量）最大幅度提高S的租户
    如果候选者中只有一个客户，则返回该客户
    foreach C in Candidates do
        实验步骤的节流C
        实验后跟踪S延时的变化，将所有节流
        恢复到以前的状态
    返回帮助S获得最大改善的L
```

该实现允许管理员配置*容忍*和*勾选*长度，以权衡检查计数器的开销和反应时间。

服务质量。Nyx可以确保延迟关键型（LC）租户满足服务水平目标，同时保持同一服务器上最佳努力型（BE）租户的高PM利用率。正如早期的工作[36，37]，接纳控制防止工作负载的QoS目标无法实现，空间分配提供必要的命中率。

如算法2所示，Nyx采用了类似于Parties[31]和Caladan[39]的方法：对于每个LC客户，跟踪保证和当前性能之间的差异；当保证被违反时（即负松弛），竞争的租户被节流。

Nyx的不同之处在于它如何识别要被节流的客户。Caladan总是对具有最大带宽（LLC失误）的BE租户进行节流，而Nyx则对最能改善LC缓存的BE或LC缓存进行节流，在竞争的租户中支出相同。该实现允许管理员配置*ExperimentStep*，允许在积极的节流和更快的收敛之间取得平衡。

公平减速：Nyx可以在以下方面实现公平性

算法3：公平减速

```
A.getSlowDown(): 返回A的当前性能/T_alone while true do
    如果T_alone信息的时间超过P秒，则
        foreach cache A做
            刷新Talone(A)
    # 调整节流以均衡减速
    foreach cache A do
        SlowDown[A] = A.getSlowDown()
    找到具有最大和最小减慢的缓存L和S
    不公平 = SlowDown[L] / SlowDown[S]
    如果不公平> 不公平阈值，那么就降低
        L，提高S FairIntervals = 0
    否则
        # 在公平放缓的情况下，努力提高利用率
        FairIntervals ++
        如果FairIntervals > FairIntervalThreshold，那么
            扼杀所有缓存

函数refreshTalone(A)。
    A.setThrottling(0)，并暂停其他每一个缓存
    A。 Talone = 测量A的吞吐量
    将所有缓存的节制器恢复到以前的状态
```

在不同的缓存中实现均衡的减速。如同算法3[38,63]，Nyx通过逐步增加MinSlowDown缓存的节流和减少MaxSlowDown缓存的节流来最小化（MaxSlowDown/MinSlowdown）。当不公平度达到UnfairnessThreshold以下时，调整过程终止。该方案定期（每P秒）刷新估计的

每个客户的独立性能（T_{alone}）。管理员运营商可以定制P，以平衡较低的开销和动态工作负载的快速调整。

该策略可以被概括为保证加权降速和对某些缓存的降速进行硬限制。对于硬限制，Nyx在运行时跟踪特定的减慢，并在超过硬限制时对其他缓存进行节流。

比例资源分配。Nyx通过实际的比例资源分配（而不是简单的带宽分配）和干扰意识到的空闲资源再分配来实现pro-portional共享。Nyx确保每个缓存达到的性能等于或优于在一定时间内单独访问PM（时间共享[67]）。例如，如果一个高速缓存的权重是3分之2，那么它就保证获得至少2/3的独立性能。

Nyx首先按比例为每个缓存分配资源（不是带宽），并在一个周期内执行资源限制（算法4）。我们假设缓存空间已被按比例分配。在一个纪元之后，Nyx预测每个租户所需的资源量：没有使用所有给定资源的租户可能会捐赠闲置资源，而使用所有分配资源的租户可能会消耗更多的资源（一个简单的线性模型预测所需资源[77]）。

Nyx提供干扰感知的资源捐赠（Alg.中的Op- tion 2）。在PM上，空闲资源的再分配面临的困难是，捐赠的资源可能会严重干扰原捐赠者的性能。例如，如图所示

算法4：比例资源分配 减速刷新代码被省略。

DonateStep: 捐赠闲置资源的步骤（例如10%）。

```
总资源=1
虽然是真的，但做
    #
    第1步：强制执行并跟踪一个纪元的资源使用情况
    开始一个新纪元
    foreach cache A do
        执行A使用资源 <= ResourceAssigned[A]
        如果A耗尽了资源，记录多长时间。T
        imeUseUp[A]（例如，半个纪元的时
        间）。
        如果A留下闲置资源，记录ResourceUsed[A]。
    foreach cache A do
    纪元结束
    如果A有闲置资源，那么
    # 第2步#:重新分配闲置和额外资源
        DesiredResource[A] = ResourceUsed[A]
        # 选项2：干扰感知的资源捐赠
        如果A.getSlowdown() < TotalWeight / A.weight
            then # 在减速限制范围内，捐献一步
            DesiredResource[A] =
            Max(ResourceAssigned[A] * (1 - DonateStep),
            ResourceUsed[A])
        否
        则 # 在减速限制下撤销一个步骤
            DesiredResource[A] =
            Min(ResourceAssigned[A] * (1 +
            DonateStep), TotalResource * A. weight/
            TotalWeight)
        如果A耗尽了资源。DesiredResource[A] =
        ResourceAssigned[A] / TimeUseUp[A] 。
    ResourceAssigned[1...N] =
    根据权重和期望资源按比例分配资源
```

在第4.5节中，如果一个获取量大的缓存A将闲置的资源捐赠给一个写入量大的缓存，新的写入流量会极大地损害A的性能。为了防止这种干扰，Nyx以递增的方式重新分配资源，当捐赠的高速缓存的减速接近其下限时停止；如果减速超过下限，捐赠的资源的一部分将被退回。因此，Nyx保证了 "时间共享"的下限，同时使资源利用率最大化。该实施方案允许管理员设置*DonateStep*，平衡快速闲置资源捐赠和比例保证。**有了准入控制和容量分配。**简而言之，缓存实例1）被接纳，2）被分配空间，3）被Nyx管理。例如，一个PM自由空间检查，足以作为缓存的准入控制的资源限制；QoS策略需要像[36, 37]那样的逻辑来预测现有缓存的SLA兼容性。然后确定缓冲区的大小。例如，它可以根据实例的价格等级来设置；

为了执行QoS，管理员可以对客户的命中率进行分析v.s.
缓存空间关系[62]，并分配足够的空间以满足SLA。在运行时，Nyx假设接纳逻辑是正确的，并且不关心分配的空间。

3.5 缓存实例。PM优化的Pelikan

Nyx被设计成可以处理任何内存键值存储；我们目前的实现是建立在Pelikan - Twit- ter的内存KV缓存上的[17, 75]。我们描述了原始的

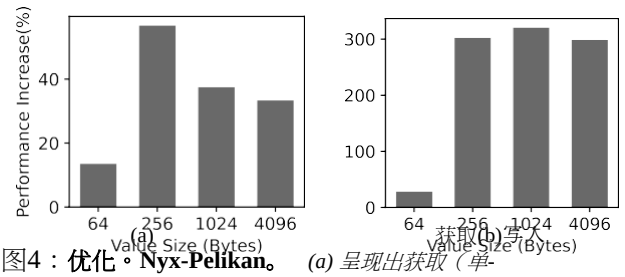


图4：**优化。Nyx-Pelikan。** *(a) 呈现出获取（单-线程）由于键值分离而提高了吞吐量。(b)呈现了由于将存储改为NT-存储而带来的写入（替换，8个线程）改进。*

Pelikan和优化的PM性能更高。Pelikan(SegCache[75])维护一个哈希表，用于索引
储存键-
值对的段。每个段包括项目，每个项目是一个（键、值、元数据）的元组。在获取操作中，Pelikan对键进行散列以找到项目。由于冲突的存在，在一次获取中可能会读取多个键。因此，Pelikan必须将每个读取的项目与键进行比较；如果键匹配，则返回值。

当Pelikan的默认版本被配置为PM时，哈希索引被保存在DRAM中，而段被保存在PM中。然而，由于PM中频繁的密钥访问，这种放置方式是低效的：缓存工作负载中的密钥通常比PM访问的粒度（256B）小得多[74]，而且小的读取在PM上表现相对较差[73]。

Nyx-
Pelikan通过将键（和元数据）与值分离成不同的部分来解决这个问题；键（和元数据）被放在DRAM中，而值则被放在PM中。这种设计需要为键和元数据提供DRAM，这很好，因为它们通常比值小得多[73]。

如表2所示，由于对PM的非时态存储可以提供比常规存储更大的吞吐量，Nyx-Pelikan使用NT-
存储。尽管非时间性存储可能无法从CPU缓存的时间定位中获益，但这种损失在大规模的缓存工作负载中是可以忽略不计的，这些工作负载通常都有大的工作集。如图4所示，Nyx-Pelikan改善了Pelikan获取性能
产量最多可减少55%，设置性能最多可减少3→。

3.6 尼克斯参数值

Nyx的参数值影响它的行为；如前所述，适当的设置取决于管理员的权衡。Nyx使用户能够配置所有这些参数，同时也设置默认值。

Nyx遵循现有的指导方针[38, 63, 77]进行策略参数值的选择。对于资源限制，Nyx使用10 ms tick和100 ticks per epoch来限制资源使用偏移量为1%。对于公平减速，Nyx将T_{alone}
刷新间隔设置为一秒钟，以实现对工作负载变化的相对快速的响应和2%以内的开销（§4.1）。

Nyx通过敏感性测试为新引入的参数提供默认值（§4.7）。Nyx QoS使用500MB/s Experi-

mentStep，因为它是产生良好干扰分析的最小步骤。在干扰感知的资源捐赠

追踪	类型	平均键/值大小（B）	操作（获取/写入比率）
S1	储存	36/799	0.86/0.13
C1	计算	67/2439	0.93/0.07
C2	计算	18/67485	0.52/0.48

表3：**Twitter的踪迹。**

为了平衡快速捐赠和稳定的捐赠者性能，Nyx设置了10%的捐赠步数。Nyx为细粒度的访问率调节设置了10ns的节流粒度，这比100ns的PM延迟要小一个数量级。我们将在第5节讨论潜在的优化，如动态/自适应参数和自动参数值选择。

4 评价

我们评估了Nyx机制的开销，以及Nyx提供资源限制、QoS、公平减速和资源分配比例等共享政策的情况。

设置。我们使用一个16核、单插槽的Intel Xeon Gold 5128 CPU @ 2.3GHz的服务器（Ubuntu 18.04），有22MB L3 Cache，2x16GB DRAM，以及2x128GB Intel Optane DC PMM，处于应用直接模式。我们在PM上以DAX模式挂载一个ext4文件系统。

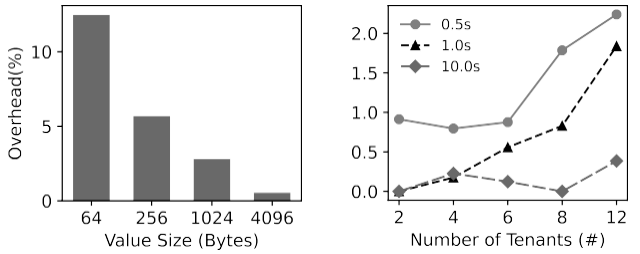
合成工作负载。我们首先用合成工作负载来说明关键特征。除非特别说明，工作负载对每个缓存实例有统一的随机访问，每个实例有10GB的工作集，有4B的键和可变大小的值。为了关注PM访问，我们使用具有高命中率（>99%）的工作负载。我们对写入量大的工作负载使用就地替换；缓存写入意味着替换。缓存开始时是热的。

现实的工作负载。我们用Twitter[74]中的三个大规模的缓存追踪来总结（表3）。追踪涵盖了不同价值大小（799B到67845B）和获取百分比（93%到52%）的缓存。我们从追踪中预装了100万个操作，并在这些操作中循环。

4.1 机制的开销

请求调节和资源使用估算。在Nyx中，每个PM的访问都会引起对Nyx-lib的调用，并产生rot-tling逻辑和资源计算。图5.a显示，对于极小的数值大小（例如，一个高速缓存行），这可能会增加12%的开销，但对于超过256B的访问大小，则低于6%。鉴于请求调节和资源使用核算的好处，我们认为这种开销是合理的。

干扰分析。由于观察尾部延迟所需的滞后性，确定干扰最大的客户比简单地选择带宽最大的客户需要更长的时间。在第4.3节中，我们将展示用增加的分析时间换取更精确的信息的好处。**减速估计。**减速估算的开销受到测量每个实例T_{alone}的时间、测量的频率和缓存实例的数量的影响。我们确定1ms是一个足够的暂停时间，以准确确定客户端的T_{alone}。图5.b显示，计算多达12个实例的T_{alone}所增加的时间少于2.5%的开销，即使每500ms执行一次。



(a) 监管、会计管理费 (b) 减速估算管理费

图5：**机制的开销。***(a)显示Nyx请求调节和资源使用核算开销（吞吐量）。它是用8线程只获取缓存测量的。观察到类似比例的延迟开销。(b)显示了Nyx的减速估计开销（吞吐量）。它是在1ms的T_{alone}暂停时间，不同数量的客户（X轴）和不同频率（0.5/1/10s）的更新T_{alone}的所有缓存的情况下测量的。*

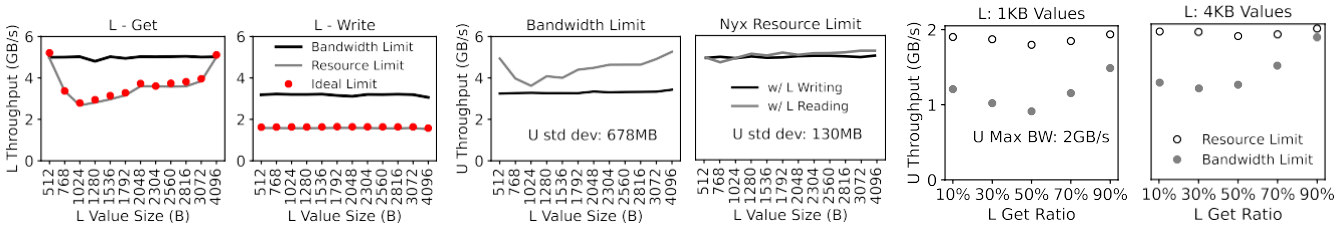
4.2 资源限制

我们证明Nyx可以对PM实施真正的资源限制，这与仅基于带宽的方法不同。我们从一个包含一个无限（U）缓存和一个有限（L）缓存的工作负载开始。缓存U是一个重获取的缓存实例，而缓存L则是变化的：只获取或只写入，有不同的值大小。L的资源限制是1.25M的4KB随机负载OPS，或者说占设备总资源的42%，因为4KB随机负载的最大IOPS是300万。根据带宽的定义，这相当于这些4KB随机负载的5GB/s；然而，这个IOPS限制导致了其他工作负载的不同带宽。

图6.a显示了L的带宽；目标IOPS，其中不超过42%的设备资源被使用，显示为红色。如愿以偿，Nyx总是将L的吞吐量限制在目标限制之内，而不管L的访问模式（由值大小和读/写决定）。相反，一个只基于带宽的策略会错误地允许L大大超出目标限制，直到5GB/s的最大带宽。当L是get-only时，当值的大小在1KB左右时，这个问题是最值得注意的；如前所述，1KB的访问会导致大量的CPU预取浪费，而软件级的带宽核算并没有发现。另一方面，Nyx的MaxIOPS成本模型准确地捕获了资源的使用。同样地，带宽不能捕捉到PM写入成本，也不能正确地限制L的吞吐量。

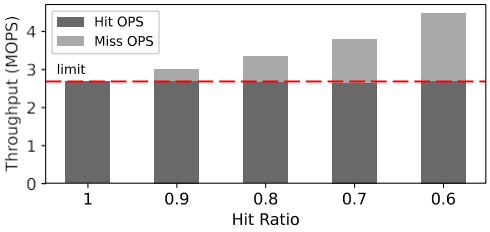
图6.b显示了在相同的L工作负载下对无限客户端（U）的影响。在带宽策略下，U的性能取决于L的访问模式。由于PM的读/写成本不对称，L是执行读还是写对U有很大影响；同样，每种访问模式的不同预取浪费对U的影响高达45%。

U.相比之下，Nyx为U提供了稳定和可预测的性能，无论L的访问模式如何：在L的所有工作负载中，U的性能的标准偏差只有130MB/s（带宽限制的偏差是678MB/s）。最后，图6.c显示，当L中Get的百分比变化时，Nyx为U提供了稳定的性能，而基于PM-oblivious带宽的方法则不然。



(a) L 吞吐量(b)U 吞吐量(c)U + Broader L 设置

图6：**资源限制：缓存U（无限）+缓存L（有限）。** *缓存U是只获取的。(a) 当资源限制为5GB/s时，缓存L的吞吐量（1.25M 4KB的随机负载OPS，或总设备资源的42%）。红色虚线表示L在 "理想限制 "下的性能，计算为当前访问模式最大IOPS的42%。L是只获取或只写入的，其数值大小不一（X轴）。(b) 缓存U与(a)中相同的L共处时的性能，比较了带宽限制和Nyx资源限制。(c) 其他L的设置。1KB/4KB的值大小和10%-90%的获取。U是一个比(a)和(b)更轻的高速缓存。标签表示U的最大带宽，当与5GB/s的缓存实例共处时（4KB值，只获取）。*



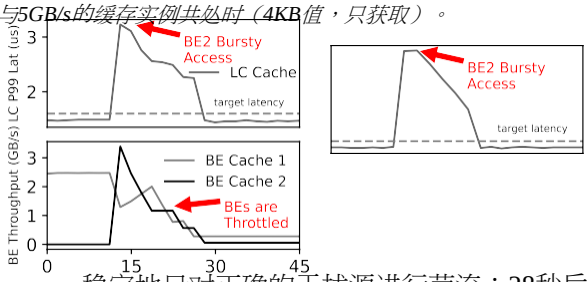


图7：**资源限制：点击率变化的行为。** *当资源限制为5GB/s时，具有1 KB Get-only工作负载的缓存的每秒操作数（OPS）。我们用不同的工作集来改变工作负载，以实现不同的命中率；注意每次错过后都没有插入。*

图7展示了Nyx的资源限制行为，因为缓存命中率的变化。如图所示，Nyx限制了PM再源的使用，从（获取）命中。旁观缓存（如Pelikan）中的失误只是在检查索引（在DRAM中）后返回，不使用PM资源，所以它们不受限制。

4.3 具备QoS功能的

Nyx可以为延迟关键型（LC）缓存提供QoS保证，同时通过干扰感知调节为尽力型（BE）缓存提供高利用率；相反，像Caladan中的PM遗忘方法可能无法为BE缓存提供同样的性能。为了进行比较，我们在Nyx-Caladan中实现了Caladan的方法。图8显示了一个LC高速缓存（P99延迟目标为1.5μs）与两个BE高速缓存共存。BE1是重度获取，BE2是重度写入。最初，当BE2的吞吐量较低，而BE1的吞吐量适中，为2.4GB/s，LC满足其P99目标；然而，在12s时，BE2进行了许多突发性的写入，导致LC的P99延迟超过3μs，违反其目标。Nyx-Caladan和Nyx都通过迭代节流BE缓存来解决这种情况。Nyx-Caladan对当前消耗带宽最多的缓存进行节流，如左边两个子图所示；因此，Nyx-Caladan同时对BE1和BE2进行节流，导致BE1的带宽减少6。另一方面，Nyx将对LC干扰最大的缓存识别为BE2，即写量大的缓存。因此，Nyx稳定地只对正确的干扰源进行节流；28秒后，只有BE2被节流，而BE1恢复到原来的产量。总而言之，Nyx为以下情况提供了高利用率多个缓存，同时保证每个目标。

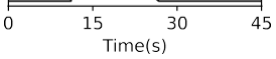


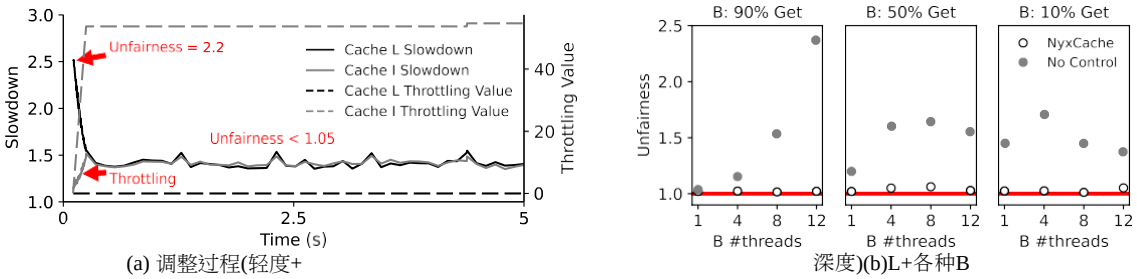
图8：**QoS。Nyx-Caladan与Nyx的调整。** *该图显示了Nyx和Nyx-Caladan是如何节制BE缓存以确保LC缓存的P99延迟的。LC缓存与两个BE缓存同处一地；BE1是get-heavy，B2是write-heavy（即对LC的干扰更多）。BE2在12s时爆发，打破了LC延迟的目标。Nyx-Caladan(左)节制最高bw的客户端，而Nyx(右)节制对LC有最多干扰的客户端。Nyx-Caladan错误地对BE1进行了节流，导致BE1的带宽减少了↛6。*

Nyx的收敛时间为几十秒，与之前的工作如Ports[31]相似：大部分收敛时间用于监测尾部延迟。与Parties一样，Nyx对尾部延迟的测量时间为500ms，因为更短的间隔会导致测量结果出现噪声。我们把在网络数据包队列中更快的尾部延迟测量（如在最初的Caladan[39]中利用的）留给未来的调查。

我们的实验显示，Nyx对收敛时间有一个耐人寻味的影响：如图所示，Nyx可以在相当长的时间内使LC缓存达到其目标性能，而只是选择具有最高带宽的缓存（这不需要任何微观实验时间）。这些结果的含义是，Nyx不是简单地快速行动和节制任何竞争的实例，而是正确地行动和节制干扰的来源。

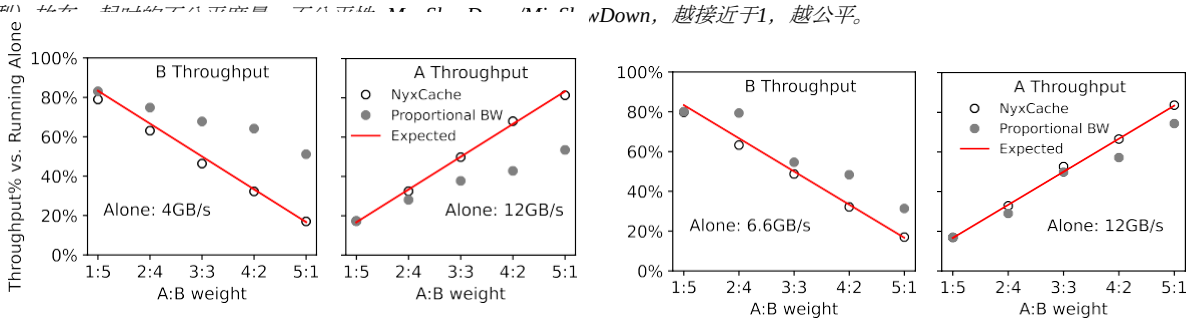
4.4 展会放缓

Nyx通过根据每个客户的测量速度迭代调节请求来实现公平减速（即$\frac{T_{alone}}{T_{分享}}$）。图9.a显示了Nyx的调整，给定了集中的轻型和密集的get-heavy缓存。最初，轻型缓存的速度比密集型缓存的速度高2.2倍。随着时间的推移，Nyx动态地增加了具有最小减速的高速缓存的节流，减少了具有最大减速的高速缓存的节流。相对来说



(a) 调整过程(轻度+ 深度)(b)L+各种B

图9：**公平减速。** *(a)显示了Nyx是如何平衡两个缓存实例（一个轻度缓存(L)和一个密集型缓存(I)）的减慢时间的。(b)显示了将L（轻度重度缓存）与不同的B实例（重度->写重度，轻度->密集型）*



(a)A: Get-heavy, B: Write-heavy(b)A: Efficient-Get, B: Inefficient-Get

图10：**比例共享。** *(a)显示了A（获取重度缓存）和B（写入重度缓存）在不同权重配置下的吞吐量。标签表示A和B单独运行的吞吐量。在带宽分配的情况下，B超过了其分配的比例性能。(b)显示了A（高效的获取密集型缓存，4KB值大小）和B（低效的获取密集型缓存，1KB值大小）。*

很快，两个缓存都收敛到了1.5附近的减速，MaxSlowdown的不公平指标在1.05附近解决。

图9.b显示了Nyx在一系列高速缓存上的公平减速策略。缓存L仍然是一个轻度的get-heavy缓存；缓存B改变了线程的数量，可以是get-heavy，50%混合，或者write-heavy。在没有Nyx的情况下，L可能会经历急剧的不公平的减速（由于PM的复杂性能）；例如，将A与多线程的get-heavy缓存B放在一起，会有接近2.4的不公平。相比之下，Nyx在所有12种情况下都实现了公平的减速（<1.05的不公平性）。

4.5 成比例的资源分配

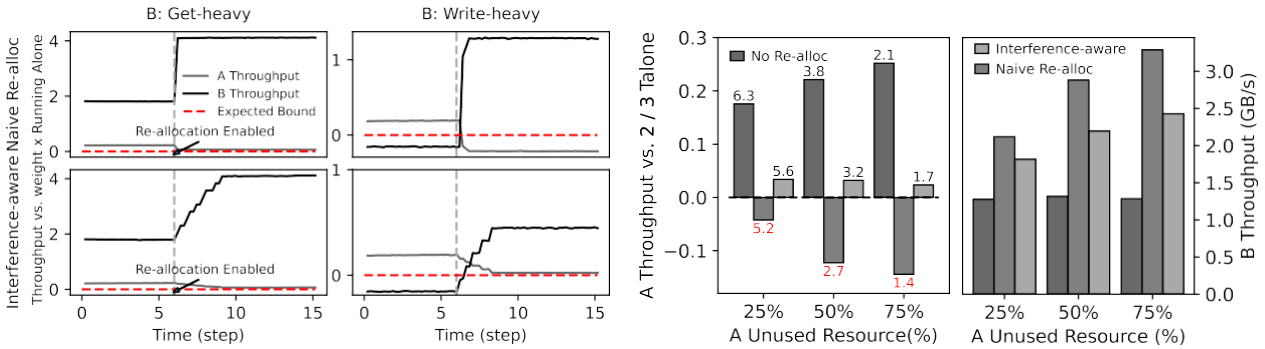
Nyx实现了按比例的资源分配，并保证了时间共享的下限，同时进行了闲置资源的再分配。我们从两个使用所有分配资源的缓存的简单场景开始。图10中的场景沿x轴改变了A和B的理想比例份额；红线表示在单独运行时的理想比例吞吐量。图10.a显示，不关心PM的带宽方法不能保证比例份额；特别是，写密集型的B缓存获得的吞吐量比期望的多3倍，获取密集型的A缓存受到很大影响（40%）。然而，通过正确估计资源的使用，Nyx为每个高速缓存提供了所需的分配。图10.b显示了当高效获取（值：4KB）和低效获取（值：1KB）的缓存被放在一起时发生的类似效果。

当有闲置资源需要重新分配时，按比例分配更具挑战性。图11.a显示了两个缓存A和B，其中A只使用其25%的份额。当B是get-heavy（左上角的子图）时，A可以把它所有的闲置资源捐给B；A的性能略有下降，但B得到的吞吐量要高很多。然而，当B的写入量很大时（右上角的子图），如果A捐出它所有的闲置资源，B的更高的吞吐量会大大干扰A，突破A的时间共享下限（A的独立吞吐量的2/3）。因此，Nyx没有进行天真的捐赠；相反，Nyx在监测每个缓存的减慢时，逐步捐赠闲置资源。如下面两张图所示，Nyx保证了每个高速缓存的时间共享下限，同时提高了利用率。

我们接下来研究了不同的缓存A的闲置资源比例的工作负载。当缓存B是get-heavy时，A的所有闲置资源都可以安全地重新分配给B，Nyx对缓存B实现了与简单捐赠相同的性能（由于空间限制，图没有显示）。然而，当缓存B是写量大的时候，简单地将A的闲置资源捐赠给B违反了A的时间共享约束（图11.b）；Nyx准确地保护了缓存A的性能，同时相对于没有捐赠，仍然提高了缓存B的性能。

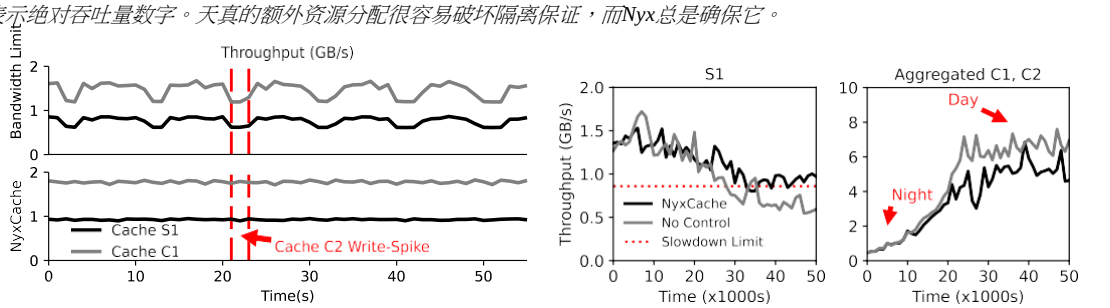
4.6 真实的痕迹

Nyx为现实的工作负载提供了隔离功能。我们展示了资源限制和减速限制的用例。在生产工作负载中，写峰值是常见的；例如，当缓存用于ML模型时，写峰值会随着模型参数的定期刷新而出现[74]。图12.a显示了Nyx如何隔离高速缓存S1和C1



(a) 天真的 "全部捐出 " 策略的问题 (b)在获取和写入量大的缓存之间的重新分配

图11：**按比例分配。额外的资源再分配。***缓存的权重A：B是2：1。(a)显示了高速缓存A（轻度，得到重度）捐赠其额外分配的资源之前和之后的吞吐量。A有75%的闲置资源。Y轴是归一化的差异。当高速缓存B是get-heavy的时候（左上图），A由于捐赠而得到名义上的性能下降。然而，当缓存B是写量大的时候（右上图），捐赠会导致A的速度严重下降。与天真的额外再分配不同，Nyx（底部两图）确保租户A的性能总是超过其单独运行性能的三分之二。(b)显示了A捐赠额外资源之前和之后A的减速（左图）和B的吞吐量（右图）。A是get-heavy，B是write-heavy。标签表示绝对吞吐量数字。天真的额外资源分配很容易破坏隔离保证，而Nyx总是确保它。*



(a) 保护缓存不受写入高峰的影响 (b)限制S1在白天和晚上的速度减慢

图12：**现实的跟踪。***(a)显示了高速缓存S1和C1与高速缓存C2共处时的性能。缓存C2有写峰值。Nyx（下图）可以隔离写尖峰，而带宽限制不能（上图）。(b)显示了高速缓存S1的性能（我们保证其减速总是小于1.5的高速缓存）。S1与C1和C2共处一室；C1和C2都有很强的昼夜规律（晚上轻，白天重）。如果没有Nyx，S1的性能在白天会急剧下降（因为C1和C2的影响），不鼓励共享。然而，Nyx总是可以提供合理的性能（例如，与单独运行相比，在x1.5的速度内）。红线代表S1的性能保证。*

从高速缓存C2的（增加的）写峰值。如果资源限制只基于C2的带宽，当C2出现写峰值时，S1和C1就会受到影响。然而，Nyx的资源限制策略可以限制C2的资源使用（4GB/s，定义为1M 4KB随机负载OPS），以保持S1和C1的稳定。

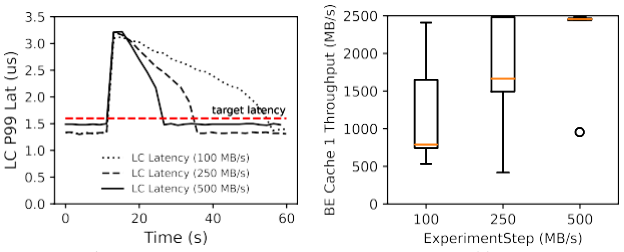
Nyx还可以保护关键缓存的性能。为了鼓励租户使用多租户的PM环境，必须保证一些缓存的性能与PM设备的独占使用类似。在图12.b所示的实验中，S1（关键缓存）与C1和C2放在一起，这两个缓存有昼夜模式[74]。在没有控制的情况下（灰线），由于C1和C2的大量访问，S1的性能在白天下降到目标以下。然而，Nyx可以为S1建立一个减速的硬限制（例如，1.5）。正如所观察到的，Nyx将S1的性能损失保持在一个合理的范围内。

4.7 参数敏感度分析

在这里，我们提出了不同的ExperimentStep和DonateStep值对Nyx行为的敏感性分析。

ExperimentStep会影响Nyx干扰分析的准确性。如图13.b所示，使用与图8相同的配置，较小的ExperimentStep更加可能会导致较低的BE 1最终吞吐量。当ExperimentStep较小时，尾部延迟的变化更可能是由于测量噪声而不是干扰，导致干扰分析的准确性降低。我们的实验建议ExperimentStep至少为500MB/s。ExperimentStep还影响到Nyx QoS能多快地确保LC尾部延时。如图13.a所示，更大的ExperimentStep表示更快的收敛。然而，它增加了过度缩减BE缓存和降低系统利用率的风险。Nyx QoS中的ExperimentStep默认为500MB/s，用于良好的干扰分析和高系统利用率，同时保持合理的收敛时间。

图14显示了DonateStep如何影响Nyx的资源分配比例。较大的DonateStep会导致更快的闲置资源捐赠，但也可能导致较大的性能波动（例如，图中60%的DonateStep，12s和18s）。在运行时，高速缓存的吞吐量总是略有变化，导致做国家调整。这些调整在小的捐赠步骤中是微妙的，但在大的捐赠步骤中是显著的。这种波动损害了捐赠者的利益，因为他们有时会减慢速度（超过限制）。Nyx使用10%的捐赠步数，在快速的资源捐赠和稳定的捐赠者性能之间取得平衡。



(a) 收敛　　时间(b)BE 1 最终吞吐量

图13:**QoS。实验步骤敏感度分析。与***图8相同。（a）显示了Nyx QoS可以确保LC P99延迟的速度，不同的ExperimentSteps。(b)显示了BE 1的最终吞吐量(boxplot，五次运行)变化的ExperimentSteps；BE 2在所有情况下的最终吞吐量接近零。*

5 讨论

超越基本政策。Nyx可以被扩展到更复杂的策略，以适应更复杂的设置。例如，一个比例的共享策略可以在各组缓存中应用。然后，在一个组内，可以执行另一种共享策略（例如，QoS）。我们把全面的研究作为未来的工作来做。**多租户缓存的替代方案。**Nyx管理缓存，每个缓存都有自己的空间。有一些替代共享缓存的方法；例如，一个大的实例可以由多个用户共享[60]。这种模式可以利用Nyx的资源使用核算和干扰分析技术。然而，它可能会产生新的问题，如：用户对共同缓存对象的PM写入应该如何收费？**更聪明的参数值选择：**i)自适应参数可能是有益的，例如，当它远离阈值时，DonateStep可以更大（用于快速捐赠），当它接近时，可以更小（以避免性能波动）。我们把这些优化作为未来的工作。

安全性。Nyx政策可能是可攻击的，例如，在资源限制中，一个敌对的客户端可能会在第一个测量点限制其访问，而在最后一个测量点投入大量的负载。一个解决方案是使用随机的测量点而不是固定的测量点。我们把Nyx的安全研究作为未来的工作。

6 相关工作

多租户内存键值缓存：我们的工作建立在过去多租户内存键值缓存系统的研究之上。这些努力包括跨租户分配空间的技术[29, 32, 34, 60, 62]，以及单个缓存实例的优化[25,27,28,33,42,54,75]。我们的工作则侧重于许多缓存共享PM时的访问调节和信息提取的挑战。

PM缓存：人们一直在努力将PM与各个缓存系统结合起来。以前的工作包括数据库[50, 72, 81]、文件系统[22, 48, 80]、内存键值缓存[6, 19, 21]和一般策略[26, 27]。然而，据我们所知，我们是第一个在多租户缓存设置中解决项目管理问题的人。

PM干扰。有几项工作描述了PM d-的特点。

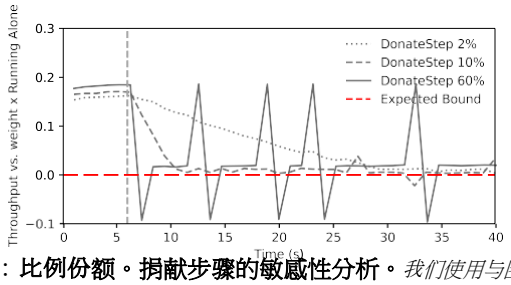


图14：**比例份额。捐献步骤的敏感性分析。***我们使用与图11中相同的设置，当B是写量大的时候。该图显示了A，捐赠者在不同捐赠步骤下的吞吐量。*

罪行[45, 69, 71, 73]。然而，只有少数人研究了PM中的干扰效应。据我们所知，Dicio[55]是这个领域的第一个工作。Dicio和我们的工作都发现了PM中不同的读写干扰效应。然而，Dicio和Nyx的目标不同。Dicio的目的是确定PM DIMM带宽何时饱和。Dicio通过使用待写队列（WPQ）延迟作为启发式方法来接近这个目标。另一方面，我们的目标是为每个客户（而不是每个DIMM）提供资源使用计数、减速估计和跨客户干扰分析机制。Dicio保护单个LC任务不受单个BE任务的影响，而我们的QoS政策适用于多个客户端。Dicio承认，用PM媒体级的统计数据来决定哪一个尽力任务要节流是很有挑战性的（因此没有做）；我们用运行时的干扰分析方法来解决这个问题。最后，Dicio扩展了Caladan[39]，使用CPU调度来调节PM访问。这种方法适用于所有的应用程序，包括缓存，但需要对应用程序进行修改以使用Caladan独特的运行时系统（不完全兼容Linux）。我们把CPU调度调节PM的方法留给未来的工作。

共享其他资源。在管理和共享其他资源方面已经做出了努力，如网络、CPU、LLC、存储设备和锁等[31,39-41,43,44,51,56,57,59,65]。它们与我们的工作基本上是正交的；我们计划在未来将项目管理纳入这些系统。

7 总结

我们证明，由于PM的独特属性，先前的DRAM或存储设备的访问调节、资源使用估计和干扰分析的方法在PM上无法发挥作用。我们介绍了Nyx，它以一种轻量级的方式实现了这些机制，而不需要硬件支持。我们表明，Nyx可以支持各种多租户缓存共享政策，比早期的DRAM或存储方法更能满足性能或共享目标。

鸣谢。我们感谢Ali R. Butt（我们的牧者）、匿名评审员和ADSL成员的宝贵意见。这份材料得到了美国国家科学基金会CNS-1838733、CNS-1763810、谷歌、VMware、英特尔、Seagate、三星和微软的资金支持。作者的观点和发现可能不反映国家科学基金会或其他机构的观点和发现。

参考文献

[1] 亚马逊的弹性。https://aws.amazon.com/elasticache/。

[2] 亚马逊弹性定价。https://aws.amazon.com/elasticache/pricing/。

[3] Aws elasticache。https://aws.amazon.com/elasticache/redis/customers/。

[4] Budgetfairqueueingi/ oscheduler。http://algo.ing.unimo.it/people/paolo/disk_sched/。

[5] 缓存在reddit。https://redditblog.com/2017/1/17/caching-at-reddit/。

[6] Caching on pmem: an iterative approach. yue yao. https://www.snia.org/educational-library/ caching-pmem-iterative-approach-2020.

[7] Googlememcacheresourcelimit。https://cloud.google.com/appengine/ docs/standard/python/memcache。

[8] Intel mba issue with pm. https://github.com/intel/intel-cmt-cat/issues/170.

[9] Intelmemorybandwidthallocation(mba。https://software.intel.com/content/ www/cn/zh/develop/articles/introduction- to-memory-bandwidth-allocation.html。

[10] Intel Optane DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html。

[11] Intel Optane DIMM定价。https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html。

[12] I/o调度。https://en.wikipedia.org/wiki/i/o_scheduling。

[13] ipmctl mediareads, mediawrites。https://docs.pmem.io/ipmctl-user-guide/instrumentation/show-device performance。

[14] Memcached。https://memcached.org/。

[15] Memcached统计。https://docs.oracle.com/cd/E17952_01/ mysql-5.6-en/ha-memcached-stats.html。

[16] Memcachier。https://www.memcachier.com/。

[17] Pelikan - twitter. https://twitter.github.io/pelikan/.

[18] Pelikan缓存-- 驯服尾部延迟并实现可预测性。https://twitter.github.io/pelikan/2020/benchmark-adq.html。

[19] Pmem redis。https://github.com/pmem/pmem-redis。

[20] Redis企业云。https://redis.com/redis-enterprise- cloud/overview/。

[21] 持久性内存的波动性好处 - memcached。https://memcached.org/blog/persistent-memory/。

[22] Thomas E Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N Schuh, and Em-mett Witchel.Assise:通过分布式文件系统中的客户端-本地NVM的性能和可用性。在*第14届USENIX操作系统设计与实现研讨会（OSDI 20）上*，第1011-1027页，2020年。

[23] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny.一个大规模键值存储的工作量分析。在*第12届ACM SIGMETRICS/PERFORMANCE计算机系统测量和建模联合国际会议上*，第53-64页，2012年。

[24] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan.杀手微秒的攻击。*Communications of the ACM*, 60(4):48-54, 2017.

[25] Nathan Beckmann, Haoxian Chen, and Asaf Cidon.LHD：通过最大化命中密度提高缓存命中率。在*第15届USENIX网络系统设计与实现研讨会（NSDI 18）上*，第389-403页，2018。

[26] Nathan Beckmann, Phillip B Gibbons, Bernhard Haeupler, and Charles McGuffey.写回感知缓存。在*计算机系统的算法原理研讨会上*，第1-15页。SIAM, 2020.

[27] Nathan Beckmann, Phillip B Gibbons, and Charles McGuffey.块粒度感知的缓存。在*第33届ACM算法和架构并行化研讨会论文集中*，第414-416页，2021年。

[28] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Palter, et al. The cachelib caching engine:设计和规模化的经验。在*第14届USENIX操作系统设计与实现研讨会（OSDI 20）上*，第753-768页，2020。

[29] Daniel S Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter.Robinhood:尾部延迟感知的缓存--从缓存丰富到缓存贫乏的动态重新分配。In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 195-212, 2018.

[30] E Chen, D Apalkov, Z Diao, A Driskill-Smith, D Druist, D Lottis, V Nikitin, X Tang, S Watts, S Wang, et al. Advances and future prospects of spin-transfer torque random access memory.*IEEE Trans- actions on Magnetics*, 46(6):1873-1878, 2010.

[31] 陈爽，Christina Delimitrou，和José F Martínez。缔约方。多个互动服务的Qos感知资源分区。在*第二十四届国际编程语言和操作系统架构支持会议上*，第107-120页，2019年。

[32] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti.Dynacache:动态云缓存。在*第七届USENIX云计算热点研讨会（HotCloud 15）上*，2015年。

[33] 阿萨夫-西顿、阿萨夫-艾森曼、穆罕默德-阿里扎德和萨钦-卡蒂。Cliffhanger:扩展网络内存缓存的性能悬崖。在*第13届USENIX网络系统设计与实现研讨会（NSDI 16）上*，第379-392页，2016。

[34] Asaf Cidon, Daniel Rushton, Stephen M Rumble, and Ryan Stutsman.Memshare：一个动态的多租户键值高速缓存。在*2017年USENIX年度技术会议（USENIX ATC 17）上*，第321-334页，2017。

[35] Craciunas, Silviu S and Kirsch, Christoph M and Röck, Harald.通过系统调用调度进行I/O资源管理。*ACM SIGOPS Operating Systems Review*，42（5）：44-54，2008。

[36] Christina Delimitrou, Nick Bambos, and Christos Kozyrakis.异构数据中心中的Qos-感知接纳控制。在*第十届自主计算国际会议（ICAC 13）上*，第291-296页，2013。

[37] Christina Delimitrou和Christos Kozyrakis。Paragon:异构数据中心的Qos感知调度。*ACM SIGPLAN通告*，48（4）：77-88，2013。

[38] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N Patt.通过源节流的公平性：多核内存系统的可配置和高性能的公平性底层。*ACM Sigplan通告*, 45(3):335-346, 2010.

[39] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay.Caladan:缓解微秒级时间尺度的干扰。在*第14届USENIX操作系统设计与实现研讨会（OSDI 20）上*，第281-297页，2020。

[40] Ajay Gulati, Irfan Ahmad, Carl A Waldspurger, et al. Parda:分布式存储访问的资源比例分配。在*FAST*，第9卷，第85-98页，2009。

[41] Ajay Gulati, Arif Merchant, and Peter J Varman. mclock:处理hypervisor io调度的吞吐量变化。在*OSDI*，第10卷，第437-450页，2010年。

[42] 胡夏梦，王晓林，李亦晨，周岚，罗英伟，丁晨，姜松，王振林。Lama:优化的键值缓存的局部感知内存分配。In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 57-69, 2015.

[43] 黄健, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K Qureshi.Flashblox:实现病毒化ssds的性能隔离和统一寿命。In *15th USENIX Conference on File and Storage Tech- nologies (FAST 17)*, pages 375-390, 2017.

[44] Călin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, 等. Perfiso:商业延迟敏感型服务的性能隔离。In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 519-532, 2018.

[45] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amir- saman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv: 1903.05714*, 2019.

[46] 贾一辰和陈锋。一石二鸟:高带宽和低延迟的自动调整rocksdb.In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 652-664.IEEE, 2020.

[47] 拉姆-斯里瓦察-卡南，迈克尔-劳伦扎诺，安贞燮，杰森-马尔斯，和唐凌嘉。Caliper:共享架构资源的多租户环境的干扰估计器。*ACM Transactions on Architecture and Code Optimization（TACO）*，16（3）：1-25，2019。

[48] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson.Strata:一个跨媒体文件系统。在*第26届ACM操作系统原理研讨会（SOSP'17）论文集中*，中国上海，2017年10月。

[49] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger.架构相变存储器作为一种可扩展的戏剧性替代方案。在*第36届计算机架构年度国际研讨会上*，第2-13页，2009年。

[50] Gang Liu, Leying Chen, and Shimin Chen.Zen:一个用于非易失性主存储器的高吞吐量无日志的OLTP引擎。*VLDB捐赠会议论文集*, 14(5):835-848, 2021.

[51] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ran- ganathan, and Christos Kozyrakis.Heracles:提高资源效率的规模。在*第42届计算机架构年度国际研讨会论文集中*，第450-462页，2015年。

[52] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa.Bubble-up:通过合理的共同定位提高现代仓库规模计算机的利用率。在*第44届IEEE/ACM国际微架构研讨会上*，第248-259页，2011年。

[53] Kyle J Nesbit, Nidhi Aggarwal, James Laudon, and James E Smith.公平排队的内存系统。In *2006 39th Annual IEEE/ACM Interna- tional Symposium on Microarchitecture（MICRO'06）*, pages 208-222.IEEE, 2006.

[54] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Her- man Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook.在*第十届网络系统设计与实施研讨会（NSDI'13）上*，第385-398页，伊利诺伊州朗伯德，2013年4月。

[55] Jinyoung Oh and Youngjin Kwon.使用Dicio的持久性内存感知性能隔离。在*第12届ACM SIGOPS亚太系统研讨会上*，第97-105页，2021年。

[56] 朴镇洙, 朴成范, 和Woongki Baek.Copart:最后一级缓存和内存带宽的协调分区，用于商品服务器上公平意识的工作负载整合。在*2019年第十四届EuroSys会议论文集中*，第1-16页，2019年。

[57] Jinsu Park, Seongbeom Park, Myeonggyun Han, Jihoon Hyun, and Woongki Baek.Hypart：商品服务器上实用内存带宽划分的混合技术。在第*27届国际并行架构和编译技术会议上*，第1-14页，2018年。

[58] Stan Park和Kai Shen.Fios：一个公平、高效的闪存i/o调度器。在*FAST*，第12卷，第13-13页，2012。

[59] Yuvraj Patel, Leon Yang, Leo Arulraj, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Michael M Swift.使用调度器合作锁来避免调度器的颠覆。在*第十五届欧洲计算机系统会议论文集中*，第1-17页，2020。

[60] 濮存昕，李浩源，Matei Zaharia，Ali Ghodsi，和Ion Stoica。Fairride:近乎最优的、公平的缓存共享。在*第13届USENIX网络系统设计与实现研讨会（NSDI 16）上*，第393-406页，2016。

[61] Kai Shen and Stan Park.Flashfq：基于闪存的公平排队i/o调度器。在*2013年USENIX年度技术会议（USENIX ATC 13）上*，第67-78页，2013。

[62] Ioan Stefanovici, Eno Thereska, Greg O'Shea, Bianca Schroeder, Hitesh Ballani, Thomas Karagiannis, Antony Rowstron, and Tom Talpey.软件定义的缓存：管理多租户数据中心的缓存。在*第六届ACM云计算研讨会上*，第174-181页。ACM，2015。

[63] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu.应用减速模型。量化和控制共享缓存和主内存的应用间干扰的影响。在*2015年第48届IEEE/ACM国际微架构研讨会（MICRO）上*，第62-75页。IEEE，2015年。

[64] Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu.Mise:在共享主存储器系统中提供性能预测和提高公平性。在*2013年IEEE第19届高性能计算机架构国际研讨会（HPCA）上*，第639-650页。IEEE, 2013.

[65] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gómez-Luna, and Onur Mutlu.Flin:在现代nvme固态硬盘中实现公平性和提高性能。在*2018年ACM/IEEE第45届计算机架构年度国际研讨会（ISCA）上*，第397-410页。IEEE，2018。

[66] Redis的团队。Redis。https://redis.io/，2021。

[67] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R Ganger.Argon:共享存储服务器的性能隔离。在*FAST*，第7卷，第5-5页，2007。

[68] Benjamin Wagner, André Kohn, and Thomas Neumann.分析性工作负载的自调控查询调度。在*2021年国际数据管理会议上*，第1879-1891页，2021年。

[69] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swan- son, and Jishen Zhao.非易失性存储器系统的特征和建模。In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 496-508.IEEE, 2020.

[70] H-S Philip Wong, Heng-Yuan Lee, Shimeng Yu, Yu-Sheng Chen, Yi Wu, Pang-Shiu Chen, Byoungil Lee, Frederick T Chen, and Ming- Jinn Tsai.金属氧化物RAM。*IEEE论文集》*，100（6）：1951-1970，2012。

[71] 吴侃，Andrea Arpaci-Dusseau，和Remzi Arpaci-Dusseau。监管英特尔Optane SSD的一个不成文的合同。在*第11届USENIX存储和文件系统热门话题研讨会（HotStorage 19）上。USENIX协会，华盛顿州伦顿*，2019年。

[72] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau.存储层次不是一个层次。用orthus优化现代存储设备上的缓存。In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 307-323, 2021.

[73] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson.可扩展持久性存储器的行为和使用的经验指南。*arXiv预印本arXiv:1908.03583*，2019。

[74] Juncheng Yang, Yao Yue, and KV Rashmi.在twitter上对数百个内存缓存集群的大规模分析。在*第14届USENIX操作系统设计与实现研讨会（OSDI 20）上*，第191-208页，2020。

[75] Juncheng Yang, Yao Yue, and Rashmi Vinayak.Segcache：一个小对象的内存高效和可扩展的内存键值缓存。在*NSDI*，第503-518页，2021年。

[76] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T Kaushik, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau.Split-level i/o scheduling.在*第25届操作系统原理研讨会论文集中*，第474-489页，2015。

[77] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha.Memguard:用于多核平台高效性能隔离的内存带宽保留系统。在*2013年IEEE第十九届实时和嵌入式技术与应用研讨会（RTAS）上*，第55-64页。IEEE，2013年。

[78] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, et al. Flashshare: punching through server storage stack from kernel to firmware for ultra-low latency ssds。在*第13届USENIX操作系统设计与实现研讨会（OSDI 18）上*，第477-492页，2018。

[79] 赵继申，Onur Mutlu，和谢元。Firm:为持久性内存系统提供公平和高性能的内存控制。In *2014 47th Annual IEEE/ACM International Symposium on Microarchitec- ture*, pages 153-165.IEEE，2014年。

[80] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson.Ziggurat：用于非易失性主存储器和磁盘的分层文件系统。In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 207-219, 2019.

[81] 周心静，Joy Arulraj，Andrew Pavlo，和David Cohen。Spitfire:一个用于易失性和非易失性内存的三层缓冲区管理器。在*2021年国际数据管理会议上*，第2195-2207页，2021年。