Cluster Computing (2020) 23:895–910
https://doi.org/10.1007/s10586-019-02964-8

AutoBahn: a concurrency control framework for non-volatile file buffer

Hyeongwon Jang[1] · Sang Youp Rhee[1] · Jae Eun Kim[1] · Yoonhee Kim[2] · Hyuck Han[3] · Sooyong Kang[1] · Hyungsoo Jung[1]

Received: 1 March 2018 / Revised: 9 April 2019 / Accepted: 21 July 2019 / Published online: 29 July 2019

Abstract
Storage systems in general and databases in particular usually balance between write durability and performance. It is not uncommon that write durability often relies on transaction systems that also offer a relaxed model of durability for performance. As hardware vendors provision more cores and faster storage devices, attaining fast data durability for concurrent file writes is demanding to high-performance storage systems in large-scale cluster systems. We approach the challenge by proposing a system that uses a small amount of fast persistent memory for buffering concurrent file writes while preserving data durability. The main technical issue in designing a durable file buffer is allowing concurrent file writes to store data in a shared and limited space of persistent memory without incurring lock or resource contention. This article addresses such issue and presents AutoBahn, a durable file buffer that expedites file I/O operations. To prove practicality and effectiveness, we implemented a prototype of AutoBahn in Linux-4.8.7 and ran several key-value systems—Redis, RocksDB, and WiredTiger—on AutoBahn. Evaluation results on a multicore server demonstrate that all the key-value systems achieved performance levels almost matching the non-durable counterpart. AutoBahn is a useful method that can efficiently deal with concurrent file I/O streams on multicores and fast storage devices.

Keywords Storage system · Concurrency control · File buffer · Non-volatile memory

1 Introduction

Synchronous I/O delay has been of pivotal concern to system architects since the emergence of non-volatile storage. Although modern storage devices such as NVMe SSDs strive to reduce I/O latency, there remains an intractable barrier between volatile memory and stable storage as the major cause of the limited performance of computer systems. One of the main hurdles in overcoming the latency gap using a software technique is providing durability for the write-requested data. In the aspect of I/O performance, durability is expensive, and there is a trade-off between them; the `ext4` file system uses the metadata journaling mode as its default mode instead of the data journaling mode, and some commercial database management systems (DBMS) provide several options that alleviate durability on purpose and gain some performance benefit.

With the advent of non-volatile memory (shortly, NVRAM), traditional memory hierarchy is gradually changing its paradigm. Data in NVRAM can be accessed in a comparable time as DRAM access, while preserving

✉ Hyungsoo Jung
hyungsoo.jung@hanyang.ac.kr

Hyeongwon Jang
hyeongwon@hanyang.ac.kr

Sang Youp Rhee
sangrhee@hanyang.ac.kr

Jae Eun Kim
jaeeunkim@hanyang.ac.kr

Yoonhee Kim
yulan@sookmyung.ac.kr

Hyuck Han
hhyuck96@dongduk.ac.kr

Sooyong Kang
sykang@hanyang.ac.kr

[1] Department of Computer Science, Hanyang University, Seoul, Korea

[2] Deptartment of Computer Science, Sookmyung Women's University, Seoul, Korea

[3] Department of Computer Science, Dongduk Women's University, Seoul, Korea

durability at the same time [1]. We can effectively exploit these characteristics of NVRAM for enhancing the overall I/O performance by using NVRAM as a buffer cache. However, in multicore systems, designing an efficient NVRAM-based buffer cache is not a simple task. Since the buffer cache is a kind of shared kernel data structure among the processes (or threads) involved in I/Os, concurrent access to the buffer needs to be controlled. This *concurrency control* problem has attracted little attention in designing an I/O buffer cache management scheme, until now, since a simple synchronization protocol is believed to be enough to solve the problem. However, we found that it is not a simple problem anymore because (1) current high-end systems have tens of or hundreds of cores that can invoke unprecedented number of I/Os concurrently, and (2) state-of-art storage devices can accommodate up to multiple gigabytes of data I/O in each second. A simple coarse-grained synchronization scheme, such as using a global lock or mutex, can act as a new performance bottleneck in multicore systems equipped with multiple high performance storage devices (e.g., NVMe SSDs). Therefore, a carefully designed concurrency control scheme is desired to be incorporated in the NVRAM buffer cache management scheme.

In this article, we concentrate on the concurrency control problem in the NVRAM buffer under a multicore environment. Specifically, we present AutoBahn, a kernel-level concurrency control framework, that both maximizes I/O throughput to an existing storage device and provides the fast guarantee of durability using a small capacity of non-volatile memory. The NVRAM buffer cache in AutoBahn, located between DRAM and the storage media, also acts as a burst buffer in the middle of the write operation so that it can efficiently accommodate intensive writes from applications. AutoBahn accelerates requested I/O in NVRAM and maximizes the utilization of storage devices. The centerpiece of AutoBahn is the pipeline technique using a lock-free FIFO queue, which is a highly concurrent data structure.

For the evaluation, we implemented a prototype of AutoBahn as a kernel module for Linux-4.8.7, and AutoBahn module ensures fast durability for regular file writes conducted by multithreaded programs or multiple processes. To measure the performance, we use three open-source key-value storage systems: Redis [19], RocksDB [20] and WiredTiger [25]. The results show that AutoBahn attains fast durability for regular file writes with negligible synchronization overhead.

This article is an extended version of our prior work [10]. The details of the enhancements are as follows: (1) we present the performance bottleneck in the current `ext4` file system when multiple threads access their *private* files and identify the main culprit for the problem, (2) we list implementation issues of AutoBahn to Linux and present our solutions to them, and (3) we conduct additional experiments using YCSB benchmark to evaluate the performance of AutoBahn under real world key-value workload.

This article is organized as follows: we first present related prior work in Sect. 2. Section 3 presents the synchronization overhead in the current I/O stack, which motivated this work. Section 4 introduces the overall design of AutoBahn. In Sect. 5, issues and our solutions in implementing AutoBahn to Linux are presented. Section 6 evaluates AutoBahn's synchronous write performance with YCSB workload on Redis, RocksDB and WiredTiger as well as synchronous random and skewed performance with our microbenchmark. Finally, we conclude in Sect. 7.

2 Related work

File systems design rooted in the nature of slowness of storage devices has suffered from poor scalability in modern multicore platforms. This is unwelcome to both vendors and customers as the cluster equipped with expensive devices can become an white elephant. One of early approaches to address the performance concern is to improve system utilization by using *latency-hiding techniques* [5, 17]. This has gained significant attention from both database and systems communities, since durability in system software is regarded as an inevitable property for ensuring persistence of data. Prior proposals, developed based on latency-hiding techniques, are trying their best to reduce the I/O latency without sacrificing durability, but still cannot overcome the nature of the slowness of persistent storage. Proposals [3, 24] in the HPC community also pursue similar goals.

Trying to approach the issue from a different perspective, researchers found that the scalability problem inherent in file systems is mainly based on using shared data structures among multiple actors, and many studies have been conducted to improve the scalability of file systems in various ways. One example is SpanFS [12] that partitions the device blocks into multiple domains and makes each of them to manage its micro file system independently to disperse contention on shared data structures. Another scalable file system is ScaleFS [2] which separates in-memory file system from on-disk one and uses operation logs per core to avoid cache conflicts.

Exploiting the attractive characteristics of NVRAM presented in Sect. 1, a large number of schemes, in which NVRAM is used for write buffering and caching, have been proposed [3, 6, 7, 11, 13, 14, 21]. This approach, that is exploiting NVRAM as an I/O buffer cache, has been explored with the assumption that NVRAM can enhance

overall I/O performance due to improved cache hit ratio, while preserving data durability. Therefore, researchers have primarily focused on maximizing the hit ratio to minimize the storage access under the limited capacity of NVRAM. However, they did not pay attention to the *shared* nature of the buffer cache, and did not present any solution to the concurrency control problem.

Despite all these efforts, contention on the shared buffer cache and the performance bottleneck caused by the contention do still exist. Hence the primary focus of this article is on addressing such long-standing issues in order to make file systems scalable and better suitable for high performance storage in HPC systems.

3 Nonscalability in concurrent file I/O

Prior studies [13–15, 18] on a fast, durable write buffer using non-volatile memory have placed conventional buffer cache on fast non-volatile memory. This design has its own advantage in that it needs not understand the file system semantics to preserve the correctness, and this approach works well on a single-socket server platform where inter-socket cache invalidation would not be necessary. As core count has increased dramatically since early 2000s, all the then-latent performance bottleneck issues have now begun to arise in numerous places in file systems [16]. This would definitely render the previous approach less effective in maximizing both CPU and storage device utilization. And scaling the performance of concurrent file writes with data durability is of importance to general-purpose (or key-value) storage systems deployed in the large-scale cluster systems, where such workloads often take place. This section describes the motivation of the present work by exploring the performance bottleneck issues in file systems and identifying the main culprit for the problem. To this end, we conduct a few experiments on the unmodified `ext4` file system and present evaluation results. The performance issues that are left unresolved with the non-volatile buffer being placed under file systems are the main motivation of the present work.

3.1 Experimental setup

To evaluate the performance of file writes on fast storage devices, we use a simple multithreaded microbenchmark. Each thread is designed to append fixed-sized (either 4 KiB or 256 Bytes) data to the end of its private file. Our microbenchmark runs on a 36-core Supermicro server (Table 1). The server has fast non-volatile storage consisting of two enterprise NVMe SSDs, each of which produces the peak I/O speed of 3.5 GiB/s with a 50 µs delay for the sequential writes of the 16 KiB block. Each thread creates its private file randomly in one of the NVMe SSDs, thus spreading the I/O workloads evenly. The Linux kernel that our experiment is conducted on is version 4.8.7, and the ext4 file system with journaling turned off is used to maximize the writing throughput. The configuration of the background kernel flusher threads to write out dirty data to the disk is set to default as of Ubuntu 16.04.

3.2 Multithreaded file I/O to private files

Concurrent file writes to the shared file incur substantial lock contention, which undoubtedly becomes worse on multicore platforms. A recent study [16] has shown unexpected results suggesting that concurrent file reads on the same file block also incur performance collapses on a multi-socket server platform, mainly due to the spurious cache invalidation messages generated by atomic operations executed on the shared variables. We focus here on the performance of multithreaded file writes to per-thread *private* files, and the normal expectation is that such file writes should be scalable since there is no interference between concurrent writes owing to the fact that the threads act on their private file.

We run the microbenchmark with two write units: 4 KiB and 256 bytes. The experiments with a small write unit is of significance to storage system designers, since it gives an important intuition about the performance of key-value database systems that must guarantee the durability of database logs whose size is around several hundreds bytes for online transaction processing (OLTP) workloads. However, Fig. 1b shows that the throughput of ext4 with a small write unit (i.e., 256 bytes), compared to the throughput with a large write unit (i.e., 4 KiB), cannot attain commensurate improvement as load increases, although each thread is appending data to its private file. In addition, the figure indicates that small writes cannot utilize the full device bandwidth even without calling *fsync*.

An in-depth look through profiling reveals that this limited throughput is mainly ascribed to excessive atomic operations performed on shared variables in the `buffer_head` structure during concurrent writes to the ext4 file system (See CPU utilization in Fig. 1b). The `buffer_head` is a structure for buffer pages to represent blocks in the page and is maintained for fast access to superblocks, inode blocks, and discontinuous disk blocks. When writing to each file, the `buffer_head` pointing to the associated inode block needs to be marked as dirty. At this time, file system increases the reference counter for the `buffer_head` or changes some bits inside the `buffer_head` (dirty bit, etc.) atomically. As mentioned earlier, because each thread writes to its own file, the contention due to the atomic operation does not seem to occur, but the reality is different.

Table 1 Supermicro Server 6028R-TR specifications

Component	Specification	Component	Specification
Processor	18-Core Intel Xeon E5-2699 v3	Processor sockets	2 Sockets
Hardware threads	36 (hyper threading enabled)	Clock speed	2.3 GHz
L1 D-cache	32 KiB (per core)	L1 I-cache	32 KiB (per core)
L2 cache	256 KiB (per core)	L3 cache	45 MiB (per socket)
Memory	488 GiB DDR4 2400 MHz	Storage	Samsung SM1725 NVMe SSD (3.5 TB)

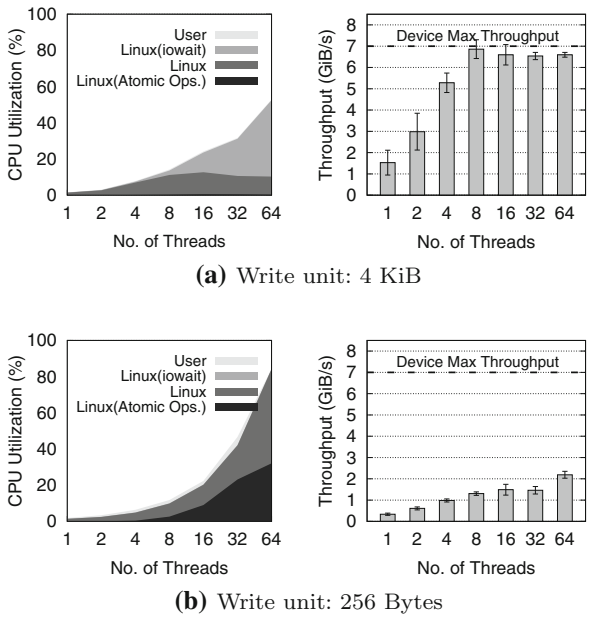


Fig. 1 The breakdown of CPU utilization and the write throughput with append-only file writes (no *fsync*)

The size of an inode block is normally smaller than that of a page buffer. In this experiment, the sizes of an inode block and a buffer page are 256 Bytes and 4 KiB, respectively, which are the default values of the Ubuntu 16.04. Hence, multiple inode blocks can be stored in the same buffer page and so multiple threads accessing their private files can share the same `buffer_head`, as depicted in Fig. 2. It is notable that concurrent atomic operations (e.g., atomic fetch-and-increment) performed by multiple threads on a shared data structure can indeed be limited by the performance upper bound of the hardware-based synchronization, as detailed in [4]. And we found the phenomenon in our experiment (Fig. 1b) when multiple threads access their shared `buffer_head` using atomic operations, as shown in Fig. 2. The codes that invoke these atomic operations exist mainly in the following functions: `ext4_mark_inode_dirty()`, `__find_get_block()`, and `__ext4_handle_dirty_metadata()`.

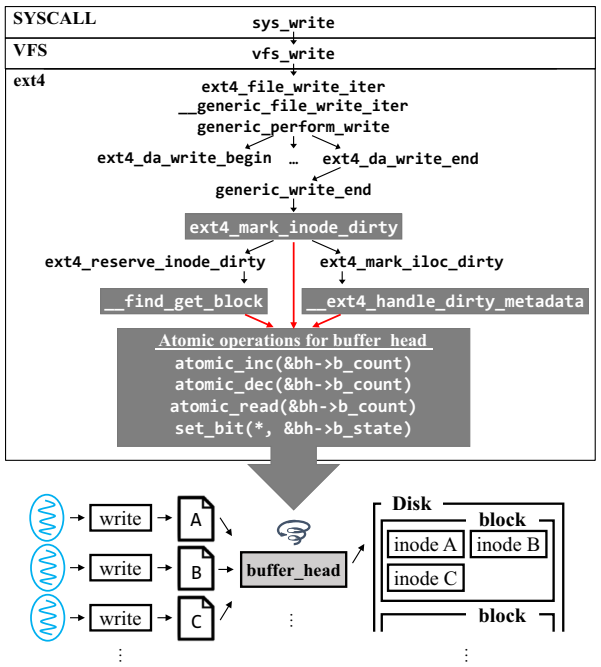
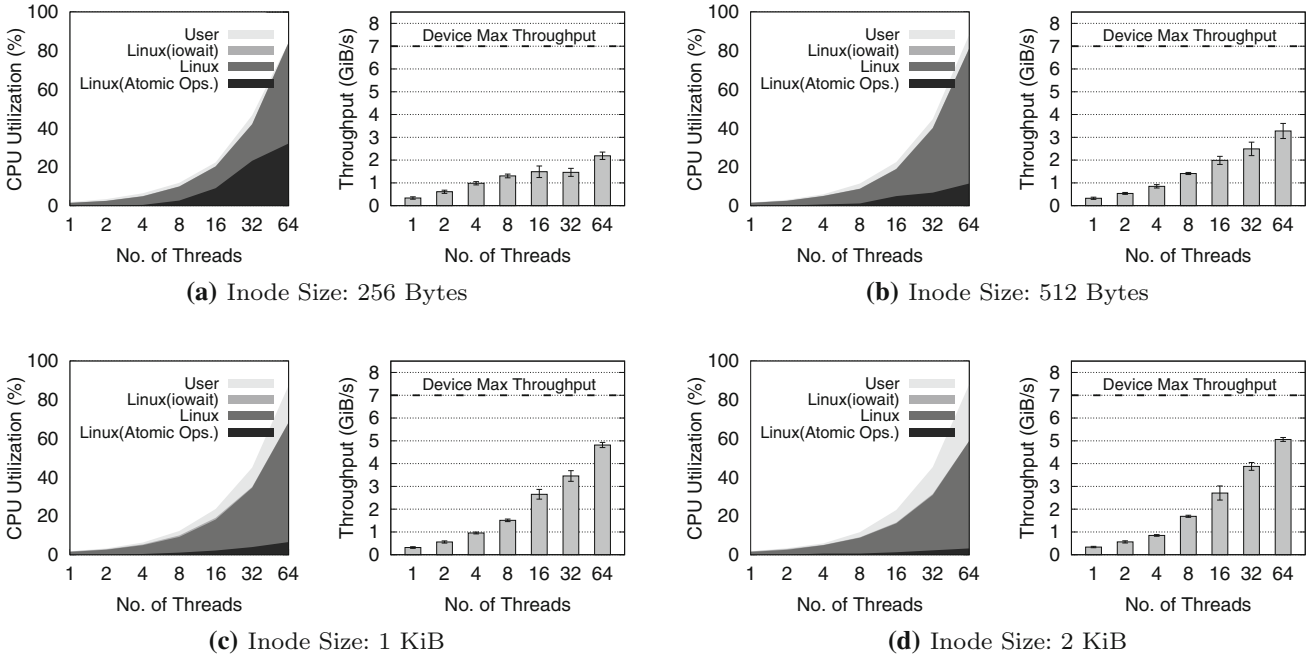


Fig. 2 Contention in the `buffer_head`

To confirm that the concurrent accesses to the shared `buffer_head` limits the write throughput of file systems on many cores, we conducted the same experiment by varying inode size from 256 Bytes to 2 KiB, with the buffer page size being set to 4 KiB. Larger inode size lets less number of threads share the same `buffer_head`, incurs less contention in it and, as a result, is expected to show higher throughput. Figure 3 confirms it. As inode size increases, the portion of the 'atomic operations' in the CPU utilization decreases while 'user' portion increases, which results in larger throughput, as anticipated. It is notable that we can hardly see the 'iowait' portion regardless of the inode size. This implies that although we use cutting-edge semiconductor-based storage devices such as enterprise SSDs, the IO performance in the high performance storage systems can be limited by the software bottlenecks rather than the storage device.



(a) Inode Size: 256 Bytes

(b) Inode Size: 512 Bytes

(c) Inode Size: 1 KiB

(d) Inode Size: 2 KiB

Fig. 3 CPU utilization and throughput with varying inode size (Buffer page: 4 KiB, Write Unit: 256 Bytes)

3.3 Challenges

We have discussed the unexpected performance issues arising in concurrent file writes to private files. The performance issues unveiled here have a few important implications: (1) placing non-volatile write buffer underneath file systems may suffer scalability bottlenecks which are inherent in file systems (e.g., bottleneck in the shared `buffer_head`) and (2) the design of a scalable non-volatile write buffer should be done delicately to maximize device utilization and to avoid possible contention issues. We address all the issues and present AutoBahn, which is designed to overcome these challenges.

4 AutoBahn: design

The inherent bottleneck issues of file systems due to the excessive execution of hardware-based synchronization constitute the main motivation of the present work. We approach the issues mainly to relieve synchronization bottlenecks. In this section, we present the overall system design of AutoBahn. We first describe the overall architecture and then explain the data structures and relevant algorithms with the design rationale.

4.1 Overall architecture

The main design rationale of AutoBahn is to escape the hardware-based synchronization overhead while ensuring data durability, when operating systems need to handle concurrent file I/O requests, which unfortunately have negative impacts on the performance of well-known general-purpose file systems. In this work, we aim at designing AutoBahn to achieve the goals, regardless of I/O size and write patterns. Figure 4 shows the overall architecture of AutoBahn with a flow diagram of a regular file write operation. As illustrated in the figure, AutoBahn, unlike other proposals sitting below the file systems, is placed above the file system layer, and works as a durable write buffer. The durable write buffer placed above the file system can substantially improve the performance of durable file writes and alleviate inherent contention in the file system.

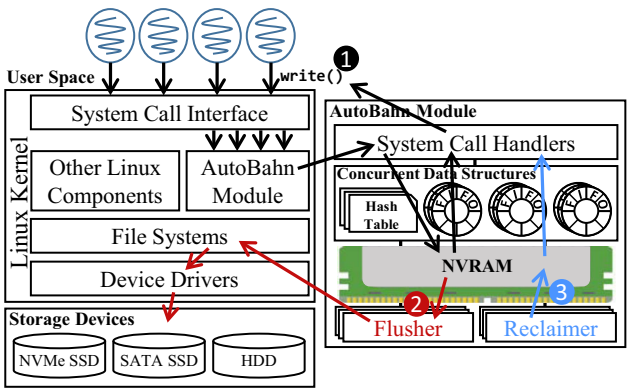


Fig. 4 The overall architecture of AutoBahn and the general flow of the synchronous `write` operation

The architecture of AutoBahn consists of two components. The first consists of buffer blocks and their associated meta-data residing in non-volatile memory (i.e., NVRAM) for ensuring fast data durability, and the second component includes concurrent data structures and related algorithms for handling concurrent file I/O requests with reduced hardware-based synchronization overhead. NVRAM-resident and DRAM-resident data structures for managing durable buffer blocks are first discussed in Sect. 4.2, and discussions on efficient algorithms for achieving high utilization and concurrency are presented in Sects. 4.3 and 4.4.

To make it easy for applications to use the system, AutoBahn intercepts a few system calls relevant to file I/O operations. For instance, the general flow of a `write()` operation is shown in Fig. 4 with colored arrows. When applications request synchronous writes to files, the AutoBahn system call handler comes in and copies user data to durable buffer blocks in NVRAM. At this point, the requested data is made durable so that the AutoBahn system call handler returns the write system call. This ensures fast durability. The next step is to sync durable buffer blocks to persistent storage devices to make room in NVRAM for other file I/O requests. Flushing the dirty buffer blocks and recycling clean buffers for future use should be handled efficiently for sustaining high device utilization, and AutoBahn processes these tasks asynchronously, using a multi-stage pipeline structure with dedicated kernel threads (see Sect. 4.3). For the efficient processing of multiple file I/O requests with reduced synchronization overhead, we use *lock-free FIFO queue* data structures (Sect. 4.4).

4.2 Data structures

Since AutoBahn works as a durable write buffer for underlying file systems, the data structures for managing buffer blocks in NVRAM should be properly designed. Two types of data structures exist in AutoBahn: (1) data structures in NVRAM and (2) data structures in DRAM. The first type refers to non-volatile buffer blocks and their control blocks, which are essential for ensuring the durability of data and need to be in NVRAM. The second type includes concurrent data structures for high concurrency without incurring a hardware-based synchronization bottleneck. Figure 5 clearly summarizes the NVRAM-resident and DRAM-resident data structures used in AutoBahn.

4.2.1 NVRAM-resident data structures

Data structures organized in NVRAM resemble disk-resident structures in file systems. Extra information needed for NVRAM is information required for the correct recovery of unflushed data buffers. NVRAM-resident data structures include durable buffer blocks to store file data. Along with the data blocks, NVRAM stores inodes for managing data blocks, one for each data block. The meta-data for identifying files is called a volume, which is assigned to each file. Table 2 provides a summary.

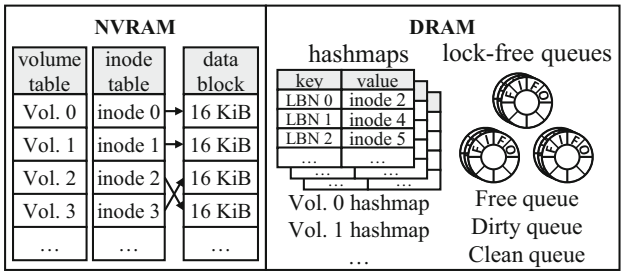


Fig. 5 Data structures of AutoBahn

4.2.2 DRAM-resident data structures

Unlike NVRAM-resident data structures, AutoBahn needs DRAM-resident ones for achieving high concurrency and so high storage utilization when multiple file I/O requests arrive. To access inodes quickly, each volume entry uses a hashmap of inodes indexed by a logical block number (LBN). Additionally, inodes representing different states of buffer blocks are managed by three lock-free queues; the three states of a data block are *free*, *dirty* and *clean*. With these structures, AutoBahn utilizes a three-stage pipeline system, which is described in next two sections.

4.3 Three-stage circular pipeline

AutoBahn uses NVRAM as a durable write buffer. However, as shown in Sect. 3, adopting naive buffer management algorithms for AutoBahn may not be efficient when processing concurrent file I/O requests. Since processing file I/O requests can be divided into multiple tasks, AutoBahn uses a three-stage circular pipeline structure to handle concurrent file I/O requests while maximizing CPU and device utilization. Figure 6 shows the overall architecture of AutoBahn pipeline structure. In AutoBahn, there are three stages separated in the pipelining: *write*, *flush* and *reclaim* stages. We assign dedicated kernel threads for each stage, which are responsible for processing the given work. We denote these dedicated threads as *writer*, *flusher* and *reclaimer* threads, respectively. Blocks in NVRAM are managed by a lock-free FIFO queue (LFQ), which is placed between the stages. Lock-free FIFO queues, depending on the state of blocks they manage, are denoted as *free*, *dirty*, or *clean* LFQ, and a dedicated thread plays a role as either a consumer of one queue or a producer of another queue. These lock-free queues enable

Table 2 Description of NVRAM metadata structures

Structure	Field name	Description
Volume	*fullpath*	Full path name of a file
	is_partial	A flag bit that indicates if a file is opened with O_NVMPARITAL (Sect. 5.4)
	file_size	Size of a file
Inode	*volume*	Back pointer to its volume entry
	lbn	Logical block number of an inode
	state	Current state of its block (e.g., free, dirty and clean)
	block	Pointer to its block

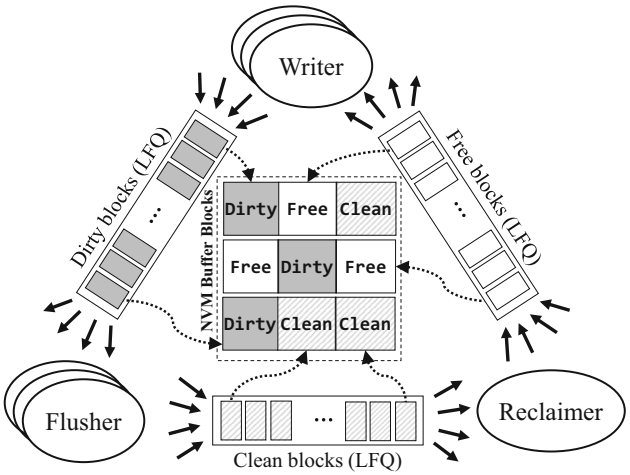


Fig. 6 The pipeline structure in AutoBahn

AutoBahn to maximize performance without incurring a synchronization bottleneck and scale well. The details of LFQ are presented in Sect. 4.4. Following is a brief description of what happens during each stage in pipelining.

4.3.1 Write stage

The first stage of AutoBahn is the *write stage*. When an application requests the write I/O with an ordinary write system call, this invokes the AutoBahn write system call handler. The AutoBahn write system call handler iterates (1) finding a free block from the *free* LFQ, (2) copying data to that block, (3) updating the block state to *dirty*, and (4) inserting the block into the *dirty* LFQ, until the entire set of data is written to the NVRAM buffer. In order to benefit from caching, AutoBahn adopts delayed flushing for dirty blocks that may be served without being reloaded from storage. AutoBahn manages the file layout with a hashmap in memory and expects to get the target block in constant time. When copying user data to NVRAM, AutoBahn perserves atomicity per block (i.e., block-level atomicity in Sect. 5.2.

4.3.2 Flush stage

The primary job of a flusher thread is to write dirty blocks in the NVRAM buffer to the persistent storage device. In the flush stage, the flusher thread (1) fetches previously written blocks from the *dirty* LFQ, (2) writes them to storage device, (3) changes their states to clean, and (4) puts them into the *clean* LFQ. The target blocks to be flushed can also be accessed by writer threads, and this may create noticeable synchronization overhead. This issue is addressed in Sect. 5.1. Also, during the flush stage, it includes the heaviest operation (i.e., write out to disk). Therefore, AutoBahn uses various optimization techniques in the flush stage. The detailed description is covered in Sect. 5.3.1.

4.3.3 Reclaim stage

The main job of the reclaimer thread is to reclaim *clean* blocks and convert them to *free* blocks so that writer threads can handle write requests without showing a performance hiccup due to the limited capacity of the NVRAM buffer. The reclaimer thread (1) dequeues the blocks from the *clean* LFQ and (2) marks their corresponding hashmap entries as invalid. Then the reclaimer thread (3) initializes the inodes of the blocks and (4) inserts the blocks into the *free* LFQ. The initialized inode contains the `NULL` *volume* and *lbn*, *free* block state, and a pointer to the block. Once the hashmap entry is invalidated, the initialized inode and its corresponding block can be returned for reuse. Validation occurs when the writer thread reenters that entry, assigning a new inode value to that entry. As in the flush stage, the writer thread and the reclaimer thread can simultaneously access the same block during the reclaim stage; the synchronization between them is addressed in Sect. 5.1.

4.4 Lock-free FIFO queue (LFQ)

In AutoBahn pipeline structure, durable shared buffers are processed by a dedicated worker thread and then delivered to the next pipeline stage. This naturally raises a contention

issue between concerned workers. Since a naive design for buffer management brings the synchronization overhead again, we designed a concurrent multi-producer and multi-consumer lock-free FIFO queue that is a building block for AutoBahn pipeline system. An LFQ utilizes only one hardware-based synchronization instruction (i.e., atomic fetch_and_increment) and is intended to manipulate durable buffer blocks without causing lock contention. It is implemented with a bounded circular array. An enqueue operation holds a unique producer ticket number (i.e., `p_position`), and stores its data in the circular array indexed by that ticket number. A dequeue operation, then, gets its unique consumer ticket number (i.e., `c_position`) and consumes the buffer with the matching ticket number. Both enqueue and dequeue operations, if the target array entry is not ready for use, do spin-wait shortly until it becomes ready (i.e., `while` loop). Figure 7 shows the implementation overview of our LFQ with pseudocodes for enqueue and dequeue operations. Since assigning a unique ticket to producers and consumers enforces workers to check their target entry only, an LFQ would never allow excessive execution of hardware-based synchronization on a single hot spot, thus alleviating synchronization bottlenecks observed in file systems before.

4.5 Recovery

Correct data recovery is of importance in ensuring data persistence in storage devices, not in NVRAM, and recovery must preserve data persistence in the face of unexpected failure. In AutoBahn, data is temporarily and durably stored in NVRAM and later flushed to the designated place in storage. AutoBahn always runs the recovery procedure, once the operating system restarts, to make sure that data in NVRAM is correctly synced to the storage devices. By doing this, AutoBahn can make durable buffer blocks *clean*, thus persisting all data in the stable storage devices. To this end, AutoBahn upon restart iterates through the volume entries and inodes to find candidate buffer blocks to flush. Dirty blocks, if found, are flushed to storage. When a buffer block is successfully flushed, its state is set to *clean*. Since the recovery process is *idempotent*, flushing the dirty buffer blocks multiple times due to consecutive failure events preserves the correctness of the AutoBahn recovery process.

5 AutoBahn: implementation issues

We have presented the overall system design of AutoBahn. This section describes the implementation details for robustly materializing the general designs and advanced techniques used for overcoming a few technical challenges.

5.1 Destressing synchronization bottleneck

As the prior study [4] thoroughly acknowledged and as our preliminary experiments showed the side effects in file systems, the maximum number of atomic instructions that can be executed on a shared variable is limited by the performance of hardware-based synchronization. This definitely has negative impacts on the performance of multithreaded file I/O, especially when the I/O unit is small. Destressing such a hardware-based synchronization bottleneck is one of the technical issues we must deal with in designing AutoBahn. Although the pipeline structure of AutoBahn makes the best use of lock-free queues when different worker threads access shared data structures with alleviated synchronization overhead, there are two worrisome situations where synchronization overhead between different worker threads may grow. The essential problem in two cases involves the enqueuing of a buffer block to a lock-free queue, which inevitably needs to change the state of the buffer, and changing the state variable should be done correctly (i.e., mutual exclusion). A naive solution is to use a big lock, which again brings lock contention problems. To mitigate the issue, we maintain a per-block lock variable.

The first case is the synchronization between the writer thread and the flusher thread. This situation occurs when the writer thread tries to update the dirty data block in a *dirty* LFQ, which is also the target block to be flushed to

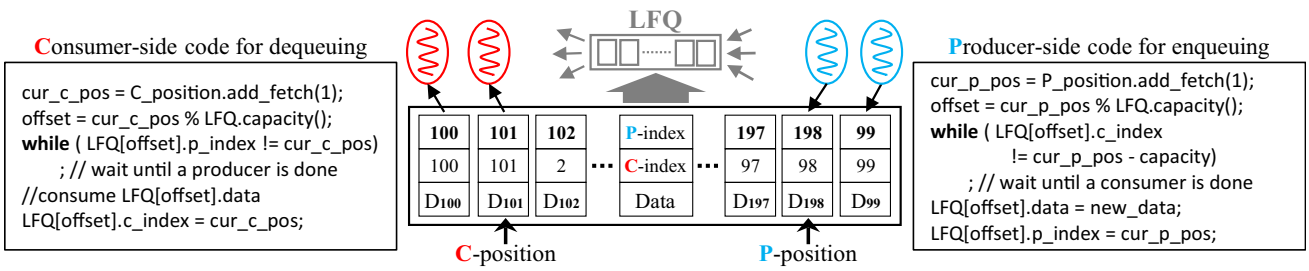


Fig. 7 Operations in the lock-free FIFO queue: Producers (or consumers), once they get a unique ticket, check their target entry to detect that either a queue is empty or full. If the entry is ready, they enqueue/dequeue a data buffer

storage by the flusher thread. To solve this case with reduced synchronization overhead, AutoBahn sets the policy when both the writer thread and the flusher thread try to access the same block. When the writer thread acquires the lock first, the writer can update the block as shown in Fig. 8. The flusher thread that failed to acquire the lock cannot flush the block and reinserts the block to the end of the dirty queue, hoping that it can flush that dirty block next time. Since the flusher thread would insert the block to the queue, the writer thread does not have to enqueue the block again. Otherwise, if the flusher thread acquires the lock first, the writer thread is waiting for the block to be flushed. This looks odd at first, but our deferred flushing policy (i.e., *batched flushing* in Sect. 5.3.1) would reduce the occurrence of such spin-waiting under a variety of workloads.

The second case is the synchronization between the writer thread and the reclaimer thread, when the writer tries to reuse the cached block in the *clean* LFQ. Similar to the previous case, both the writer thread and the reclaimer thread compete to acquire a per-block lock, but this case is much simpler. If the writer thread successfully acquires the lock, it can reuse the *clean* block that still holds file data (i.e., *data cache*). The reclaimer thread that failed to acquire the lock skips that block and proceeds to check the next block from the clean queue (i.e., *opportunistic locking*). Otherwise, the winner (i.e., the reclaimer thread) just invalidates the block, and the loser (i.e., the writer thread) obtains a new block from the *free* LFQ, instead of spin-waiting. As such, *opportunistic locking* by the reclaimer and the *opportunistic trial* for grabbing the same block by the writer thread would indeed need the one-time execution of a hardware-based *read-modify-write* (RMW) instruction (e.g., *compare-and-swap*, or CAS). The optimized technique used here eliminates bad spin-waiting on a lock variable, thus avoiding a potential synchronization bottleneck.

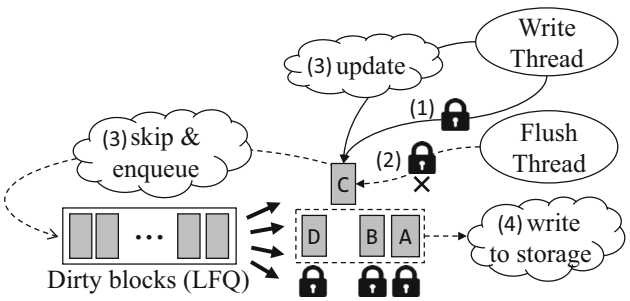


Fig. 8 Synchronization between the writer and the flusher on the same dirty block: Both the writer and the flusher tries to acquire the per-block lock of a dirty block C. (1) The writer successfully acquired the lock, (2) the flusher failed. (3) The writer updates the dirty (cached) block while the flusher skips the block and re-enqueue the block to the end of the *dirty* LFQ. (4) The flusher writes dirty blocks whose per-block locks have been acquired

5.2 Guaranteeing block-level atomicity

Write operations inside SSDs are aligned to page size, which ranges from 4 to 16 KiB depending on the model of the SSD. While some SSDs show shorn write behavior on system crash [26], in many SSDs, page-level atomicity is a guaranteed behavior owing to their power-loss protection functionalities [9, 23]. AutoBahn also guarantees block-level atomicity when data is written to NVRAM.

5.2.1 Out-of-place update

The well-known shadow paging (or out-of-place update) technique to ensure a block-level atomicity was originally proposed in the mid 1970s in the database community. The out-of-place update forces new updates to be written to a free buffer block first and then atomically updates the block pointer of an inode to point the new block. For a partial block write, the data in the old block needs to be copied to the free block before updating the new data, and this incurs an extra memory copy operation. Due to this, random write workloads incur substantial overhead, thus slowing down the performance.

5.2.2 Optimized in-place update

The inherent copy overhead in the out-of-place update technique that is particularly notable in partial write workloads can be significantly alleviated when write workloads are append-only. The optimization we devise here is to allow an *in-place update* with a file offset being atomically updated after copying the data. The optimized in-place update technique indeed improves the performance of append-only file I/O workloads, such as database logging and file system journaling (see Sect. 6). Figure 9 depicts how the optimized in-place update is performed for the new data appended to the end of a block. AutoBahn updates to an existing block first and then atomically updates the offset of a file. Under any failure situation, the block can be recovered to either *old* data or *new* data, not something in between. This ensures the block-level atomicity while attaining better performance than the out-of-place update.

5.3 Optimizing write performance

The trade-off between maximizing device utilization and attaining short latency is a well-known issue in storage systems. Placing the NVRAM buffer cache between DRAM and the storage device in AutoBahn can naturally

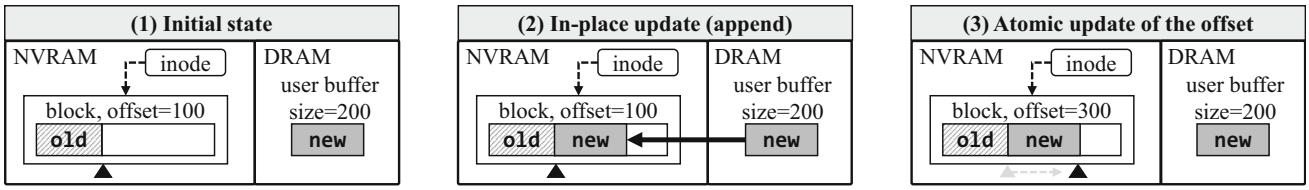


Fig. 9 Optimized in-place update for ensuring atomic append

reduce the latency of durable I/O, but the throughput improvement is another challenge we have to deal with. Since the flusher in AutoBahn performs I/O intensive tasks that basically write dirty blocks to storage devices, we devise a few optimization techniques in establishing flushing strategies to maximize the I/O bandwidth of given storage devices. In general, to expedite the flush-out phase, AutoBahn sets up several *dirty* LFQs to let multiple flusher threads run concurrently.

5.3.1 Write batching and batched flushing

In AutoBahn, write takes place on a batch unit in the flush stage. The batch unit consists of continuous dirty blocks for one volume (i.e., file). We call this *write batching*. This technique is the best fit strategy for append-only workloads, especially database logging. To implement this, AutoBahn manages the dirty blocks from one volume to be collected to the same *dirty* LFQ. In this manner, one flusher thread is responsible for one volume and flushes a group of continuous dirty blocks altogether, thus achieving better device utilization through batching the I/O unit in append-only workloads.

In addition to the *write batching*, we optimize flushing further to improve the storage device utilization. A naive flushing policy is to write dirty blocks in *dirty* LFQ whenever the flusher detects any candidate dirty blocks, and this eager flushing helps reducing the queueing delay for dirty blocks. However, the eager flushing of dirty blocks to storage devices raises an important issue in AutoBahn. Eager flushing forces flusher thread to check the dirty LFQ continuously, although there are no dirty blocks to flush, thus wasting CPU cycles due to the spin-waiting. To avoid this, the flusher thread is normally inactive, and it is awakened when the number of blocks in the *dirty* LFQ exceeds a certain threshold (i.e., the *watermark*). Figure 10 shows that only the activated flusher thread processes its associated *dirty* LFQ if the number of dirty blocks in the queue exceeds the threshold. Once the flusher thread is activated, it writes dirty blocks in its *dirty* LFQ to the storage devices, and then it will be inactive again. This flushing technique is called *batched flushing* or deferred flushing.

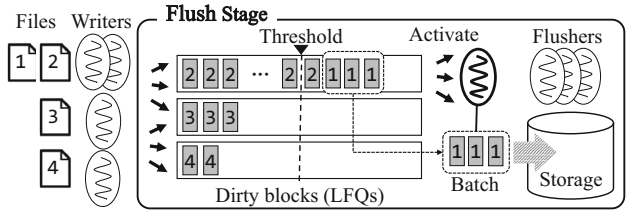


Fig. 10 Write batching and batched flushing in AutoBahn

Deferring the actual flushing time improves storage utilization without sacrificing write latency since AutoBahn keeps dirty blocks in NVRAM as a durable state. The *batched flushing* can be of advantage as well to the synchronization between the flusher thread and the writer thread, which is presented in Sect. 5.1. There are two important situations where both the writer thread and the flusher thread compete on access to the same block.

The first situation is when writing on hot spot blocks in a skewed workload. This situation causes severe contention on the same block with the naive flushing policy (i.e., eager flushing), especially when the flush thread has acquired the lock and the writer thread is supposed to wait until the lock holder (i.e., the flusher) releases the lock after flushing the dirty block. However, in batched flushing, the actual flushing is deferred until the number of dirty blocks exceeds the threshold so that the chance of contention can be substantially reduced. Also, the threshold is configurable in AutoBahn so that the hot spot data can remain in NVRAM, if applicable.

The second situation is when writing small data frequently in the append workload. As mentioned in Sect. 5.2, the writer thread may access the same block repeatedly for in-place updates. Then, eager flushing of the updated blocks may cause severe lock contention and result in poor throughput. Moreover, flushing small dirty data may amplify the total amount of written data observed in the device because writing a data block should be aligned to the block size in the storage devices. AutoBahn mitigates those issues by *batched flushing* since the flush thread would not be active until the number of dirty blocks reach the threshold. When it is activated, the blocks located in the front of the *dirty* LFQ at that time would start to be flushed to storage while the block updated by the writer thread is located in the behind.

5.3.2 Caching

In AutoBahn, NVRAM acts as a burst buffer that handles explosive I/O requests and designed to exploit maximum bandwidth in sequential writes. In addition, it is also designed to get a caching effect on frequently used blocks in random writes. As described in Sect. 4.3.1, a writer thread can safely rewrite blocks in NVRAM. Also, by setting the threshold for batched flushing at the flush stage, dirty blocks remain in NVRAM until being flushed. Since this threshold is configurable, it is possible to buffer as much as the capacity of NVRAM. Using this caching technique, AutoBahn can effectively maximize I/O performance without accessing the storage device for hot-spot blocks. In that point of view, NVRAM in AutoBahn takes the same role as the existing buffer cache. However, since AutoBahn assumes small sized NVRAM, it proactively flushes out dirty blocks to reclaim the nonvolatile buffer block via batched flushing.

5.4 System calls implementation

Autobhan is implemented as a pluggable kernel module on the Linux VFS layer and uses ext4 as the underlying file system for evaluation. As shown in Fig. 4, Autobahn module is located in between system call interface and file system layer. In this way, Autobahn can intercept some fundamental system calls related to file system and handle those in its way alleviating bottlenecks in existing file system. For applications to access a file via Autobahn, the file must be opened using the O_NVM flag. The files opened with this flag are managed by Autobahn using NVRAM as a fast durable buffer with its own caching instead of existing buffer caching in file system. Applications requesting fast durable file writes using Autobahn are successfully returned once user data are safely copied to NVRAM. Physical data copy to NVRAM is guaranteed by flushing cache line using either of `CLWB` or `CLFLUSHOPT` instructions [8], followed by proper memory fence instructions. To preserve block-level atomicity (Sect. 5.2), AutoBahn by default only accepts block size aligned write requests. However, it can accept unaligned writes (i.e., partial writes) when a file is opened with another AutoBahn-specific flag, O_NVMPARTIAL. This flag is useful for append-only workloads, such as database logging and file system journaling. Since NVRAM in Autobahn keeps the durable buffer blocks intact with their own recently written versions, application can read blocks instantly from NVRAM without incurring disk I/O unless those blocks are not flushed by the internal mechanism of Autobahn (Sect. 4.3.2).

5.5 Limitations

Since the prototype AutoBahn system is implemented as a kernel module for Linux to focus on validating important functionality, AutoBahn may not be fully compatible with the POSIX standard. Our future work will seek to make AutoBahn POSIX compatible so that there is no disruption to legacy applications to make use of AutoBahn.

Currently, we do not allow multiple opens on a file to prevent concurrent writes to a single file. To the best of our knowledge, concurrently writing in a shared file does not perform very well in all file systems due to lock contention [16]. In this study, we focus on the performance improvement of concurrent writes in private files and scope out concurrent writes in a shared file.

6 Evaluation

In this section, we evaluate the performance of AutoBahn. For the evaluation, we use our synthetic I/O microbenchmark and the real-world benchmark—the key-value workload (YCSB) running on three open source key-value systems: WiredTiger [25], RocksDB [20], and Redis [19]. We have implemented a prototype of AutoBahn in Linux kernel 4.8.7. For the evaluation, NVRAM is emulated as a contiguous physical memory space of DRAM, with the size of NVRAM being set to 4 GiB and the block size being set to 16 KiB for the I/O unit. As we described in Sect. 3.1, we ran experiments on a machine with two sockets of 18-core Intel Xeon E5-2699 v3 (with hyperthreading being enabled) and two NVMe SSD devices each of which achieves the peak I/O bandwidth of 3.5 GiB/s. For distributing file I/O workload over two storage devices evenly, we configure the benchmark (or NoSQL databases) to create regular (or log) files in a randomly chosen device. Unless stated otherwise, for the rest of the evaluations, our experiments are conducted with the ext4 file system with a default journaling mode: the *ordered mode*.

6.1 YCSB workload

Distributing partitioned data to multiple database server instances, called database sharding, is widely used for load-balancing, scalability, high-performance, and high availability. In our key-value workload evaluation, we used the database sharding configuration in a single machine; multiple database processes run each of their own database instances individually. As described in [22], our sharding configuration has better performance benefits than the configuration with a database instance that manages

multiple database tables, mainly because lock contention in many places can be substantially reduced (or avoided).

For the performance evaluation with a key-value workload, we used the YCSB benchmark with the type-A workload (50% PUT and 50% GET), and database sharding is applied to all databases tested. For the experiments, we used three popular key-value storage engines: WiredTiger [25], RocksDB [20], and Redis [19]. In all the experiments, each update transaction is forced to flush its log to the storage device when it commits (i.e., *strict durability*). For the experiments with AutoBahn, we slightly modified the logging components so that the I/O operations for database logs are processed through AuthoBahn. We compare AutoBahn-based logging to two existing systems; ext4-based logging on NVMe SSD and tmpfs-based logging.

Figure 11 shows the transactions per second of each DBMS engine. In WiredTiger, AutoBahn (9.92 million Txns/s) shows 19.9× better performance than ext4 (497 kTxns/s) in 64 database instances. This is attributed largely to the fast synchronous I/O operations of AutoBahn for the database logging. The performance of tmpfs in 64 database instances is 11.6 million Txns/s, which is 17.3% better than AutoBahn. A similar result is shown with RocksDB.

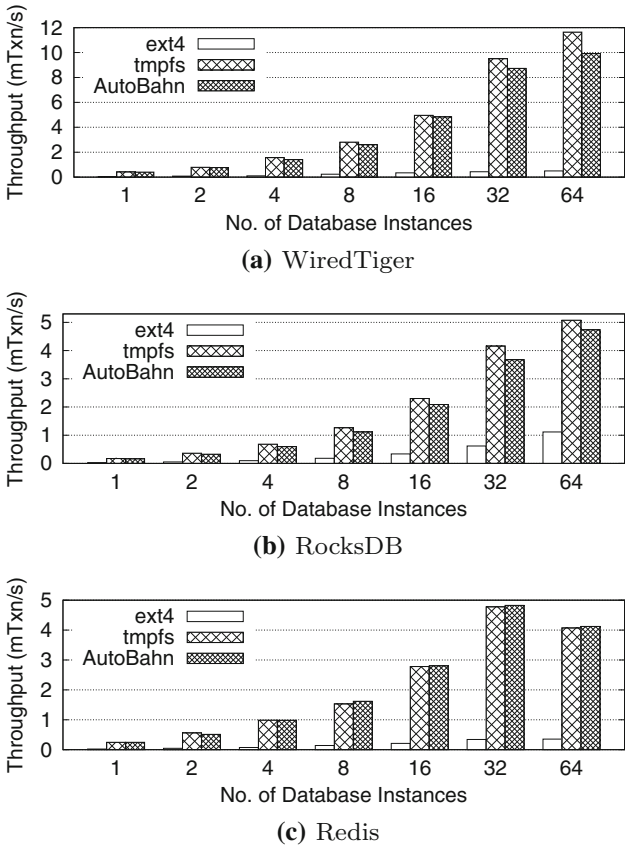


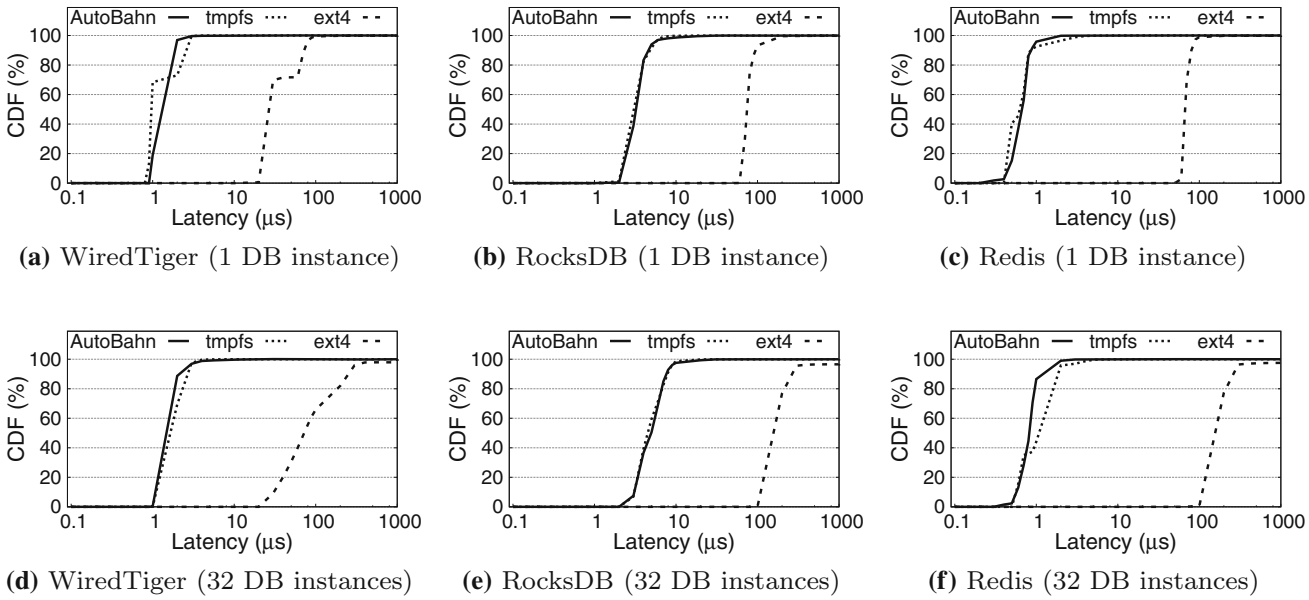
Fig. 11 Throughput on a key-value workload

AutoBahn (4.74 million Txns/s) performs 4.25× better than ext4 (1.11 million Txns/s) in 64 database instances. Meanwhile, tmpfs (5.07 million Txns/s) performs 7.16% better than AutoBahn. In the case of Redis, the peak performance is reached under 32 database instances in all three cases. The peak performance of AutoBahn is 4.82 million Txns/s while those of ext4 and tmpfs are 342 kTxns/s and 4.78 million Txns/s, respectively. AutoBahn is 14× faster than ext4 and no slower than tmpfs. The performance with more than 32 database instances drops for all cases since the Java-based Redis client takes up the CPU that could be utilized by the Redis server.

Figure 12 shows the commit latency of the YCSB benchmark on the three database engines. The experimental configuration is the same as the configuration for the previous experiment, except the benchmark process executes only update transactions. Overall, the average latency of the system with AutoBahn is much smaller than that of the ext4-based system and close to the latency measured in the tmpfs-based system, showing that the AutoBahn-based system can deliver transaction responses faster than the competitors. The average commit latency of AutoBahn in WiredTiger with 1 database instance is 1.23 µs, whereas ext4 and tmpfs show 38.207 µs and 1.31 µs, respectively. When the number of WiredTiger database instances is 32, AutoBahn shows 2.94 µs of the average commit latency, whereas ext4 and tmpfs have 132.8 µs and 1.69 µs, respectively. In the cases of RockDB and Redis, we see similar behaviours. In RocksDB with 1 instance (32 instances), the average commit latency measured in AutoBahn is 4.04 (6.62) µs while ext4 and tmpfs are measured as 80.56 (230.2) µs and 3.61 (5.43) µs. In Redis, the average latency of AutoBahn with 32 instances is 0.88 µs, and it is 99.6% reduced compared to ext4 (214.9 µs).

6.2 Microbenchmarks

To explore the various performance metrics of AutoBahn, we use our write-based synthetic microbenchmark. In the microbenchmark, we set multiple threads to write 16 KiB blocks repeatedly to its private file. We run three different write workloads; append-only, uniform, and skewed (i.e., zifian). We run our benchmark program under 16 GiB of DRAM to exclude the effect of the larger buffer cache size and set the size of NVRAM to 4 GiB of DRAM for experiments with AutoBahn. To evaluate AutoBahn, we compare the I/O throughput and latency of the AutoBahn-based system with those of two ext4-based systems; systems with ext4 (w/ fsync) and ext4 (w/o fsync). In the case of ext4 (w/ fsync), the benchmark program is designed to open files with the O_SYNC option, and this forces each write call of the benchmark process to immediately flush data (*fsync*) to the storage device for durable I/O. In



(a) WiredTiger (1 DB instance) **(b)** RocksDB (1 DB instance) **(c)** Redis (1 DB instance)

(d) WiredTiger (32 DB instances) **(e)** RocksDB (32 DB instances) **(f)** Redis (32 DB instances)

Fig. 12 CDF for the commit latency in YCSB (update 100%)

contrast, the system with the ext4 (w/o fsync) fully utilizes the buffer cache, and the durability depends on the buffer replacement policy of the Linux kernel.

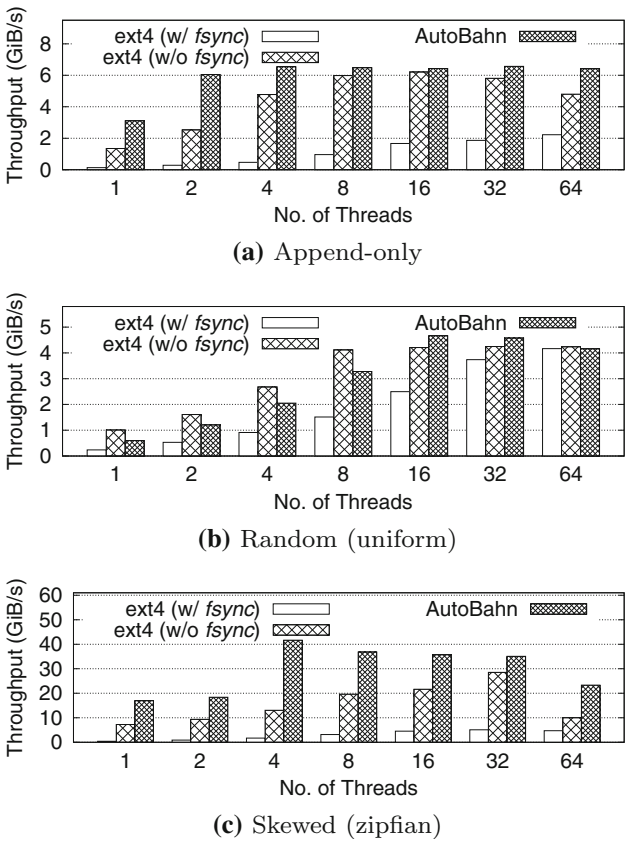
6.2.1 I/O throughput

Figure 13 shows the performance of AutoBahn, ext4 (w/ fysnc), and ext4 (w/o fsync). When we use AutoBahn for the append-only workload, the device throughput is maximized owing to the optimization techniques described in Sect. 5.3.1, and AutoBahn achieves maximum bandwidth (i.e., 6.7 GiB/s) starting from 4 threads. With 4 threads, the throughput of AutoBahn (6.7 GiB/s) is 13.9× and 1.36× better than that of ext4 with fsync (0.48 GiB/s) and ext4 without fsync (4.9 GiB/s), respectively. Under the skewed workload with 4 threads, AutoBahn (42.65 GiB/s) performs 3.19× better than ext4 without fsync (13.35 GiB/s) owing to the caching hot spot data on NVRAM described in Sect. 5.3.1.

In some cases, AutoBahn shows lower performance than ext4 (w/o fsync) with the random workload, due to the small size of NVRAM compared to the buffer cache size of the `ext4` file system. However, AutoBahn shows substantial improvements in synchronous write performance, and it can utilize the maximum bandwidth of the high-performance storage devices.

6.2.2 CDF for I/O latency

We also measured the latency of write operations in the append-only workload. Figure 14 shows the latency distributions of AutoBahn and ext4 (w/ and w/o fsync). The average latency of AutoBahn with 1 thread is 4.56 µs,



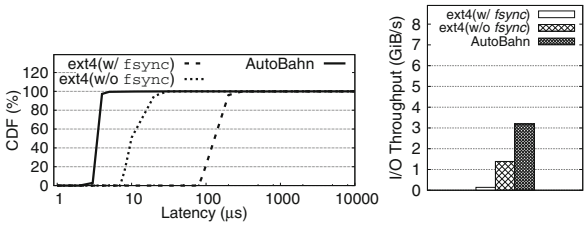
(a) Append-only

(b) Random (uniform)

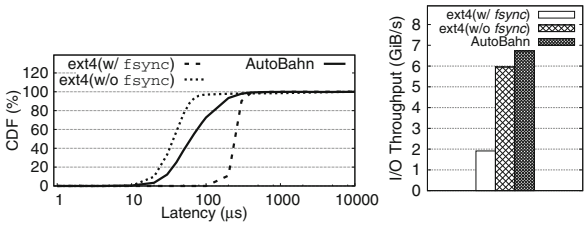
(c) Skewed (zipfian)

Fig. 13 Write performance with append, random, and skewed workloads. The total size of the private files is 196 GiB in the random and skewed workloads

whereas ext4 (w/ sync) and ext4 (w/o sync) show average latency of 126.21 µs and 11.36 µs, respectively. In this case, most I/O requests ($\sim$ 99.9 %) were processed within

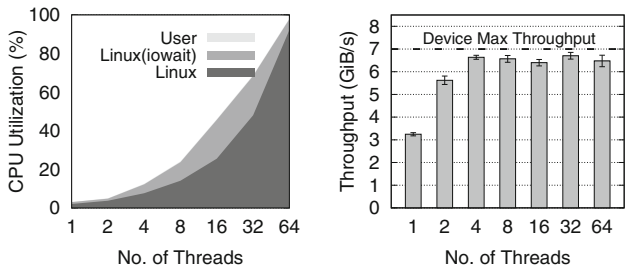


(a) 1 writer thread

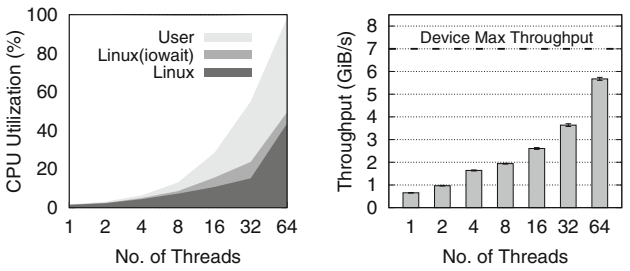


(b) 32 writer threads

Fig. 14 The CDF for the write latency and the throughput of the microbenchmark on an append-only workload



(a) Write unit: 4 KiB



(b) Write unit: 256 B

Fig. 15 The breakdown of CPU utilization and the write throughput with the append-only workload

20 µs in AutoBahn, compared to 30 µs in ext4 (w/o sync). The average latency in 32 threads is reduced by 61% and 7% in AutoBahn (100.15 µs), compared to ext4 with fsync (253.07µs) and ext4 without fsync (106.56 µs), respectively. The throughput of AutoBahn with 32 threads (6.73 GiB/s) is 3.52× and 1.12× better than ext4 with fsync (1.91 GiB/s) and ext4 without fsync (5.96 GiB/s).

6.2.3 The breakdown of CPU utilization

Next, we analyze the CPU usage of AutoBahn with the append-only workload, which is similar to experiments in Sect. 3. From Fig. 15, we see that the CPU usage portion for atomic instructions (e.g., atomic fetch-and-increment) is substantially reduced even when the I/O unit is 256 bytes. This leads the performance of file I/O to private files to have better scalability, compared to the results shown in Fig. 1. When the number of threads is 64 and the I/O unit is 256 bytes, the performance of AUTOBAHN peaks at 5.67 GiB/s while ensuring the write durability, thus utilizing the storage bandwidth 1.6× better than the ext4-based system without fsync and journaling (i.e., 3.51 GiB/s). The main benefit comes from the durable write buffer of Autobahn in alleviating excessive hardware-based synchronization on the shared `buffer_head` in the ext4 file system.

7 Conclusion

It is not uncommon to see that a cloud-based or in-house server infrastructure is built on a cluster of computing platforms with dozens or even hundreds of cores on a single chip. This trend of building high-end computing platforms poses significant challenges, especially in scaling the performance of cluster file systems. In this article, we address the scalability issue of file systems on high-end servers and propose AutoBahn designed to enhance the performance of concurrent write operations with durability being strictly ensured, by using a small amount of NVRAM as durable buffer cache. Prior proposals on the design of file systems using NVRAM has overlooked latent performance issues often arising from hardware-based synchronization bottlenecks occurring inside a shared buffer. We found that this can substantially degrade the performance and scalability of modern cluster-based infrastructures even when multiple threads attempt to write data even to their own private files. Autobahn is based on lock-free queue data structures in order to deal with a burst of durable write requests with an efficient three-stage pipelining technique. We implemented Autobahn as a Linux kernel module, and evaluated its performance under macro- and micro-benchmarks running against database engines. Evaluation results demonstrated that Autobahn can improve the transaction throughput of existing key-value storage systems by up to 19.9×, showing its practicality and effectiveness.

Funding The Funding was provided by National Research Foundation of Korea (KR) (Grant Nos. 2015M3C4A7065646, 2017R1A2B4006134).

References

1. Agigaram Non-volatile System. http://www.agigatech.com/agigaram.php
2. Bhat, S.S., Eqbal, R., Clements, A.T., Kaashoek, M.F., Zeldovich, N.: Scaling a file system to many cores using an operation log. In: Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17, pp. 69–86. ACM, New York (2017). https://doi.org/10.1145/3132747.3132779
3. Congiu, G., Narasimhamurthy, S., Süß, T., Brinkmann, A.: Improving collective i/o performance using non-volatile memory devices. In: 2016 IEEE International Conference on Cluster Computing (CLUSTER), pp. 120–129 (2016). https://doi.org/10.1109/CLUSTER.2016.37
4. David, T., Guerraoui, R., Trigonakis, V.: Everything you always wanted to know about synchronization but were afraid to ask. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13, pp. 33–48. ACM, New York (2013). https://doi.org/10.1145/2517349.2522714
5. DeWitt, D.J., Katz, R.H., Olken, F., Shapiro, L.D., Stonebraker, M.R., Wood, D.A.: Implementation techniques for main memory database systems. In: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, SIGMOD '84, pp. 1–8. ACM, New York (1984). https://doi.org/10.1145/602259.602261
6. Doh, I.H., Lee, H.J., Moon, Y.J., Kim, E., Choi, J., Lee, D., Noh, S.H.: Impact of NVRAM write cache for file system metadata on I/O performance in embedded systems. In: Proceedings of the 2009 ACM Symposium on Applied Computing, SAC '09, pp. 1658–1663. ACM, New York (2009)
7. Gill, B.S., Modha, D.S.: WOW: wise ordering for writes—combining spatial and temporal locality in non-volatile caches. In: Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies, FAST'05, pp. 10–10 (2005)
8. Intel Corporation: Intel® 64 and IA-32 architectures software developer's manual. www.intel.com/Assets/en_US/PDF/manual/253666.pdf
9. Intel-power loss protection. https://newsroom.intel.com/wp-content/uploads/sites/11/2016/01/Intel_SSD_320_Series_Enhance_Power_Loss_Technology_Brief.pdf
10. Jang, H., Rhee, S.Y., Kim, J.E., Kang, S., Han, H., Jung, H.: AutoBahn: accelerating concurrent, durable file I/O via a non-volatile buffer. In: Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER) (2017)
11. Kang, S., Park, S., Jung, H., Shim, H., Cha, J.: Performance trade-offs in using NVRAM Write buffer for flash memory-based storage devices. IEEE Trans. Comput. **58**(6), 744–758 (2009)
12. Kang, J., Zhang, B., Wo, T., Yu, W., Du, L., Ma, S., Huai, J.: Spanfs: A scalable file system on fast storage devices. In: Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC '15, pp. 249–261. USENIX Association, Berkeley (2015). http://dl.acm.org/citation.cfm?id=2813767.2813786
13. Lee, E., Bahn, H., Noh, S.H.: Unioning of the buffer cache and journaling layers with non-volatile memory. In: Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13), pp. 73–80. USENIX, San Jose (2013). https://www.usenix.org/conference/fast13/technical-sessions/presentation/lee
14. Lee, E., Kang, H., Bahn, H., Shin, K.G.: Eliminating periodic flush overhead of file I/O with non-volatile buffer cache. IEEE Trans. Comput. **65**(4), 1145–1157 (2016)
15. Liu, Z., Wang, B., Yu, W.: HALO: a fast and durable disk write cache using phase change memory. Clust. Comput. **21**, 1275–1287 (2017)
16. Min, C., Kashyap, S., Maass, S., Kim, T.: Understanding many-core scalability of file systems. In: 2016 USENIX Annual Technical Conference (USENIX ATC 16), pp. 71–85. USENIX Association, Denver (2016). https://www.usenix.org/conference/atc16/technical-sessions/presentation/min
17. Nightingale, E.B., Veeraraghavan, K., Chen, P.M., Flinn, J.: Rethink the sync. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI '06, pp. 1–14. USENIX Association, Berkeley (2006). http://dl.acm.org/citation.cfm?id=1298455.1298457
18. Pnevmatikatos, D., Markatos, E.P., Magklis, G., Ioannidis, S.: On using network RAM as a non-volatile buffer. Clust. Comput. **2**(4), 295–303 (1999)
19. Redis. https://redis.io/
20. Rocksdb. http://rocksdb.org/
21. Ryu, Y.: Performance evaluation of page migration scheme for NVRAM-based wireless sensor nodes. Int. J. Distrib. Sens. Netw. **9**(11), 278132 (2013)
22. Salomie, T.I., Subasu, I.E., Giceva, J., Alonso, G.: Database engines on multicores, why parallelize when you can distribute? In: Proceedings of the Sixth European Conference on Computer Systems, EuroSys '11 (2011)
23. Samsung-power loss protection. http://www.samsung.com/semiconductor/minisite/ssd/downloads/document/Samsung_SSD_845DC_05_Power_loss_protection_PLP.pdf
24. Wang, T., Oral, S., Pritchard, M., Wang, B., Yu, W.: Trio: Burst buffer based i/o orchestration. In: 2015 IEEE International Conference on Cluster Computing, pp. 194–203 (2015). https://doi.org/10.1109/CLUSTER.2015.38
25. Wiredtiger. http://www.wiredtiger.com/
26. Zheng, M., Tucek, J., Qin, F., Lillibridge, M.: Understanding the robustness of ssds under power fault. In: Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13), pp. 271–284. USENIX, San Jose (2013). https://www.usenix.org/conference/fast13/technical-sessions/presentation/zheng

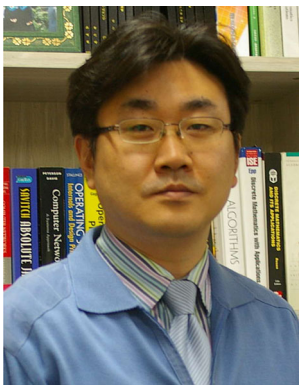
Hyeongwon Jang received his B.S. in Computer Science and Engineering from Hanyang University, Seoul, South Korea, in 2017. Currently, he is a M.S. student at Hanyang University. His research interests are file system designs for fast NVMe storage devices and high-performance database systems.

Sang Youp Rhee received his B.S. in Computer Science from Purdue University, West Lafayette, IN, USA, in 2015. Currently, he is a M.S. candidate at Hanyang University. His research interests are file system designs for highly concurrent systems.

Jaeeun Kim received his B.S. in Computer Science and Engineering from Hanyang University, Seoul, South Korea, in 2017. Currently, he is in the Master's course at Hanyang University. His research interests are distributed systems and high-performance database systems.

Yoonhee Kim is a professor in the Department of Computer Science, Sookmyung Women's University. She received her B.S. degree from Sookmyung Women's University in 1991, her M.S. degree and Ph.D. from Syracuse University in 1996 and 2001, respectively. She was a Research Staff Member at the Electronics and Telecommunication Research Institute during 1991 and 1994. Before joining the faculty of Sookmyung Women's University in 2001, she was on the faculty of the Computer Engineering Department at Rochester Institute of Technology in NY, USA. Her research interests span many aspects of runtime support and management in distributed computing systems.

Hyuck Han received his B.S., M.S., and Ph.D. degrees in Computer Science and Engineering from Seoul National University, Seoul, Korea, in 2003, 2006, and 2011, respectively. Currently, he is an assistant professor in the Department of Computer Science, Dongduk Women's University. His research interests are operating systems, database systems, and distributed systems.



Sooyong Kang is a professor in the Department of Computer Science, Hanyang University, Seoul, Korea. He received his B.S. degree in mathematics and the M.S. and Ph.D. degrees in Computer Science, from Seoul National University (SNU), Seoul, Korea, in 1996, 1998, and 2002, respectively. He was then a Postdoctoral Researcher in the School of Computer Science and Engineering, SNU. From March 2003, he joined the faculty of Hanyang University. His research interests include storage system, distributed computing system, mobile cloud computing and big data processing.

Hyungsoo Jung received the B.S. degree in mechanical engineering from Korea University, Seoul, in 2002, and the M.S. and Ph.D. degrees in computer science from Seoul National University, Korea in 2004 and 2009, respectively. From 2010 to 2012, he was with the University of Sydney, Sydney, Australia, as a postdoctoral research associate. From April to September 2012, he was a researcher in NICTA. From October 2012 to August 2015, he worked for Amazon Web Services as a (senior) Software Development Engineer. From September 2015, he joined Hanyang University as an assistant professor. His research interests are in the areas of distributed systems, database systems, and transaction processing.