Non-blocking Writes to Files

Daniel Campello[†] *Hector Lopez*[†] *Luis Useche*[◇*] *Ricardo Koller*[‡*] *Raju Rangaswami*[†]
[†]Florida International University [◇]Google Inc. [‡]IBM TJ Watson Research Center

Abstract

Writing data to a page not present in the file-system page cache causes the operating system to synchronously fetch the page into memory first. *Synchronous* page fetch defines both policy (*when*) and mechanism (*how*), and always blocks the writing process. *Non-blocking writes* eliminate such blocking by buffering the written data elsewhere in memory and unblocking the writing process immediately. Subsequent *reads* to the updated page locations are also made non-blocking. This new handling of writes to non-cached pages allow processes to overlap more computation with I/O and improves page fetch I/O throughput by increasing fetch parallelism. Our empirical evaluation demonstrates the potential of non-blocking writes in improving the overall performance of systems with no loss of performance when workloads cannot benefit from it. Across the Filebench write workloads, non-blocking writes improve benchmark throughput by 7X on average (up to 45.4X) when using disk drives and by 2.1X on average (up to 4.2X) when using SSDs. For the SPECsfs2008 benchmark, non-blocking writes decrease overall average latency of NFS operations between 3.5% and 70% and average write latency between 65% and 79%. When replaying the MobiBench file system traces, non-blocking writes decrease average operation latency by 20-60%.

1 Introduction

Caching and buffering file data within the operating system (OS) memory is a key performance optimization that has been prevalent for over four decades [7, 43]. The OS caches file data in units of pages, seamlessly fetching pages into memory from the backing store when necessary as they are read or written to by a process. This basic design has also carried over to networked file systems whereby the client issues page fetches over the network to a remote file server. An undesirable outcome of this design is that processes are blocked by the OS during the page fetch.

While blocking the process for a page fetch cannot be avoided in case of a read to a non-cached page, it can be entirely eliminated in case of writes. The OS could buffer the data written temporarily elsewhere in memory and unblock the process immediately; fetching and updating the page can be performed asynchronously. This decoupling of page write request by the application process from the OS-level page update allows two crucial performance enhancements. First, the process is free to make progress without having to wait for a slow page fetch I/O operation to complete. Second, the parallelism of page fetch operations increases; this improves page fetch throughput since storage devices offer greater performance at higher levels of I/O parallelism.

In this paper, we explore new design alternatives and optimizations for non-blocking writes, address consistency and correctness implications, and present an implementation and evaluation of these ideas. By separating page fetch policy from fetch mechanism, we implement and evaluate two page fetch policies: *asynchronous* and *lazy*, and two page fetch mechanisms: *foreground* and *background*. We also develop *non-blocking reads* to recently written data in non-cached pages.

We implemented non-blocking writes to files in the Linux kernel. Our implementation works seamlessly inside the OS requiring no changes to applications. We integrate the handling of writes to non-cached file data for both local file systems and network file system clients within a common design and implementation framework. And because it builds on a generic design, our implementation provides a starting point for similar implementations in other operating systems.

We evaluated non-blocking writes using several file system workloads. Across Filebench workloads that perform writes, non-blocking writes improve average benchmark throughput by 7X (up to 45.4X) when using disk drives and by 2.1X (up to 4.2X) when using SSDs. For the SPECsfs2008 benchmark workloads, non-blocking writes decrease overall average latency of NFS operations between 3.5% and 70% and average write latency between 65% and 79% across configurations that were obtained by varying the proportion of NFS write operations and NFS read operations. When replaying the MobiBench file system traces, non-blocking writes decrease average operation latency by 20-60%. Finally, the overhead introduced by non-blocking writes is negligible with no loss of performance when workloads cannot benefit from it.

[*]Work done while at Florida International University.

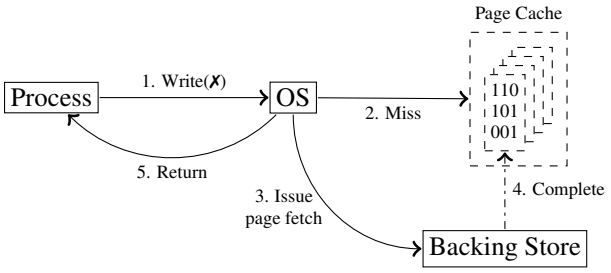

Figure 1: **Anatomy of a write.** The first step, a write reference, fails because the page is not in memory. The process resumes execution (Step 5) only after the blocking I/O operation is completed (Step 4). The dash-dotted arrow represents a slow transition.

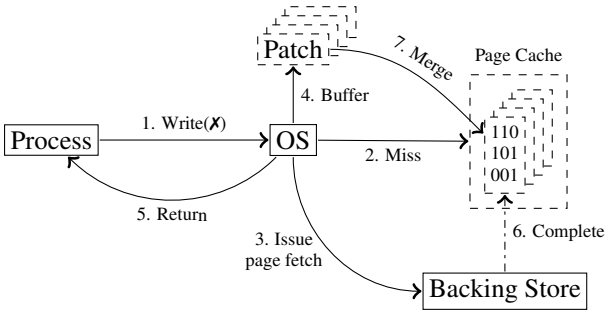

Figure 2: **A non-blocking write employing *asynchronous fetch*.** The process resumes execution (Step 5) after the patch is created in memory while the originally blocking I/O completion is delayed until later (Step 6). The dash-dotted line represents a slow transition.

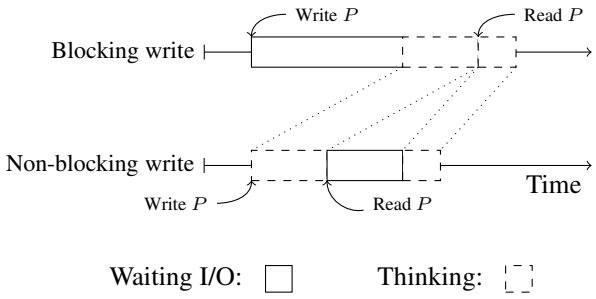

Figure 3: **Page fetch asynchrony with non-blocking writes.** Page P, not present in the page cache, is written to. The application waits for I/O completion. A brief thinktime is followed by a read to P to a different location than the one written to earlier. With non-blocking writes, since the write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

2 Motivating Non-blocking Writes

Previous studies that have analyzed production file system workloads report a significant fraction of write accesses being small or unaligned writes [11, 30, 39, 44]. Technology trends also indicate an increase in page fetch rates in the future. On the server end, multi-core systems and virtualization now enable more co-located workloads leading to larger memory working sets. As the effective memory working sets [8, 25] of workloads continue to grow, page fetch rates also continue to increase. A host of flash-based hybrid memory systems and storage caching and tiering systems have been inspired, and find relevance in practice, because of these trends [3, 4, 13, 16, 17, 18, 22, 24, 35, 40, 45, 55, 57]. On the personal computing end, newer data intensive desktop/laptop applications place greater I/O demands [20]. In mobile systems, page fetches have been found to affect the performance of the data-intensive applications significantly [23]. Second, emerging byte-addressable persistent memories can provide extremely fast durability to applications and systems software [6, 10, 17, 27, 28, 42, 54, 56, 58]. Recent research has also argued in favor of considering main memory in smartphones as *quasi* non-volatile [32]. When used as file system caches [29, 32], such memories can make the durability of in-memory data a non-blocking operation. Eliminating any unwanted blocking in the front end of the durability process, such as fetch-before-write, becomes critical.

2.1 The *fetch-before-write* problem

Page fetch behavior in file systems is caused because of the mismatch in data access granularities: *bytes* accessed by the application, and *pages* accessed from storage by the operating system. To handle write references, the target page is synchronously fetched before the write is applied, leading to a *fetch-before-write* requirement [34, 51]. This is illustrated in Figure 1. This *blocking* behavior affects performance since it requires fetching data from devices much slower than main memory. Today, main memory accesses can be performed in a couple of nanoseconds whereas accesses to flash drives and hard drives can take hundreds of microseconds to a few milliseconds respectively. We confirmed the page *fetch-before-write* behavior for the latest open-source kernel versions of BSD (all variants), Linux, Minix, OpenSolaris, and Xen.

2.2 Addressing the *fetch-before-write* problem

Non-blocking writes eliminate the *fetch-before-write* requirement by creating an in-memory patch for the updated page and unblocking the process immediately. This modification is illustrated in Figure 2.

2.2.1 Reducing Process blocking

Processes block when they partially overwrite one or more non-cached file pages. Such overwrites may be of any size as long as they are not perfectly aligned to page

Workload	Description
ug-filesrv	Undergrad NFS/CIFS fileserver
gsf-filesrv	Grad/Staff/Faculty NFS/CIFS fileserver
moodle	Web & DB server for department CMS
backup	Nightly backups of department servers
usr1	Researcher 1 desktop
usr2	Researcher 2 desktop
Facebook	MobiBench Facebook trace [14]
twitter	MobiBench twitter trace [14]

Table 1: **Workloads and descriptions.**

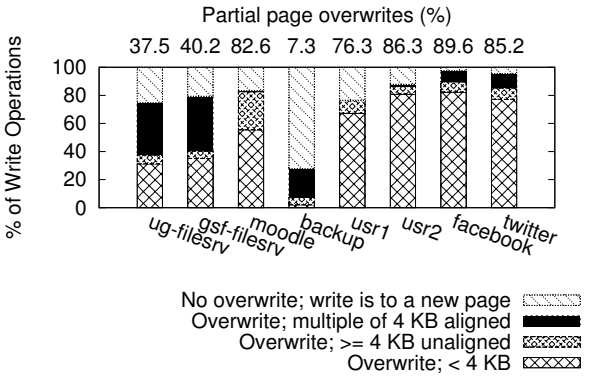

Figure 4: **Breakdown of write operations by amount of page data overwritten.** Each bar represents a different trace and the number above each bar is the percentage of write operations than involve at least one partial page overwrite.

boundaries. Figure 3 illustrates how non-blocking writes reduce process blocking. Previous studies have reported about the significant fraction of small or unaligned writes in production file system workloads [11, 30, 39, 44]. However, little is known about partial page overwrite behavior. To better understand the prevalence of such file writes in production workloads, we developed a Linux kernel module that intercepts file system operations and reports sizes and block alignment for writes. We then analyzed one day's worth of file system operations collected from several production machines at Florida International University's Computer Science department. Besides these we also analyzed file system traces of much shorter duration (two minutes each) available in MobiBench [14, 21]. Table 1 provides a description of all the traces we analyzed.

Figure 4 provides an analysis of the write traffic on each of these machines. On an average, 63.12% of the writes involved partial page overwrites. Depending on the size of the page cache, these overwrites could result in varying degrees of page fetches prior to the page update. The degree of page fetches also depends on the locality of data accesses in the workload wherein a write may follow a read in short temporal order. To account for access locality, we refined our estimates using a cache simulator to count the number of writes that ac-

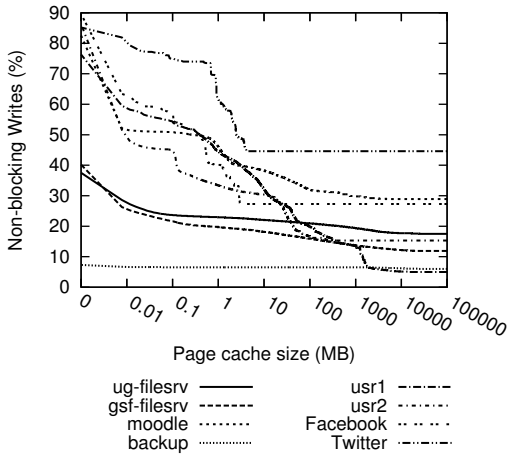

Figure 5: **Non-blocking writes as a percentage of total write operations when varying the page cache size.**

tually lead to page fetches at various memory sizes. Such writes can be made non-blocking. The cache simulator used a modified Mattson's LRU stack algorithm [33] and uses the observation that a non-blocking write at a given LRU cache size would also be a non-blocking write at all smaller cache sizes. Modifications to the original algorithm involved counting all partial page overwrites to pages not in the cache as non-blocking writes. Figure 5 presents the percentage of total writes that would benefit from non-blocking writes for the workloads in Table 1. For a majority of the workloads, this value is at least 15% even for a large page cache of size 100GB. A system that can make such writes non-blocking would make the overall write performance less dependent on the page cache capacity.

2.2.2 Increasing Page fetch parallelism

Processes that access multiple pages not resident in memory during their execution are blocked by the operating system, once for each page while fetching it. As a result, operating systems end up serializing page fetches for accesses that are independent of each other. With non-blocking writes, the operating system allows a process to fetch independent pages in parallel taking better advantage of the available I/O parallelism at the device level. Figure 6 depicts this improvement graphically. Higher levels of I/O parallelism lead to greater device I/O throughput which ultimately improves page fetch throughput for the application.

2.2.3 Making Durable Writes Fast

Next-generation byte-addressable persistent memories are likely to be relatively small compared to today's block-based persistent stores, at least initially. Main memory in today's smartphones has been argued to be *quasi* non-volatile [32]. When such memories are used

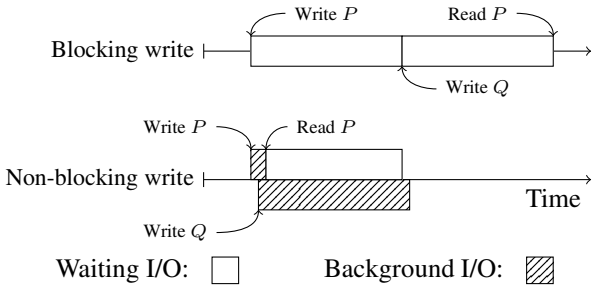

Figure 6: **Page fetch parallelism with non-blocking writes.** Two non-cached pages, P and Q, are written in sequence and the page fetches get serialized by default. With non-blocking writes, P and Q get fetched in parallel increasing device I/O parallelism and thus page fetch throughput.

as a persistent file system cache [29, 32], the containing devices have the ability to provide extremely fast durability (i.e., *sync* operations), a function that would typically block process execution. In such systems, any blocking in the front end of the durability mechanism, such as the fetch-before-write, becomes detrimental to performance. Since non-blocking writes would allow updates without having to fetch the page, it represents the final link in extremely fast data durability when byte addressable persistent memories become widely deployed.

2.3 Addressing Correctness

With non-blocking writes, the ordering of read and write operations within and across processes in the system are liable to change. As we shall elaborate later (§3.3), the *patch creation* and *patch application* mechanisms in non-blocking writes ensure that the ordering of causally dependent operations is preserved. The key insights that we use are: *(i)* reads to recent updates can be served correctly using the most recently created patches, *(ii)* reads that block on page-fetch are allowed to proceed only after applying all the outstanding patches, and *(iii)* reads and writes that are independent and issued by the same or different threads can be reordered without loss of correctness.

Another potential concern with non-blocking writes is data durability. For file data, we observe that the asynchronous write operation only modifies volatile memory and the OS makes no guarantees that the modifications are durable. With non-blocking writes, synchronous writes (on account of `sync/fsync` or the periodic page-flusher daemon) block to wait for the required fetch, apply any outstanding patches, and write the page to storage before unblocking the process. Thus, the durability properties of the system remain unchanged with non-blocking writes.

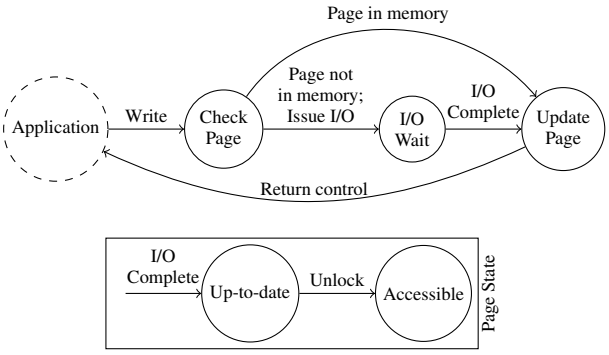

Figure 7: **Process and page state diagram for page fetch with blocking writes.**

3 Non-blocking Writes

The operating system services an application write as depicted in Figure 7. In the *Check Page* state, it looks for the page in the page cache. If the page is already in memory (as a result of a recent fetch completion), it moves to the *Update Page* state which also marks the page as *dirty*. If the page is not in memory, it issues a page fetch I/O and enters the *Wait* state, wherein it waits for the page to be available in memory. When the I/O completes, the page is up-to-date and ready to be unlocked (states *Up-to-date* and *Accessible* in the page state diagram). In the *Update Page* state, the OS makes the page accessible. Finally, control flow returns to the application performing the page write.

3.1 Approach Overview

The page fetch process blocks process execution, which is undesirable. Non-blocking writes work by buffering updates to non-cached pages by creating *patches* in OS memory to be applied later. The basic approach modifies the page fetch path as illustrated in Figure 8. In contrast to current systems, non-blocking writes eliminate the *I/O Wait* state that blocks the process until the page is available in memory. Instead, a non-blocking write returns immediately once a patch of the update is created and queued to the list of pending page updates. non-blocking writes add a new state in the page state, *Outdated*, that reflects the state of the page after it is read into memory but before pending patches are applied. The page transitions into the *Up-to-date* state once all the pending patches are applied.

Non-blocking writes alter write control flow, thus affecting reads to recently written data. Further, they require managing additional cached data in the form of patches. The rest of this section discusses these details in the context of general systems design as well as implementations specific to Linux.

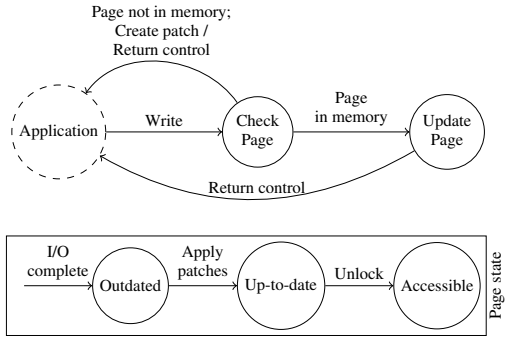

Figure 8: **Process and page state diagram for page fetch with non-blocking writes.**

3.2 Write Handling

Operating systems allow writes to file data via two common mechanisms: supervised *system calls* and unsupervised *memory mapped access*.

To handle supervised writes, the OS uses the system call arguments — the address of the data buffer to be written, the size of the data, and the file (and implicitly, the offset) to write to — and resolves this access to a data page write internally. With non-blocking writes, the OS extracts the data update from the system call arguments, creates a patch, and queues it for later use. This patch is applied later when the data page is read into memory.

Unsupervised file access can be provided by memory mapping a portion of a file to the process address space. In our current design, memory mapped access are handled as in current systems by blocking the process to service the page fault.

3.3 Patch Management

We now discuss how patches are created, stored in the OS, and applied to a page after it is fetched into memory.

3.3.1 Patch Creation

A patch must contain the data to be written along with its target location and size. Since commodity operating systems handle data at the granularity of pages, we chose a design where each patch will apply to a single page. Thus, we abstract an update with a *page patch* data structure that contains all the information to patch and bring the page up-to-date. To handle multiple disjoint overwrites to the same page, we implement *per-page patch queues* wherein page patches are queued and later applied to the page in FIFO order. Consequently, sharing pages via page tables or otherwise is handled correctly. This is possible since operating systems maintain a one-to-one mapping of pages to physical memory frames (e.g., `struct page` in Linux or `struct vm_page` in OpenBSD). When new data is adjacent or overwrites existing patches, it is merged into existing patches accordingly. This makes patch memory overhead and patch application overhead proportional to the number of page bytes changed in the page instead of the number of bytes written to the page since the page was last evicted from memory.

3.3.2 Patch Application

Patch application is rather straightforward. When a page is read in either via a system call induced page fetch or a memory-mapped access causing a page fault, the first step is to apply outstanding patches, if any, to the page to bring it up-to-date before the page is made accessible. Patches are applied by simply copying patch data to the target page location. Patch application occurs in the bottom-half interrupt handling of the page read completion event (further discussed in §5). Once all patches are applied, the page is unlocked which also unblocks the processes waiting on the page, if any.

3.4 Non-blocking Reads

Similar to writes, reads can be classified as *supervised* and *unsupervised* as well. Reads to non-cached pages block the process in current systems. With non-blocking writes, a new opportunity to perform *non-blocking reads* becomes available. Specifically, if the read is serviceable from one of the patches queued on the page, then the reading process can be unblocked immediately without incurring a page fetch I/O. This occurs with no loss of correctness since the patch contains the most recent data written to the page. The read is not serviceable if any portion of the data being requested is not contained within the patch queue. In such a case, the reading process blocks for the page to be fetched. If all data being requested is contained in the patch queue, the data is copied into the target buffer and the reading process is unblocked immediately. For unsupervised reads, our current design blocks the process for the page fetch in all cases.

4 Alternative Page Fetch Modes

Let us consider the page fetch operation issued in Step 3 when performing a non-blocking write as depicted in Figure 2. This operation requires a physical memory allocation (for the page to be fetched) and a subsequent asynchronous I/O to fetch the page so that the newly created patch can be applied to the page. However, since blocking is avoided, process execution is not dependent on the page being available in memory. This raises the question: *can page allocation and fetch be deferred or even eliminated?* Page fetch deferral and elimination allow reduction and shaping of memory consumption and page fetch I/O to storage. While page fetch deferral is opportunistic, page fetch elimination is only possible if the patches created are sufficient to overwrite the page entirely *or* if page persistence becomes unnecessary. We

now explore the page fetch modes that become possible with non-blocking writes.

4.1 Asynchronous Page Fetch

In this mode, page fetch I/O is queued to be issued at the time of the page write. The appeal of this approach is its simplicity. Since the page is brought into memory in a timely fashion similar to the synchronous fetch, it is transparent to timer-based durability mechanisms such as dirty page flushing [2] and file system journaling [19].

Asynchronous page fetch defines policy. However, its mechanism may involve additional blocking prior to issuing the page fetch. We discuss two alternative page fetch mechanisms that highlight this issue.

1. Foreground Asynchronous Page Fetch (NBW-Async-FG). The page fetch I/O is issued in the context of process performing the write to the file page. Our discussion in previous sections was based on this mechanism. Although the process does not wait for the completion of the data fetch, issuing the fetch I/O for the data page may itself involve retrieving additional metadata pages to locate the data page if these metadata pages are not cached in OS memory. If so, the writing process would have to block for the necessary metadata fetches to complete, thereby voiding most of the benefits of the non-blocking write.

2. Background Asynchronous Page Fetch (NBW-Async-BG). The writing process moves all work necessary to issue the page fetch to a different context by using kernel worker threads. This approach eliminates any blocking of the writing process owing to metadata misses; a worker thread blocks for all fetches while the issuing process continues its execution.

Synchronous fetch is a valuable improvement. However, it consumes system resources, allocating system memory for the page to be fetched and using storage I/O bandwidth to fetch the page.

4.2 Lazy Page Fetch (NBW-Lazy)

When a process writes to a non-cached data page, its execution is not contingent on the page being available in memory. With *lazy page fetch*, the OS delays the page fetch until it becomes unavoidable. Lazy page fetch has the potential to further reduce the system's resource consumption. Figure 9 illustrates this alternative.

Lazy page fetch creates new system scenarios which must be considered carefully. If a future page read cannot be served using the currently available patches for the non-cached page, the page fetch becomes unavoidable. In this case, the page is fetched synchronously and patches are applied first before unblocking the reading process. If the page gets overwritten in its entirety or if page persistence becomes unnecessary for another reason (e.g., the containing file is deleted), the original page fetch is eliminated entirely.

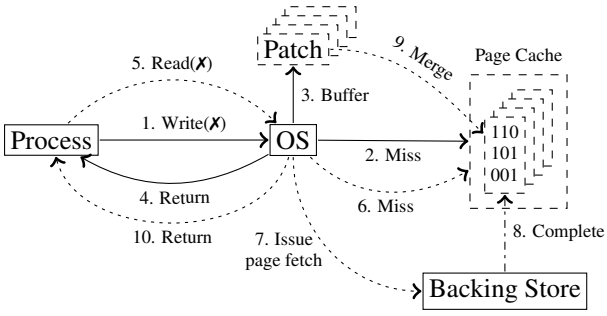

Figure 9: **A non-blocking write employing *lazy fetch*.** The process resumes execution (Step 4) after the patch is created in memory. The Read operation in Step 5 optionally occurs later in the execution while the originally blocking I/O is optionally issued and completes much later (Step 8). The dash-dotted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

Page data durability can become necessary in the following instances: *(i)* synchronous file write by an application, *(ii)* periodic flushing of dirty pages by the OS [2], or *(iii)* ordered page writes to storage as in a journaling file system [19, 41]. In all these cases, the page is fetched synchronously before being flushed to the backing store. Lastly, non-blocking writes are not engaged for metadata pages which use the conventional durability mechanisms. Durability related questions are discussed further in §5.2.

5 Implementation

Non-blocking writes alter the behavior and control flow of current systems. We present an overview of the implementation of non-blocking writes and discuss details related to how it preserves system correctness.

5.1 Overview

We implemented non-blocking writes for file data in the Linux kernel (version 2.6.34.17) by modifying the generic virtual file system (VFS) layer. Unlike the conventional Linux approach, all handling of fetch completion (such as applying patches, marking the page dirty, processing a journaling transaction, and unlocking the page) occurs in the bottom-half I/O completion handler.

5.2 Handling Correctness

OS-initiated Page Accesses. Our implementation does not implement non-blocking writes for accesses (writes and reads) to un-cached pages initiated internally by the OS. These include file system metadata page updates, and updates performed by kernel threads. This implementation trivially provides the durability properties expected by OS services to preserve semantic correctness.

Journaling File Systems. Our implementation of non-blocking writes preserves the correctness of journaling file systems by allowing the expected behavior for various journaling modes. For instance, non-blocking writes preserve ext4's ordered mode journaling invariant that data updates are flushed to disk before transactions containing related metadata updates. Metadata transactions in ext4 do not get processed until after the related data page is fetched into memory, outstanding patches are applied, the page is marked dirty, and dirty buffers added to the transaction handler. Thus, all dirty data pages related to a metadata transaction are resident in memory and flushed to disk by ext4's ordered mode journaling mechanism prior to committing the transaction.

Handling Read-Write Dependencies. While a non-blocking write is being handled within the operating system, multiple operations such as read, prefetch, synchronous write, and flush, can be issued to the page involved. Operating systems carefully synchronize these operations to maintain consistency and return only up-to-date data to applications. Our implementation respects the Linux page locking protocol. A page is locked after it is allocated and before issuing a fetch for it. As a result, kernel mechanisms such as fsync and mmap are also supported correctly. These mechanisms block on the page lock which becomes available only after the page is fetched and patches applied before proceeding to operate on the page. When delayed page fetch mechanisms (as in NBW-Async-BG and NBW-Lazy) are used, an `NBW` entry for the page involved is added in the page cache mapping for the file before the page is allocated. This `NBW` entry allows for locking the page to maintain the ordering of page operations. When necessary (e.g., a *sync*), pages indexed as `NBW` get fetched which in turn involves acquiring the page lock, thus synchronizing future operations on the page. The only exception to such page locking is writing to a page already in the non-blocking write state; the write does not lock the page but instead queues a new patch.

Ordering of Page Updates. Non-blocking writes may alter the sequence in which patches *to different pages* get applied since the page fetches may complete out-of-order. Non-blocking writes only replace writes that are to memory that are not guaranteed to be reflected to persistent storage in any particular sequence. Thus, ordering violations in updates of in-memory pages are crash-safe.

Page Persistence and Syncs. If an application would like explicit disk ordering for memory page updates, it would execute a blocking flush operation (e.g., `fsync`) subsequent to each operation. The flush operation causes the OS to force the fetch of any page indexed as `NBW` even if it has not been allocated yet. The OS then obtains the page lock, waits for the page fetch, and applies any outstanding patches, before flushing the page and returning control to the application. Ordering of disk writes are thus preserved with non-blocking writes.

Handling of disk errors. Our implementation changes the semantics of the OS with respect to notification of I/O errors when handling writes to non-cached pages. Since page fetches on writes are done asynchronously, disk I/O errors (e.g., `EIO` returned for the UNIX write system call) during the asynchronous page fetch operation would not get reported to the writing application process. Any action that the application was designed to take based on the error reported would not be performed. Semantically, the application write was a memory write and not to persistent storage; an I/O error being reported by current systems is an artifact of the *fetch-before-write* design. With non-blocking writes, if the write were to be made persistent at any point via a flush issued by the application or the OS, any I/O errors during page flushing would be reported to the initiator.

Multi-core and Kernel Preemption. Our implementation fully supports SMP and kernel preemption. For a given non-cached page, the patch creation mechanism (when processing the write system call) can contend with the patch application mechanism (when handling page fetch completion). Our implementation uses a single additional lock to protect a patch queue from simultaneous access.

6 Evaluation

We address the following questions:

(1) What are the benefits of non-blocking writes for different workloads?

(2) How do the fetch modes of non-blocking writes perform relative to each other?

(3) How sensitive are non-blocking writes to the underlying storage type?

(4) How does memory size affect non-blocking writes?

We evaluate four different solutions. Blocking writes (*BW*) is the conventional approach to handling writes and uses the Linux kernel implementation. Non-blocking writes variants include asynchronous mode using foreground (*NBW-Async-FG*) and background (*NBW-Async-BG*) fetch, and lazy mode (*NBW-Lazy*).

Workloads and Experimental Setup. We use the Filebench micro-benchmark [50] to address (1), (2), (3), and (4) using controlled workloads. We use the SPECsfs2008 benchmark [49] and replay the MobiBench traces [14] to further analyze questions (1) and (2); the MobiBench trace replay also helps answer question (3). The Filebench and MobiBench evaluations were performed on a machine with Quad-Core 2.50 GHz AMD Opteron(tm) 1381 processors, 8GB of RAM, a 500

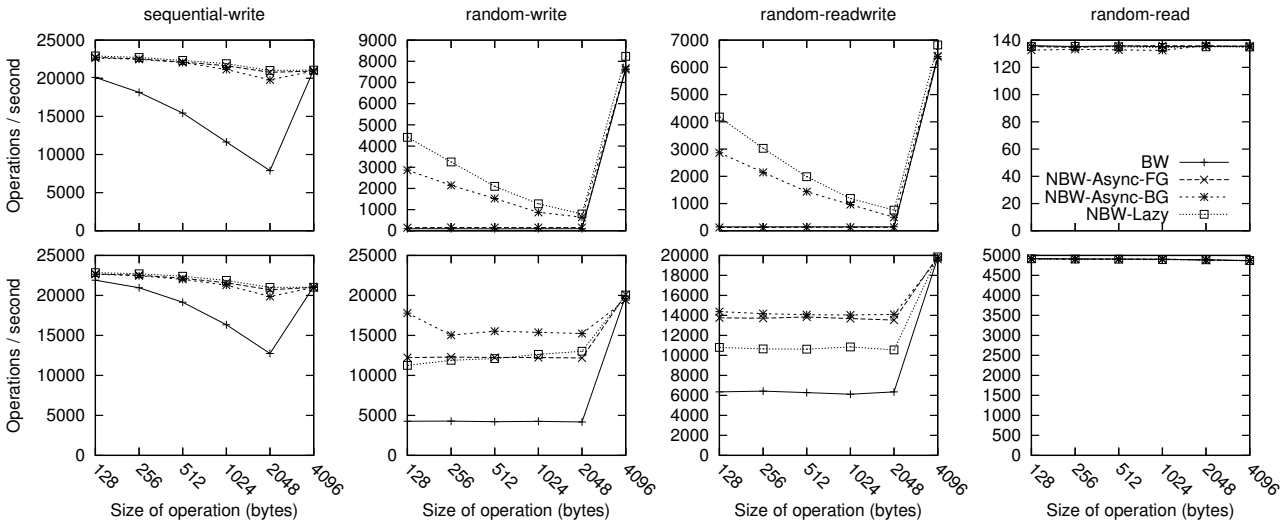

Figure 10: **Performance for various Filebench personalities when varying the I/O size.** The two rows correspond to two different storage back-ends: hard disk-drive (top) and solid-state drive (bottom).

GB WDC WD5002ABYS hard disk-drive, a 32 GB Intel X25-E SSD, and Gigabit Ethernet, running Gentoo Linux (kernel 2.6.34.14) . The above setup was also used to run the client-side component of the SPECsfs2008 benchmark. Additionally, for the SPECsfs2008 benchmark, the NFS server used a 2.3 GHz Quad-Core AMD Opteron(tm) Processor 1356, 7GB of RAM, 500 GB WDC and 160 GB Seagate disks, and Gigabit Ethernet, running Gentoo Linux (kernel 2.6.34.14). The 500GB hard disk housed the root file system while the 160GB hard disk stored the NFS exported data. The network link between client and server was Gigabit Ethernet.

6.1 Filebench Micro-benchmark

For all the following experiments we ran five Filebench personalities for 60 seconds using a 5GB pre-allocated file after clearing the contents of the OS page cache. Each personality represents a different type of workload. The system was configured to use 4GB of main memory and memory used for patches was limited to 64MB, a small fraction of DRAM, to avoid significantly affecting the DRAM available to the workload and the OS. We report the Filebench performance metric, the number of operations per second. Each data-point is calculated using the average of 3 executions.

6.1.1 Performance Evaluation

We first examine the performance of Filebench when using a hard disk as the storage back-end. The top row of Figure 10 depicts the performance for four Filebench personalities when varying the size of the Filebench operation. Each data point reports the average of 3 executions. Standard error of measurement was less than 3% of the average for 96.88% of the cases and were less than 10% for the rest.

The first three plots involve personalities that perform write operations. At 4KB I/O size, there is no *fetch-before-write* behavior because every write results in an overwrite of an entire page; thus, non-blocking writes are not engaged and do not impose any overhead either.

For the *sequential-write* personality, performance with blocking writes (BW) depends on the operation size, and is limited by the number of page misses per operation. In the worst case, when the I/O size is equal to 2KB, every two writes involve a blocking fetch. On average, the different non-blocking write modes provide a performance improvement of 13-160% depending on the I/O size.

The second and third personalities represent random access workloads. *Random-write* is a write-only workload, while *random-readwrite* is a mixed workload; the latter uses two threads, one for issuing reads and the other for writes. For I/O sizes smaller than 4KB, BW provides a constant throughput of around 97 and 146 operations/sec for *random-write* and *random-readwrite* personalities respectively. Performance is consistent regardless of the I/O size because each operation is equally likely to result in a page miss and fetch. *Random-readwrite* performs better than *random-write* due to the additional available I/O parallelism when two threads are used. Further, for *random-write*, NBW-Async-FG provides 50-60% performance improvement due to reduced blocking for page fetches of the process. However, this improvement does not manifest for *random-readwrite* wherein read operations incur higher latencies due to additional blocking for pages with fetches in progress. In both cases, the benefits of NBW-Async-FG are significantly lower when compared to other non-blocking write modes since NBW-Async-FG blocks on many of the ini-

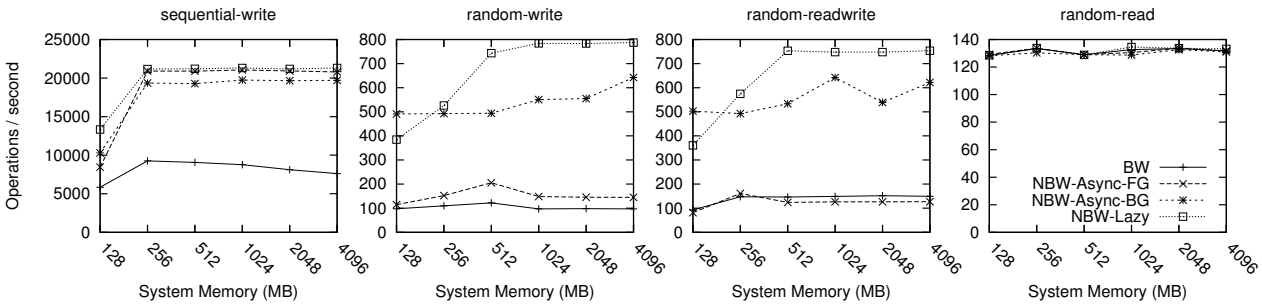

Figure 11: **Memory sensitivity of Filebench.** The I/O size was fixed at 2KB and patch memory limit was set to 64MB.

tial file-system metadata misses during this short-running experiment.

In contrast, NBW-Async-BG unblocks the process immediately while a different kernel thread blocks for the metadata fetches as necessary. This mode shows a 6.7x-29.5x performance improvement for *random-write*, depending on the I/O size. These performance gains reduce as the I/O size increases since non-blocking writes can create fewer outstanding patches to comply with the imposed patch memory limit of 64MB. A similar trend is observed for *random-readwrite* with performance improvements varying from 3.4x-19.5x depending on the I/O size used. NBW-Lazy provides up to 45.4X performance improvement over BW by also eliminating both data and metadata page fetches when possible. When the available patch memory limit is reached, writes are treated as in BW until more patch memory is freed up.

The final two personalities, *random-read* and *sequential-read* (not shown), are read-only workloads. These workloads do not create write operations and the overhead of using a non-blocking writes kernel is zero. Non-blocking writes deliver the same performance as blocking writes.

6.1.2 Sensitivity to system parameters

Our sensitivity analysis of non-blocking writes addresses the following specific questions:

(1) What are the benefits of non-blocking writes when using different storage back-ends?

(2) How do non-blocking writes perform when system memory size is varied?

Sensitivity to storage back-ends

To answer the first question, we evaluated non-blocking writes using a solid state drive (SSD) based storage back-end. Figure 10 (bottom row) presents results when running Filebench personalities using a solid state drive. Each data point reports the average of 3 executions. Standard error of measurement was less than 2.25% of the average in all cases except one for which it was 5%.

Performance trends with the *sequential-write* workload are almost identical to the hard disk counterparts (top row in Figure 10) for all modes of non-blocking writes. This is because non-blocking writes completely eliminate the latency of accessing storage for every operation in both systems. On the other hand, because the SSD offers better throughput than the hard disk drive, BW offers an increase in throughput for every size below 4KB. In summary, the different non-blocking write modes provide between 4% and 61% performance improvement depending on the I/O size.

For the *random-write* and *random-readwrite* workloads, the non-blocking write variants all improve performance but to varying degrees. The SSD had significantly lower latencies servicing random accesses relative to the hard disk drive which allowed for metadata misses to be serviced much quicker. The efficiency of NBW-Async-FG relative to BW is further improved relative to the hard disk system and it delivers 188% and 117% performance improvement for *random-write* and *random-readwrite* respectively. NBW-Async-BG improves over NBW-Async-FG for reasons similar to those with hard disks. NBW-Async-BG delivers 272% (up to 4.2X in the best case) and 125% performance improvement over BW on average for *random-write* and *random-readwrite* respectively. Lastly, although NBW-Lazy performs significantly better than BW, contrary to our expectations, its performance improvements were lower when compared to the NBW-Async modes. Upon further investigation, we found that when the patch memory limit is reached, NBW-Lazy takes longer than the other modes to free its memory given that the fetches are issued only when blocking cannot be avoided anymore. While the duration of the experiment is the same as disk drives, a faster SSD results in the patch memory limit being met more quickly. In our current implementation, after the patch memory limit is reached and no more patches can be created, NBW-Lazy defaults to a BW behavior issuing fetches synchronously for handling writes to non-cached pages. Despite this drawback, NBW-Lazy mode shows 163%-211% and 70% improvement over BW for *random-write* and *random-readwrite* respectively.

Write size	%	Write size	%
1 - 4095 bytes	28	8193 - 16383 bytes	7
4KB	11	16KB	5
4097 - 8191 bytes	3	16385 - 32767 bytes	1
8KB	30	32, 64, 96, 128, 256 KB	15

Table 2: **SPECsfs2008 write sizes.**

Sensitivity to system memory size

We answer the second question using the Filebench workloads and varying the amount of system memory available to the operating system. For these experiments, we used a hard disk drive as the storage back-end and fixed the I/O size at 2KB. Figure 11 presents the results of this experiment. Each data point reports the average of 3 executions. Standard error of measurement was less than 4% of the average for 90% of the cases and were less than 10% for the rest.

For the *sequential-write* workload, the non-blocking writes variants perform 45-180% better than BW. Further NBW-Lazy performs better and can be considered optimal because *(i)* it uses very little patch memory, sufficient to hold enough patches until a single whole page is overwritten, and *(ii)* since pages get overwritten entirely in the sequential write, it eliminates all page fetches.

For *random-write* and *random-readwrite* workloads, NBW-Async-FG delivers performance that is relatively consistent with BW; the I/O performance achieved by these solutions is not high enough to make differences in memory relevant. NBW-Async-BG and NBW-Lazy offer significant performance gains relative to BW of as much as 560% and 710% respectively. With NBW-Lazy, performance improves with more available memory but only up to the point at which the imposed patch memory limit is reached prior to the completion of the execution; increasing the patch memory limit would allow NBW-Lazy to continue scaling its performance.

6.2 SPECsfs2008 Macro-benchmark

The SPECsfs2008 benchmark tests the performance of NFS servers. For this experiment, we installed a non-blocking writes kernel in the NFS server which exported the network file system in *async* mode. SPECsfs2008 uses a client side workload generator that bypasses the page cache entirely. The client was configured for a target load of 500 operations per second. The target load was sustained in all evaluations; thus the SPECsfs2008 performance metric is the operation latency reported by the NFS client. While the evaluation results are encouraging, the relative performance results we report for NFS workloads are likely to be an underestimate. This is because our prototype was used only at the NFS server; the client counterpart of non-blocking writes was not engaged by this benchmark.

SPECsfs2008 operations are classified as *write, read,* and *others* which includes metadata operations such as *create, remove,* and *getattr*. For each variant solution, we report results for the above three classes of operations separately as well as the overall performance that represents the weighted average across all operations. Further, we evaluated performance when varying the relative proportion of NFS operations issued by the benchmark. The default configuration as specified in SPECsfs2008 is: reads (18%), writes (10%) and others (72%). We also evaluated three modified configurations: no-writes, no-reads, and one that uses: reads (10%), writes (18%), and others (72%) to examine a wider spectrum of behaviors.

We first perform a brief analysis of the workload to determine expected performance. Even for configurations that contained more writes than reads (e.g., 18% writes and 10% reads) the actual fraction of cache misses upon writes is far lower than the fraction of misses due to reads (i.e 16.9% write misses vs. 83.1% read misses). This mismatch is explained by noting that each read access to a non-cached page results in a read miss but the same is not true for write accesses when they are page-aligned. Further, Table 2 reports that only 39% of all writes issued by the SPECsfs2008 are partial page overwrites which may result in non-blocking writes.

Figure 12 presents the average operation latencies normalized using the latency with the BW solution. Excluding the read-only workload, the dominant trend is that the non-blocking write modes offer significant reductions in write operation latency with little or no degradation in read latencies. Further, the average overall operation latency is proportional to the fraction of write misses and to the latency improvements for NFS write operations. For the three configurations containing write operations, the latency of the write operations is reduced between 65 and 79 percent when using the different modes of non-blocking writes. Read latencies are slightly affected negatively due to additional blocking on certain pages. With BW, certain pages could have been fetched into memory by the time the read operation was issued. With non-blocking writes, the corresponding fetches could be delayed or not issued at all until the blocking read occurs. For the configuration with no write operations the average overall latency remained relatively unaffected.

6.3 MobiBench Trace Replay

The MobiBench suite of tools contains traces obtained from an Android device when using the Facebook and Twitter apps [14]. We used MobiBench's timing-accurate replay tool to replay the traces. We fixed a bug in the replay tool prior to using it; the original replayer used a fixed set of flags when opening files regardless of the trace information. MobiBench reports the average file system call operation latency as the performance metric. We replayed the traces five times and report the

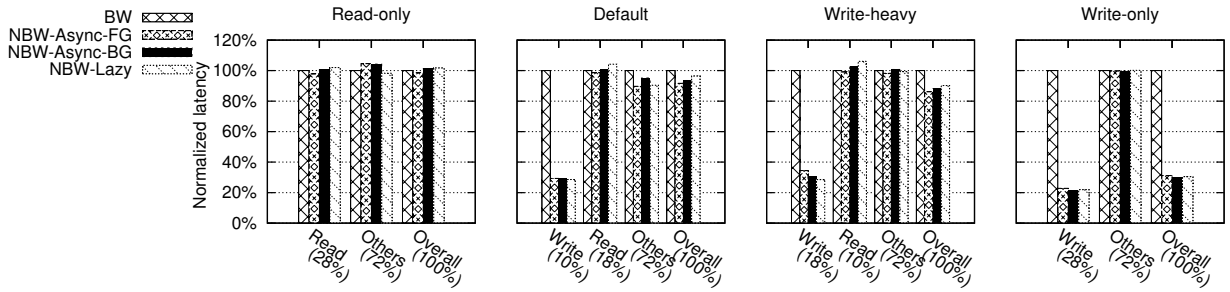

Figure 12: **Normalized average operation latencies for SPECsfs2008.**

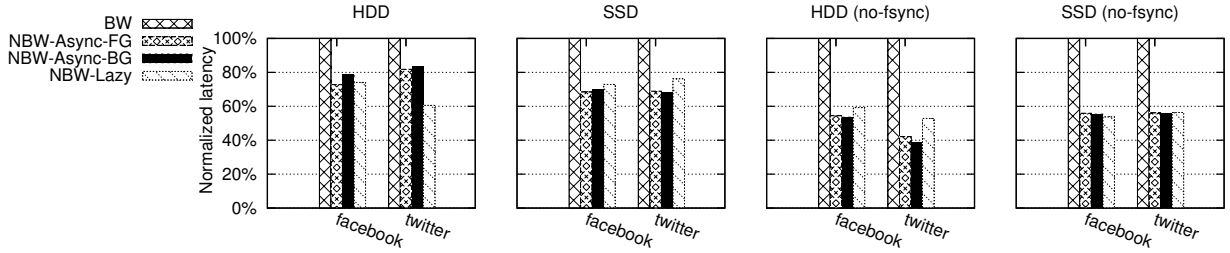

Figure 13: **Normalized average operation latencies when replaying MobiBench traces [14].**

average latency observed. Standard error of measurement was less than 4% of the average in all cases except one for which it was 7.18%. The two left-most graphs of Figure 13 present the results for this evaluation for both hard disks and solid-state drives respectively. Non-blocking writes exhibit a reduction in operation latencies between 20% and 40% depending on the mode and back-end storage used for both Facebook and Twitter traces.

When we analyzed the MobiBench traces, we found that they contained a significant amount of sync operations. Sync operations do not allow exploiting the full potential of non-blocking writes because they block the process to fetch pages synchronously. As discussed previously, recent work on byte-addressable persistent memories and qNVRAM [32] provide for extremely fast, durable, in-memory operations. In such systems, the blocking fetch-before-write behavior in OSes becomes an even more significant obstacle to performance. To estimate[1] the impact of non-blocking writes in such an environment, we modified the original traces by discarding all *fsync* operations to simulate extremely fast durability of in-memory data. The rightmost two graphs present the results obtained upon replaying the modified traces. non-blocking writes reduce latencies by 40-60% depending on the mode and the storage back-end used.

7 Related Work

Non-blocking writes have existed for almost three decades for managing CPU caches. Observing that entire cache lines do not need to be fetched on a word write-miss thereby stalling the processor, the use of additional registers that temporarily store these word updates was investigated [26] and later adopted [31].

Recently, non-blocking writes to main memory pages was motivated using full system memory access traces generated by an instrumented QEMU machine emulator [53]. This prior work outlined some of the challenges of implementing non-blocking writes in commodity operating systems. We improve upon this work by presenting a detailed design and Linux kernel implementation of non-blocking writes, addressing a host of challenges as well as uncovering new design points. We also present a comprehensive evaluation with a wider range of workloads and performance numbers from a running system.

A candidate approach to mitigate the *fetch-before-write* problem involves provisioning adequate DRAM to minimize write cache misses. However, the file system footprint of a workload over time is usually unpredictable and potentially unbounded. Alternatively, prefetching [46] can reduce blocking by anticipating future memory accesses. However, prefetching is typically limited to sequential accesses. Moreover, incorrect decisions can render prefetching ineffective and pollute memory. Non-blocking writes is complementary to these approaches. It uses memory judiciously and only fetches those pages that are necessary for process execution.

There are several approaches proposed in the literature that reduce process blocking specifically for system call induced page fetches. The goal of the asynchronous I/O library (e.g., POSIX AIO [1]) available on Linux and a few BSD variants is to make file system writes asynchronous; a helper library thread blocks on behalf of the

[1] We did not enforce ordered CPU cache flushing to persistent memory to ensure in-memory durability upon fsync.

process. LAIO [12] is a generalization of the basic AIO technique to make all system calls asynchronous; a library checkpoints execution state and relies on scheduler activations to get notified about the completion of blocking I/O operations initiated inside the kernel. Recently, FlexSC [48] proposed asynchronous exceptionless system calls wherein system calls are queued by the process in a page shared between user and kernel space; these calls are serviced asynchronously by syscall kernel threads which report completion back to the user process.

The scope of non-blocking writes in relation to the above proposals is different. Its goal is to entirely eliminate the blocking of memory writes to pages not available in the file system page cache. A non-blocking write does not need to checkpoint state thereby consuming lesser system resources. Further, it can be configured to be lightweight so that it does not use additional threads (often a limited resource in systems) to block on behalf of the running process. Finally, unlike these approaches which require application modifications to use specific libraries, non-blocking writes work seamlessly in the OS transparent to applications.

There are works that are related to non-blocking writes, but quite different in their accomplished goal. Speculative execution (or Speculator) as proposed by Nightingale *et al.* [36] eliminates blocking when synchronously writing cached in-memory page modifications to a network file server using a process checkpoint and rollback mechanism. Xsyncfs [37] eliminates the blocking upon performing synchronous writes of in-memory pages to disk by creating a commit dependency for the write and allowing the process to make progress. Featherstitch [15] improves the performance of synchronous file system page updates by scheduling these page writes to disk more intelligently. Featherstitch employed patches but for a different purpose – to specify dependent changes across disk blocks at the byte granularity. OptFS [5] decouples the ordering of writes of in-memory pages from their durability, thus improving performance. While these approaches optimize the writing of in-memory pages to disk they do not eliminate the blocking page fetch before in-memory modifications to a file page can be made.

BOSC [47] describes a new disk update interface for applications to explicitly specify disk update requests and associate call back functions. Opportunistic Log [38] describes the fetch-before-write problem for objects and uses a second log to record updates. Both of these reduce application blocking allowing updates to happen in the background but they require application modification and do not support general-purpose usage. Non-blocking writes is complementary to the above body of work because it runs seamlessly inside the OS requiring no changes to applications.

8 Conclusions and Future Work

For over four decades, operating systems have blocked processes for page fetch I/O when they write to non-cached file data. In this paper, we revisited this well-established design and demonstrated that such blocking is not just unnecessary but also detrimental to performance. Non-blocking writes decouple the writing of data to a page from its presence in memory by buffering page updates elsewhere in OS memory. This decoupling is achieved with a self-contained operating system improvement seamless to the applications. We designed and implemented *asynchronous* and *lazy* page fetch modes that are worthwhile alternatives to blocking page fetch. Our evaluation of non-blocking writes using Filebench revealed throughput performance improvements of as much as 45.4X across various workload types relative to blocking writes. For the SPECsfs2008 benchmark, non-blocking writes reduced write operation latencies by as much as 65-79%. When replaying the MobiBench file system traces, non-blocking writes decreased average operation latency by 20-60%. Further, there is no loss of performance when workloads cannot benefit from non-blocking writes.

Non-blocking writes open up several avenues for future work. First, since they alter the relative importance of pages in memory in a fundamental way, new page replacement algorithms are worth investigating. Second, by intelligently scheduling page fetch operations (instead of simply asynchronously or lazily), we can reduce and shape both memory consumption and the page fetch I/O traffic to storage. Third, I/O related to asynchronous page fetching due to non-blocking writes can be scheduled more intelligently (e.g., as background operations [52] or semi-preemptibly [9]) to speed up blocking page fetches. Finally, certain OS mechanisms such as dirty page flushing thresholds and limits on per-process dirty data would need to be updated to also account for in-memory patches.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Donald Porter, for their detailed and thoughtful feedback which improved the quality of this paper significantly. Many thanks to Eric Johnson who aided in the collection of file system traces from FIU's production servers. This work is supported in part by NSF awards CNS-1018262 and CNS-1448747, the Intel ISRA program, and a NetApp Faculty Fellowship.

Traces

The traces used in this paper are available at: `http://sylab-srv.cs.fiu.edu/dokuwiki/doku.php?id=projects:nbw:start`

References

[1] AMERICAN NATIONAL STANDARDS INSTITUTE. *IEEE standard for information technology: Portable Operating Sytem Interface (POSIX). Part 1, system application program interface (API) — amendment 1 — realtime extension [C language]*. IEEE, 1994. IEEE Std 1003.1b-1993 (formerly known as IEEE P1003.4; includes IEEE Std 1003.1-1990). Approved September 15, 1993, IEEE Standards Board. Approved April 14, 1994, American National Standards Institute.

[2] BACH, M. J. *The Design of the UNIX Operating System*, 1st ed. Prentice Hall Press, 1986.

[3] BYAN, S., LENTINI, J., MADAN, A., PABON, L., CONDICT, M., KIMMEL, J., KLEIMAN, S., SMALL, C., AND STORER, M. Mercury: Host-side flash caching for the data center. In *Proceedings of the 28th IEEE Conference on Massive Data Storage* (April 2012), MSST '12.

[4] CHEN, F., KOUFATY, D. A., AND ZHANG, X. Hystor: making the best use of solid state drives in high performance storage systems. In *Proceedings of the International Conference on Supercomputing* (May-June 2011), ICS '11.

[5] CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Optimistic crash consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles* (November 2013), SOSP '13.

[6] COBURN, J., CAULFIELD, A., AKEL, A., GRUPP, L., GUPTA, R., JHALA, R., AND SWANSON, S. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), ASPLOS XVI.

[7] DALEY, R. C., AND NEUMANN, P. G. A general-purpose file system for secondary storage. In *Proceedings of the Fall Joint Computer Conference, Part I* (1965), AFIPS '65 (Fall, part I), pp. 213–229.

[8] DENNING, P. J. The working set model for program behavior. *Communications of the ACM 11*, 5 (1968), 323–333.

[9] DIMITRIJEVIC, Z., RANGASWAMI, R., AND CHANG, E. Systems support for preemptive RAID scheduling. *IEEE Transactions on Computers 54*, 10 (October 2005), 1314–1326.

[10] DULLOOR, S. R., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System software for persistent memory. In *Proceedings of the European Conference on Computer Systems* (2014), EuroSys '14.

[11] ELLARD, D., LEDLIE, J., MALKANI, P., AND SELTZER, M. Passive NFS tracing of email and research workloads. In *Proceedings of the USENIX Conference on File and Storage Technologies* (2003), FAST '03.

[12] ELMELEEGY, K., CHANDA, A., COX, A. L., AND ZWAENEPOEL, W. Lazy asynchronous I/O for event-driven servers. In *Proceedings of the 2004 USENIX Annual Technical Conference* (2004), ATC '04.

[13] EMC. VFCache. http://www.emc.com/storage/vfcache/vfcache.htm, 2012.

[14] ESOS LABORATORY. Mobibench traces. https://github.com/ESOS-Lab/Mobibench/tree/master/MobiGen, 2013.

[15] FROST, C., MAMMARELLA, M., KOHLER, E., DE LOS REYES, A., HOVSEPIAN, S., MATSUOKA, A., AND ZHANG, L. Generalized file system dependencies. In *Proceedings of the ACM Symposium on Operating Systems Principles* (October 2007), SOSP '07, pp. 307–320.

[16] FUSION-IO. ioTurbine. http://www.fusionio.com/systems/ioturbine/, 2012.

[17] GUERRA, J., MARMOL, L., CAMPELLO, D., CRESPO, C., RANGASWAMI, R., AND WEI, J. Software persistent memory. In *Proceedings of the USENIX Annual Technical Conference* (June 2012), ATC '12.

[18] GUERRA, J., PUCHA, H., GLIDER, J., BELLUOMINI, W., AND RANGASWAMI, R. Cost effective storage using extent-based dynamic tiering. In *Proceedings of the USENIX Conference on File and Storage Technologies* (February 2011), FAST '11.

[19] HAGMANN, R. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the ACM Symposium on Operating Systems Principles* (November 1987), SOSP '87.

[20] HARTER, T., DRAGGA, C., VAUGHN, M., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. A file is not a file: Understanding

the I/O behavior of Apple desktop applications. In *Proceedings of the ACM Symposium on Operating Systems Principles* (October 2011), SOSP '11.

[21] JEONG, S., LEE, K., HWANG, J., LEE, S., AND WON, Y. Framework for analyzing android I/O stack behavior: From generating the workload to analyzing the trace. *Future Internet 5*, 4 (2013), 591–610.

[22] KGIL, T., AND MUDGE, T. FlashCache: a NAND flash memory file cache for low power web servers. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (October 2006), CASES '06.

[23] KIM, H., AGRAWAL, N., AND UNGUREANU, C. Revisiting storage for smartphones. In *Proceedings of the USENIX Conference on File and Storage Technologies* (February 2012), FAST '12.

[24] KOLLER, R., MARMOL, L., RANGASWAMI, R., SUNDARARAMAN, S., TALAGALA, N., AND ZHAO, M. Write policies for host-side flash caches. In *Proceedings of the USENIX Conference on File and Storage Technologies* (February 2013), FAST '13.

[25] KOLLER, R., VERMA, A., AND RANGASWAMI, R. Generalized ERSS tree model: Revisiting working sets. *Performance Evaluation 67*, 11 (2010), 1139–1154.

[26] KROFT, D. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th annual symposium on Computer Architecture* (1981), ISCA '81, IEEE Computer Society Press, pp. 81–87.

[27] LANTZ, P., DULLOOR, S., KUMAR, S., SANKARAN, R., AND JACKSON, J. Yat: A validation framework for persistent memory software. In *Proceedings of the USENIX Annual Technical Conference* (June 2014), ATC '14.

[28] LEE, B., IPEK, E., MUTLU, O., AND BURGER, D. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the International Symposium on Computer Architecture* (2009), ISCA '09.

[29] LEE, E., BAHN, H., AND NOH, S. H. Unioning of the buffer cache and journaling layers with non-volatile memory. In *Proceedings of the USENIX Conference on File and Storage Technologies* (February 2013), FAST '13.

[30] LEUNG, A. W., PASUPATHY, S., GOODSON, G., AND MILLER, E. L. Measurement and analysis of large-scale network file systemworkloads. In *Proceedings of the USENIX Annual Technical Conference* (2008).

[31] LI, S., CHEN, K., BROCKMAN, J. B., AND JOUPPI, N. P. Performance impacts of non-blocking caches in out-of-order processors. Tech. rep., Hewlett-Packard Labs and University of Notre Dame, July 2011.

[32] LUO, H., TIAN, L., AND JIANG, H. qNVRAM: quasi non-volatile RAM for low overhead persistency enforcement in smartphones. In *6th USENIX Workshop on Hot Topics in Storage and File Systems* (June 2014), HotStorage '14.

[33] MATTSON, R. L., GECSEI, J., SLUTZ, D. R., AND TRAIGER, I. L. Evaluation techniques for storage hierarchies. *IBM Systems Journal* (1970).

[34] MCKUSICK, M. K., BOSTIC, K., KARELS, M. J., AND QUARTERMAN, J. S. *The Design and Implementation of the 4.4 BSD Operating System*. Addison Wesley, 1996, pp. 163, 196.

[35] NETAPP. Flash Accel. http://www.netapp.com/us/products/storage-systems/flash-accel/, 2013.

[36] NIGHTINGALE, E. B., CHEN, P. M., AND FLINN, J. Speculative execution in a distributed file system. *ACM Transactions on Computer Systems* (2006), 361–392.

[37] NIGHTINGALE, E. B., VEERARAGHAVAN, K., CHEN, P. M., AND FLINN, J. Rethink the sync. In *Proceedings of the 7th USENIX Conference on Operating Systems Design and Implementation* (November 2006), OSDI '06.

[38] O'TOOLE, J., AND SHRIRA, L. Opportunistic log: Efficient installation reads in a reliable storage server. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation* (1994), OSDI '94, pp. 39–48.

[39] OUSTERHOUT, J. K., DA COSTA, H., HARRISON, D., KUNZE, J. A., KUPFER, M., AND THOMPSON, J. G. A trace-driven analysis of the UNIX 4.2 BSD/ file system. In *Proceedings of the ACM Symposium on Operating Systems Principles* (1985), SOSP '85.

[40] PATIL, M., VILAYANNUR, M., OSTROWSKI, M., NARKHEDE, S., KOTHAKOTA, V., JUNG, W., KOHLER, H., SOUNDARARAJAN, G., PATIL, K.,

KUMAR, C., AND BHAGWAT, D. A practical implementation of clustered fault tolerant write acceleration in a virtualized environment. In *13th USENIX Conference on File and Storage Technologies* (2015), FAST '15.

[41] PRABHAKARAN, V., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Analysis and evolution of journaling file systems. In *Proceedings of the USENIX Annual Technical Conference* (June 2005), ATC '05.

[42] QURESHI, M. K., SRINIVASAN, V., AND RIVERS, J. A. Scalable high-performance main memory system using phase-change memory technology. In *Proceedings of the International Symposium on Computer Architecture* (2009), ISCA '09.

[43] RITCHIE, D. M., AND THOMPSON, K. The UNIX time-sharing system. *Commun. ACM 17* (July 1974), 365–375.

[44] ROSELLI, D., AND ANDERSON, T. E. A comparison of file system workloads. In *Proceedings of the USENIX Annual Technical Conference* (2000), ATC '00.

[45] SAXENA, M., AND SWIFT, M. M. FlashVM: Revisiting the virtual memory hierarchy. In *Proceedings of the USENIX Annual Technical Conference* (June 2010), ATC '10.

[46] SHRIVER, E., SMALL, C., AND SMITH, K. A. Why does file system prefetching work? In *Proceedings of the USENIX Annual Technical Conference* (1999), ATC '99.

[47] SIMHA, D. N., LU, M., AND CHIUEH, T.-C. An update-aware storage system for low-locality update-intensive workloads. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), ASPLOS XVII.

[48] SOARES, L., AND STUMM, M. FlexSC: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation* (2010), OSDI'10, USENIX Association, pp. 1–8.

[49] STANDARD PERFORMANCE EVALUATION CORPORATION. SPECsfs2008. http://www.spec.org/sfs2008/, 2008.

[50] SUN MICROSYSTEMS. Filebench 1.4.9.1. http://sourceforge.net/projects/filebench/, 2011.

[51] TANENBAUM, A. S. *Modern Operating Systems*, 3rd ed. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[52] THERESKA, E., SCHINDLER, J., BUCY, J., SALMON, B., LUMB, C. R., AND GANGER, G. R. A framework for building unobtrusive disk maintenance applications. In *Proceedings of the USENIX Conference on File and Storage Technologies* (March 2004), FAST '04.

[53] USECHE, L., KOLLER, R., RANGASWAMI, R., AND VERMA, A. Truly non-blocking writes. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems* (June 2011), HotStorage '11.

[54] VENKATARAMAN, S., TOLIA, N., RANGANATHAN, P., AND CAMPBELL, R. H. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the USENIX Conference on File and Storage Technologies* (February 2011), FAST '11.

[55] VMWARE, INC. VMware Virtual SAN. http://www.vmware.com/products/virtual-san/, 2013.

[56] VOLOS, H., TACK, A. J., AND SWIFT, M. Mnemosyne: Lightweight persistent memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2011), ASPLOS XVI.

[57] WU, X., AND REDDY, A. L. N. Exploiting concurrency to improve latency and throughput in a hybrid storage system. In *Proceedings of the IEEE International Symposium in Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (September 2010), MASCOTS '10.

[58] ZHOU, P., ZHAO, B., YANG, J., AND ZHANG, Y. A durable and energy efficient main memory using pcm technology. In *Proceedings of the International Symposium on Computer Architecture* (2009), ISCA '09.