非阻塞写入文件

Daniel Campello and Hector Lopez

Abstract—将数据写入文件系统页面缓存中不存在的页面会导致操作系统首先同步地将页面提取到内存中。同步页面提取定义策略（何时）和机制（如何），并总是阻止写入过程。非阻塞写入通过将写入数据缓存在存储器中的其它地方并立即解除写入过程而消除了这种阻塞。后续读取到更新的页面位置也是非阻塞的。这种对非高速缓存页面进行写操作的新处理允许进程将更多计算与 I / O 重叠，并通过增加提取并行性来提高页面提取 I/O 吞吐量。我们的实证评估表明，当工作负载无法从中受益时，无阻塞写入可以提高系统的整体性能，而不会降低性能。在 Filebench 写入工作负载中，无阻塞写入使用磁盘驱动器时的基准测试吞吐量平均提高了 7 倍（高达 45.4 倍），使用 SSD 时平均提高了 2.1 倍（高达 4.2 倍）。对于 SPECsfs2008 基准测试，非阻塞式写入可将 NFS 操作的整体平均延迟时间缩短 3.5%至 70%，平均写入延迟时间可降至 65%至 79%。重放 MobiBench 文件系统跟踪时，非阻塞写入将平均操作延迟降低 20-60%。

Index Terms—

1 Introduction

在操作系统（OS）内存中缓存和缓存文件数据是一项关键性能优化，已经流行了四十多年。操作系统以页面为单位缓存文件数据，在需要的时候将页面无缝地从后台存储器中读取到存储器中，这是由进程读取或写入的。这个基本的设计也延续到网络文件系统，客户端通过网络将页面提取发送到远程文件服务器。这种设计的一个不希望的结果是，在页面提取期间进程被 OS 阻塞。

虽然在读取非缓存页面的情况下阻止页面获取的过程是不可避免的，但在写入的情况下可以完全消除。操作系统可以缓冲暂时写在内存中的数据，并立即解除进程的阻塞。读取和更新页面可以异步执行。应用程序进程对页面写入请求与 OS 级别页面更新的解耦允许两个关键的性能增强。首先，进程可以自由地进行，而不必等待缓慢的页面读取 I/O 操作完成。其次，页面提取操作的并行性增加; 这提高了页面读取吞吐量，因为存储设备在更高级别的 I/O 并行性上提供更高的性能。

在本文中，我们将探索新的设计方案，并针对非阻塞式写入进行优化，解决一致性和正确性问题，并对这些思想进行实现和评估。通过将页面提取策略与提取机制分开，我们实现并评估两个页面提取策略：异步和懒惰，以及两个页面提取机制：前景和背景。我们还开发非阻塞式读取到最近在非缓存页面中写入数据。

我们在 Linux 内核的文件中实现了非阻塞写操作。我们的实现可以在操作系统内无缝运行，不需要修改应用程序。我们将处理写入本地文件系统和网络文件系统客户端的非缓存文件数据集成到一个通用的设计和实现框架中。而且由于它建立在一个通用的设计上，我们的实现为其他操作系统中的类似实现提供了一个起点。

我们使用多个文件系统工作负载评估了非阻塞式写入。在执行写入操作的 Filebench 工作负载中，使用非阻塞式写入时，平均基准测试吞吐量在使用磁盘驱动器时提高了 7 倍（高达 45.4 倍），在使用 SSD 时提高了 2.1 倍（高达 4.2 倍）。对于 SPECsfs2008 基准测试工作负载，通过改变 NFS 写入操作和 NFS 读取操作的比例，非阻塞写入减少了 NFS 操作的整体平均延迟 3.5%到 70%，平均写入延迟 65%到 79%。重放 MobiBench 文件系统跟踪时，非阻塞写入将平均操作延迟降低 20-60%。最后，由非阻塞写入引入的开销是微不足道的，当工作负载不能从中受益时不会损失性能。

2 Motivating Non-blocking Writes

我们使用多个文件系统工作负载评估了非阻塞式写入。在执行写入操作的 Filebench 工作负载中，使用非阻塞式写入时，平均基准测试吞吐量在使用磁盘驱动器时提高了 7 倍（高达 45.4 倍），在使用 SSD 时提高了 2.1 倍（高达 4.2 倍）。对于 SPECsfs2008 基准测试工作负载，通过改变 NFS 写入操作和 NFS 读取操作的比例，非阻塞写入减少了 NFS 操作的整体平均延迟 3.5%到 70%，平均写入延迟 65%到 79%。重放 MobiBench 文件系统跟踪时，非阻塞写入将平均操作延迟降低 20-60%。最后，由非阻塞写入引入的开销是微不足道的，当工作负载不能从中受益时不会损失性能。

2.1 The fetch-before-write problem

文件系统中的页面读取行为是由于数据访问粒度的不匹配造成的：应用程序访问的字节和操作系统从存储访问的页面。为了处理写入引用，目标页面在应用写入之前被同步提取，导致提前读取要求。如图 1 所示。这种阻塞行为会影响性能，因为它需要从主设备中获取比主设备慢得多的数据。今天，主内存访问可以在几纳秒内完成，而对闪存驱动器和硬盘驱动器的访问则分别需要几百微秒到几毫秒。我们确认了 BSD（所有变体），Linux，Minix，OpenSolaris 和 Xen 的最新开源内核版本的页面预写入行为。

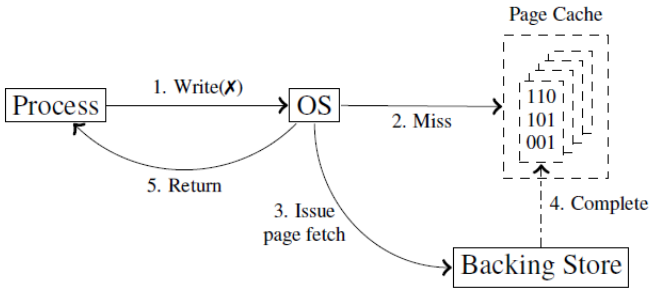

Figure 1: **Anatomy of a write.** The first step, a write reference, fails because the page is not in memory. The process resumes execution (Step 5) only after the blocking I/O operation is completed (Step 4). The dash-dotted arrow represents a slow transition.

2.2 Addressing the fetch-before-write problem

非阻塞式写入通过为更新的页面创建内存补丁并立即解除进程，消除了先取后提的要求。图 2 说明了这种修改。

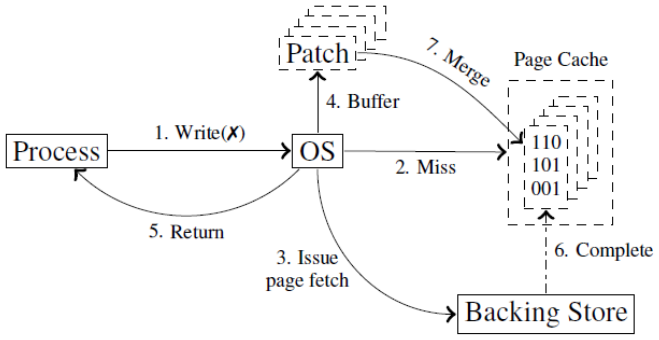

Figure 2: **A non-blocking write employing *asynchronous fetch*.** The process resumes execution (Step 5) after the patch is created in memory while the originally blocking I/O completion is delayed until later (Step 6). The dash-dotted line represents a slow transition.

2.2.1 Reducing Process blocking

当部分覆盖一个或多个非缓存文件页时，进程将被阻止。这种重写可以是任何大小，只要它们不是完全对齐到页面边界。图 3 说明了非阻塞写入如何减少进程阻塞。以前的研究已经报道了生产文件系统工作量中很大一部分小的或未对齐的写入。然而，对于部分页面覆盖行为知之甚少。为了更好地理解生产工作负载中这种文件写入的普遍性，我们开发了一个 Linux 内核模块，用于截取文件系统操作和报告大小以及写入块对齐。然后，我们分析了从佛罗里达国际大学计算机科学系的几台生产机器收集的一天的文件系统操作。除此之外，我们还分析了 MobiBench 中可用的文件系统痕迹，其持续时间短得多（两分钟）。表 1 提供了我们分析的所有痕迹的描述。

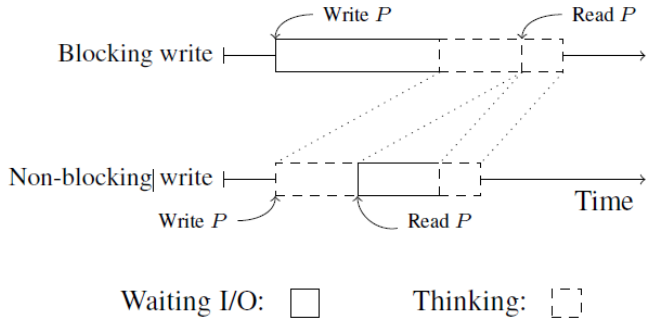

Figure 3: **Page fetch asynchrony with non-blocking writes.** Page P, not present in the page cache, is written to. The application waits for I/O completion. A brief thinktime is followed by a read to P to a different location than the one written to earlier. With non-blocking writes, since the write returns immediately, computation and I/O are performed in parallel.

图 4 提供了这些机器上每个写入流量的分析。平均而言，63.12%的写入涉及部分页面覆盖。根据页面缓存的大小，这些覆盖可能导致在页面更新之前不同程度的页面提取。页面提取的程度还取决于工作负载中的数据访问的局部性，其中写入可以按照短暂的时间顺序跟随读取。为了解释访问局部性，我们使用缓存模拟器来提高我们的估计，以计算实际上导致以各种内存大小进行页面提取的写入次数。这样的写入可以做成非阻塞的。缓存模拟器使用了一个改进的 Mattson 的 LRU 堆栈算法，并且使用这样的观察：在给定的 LRU 缓存大小下的非阻塞写入在所有更小的缓存大小下也将是非阻塞写入。对原始算法的修改涉及将所有部分页面覆盖计数为不在缓存中的页面作为非阻塞写入。图 5 显示了可以从表 1 中的工作负载的非阻塞式写入中受益的总写入的百分比。对于大多数工作负载，即使对于大小为 100GB 的大型页面高速缓存，该值也至少为 15%。一个可以使这种写入非阻塞的系统将使得整个写入性能较少依赖于页面缓存容量。

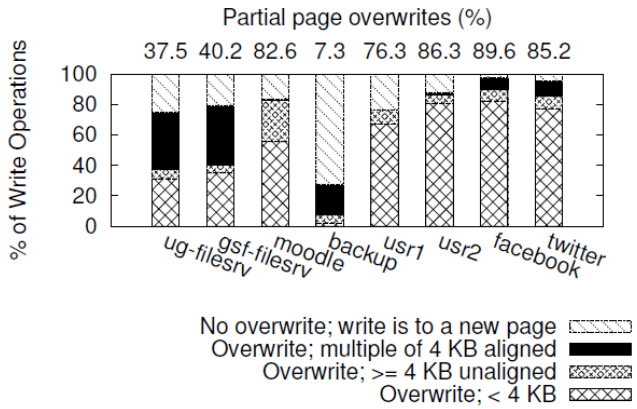

Figure 4: **Breakdown of write operations by amount of page data overwritten.** Each bar represents a different trace and the number above each bar is the percentage of write operations than involve at least one partial page overwrite.

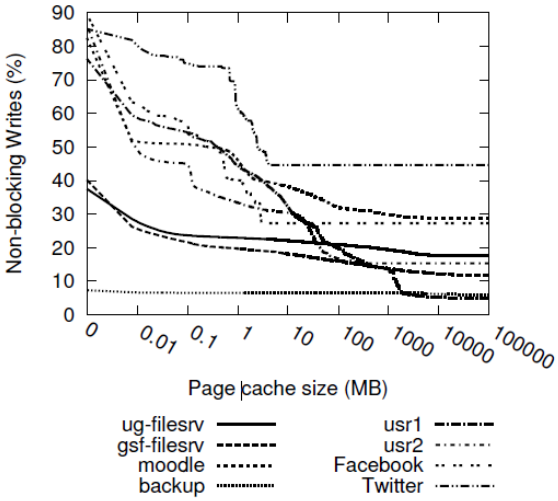

Figure 5: **Non-blocking writes as a percentage of total write operations when varying the page cache size.**

2.2.2 Incresaing Page fetch parallelism

在执行期间访问多个不驻留在内存中的页面的操作被操作系统阻止，每个页面访存一次。因此，操作系统最终会对彼此独立的访问进行序列化页面提取。通过非阻塞式写操作，操作系统允许进程并行获取独立页面，从而更好地利用设备级别的可用 I / O 并行机制。图 6 用图形描述了这种改进。更高级别的 I / O 并行性可以提高设备 I / O 吞吐量，最终提高应用程序的页面读取吞吐量。

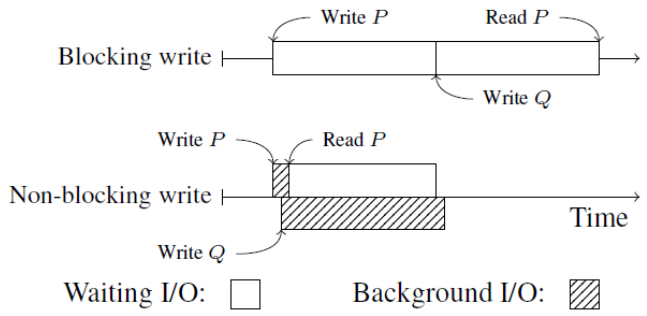

Figure 6: **Page fetch parallelism with non-blocking writes.** Two non-cached pages, P and Q, are written in sequence and the page fetches get serialized by default. With non-blocking writes, P and Q get fetched in parallel increasing device I/O parallelism and thus page fetch throughput.

2.2.3 *Making Durable Writes Fast*

至少最初，与当今基于块的持久性存储相比，下一代字节可寻址的持久性存储器可能相对较小。在今天的智能手机中的主要记忆被认为是准非易失性的。当这样的存储器被用作持久的文件系统缓存时，容纳设备能够提供极快的耐久性（即同步操作），这是一种通常会阻止进程执行的功能。在这样的系统中，耐久性机制前端的任何阻塞（例如写入之前的读取）对性能都是有害的。由于非阻塞式写入操作允许更新而不必访存页面，因此当字节可寻址的持久性存储器被广泛部署时，它代表了数据持久性极高的最终环节。

2.3 Addressing Correctness

在非阻塞写入的情况下，系统中进程内和进程之间的读写操作顺序可能会改变。正如我们稍后将详细阐述的那样（§3.3），非阻塞写入中的补丁创建和补丁应用机制确保依赖于因果关系的操作的顺序被保留。我们使用的关键见解是：（i）使用最近创建的修补程序正确地提供对最近更新的读取，（ii）读取在取回页面上的块仅在应用所有未完成修补程序之后才允许进行，以及 iii）由相同或不同线程独立并且发布的读取和写入可以被重新排序，而不会损失正确性。

另一个有关非阻塞写入的潜在问题是数据的持久性。对于文件数据，我们观察到异步写入操作仅修改易失性存储器，操作系统不保证修改是持久的。在非阻塞写入的情况下，同步写入（由于 sync / fsync 或周期性页面刷新守护程序）将等待所需的提取，应用所有未完成的修补程序，并在解除阻止进程之前将页面写入存储。因此，系统的耐久性属性在非阻塞写入时保持不变。

3 Non-blocking Writes

操作系统为应用程序写入服务，如图 7 所示。在 Check Page 状态下，它在页面缓存中寻找页面。如果页面已经在内存中（由于最近的抓取完成），它将移动到“更新页面”状态，这也将页面标记为脏。如果页面不在内存中，则发出页面提取 I / O 并进入等待状态，等待页面在内存中可用。当 I / O 完成时，页面是最新的并且准备好被解锁（在页面状态图中表示最新和可访问）。在“更新页面”状态中，操作系统使页面可访问。最后，控制流程返回到执行页面写入的应用程序。

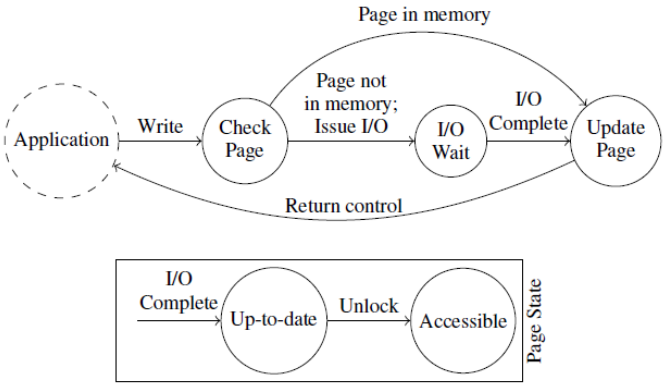

Figure 7: **Process and page state diagram for page fetch with blocking writes.**

3.1 Approach Overview

页面获取进程阻止进程执行，这是不可取的。非阻塞式写入通过在 OS 内存中创建补丁以稍后应用来缓存对非高速缓存页面的更新而起作用。如图 8 所示，基本方法修改了页面读取路径。与当前系统不同，非阻塞写入消除了 I / O 等待状态，该状态会阻塞进程，直到页面在内存中可用。相反，创建一个更新补丁后，立即返回一个非阻塞写入，并将其排队等待到页面更新列表。非阻塞写入在页面状态“过时”中添加一个新状态，该状态反映页面在读入内存之后但未应用临时修补程序之前的状态。一旦应用了所有待处理的补丁，页面就转换到最新状态。

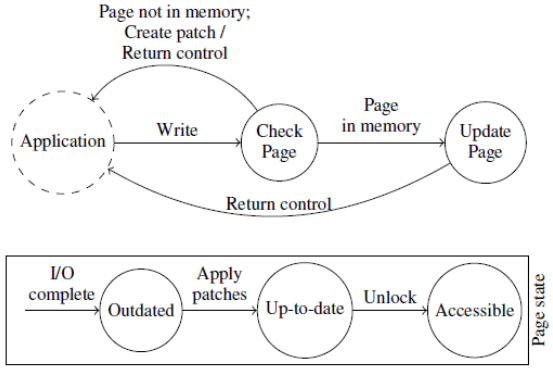

Figure 8: **Process and page state diagram for page fetch with non-blocking writes.**

非阻塞写入改变写入控制流程，从而影响对最近写入的数据的读取。而且，他们需要以补丁的形式管理额外的缓存数据。本节的其余部分将在通用系统设计的上下文中讨论这些细节，以及特定于 Linux 的实现。

3.2 Write Handling

操作系统允许通过两种常见机制写入文件数据：监督系统调用和无监督内存映射访问。

为了处理监督写操作，操作系统使用系统调用参数 - 要写入的数据缓冲区的地址，数据的大小以及要写入的文件（以及隐式地，偏移量），并解析对数据的访问页面在内部写入。在非阻塞式写操作的情况下，操作系统从系统调用参数中提取数据更新，创建一个补丁并将其排入队列供以后使用。稍后将数据页读入内存时应用此修补程序。

内存将文件的一部分映射到进程地址空间，可以提供无监督的文件访问。在我们当前的设计中，内存映射访问是通过阻止进程来处理页面错误来像当前系统一样处理的。

3.3 Patch Management

我们现在讨论如何创建补丁，存储在操作系统中，并在将其提取到内存后应用到页面。

3.3.1 Patch Creation

补丁必须包含要写入的数据及其目标位置和大小。由于商品操作系统以页面的粒度处理数据，我们选择了一个设计，每个补丁将应用于单个页面。因此，我们使用页面补丁数据结构来抽象更新，其中包含所有要修补的信息并使页面保持最新状态。为了处理多个不相交的覆盖到同一个页面，我们实现了每个页面的修补程序队列，其中页面修补程序排队并随后以 FIFO 的顺序应用到页面。因此，通过页表共享页面或以其他方式正确处理。这是可能的，因为操作系统保持页面到物理内存帧的一对一映射（例如，Linux 中的 struct page 或 OpenBSD 中的 struct vm_page）。当新数据相邻或覆盖现有的补丁时，它将相应地合并到现有的补丁中。这使修补程序内存开销和修补程序应用程序开销与在页面中更改的页面字节数成比例，而不是从页面上次从内存中删除后写入页面的字节数。

3.3.2 Patch Application

补丁程序相当简单。当通过系统调用引起的页面读取或者导致页面错误的存储器映射访问来读取页面时，第一步是将未完成的补丁（如果有的话）应用于页面以使其在页面是可访问的。通过将补丁数据简单地复制到目标页面位置来应用补丁。补丁应用发生在页面读取完成事件的下半部分中断处理中（在 §5 中进一步讨论）。一旦所有的补丁被应用，页面被解锁，这也解除了在页面上等待的进程的阻塞（如果有的话）。

3.4 Non-blocking Reads

类似于写入，读取也可以被分类为有监督的和无监督的。读取到非缓存页面会阻止当前系统中的进程。使用非阻塞式写入，可以实现非阻塞式读取的新机会。具体来说，如果读取可以从页面上排队的一个补丁中得到维护，那么读取过程可以立即解除阻塞，而不会发生页面读取 I / O。这发生在不损失正确性的情况下，因为修补程序包含写入页面的最新数据。如果所请求数据的任何部分不包含在修补程序队列中，则读取不可用。在这种情况下，读取过程会阻止页面被提取。如果请求的所有数据都包含在修补程序队列中，则将数据复制到目标缓冲区，并立即解除读取过程。对于无监督读取，我们目前的设计在所有情况下阻止页面提取的过程。

4 Alternative Page Fetch Modes

让我们考虑在执行非阻塞写入时在步骤 3 中发出的页面提取操作，如图 2 所示。此操作需要物理内存分配（用于提取页面）和后续异步 I / O 来提取页面以便新创建的补丁可以应用到页面上。但是，由于避免了阻塞，进程执行不依赖于内存中可用的页面。这就提出了一个问题：页面分配和获取可能被延期甚至被淘汰？页面取消延迟和消除允许减少和整形内存消耗和页面提取 I / O 到存储。虽然页面提取延期是机会主义的，但只有创建的修补程序足以完全覆盖页面，或者页面持久化变得不必要时，才能删除页面。我们现在探索一下在非阻塞写入的情况下可能的页面抓取模式。

4.1 Asynchronous Page Fetch

在这种模式下，页面读取 I / O 排队在写入页面时发出。这种方法的吸引力在于其简单性。由于页面以及时的方式被带入内存，类似于同步获取，因此对基于计时器的持久性机制（例如脏页面刷新和文件系统日志）是透明的。

异步页面提取定义策略。但是，在发布页面提取之前，其机制可能会涉及额外的阻塞。我们讨论两个突出这个问题的替代页面抓取机制。

1) 前台异步页面提取（NBWAsync-FG）。页面读取 I / O 是在执行写入文件页面的过程的上下文中发出的。前面几节我们的讨论是基于这个机制。尽管进程不会等待数据提取的完成，但是如果这些元数据页面未被缓存在 OS 存储器中，那么发布数据页面的提取 I / O 本身可能涉及检索额外的元数据页面以定位数据页面。如果是这样的话，写入过程将不得不阻止必要的元数据提取来完成，从而消除了非阻塞写入的大部分好处。
2) 背景异步页面提取（NBWAsync-BG）。写入过程通过使用内核工作线程将所有必要的工作移动到不同的上下文。这种方法消除了由于元数据丢失而导致的写入过程的任何阻塞; 一个工作线程阻塞所有的提取，而发行过程继续执行。

同步取指是一个有价值的改进。但是，它会占用系统资源，为要提取的页面分配系统内存，并使用存储 I / O 带宽来提取页面。

4.2 Lazy Page Fetch(NBW-Lazy)

当一个进程写入一个非缓存的数据页面时，它的执行并不取决于内存中可用的页面。由于懒惰的页面提取，操作系统延迟了页面提取，直到变得不可避免。延迟页面获取有可能进一步减少系统的资源消耗。图 9 说明了这个选择。

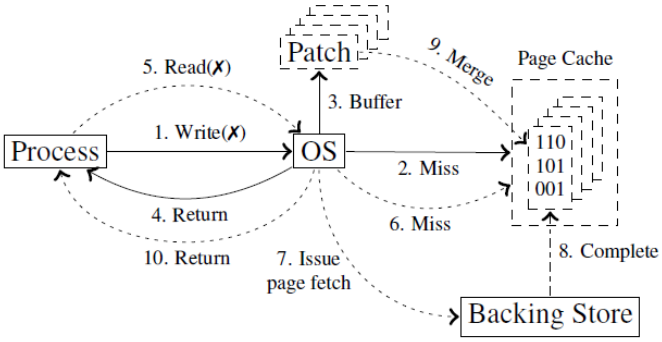

Figure 9: **A non-blocking write employing *lazy fetch*.** The process resumes execution (Step 4) after the patch is created in memory. The Read operation in Step 5 optionally occurs later in the execution while the originally blocking I/O is optionally issued and completes much later (Step 8). The dash-dotted arrow represents a slow transition.

懒惰的页面抓取创建新的系统场景，必须仔细考虑。如果使用非缓存页面的当前可用补丁无法提供未来的页面读取，页面抓取变得不可避免。在这种情况下，页面被同步取出，在解除读取过程之前先应用补丁。如果页面被完全覆盖，或者由于其他原因（例如，包含文件被删除）而页面持久性变得不必要，则原始页面提取被完全消除。

页面数据持久性在以下情况下可能变得必要：（i）由应用程序同步文件写入，（ii）由 OS 定期刷新脏页面，或者（iii）

如在日志文件中的有序页面写入系统。在所有这些情况下，页面在刷新到后备存储之前被同步取出。最后，非阻塞写入不使用传统的持久性机制的元数据页面。与持久性有关的问题将在第 5.2 节中进一步讨论。

5 Implementation

非阻塞写入会改变当前系统的行为和控制流。我们提供一个非阻塞写入的实现的概述，并讨论与它如何保持系统正确性有关的细节。

5.1 Overview

我们通过修改通用虚拟文件系统（VFS）层，在 Linux 内核（版本 2.6.34.17）中实现了文件数据的非阻塞写入。与传统的 Linux 方法不同，在下半部 I / O 完成处理程序中，所有对提取完成的处理（如应用修补程序，标记页面脏，处理日志事务以及解锁页面）都会发生。

5.2 Handling Correctness

OS 启动的页面访问。我们的实现并没有实现非阻塞写访问（写入和读取）到操作系统内部启动的未缓存的页面。这些包括文件系统元数据页面更新和内核线程执行的更新。这个实现提供了 OS 服务期望的持久性属性，以保持语义的正确性。

日志文件系统。通过允许各种日志模式的预期行为，我们的非阻塞写入的实现保留了日志文件系统的正确性。例如，非阻塞写入保留 ext4 的有序模式日志不变，即在包含相关元数据更新的事务之前将数据更新刷新到磁盘。ext4 中的元数据事务不会被处理，直到相关的数据页被提取到内存中，应用了优秀的补丁，页面被标记为脏，并且脏事务缓冲区被添加到事务处理程序。因此，与元数据事务相关的所有脏数据页都驻留在内存中，并在提交事务之前通过 ext4 的有序模式日志记录机制刷新到磁盘。

处理读写相关性。在操作系统中处理非阻塞写入时，可以向涉及的页面发出多个操作，例如读取，预取，同步写入和刷新。操作系统仔细地同步这些操作以保持一致性，并仅将最新的数据返回给应用程序。我们的实现尊重 Linux 页面锁定协议。一个页面被分配后，并在发出一个提取之前被锁定。因此，也支持 fsync 和 mmap 等内核机制。这些机制阻止页面锁定，该页面锁定仅在页面被提取并且在进行页面操作之前应用补丁之后变为可用。当使用延迟的页面提取机制（如 NBW-Async-BG 和 NBW-Lazy）时，在分配页面之前，在页面缓存映射中添加所涉页面的 NBW 条目。此 NBW 条目允许锁定页面以维护页面操作的顺序。当必要时（例如，同步），索引为 NBW 的页面被获取，这又涉及获取页面锁定，从而同步页面上的未来操作。这种页面锁定的唯一例外是写入已经处于非阻塞写入状态的页面; 写入不锁定页面，而是排队一个新的补丁。

页面更新的顺序。非阻塞写入可能会改变应用不同页面的补丁的顺序，因为页面提取可能会乱序完成。非阻塞式写入只会将不能保证以任何特定顺序反映到持久性存储的内存写入。因此，在更新内存页面时排序违规是安全的。

页面持久性和同步。如果应用程序需要显式的磁盘排序来更新内存页面，则会在每个操作之后执行一个阻塞刷新操作（例如，fsync）。刷新操作会导致操作系统强制抓取任何以 NBW 为索引的页面，即使尚未分配。然后操作系统获得页面锁定，等待页面抓取并应用任何未完成的补丁，然后刷新页面并将控制返回给应用程序。磁盘写入的顺序因此通过非阻塞写入来保存。

处理磁盘错误。当处理写入非缓存页面时，我们的实现改变了操作系统的语义，就通知 I / O 错误。由于写入页面读取是异步完成的，因此在异步页面读取操作期间磁盘 I / O 错误（例如，为 UNIX 写入系统调用返回的 EIO）将不会被报告给写入应用程序进程。应用程序根据所报错误采取的任何操作都不会执行。在语义上，应用程序写入是内存写入，而不是持久性存储; 当前系统报告的 I / O 错误是先取后写设计的伪像。在非阻塞式写入的情况下，如果通过应用程序或操作系统发出的刷新在任何时候使写入持久化，则页面刷新期间的任何 I / O 错误都将被报告给发起者。

多核和内核抢占。我们的实现完全支持 SMP 和内核抢占。对于给定的非缓存页面，修补程序创建机制（处理写入系统调用时）可以与修补程序应用程序机制（处理页面获取完成时）进行竞争。我们的实现使用一个额外的锁来保护修补程序队列免受同时访问。

6 Evaluation

我们解决以下问题：

1) 对于不同的工作负载，无阻塞写入有什么好处？
2) 非阻塞写入的获取模式如何相对于彼此执行？
3) 对基础存储类型的非阻塞写入有多敏感？
4) 内存大小如何影响非阻塞式写入？

我们评估四种不同的解决方阻塞写入（BW）是处理写入的传统方法，并使用 Linux 内核实现。非阻塞写入变体包括使用前景（NBW-Async-FG）和背景（NBW-Async-BG）提取的异步模式，以及懒惰模式（NBW-Lazy）。

工作量和实验设置。我们使用 Filebench 微基准来使用受控工作负载来解决（1），（2），（3）和（4）。我们使用 SPECsfs2008 基准并重放 MobiBench 跟踪来进一步分析问题（1）和（2）。MobiBench 跟踪重播也有助于回答问题（3）。Filebench 和 MobiBench 评估在四核 2.50GHz AMD Opteron™1381 处理器，8GB 内存，500GB WDC WD5002ABYS 硬盘，32GB Intel X25-E SSD 和千兆以太网的机器上运行，运行 Gentoo Linux（内核 2.6.34.14）。上述设置也用于运行 SPECsfs2008 基准测试的客户端组件。另外，在 SPECsfs2008 基准测试中，NFS 服务器使用了一个 2.3GHz 四核 AMD 皓龙 ™ 处理器 1356,7GB RAM，500GB WDC 和 160GB 希捷磁盘以及运行 Gentoo Linux（内核 2.6.34.14）的千兆以太网。500GB 硬盘容纳根文件系统，而 160GB 硬盘容纳 NFS 输出数据。客户端和服务器之间的网络连接是千兆以太网。

6.1 Filebench Micro-benchmark

对于以下所有实验，在清除 OS 页面缓存的内容之后，我们使用 5GB 预分配文件运行五个 Filebench 个性达 60 秒。每个人格都代表着不同类型的工作量。该系统被配置为使用 4GB 的主存储器，并且用于补丁的存储器被限制为 64MB，这是 DRAM 的一小部分，以避免显着影响可用于工作负载和 OS 的 DRAM。我们报告 Filebench 性能指标，即每秒的操作数。每个数据点使用 3 次执行的平均值计算。

6.1.1 Performance Evaluation

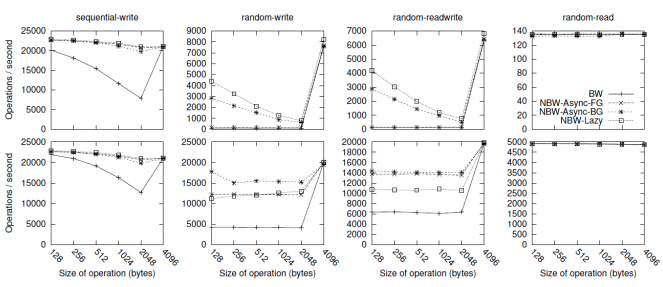

Figure 10: **Performance for various Filebench personalities when varying the I/O size.** The two rows correspond to two different storage back-ends: hard disk-drive (top) and solid-state drive (bottom).

我们首先检查使用硬盘作为存储后端时的 Filebench 性能。图 10 的第一行描述了在改变 Filebench 操作的大小时，四个 Filebench 特性的性能。每个数据点报告 3 次执行的平均值。测量的标准误差小于 96.88%的平均值的 3%，其余的小于 10%。

前三个地块涉及执行写作操作的人物。在 4KB I / O 大小的情况下，不存在先取后写（fetch-before-write）行为，因为每次写入都会覆盖整个页面; 因此，非阻塞写入不被使用，也不施加任何开销。

对于顺序写入特性，具有阻塞写入（BW）的性能取决于操作大小，并受每次操作的页面未命中次数的限制。在最坏的情况下，当 I / O 大小等于 2KB 时，每两次写入都涉及阻塞获取。平均而言，根据 I / O 大小的不同，非阻塞式写入模式可提供 13-160%的性能提升。

第二和第三个人物代表随机访问工作量。随机写入是一个只写工作负载，而随机读写是一个混合工作负载; 后者使用两个线程，一个用于发出读取，另一个用于写入。对于小于 4KB 的 I / O 大小，BW 为随机写入和随机读写特性分别提供了大约 97 和 146 次操作/秒的恒定吞吐量。无论 I / O 大小如何，性能都是一致的，因为每个操作同样可能导致页面遗漏和获取。由于使用两个线程时额外的可用 I / O 并行性，随机读取执行比随机写入更好。此外，对于随机写入，NBW-Async-FG 提供了 50-60%的性能改进，这是由于对进程的页面抓取的阻塞减少了。然而，这种改进并不表现为随机读取，其中读取操作由于对正在进行读取的页面的附加阻塞而引起较高的等待时间。在这两种情况下，NBW-Async-FG 与其他非阻塞式写入模式相比的优势明显较低，因为在此短时间运行实验期间，许多初始文件系统元数据中的 NBW-Async-FG 模块未命中。

相比之下，NBW-Async-BG 立即解除阻塞进程，而根据需要获取元数据的不同内核线程块。该模式显示随机写入性能提高了 6.7x-29.5x，具体取决于 I / O 大小。随着 I / O 大小的增加，这些性能增益会降低，因为非阻塞式写入可以创建更少的未完成的修补程序，以符合 64MB 的强制修补程序内存限制。对于随机读取，观察到类似的趋势，性能改进从 3.4x-19.5x 变化，取决于所使用的 I / O 大小。NBW-Lazy 通过同时消除数据和元数据页面抓取，可以提供高达 45.4 倍的性能提升。当达到可用补丁内存限制时，写入被视为 BW，直到更多补丁内存被释放。

随机读取和顺序读取（未示出）的最后两个特性是只读工作负载。这些工作负载不会创建写入操作，使用非阻塞写入内核的开销为零。非阻塞写入提供与阻塞写入相同的性能。

6.1.2 Sensitivity to system parameters

我们对非阻塞写入的敏感性分析解决了以下具体问题：

1) 使用不同的存储后端时，非阻塞式写入有什么好处？
2) 当系统内存大小变化时，非阻塞写入如何执行？

Sensitivity to storage back-ends

为了回答第一个问题，我们评估了使用基于固态驱动器（SSD）的存储后端的非阻塞写入。图 10（底部一行）显示使用固态驱动器运行 Filebench 个性时的结果。每个数据点报告 3 次执行的平均值。测量的标准误差小于平均值的 2.25%，除了 5%。

顺序写入工作负载的性能趋势与所有非阻塞写入模式下的硬盘对应部分（图 10 中的最上面一行）几乎相同。这是因为非阻塞写入完全消除了访问两个系统中的每个操作的存储延迟。另一方面，由于 SSD 比硬盘驱动器提供更好的吞吐量，所以 BW 为每个小于 4KB 的大小提供了吞吐量的增加。总之，根据 I / O 大小的不同，非阻塞式写入模式提供了 4%到 61%的性能提升。

对于随机写入和随机读取工作负载，非阻塞式写入变体都会不同程度地提高性能。SSD 相对于硬盘驱动器的随机访问具有显着较低的延迟，这使得元数据未命中得到更快速的服务。NBWAsync-FG 相对于 BW 的效率相对于硬盘系统有了进一步的提高，分别为随机写入和随机读取提供了 188%和 117%的性能提升。NBW-Async-BG 的改进与 NBW-Async-FG 相似，原因与硬盘类似。NBW-Async-BG 分别提供了 272%（在最好的情况下高达 4.2X）和 125%的性能改善，平均随机写入和随机读取的平均水平。最后，虽然 NBW-Lazy 的性能明显好于 BW，但与我们的预期相反，与 NBW-Async 模式相比，其性能提升更低。经过进一步的调查，我们发现当补丁内存限制达到时，NBW-Lazy 会比其他模式花费更长的时间来释放内存，因为只有在阻止无法避免时才会发出提取。虽然实验的持续时间与磁盘驱动器相同，但更快的 SSD 会使修补程序内存限制更快地被满足。在我们当前的实现中，在达到补丁内存限制并且不能创建更多补丁之后，NBW-Lazy 默认为 BW 行为发布，同步提取以处理对非高速缓存页面的写入。尽管存在这个缺点，NBW 懒惰模式分别显示随机写入和随机读写的 163%-211%和 70%的改善 BW。

Sensitivity to system memory size

我们使用 Filebench 工作负载来回答第二个问题，并改变操作系统可用的系统内存量。对于这些实验，我们使用硬盘驱动器作为存储后端，并将 I / O 大小固定为 2KB。图 11 显示了这个实验的结果。每个数据点报告 3 次执行的平均值。测量的标准误差小于 90%的平均值的 4%，其余的小于 10%。

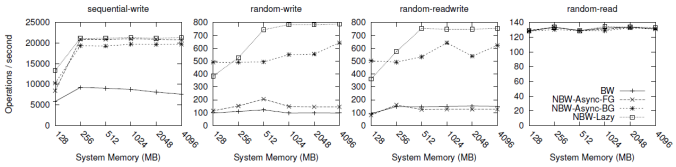

Figure 11: **Memory sensitivity of Filebench.** The I/O size was fixed at 2KB and patch memory limit was set to 64MB.

对于顺序写入工作负载，非阻塞写入变体比 BW 执行 45-180%。进一步 NBW 懒惰表现更好，可以认为是最佳的，因为（一）它使用很少的补丁存储器，足以容纳足够的补丁，直到整个页面被覆盖，（二）由于页面完全覆盖写入顺序，消除所有的页面抓取。

对于随机写入和随机读写工作负载，NBW-Async-FG 提供与 BW 相对一致的性能; 这些解决方案所实现的 I / O 性能不足以使存储器的相关性出现差异。NBW-Async-BG 和 NBW-Lazy 相对于 BW 的性能增益分别高达 560%和 710%。使用 NBW-Lazy 时，性能会随着更多可用内存的提高而改善，但只能达到在执行完成之前达到强制内存限制的时间点。增加补丁内存限制将允许 NBWLazy 继续扩展其性能。

6.2 SPECsfs2008 Macro-benchmark

SPECsfs2008 基准测试 NFS 服务器的性能。对于这个实验，我们在以异步模式导出网络文件系统的 NFS 服务器上安装了非阻塞写内核。SPECsfs2008 使用客户端工作负载生成器，完全绕过页面缓存。客户端配置为每秒 500 个操作的目标负载。目标负荷在所有评估中都得以维持; 因此 SPECsfs2008 性能指标是 NFS 客户端报告的操作延迟。尽管评估结果令人鼓舞，但我们报告的 NFS 工作负载的相对性能结果可能被低估。这是因为我们的原型仅用于 NFS 服务器; 非阻塞写入的客户对手并没有参与这个基准测试。

SPECsfs2008 操作分为写入，读取和其他包含元数据操作（如 create，remove 和 getattr）的操作。对于每个不同的解决方案，我们分别报告上述三类业务的结果以及代表所有业务的加权平均的整体业绩。此外，我们在改变基准测试所发布的 NFS 操作的相对比例时评估了性能。SPECsfs2008 中指定的默认配置是：读取（18%），写入（10%）和其他（72%）。我们还评估了三个修改后的配置：不写，不读，以及使用：读（10%），写（18%）和其他（72%）来检查更广泛的行为。

我们首先对工作量进行简要分析以确定预期的性能。即使对于包含比读取更多的写入的配置（例如，18%写入和 10%读取），写入时的实际高速缓存未命中部分远低于由读取引起的未命中部分（即 16.9%写入未命中与 83.1%读取未命中）。注意到每个对非高速缓存页面的读取访问都会导致读取未命中，但在页面对齐时写入访问不同的情况下解释了这种不匹配。此外，表 2 还报告说，SPECsfs2008 发出的所有写入中只有 39%是部分页面覆盖，这可能导致非阻塞写入。

图 12 显示了使用 BW 解决方案的延迟标准化的平均操作延迟。不包括只读工作负载，主要的趋势是非阻塞写入模式显着减少了写入操作延迟，读取延迟很少或没有降低。此外，平均整体操作延迟与写入失败的比例成正比，并与 NFS 写入操作的延迟提高成正比。对于包含写入操作的三种配置，使用不同模式的非阻塞写入时，写入操作的延迟在 65%到 79%之间减少。由于某些页面上的附加阻塞，读取延迟略有影响。对于 BW，读取操作发布时，某些页面可能已经被读取到内存中。使用非阻塞式写入时，相应的读取可能会延迟或根本不发出，直到发生阻塞读取。对于没有写操作的配置，平均总延迟相对不受影响。

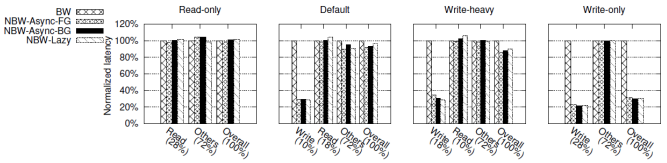

Figure 12: **Normalized average operation latencies for SPECsfs2008.**

6.3 MobiBench Trace Replay

MobiBench 工具套件包含使用 Facebook 和 Twitter 应用程序时从 Android 设备获取的痕迹。我们使用 MobiBench 的时间精确的重播工具来重播曲目。我们在使用之前修复了重放工具中的一个错误; 原始的播放器在打开文件时使用了一组固定的标志，而不管跟踪信息如何。MobiBench 将平均文件系统调用操作延迟报告为性能指标。我们重播了五次，并报告观察到的平均潜伏期。测量的标准误差小于平均值的 4%，除了 7.18%。图 13 的两个最左边的图分别为硬盘和固态驱动器提供了评估的结果。非阻塞写入展现了操作延迟在 20%和 40%之间的减少，取决于用于 Facebook 和 Twitter 追踪的模式和后端存储。

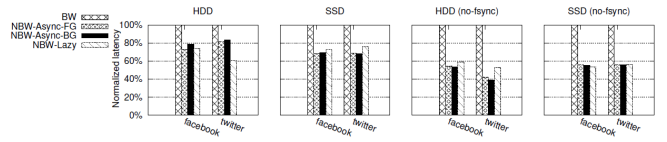

Figure 13: **Normalized average operation latencies when replaying MobiBench traces [14].**

当我们分析 MobiBench 的痕迹时，我们发现它们包含了大量的同步操作。同步操作不允许利用非阻塞式写入的全部潜力，因为它们阻止进程同步读取页面。如前所述，最近在字节寻址持久性存储器和 qNVRAM 上的工作提供了非常快速，持久的内存中操作。在这样的系统中，操作系统中的阻塞读写之前的行为成为性能的更重要的障碍。为了估计这种环境下非阻塞写入的影响，我们通过丢弃所有的 fsync 操作来修改原始记录，以模拟内存数据的极快的持久性。最右边的两个图表显示了重放修改的轨迹后获得的结果。根据使用的模式和存储后端，非阻塞写入可将延迟降低 40-60%。

7 Related Word

非阻塞式写入已经存在了近三十年，用于管理 CPU 缓存。观察到整个高速缓存行不需要被写入一个字写入未命中，从而拖延处理器，临时存储这些字更新的附加寄存器的使用进行了调查，后来通过。

最近，使用仪器化的 QEMU 机器仿真器生成的全系统内存访问跟踪，激发了对主内存页面的非阻塞写入。此前的工作概述了在商品操作系统中实现无阻塞写入的一些挑战。我们通过提供非阻塞式写入的详细设计和 Linux 内核实现来改进这项工作，解决了许多挑战，并揭示了新的设计要点。我们还提供了一个全面的评估，包含来自正在运行的系统的更广泛的工作负载和性能数据。

缓解先取后写问题的候选方法涉及配置足够的 DRAM 来最小化写入缓存未命中。但是，随着时间的推移，工作负载的文件系统占用空间通常是不可预知的，并且可能是无限的。或者，预取可以通过预测未来的内存访问来减少阻塞。但是，预取通常仅限于顺序访问。而且，错误的决策可能导致预取无效，污染记忆。非阻塞写入是对这些方法的补充。它明智地使用内存，只能获取进程执行所需的那些页面。

在文献中提出了几种专门用于系统调用引起的页面抓取的过程阻塞的方法。Linux 上可用的异步 I / O 库（例如，POSIX AIO）和一些 BSD 变体的目标是使文件系统写入异步; 一个辅助函数库线程代表进程阻塞。LAIO 是基本的 AIO 技术的一般化，使所有系统调用异步; 一个库检查点的执行状态，并依靠调度程序激活来获得有关在内核中启动的阻塞 I / O 操作完成的通知。最近，FlexSC 提出了异步无异步系统调用，其中系统调用在进程中在用户和内核空间共享的页面中排队; 这些调用是由 syscall 内核线程异步服务的，这些内核线程将完成报告回用户进程。

与上述建议有关的无阻塞写入的范围是不同的。其目标是完全消除对文件系统页面缓存中不可用页面的内存写入阻塞。非阻塞式写入不需要检查点状态，从而消耗较少的系统资源。此外，它可以配置为轻量级的，以便它不使用额外的线程（通常是系统中有限的资源）来代表正在运行的进程阻塞。最后，与这些需要修改应用程序的方法不同，非阻塞写入在对应用程序透明的操作系统中无缝工作。

有些作品是与无阻塞写作有关的，但是他们完成的目标却截然不同。投机性执行（或投机者）由 Nightingale 等人提出。在使用进程检查点和回滚机制将缓存的内存页面修改同步写入网络文件服务器时，消除了阻塞。Xsyncfs 通过创建写入的提交依赖性并允许进程进行，消除了将内存页面同步写入磁盘的阻塞。Featherstitch 通过更加智能地将这些页写入磁盘写入磁盘来提高同步文件系统页面更新的性能。Featherstitch 采用了补丁，但用于不同的目的 - 以字节粒度指定跨磁盘块的相关更改。OptFS 将内存页面的写入顺序从其耐久性中分离出来，从而提高了性能。尽管这些方法优化了将内存页面写入磁盘，但是在内存中修改文件页面之前，它们并没有消除阻塞页面取回。

BOSC 描述了一个新的磁盘更新接口，用于应用程序明确指定磁盘更新请求和关联回调函数。机会性日志描述了对象的读取前写入问题，并使用第二个日志来记录更新。这两个都减少了应用程序阻止，允许在后台进行更新，但是它们需要修改应用程序，不支持通用用途。非阻塞写入是对上述工作体系的补充，因为它在 OS 内无缝运行，不需要更改应用程序。

8 Conclusions and Future Word

四十多年来，操作系统在写入非缓存文件数据时，已经阻止了页面获取 I / O 的进程。在本文中，我们重新审视了这个完善的设计，并且证明这种阻塞不仅是不必要的，而且对性能也是有害的。非阻塞式写入通过缓冲 OS 内存中其他地方的页面更新，将数据的写入从内存中解除耦合。这种解耦是通过与应用程序无缝连接的自包含操作系统来实现的。我们设计并

实现了异步和懒惰的页面抓取模式，这些模式是阻止页面抓取的值得选择。我们使用 Filebench 对非阻塞式写入的评估显示，相对于阻塞式写入，各种工作负载类型的吞吐量性能提高了多达 45.4 倍。对于 SPECsfs2008 基准测试，非阻塞式写入操作减少了 65-79％的写操作延迟。在重放 MobiBench 文件系统跟踪时，非阻塞写入将平均操作延迟降低 20-60％。此外，当工作负载不能从非阻塞式写入中受益时，性能没有任何损失。

非阻塞写道打开了未来工作的几个途径。首先，由于它们以基本的方式改变了页面在内存中的相对重要性，所以新的页面替换算法是值得研究的。其次，通过智能调度页面抓取操作（而不是简单地异步或延迟），我们可以减少和调整内存消耗和页面抓取 I / O 流量到存储。第三，与非阻塞写入相关的异步页面读取相关的 I / O 可以更智能地调度（例如，作为后台操作或半抢先）以加速阻塞页面获取。最后，某些操作系统机制（如脏页面刷新阈值和每进程脏数据的限制）需要更新，以便也考虑内存中的修补程序。