Scalable Write Allocation in the WAFL File System

Matthew Curtis-Maury, Ram Kesavan, and Mrinal K. Bhattacharjee
NetApp, Inc

***Abstract*—**

Enterprise storage systems must scale to increasing core counts to meet stringent performance requirements. Both the NetApp® Data ONTAP® storage operating system and its WAFL® file system have been incrementally parallelized over the years, but some components remain single-threaded. The WAFL write allocator, which is responsible for assigning blocks on persistent storage to dirty data in a way that maximizes write throughput to the storage media, is single-threaded and has become a major scalability bottleneck. This paper presents a new write allocation architecture, *White Alligator*, for the WAFL file system that scales performance on many cores. We also place the new architecture in the context of the historical parallelization of WAFL and discuss the architectural decisions that have facilitated this parallelism. The resulting system demonstrates increased scalability that results in throughput gains of up to 274% on a many-core storage system.

I. Introduction

Enterprise storage systems are expected to achieve high performance, and the NetApp Data ONTAP storage operating system must constantly deliver performance gains on cutting-edge platforms to remain competitive. Such systems employ an ever-increasing number of cores, and large-scale parallelization is required to translate the increased computational potential into higher throughput and lower latency. Data ONTAP and its Write Anywhere File Layout (WAFL) file system [1] were initially single-threaded, but they have been incrementally parallelized over time to deliver scalable performance on each new platform.

A key component of any file system is its *write allocator*, which is responsible for determining where to write data on persistent storage to maximize write throughput and subsequent read performance. Over the years, WAFL processing was parallelized [2], but write allocation remained single-threaded. Because WAFL always writes data to new locations on disk, the pressure placed on the write allocator is acute, as we show in Section V. Conventional wisdom in the literature is that file systems and write allocation are not scalable and are extremely difficult to parallelize [3], [4], [5], [6], [7]. Therefore, scalable designs typically involve partitioning the file space and replicating the storage stack for each partition [5], [6], [7].

In this paper, we describe a scalable write allocator for WAFL, called *White Alligator*, that was specifically designed for the many-core era. The key to the new architecture is a division between: (1) the *infrastructure* that deals with the bookkeeping required to track free space and maximizing write throughput to the underlying storage media and (2) a set of *cleaner threads* that assign persistent storage locations to data. We present the novel approaches used to parallelize each of these components to eliminate the write allocation bottleneck. White Alligator improves throughput on a many-core platform by as much as 274%, with the potential to scale even higher on future platforms. The presented architecture has been shipping since 2011 and has been running extremely reliably on more than a quarter of a million deployed systems with different core counts and many kinds of workloads.

We demonstrate that with a scalable architecture in place, a traditional file system design can run efficiently on many cores. We make the following contributions:

- We provide an inside view of the internals of the WAFL file system and the details of WAFL write-allocation infrastructure and mechanics that are relevant to its parallelization.
- We present a collection of techniques that have been successfully deployed to parallelize the WAFL write allocator for high performance in a many-core system.
- We evaluate the performance of the parallelized infrastructure on real many-core storage systems.

Section II presents background material on WAFL and write allocation. Section III provides a brief history of parallelism in WAFL. Section IV presents the new parallel White Alligator architecture and its advantages. Section V evaluates White Alligator performance and Section VI surveys the related work.

II. Background

A. Write Allocation in Storage Systems

Storage space on devices is typically exposed as an addressable number space referencing fixed size blocks. A file system's data is laid out in this space along

with metadata to track the location of data. Metadata is frequently accessed to retrieve and operate on the user data, and high-performance storage systems use advanced techniques to optimize data and metadata accesses. *Write allocation* is the process by which a storage system chooses where on persistent media to write data and metadata, as well as the underlying mechanisms to perform the write. The write allocator must maximize the efficiency of both writing to disk and the performance of subsequent reads.

B. Background on the WAFL File System

WAFL is the file system within the Data ONTAP operating system. WAFL houses and exports multiple file systems called NetApp FlexVol® volumes from within a shared pool of storage called an *aggregate*. An aggregate is made up of a set of RAID groups, each of which is configured with one or more parity drives. In WAFL, all metadata and user data (including logical units exported to SAN clients) are stored in files, called metafiles and user files, respectively. Metadata to track free blocks and other information critical to allocating and freeing blocks is stored in *allocation metafiles*. Each file is represented by an *inode* that tracks its attributes and the location of its data blocks on persistent storage. A block of a file is represented in memory by a *buffer*. The file system is a tree of blocks rooted at the superblock.

A block in a WAFL file system is addressed by its *volume block number*, or *VBN*. WAFL uses physical VBNs to refer to blocks in the aggregate, which are mapped to a location on persistent media. A block in a FlexVol volume has both a VBN to specify the physical location of the block and a *Virtual VBN* to specify the block's offset within the volume. Previous work has provided detailed descriptions of WAFL [1], [8].

C. Write Allocation in WAFL

In file systems that write data in place, data must be assigned a block only once. However, copy-on-write (COW) file systems—such as WAFL, LFS, ZFS, and Btrfs—must allocate blocks for *every* incoming write. That is, even overwritten data results in write allocation because all data is written to a new location, as in LFS [9].

WAFL accumulates and flushes thousands of operations worth of data to persistent storage, which allows better layout decisions and amortizes the associated overhead. Instead of delaying the client reply until the data reaches persistent storage as part of the next batch, operations that update file system state are logged in nonvolatile RAM, which allows the system to reply to client writes very quickly. Writing a consistent collection of changes as a single transaction in WAFL is known as a *consistency point*, or simply *CP* [1], [8]. Each CP is a self-consistent point-in-time image of the file system that includes updates since the previous CP. Therefore, in-memory data that is to be included in a CP is atomically identified at the start of the CP and isolated from further modifications until the data reaches persistent storage. In order to allow client traffic to continue to make updates during a CP, copies of data can be made on demand. That is, any attempts to change an inode's properties or a buffer's contents during a CP result in the object being COW'd in memory, with a read-only copy being written to persistent storage and a writeable version available to accept modifications. In such cases, changes made after the start of a CP are persisted during the subsequent CP. Any metafile updates made on behalf of a CP must reach persistent storage as part of that same CP.

Writing to a file "dirties" the in-memory inode associated with the file and adds it to a list of dirty inodes to process in the next consistency point. Since the WAFL file system stores all metadata as files, the primary function of a CP is to flush changed state—i.e., all dirty buffers—from each dirty inode to persistent storage, which is known as *inode cleaning*. Each dirty buffer is cleaned by allocating a free block, writing the buffer to this chosen location, and freeing the previously used block. Once all dirty inodes for files and metafiles have been cleaned, the newly written data is atomically persisted by overwriting the superblock in place [1], [8]. If the system crashes before the superblock is written, the file system state from the most recently completed CP is loaded and all subsequent operations are replayed from the log stored in nonvolatile RAM.

III. History of Parallelism in WAFL

To understand the parallelization of write allocation work in WAFL, it is first necessary to put in place the foundation of the parallelization of WAFL file system processing and the parallelization of the Data ONTAP operating system in which it resides.

A. Early Parallelization of Data ONTAP

Data ONTAP was first parallelized by dividing each subsystem in the operating system into a private *domain*, where only a single thread from a given domain could execute at a time. Domains were defined such that sharing of global data between different domains was rare. For example, domains were created for RAID, networking, storage, file system (WAFL), and the protocols, and communication between domains used message passing. This approach allowed scaling to multiple cores with minimal code rewrite, because little explicit locking

was required. This model provided sufficient performance because systems at the time had very few cores (e.g., four), so parallelism within the file system was not required. Over time, domains were incrementally parallelized as they became scalability bottlenecks and additional synchronization was introduced.

B. Classical Waffinity

As core counts increased to eight, serialized execution of file system operations became a scalability bottleneck. To parallelize the WAFL domain, we implemented a multiprocessor model called *Waffinity*, the first version of which was called *Classical Waffinity* and shipped with Data ONTAP 7.2 in 2006. In Classical Waffinity, the file system message scheduler defined message execution contexts called *affinities*. User files were partitioned into *file stripes* that corresponded to a contiguous range of blocks in the file, and these were rotated over a set of *Stripe affinities*. This model ensured that messages operating in different Stripe affinities were guaranteed to be working on different partitions of user files, so they could be safely executed in parallel by Waffinity threads executing on different cores. Any message operating outside of a single file stripe ran in a *Serial affinity* that excluded all Stripe affinities and serialized all WAFL processing. This data partitioning provided an implicit coarse-grained synchronization that eliminated the need for explicit locking on partitioned objects, thereby greatly reducing the complexity of the programming model. Some minimal locking was required for data structures shared between Stripe affinities. The benefit of this model came from the fact that most performance-critical messages at the time, such as user file reads and writes, could be executed in Stripe affinities.

C. Parallelized Inode Cleaning

After the WAFL parallelization just described, inode cleaning ran in the Serial affinity. That is, the process of assigning VBNs to dirty buffers and writing the data out to persistent storage prevented the execution of client operations such as user file reads and writes. The next parallelization of WAFL was the introduction of a single *inode cleaner thread* that ran in parallel with Waffinity threads and cleaned inodes. This work shipped in Data ONTAP 7.3 in 2008 on platforms with 8-12 cores.

For each dirty buffer in a dirty inode, the cleaner thread identified a single VBN that satisfied the persistent storage layout objectives by reading the allocation bitmaps. The thread then updated user file metadata to reflect the new location of its data; user data and metadata buffers are COW'd so the cleaner thread modifies the current CP's copy of user metadata while client operations modified the other copy. Finally, the cleaner thread made two updates to allocation bitmaps: (1) to reflect the VBN that was allocated and (2) to free the previously used VBN, since an overwrite in WAFL frees the old block. For example, WAFL maintains a metafile containing one bit for each block in the file system to track whether the corresponding block is used or free. Thus, allocations and frees of VBNs toggle bits in this metafile and these updates were made by the cleaner thread. A variety of other metafiles exist that must also be updated. The system tracked periods during which the cleaner thread could run and disallowed access to allocation metafiles and user file metadata from Waffinity during these times, giving the cleaner thread exclusive access to everything it needed. This model allowed us to quickly parallelize inode cleaning and achieve the scalability benefits it provided, but it left us with a complex set of MP safety rules. For example, any operation that updated metafiles would have to run on the cleaner thread *during certain phases* if it could not delay its execution, even if it was unrelated to file cleaning. Such complexity continued to burden us until it was resolved in the White Alligator architecture.

Before the introduction of the cleaner thread, cleaning work running in the Serial affinity could directly update a variety of global counters that were systemwide, per aggregate, per FlexVol volume, or per file. As a simple example, inode cleaning must decrement/increment the number of free blocks in an aggregate. Once cleaning was moved in parallel with WAFL processing, synchronization was required around counter updates to prevent races. Counters could be updated frequently within the cleaner thread, and this led to excessive overhead. Thus, cleaner threads were extended to use *loose accounting*, wherein counter updates were staged in a local *token* that was later applied to the global counters in a batched fashion from within Waffinity. Loose accounting allowed the counters' values to deviate from their instantaneous logical values, and all counter accesses had to be audited and corrected to deal with temporary discrepancies.

D. Hierarchical Waffinity

Classical Waffinity was replaced by a new *Hierarchical Waffinity* model to enable increased levels of parallelism within WAFL. This model, which first shipped with Data ONTAP 8.1 in 2011 on platforms with 16-20 cores, provided a way to parallelize all types of work, beyond the dozen operations parallelized by Classical Waffinity. It did so by coordinating accesses to file system objects beyond data blocks of user files.

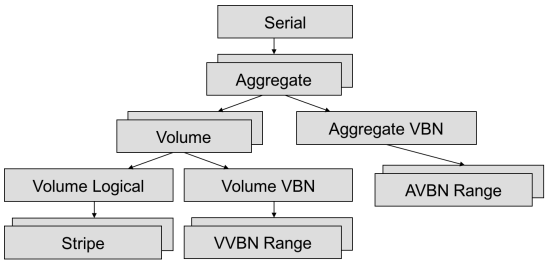

Fig. 1: The Hierarchical Waffinity hierarchy.

The new model built a hierarchy of affinities where each affinity was associated with certain permissions, such as access to metadata files, that serialized execution in the file system in Classical Waffinity (Figure 1). The scheduler enforced execution exclusivity between a given affinity and its children, so it only restricted the execution of an affinity's parents and children in the hierarchy; all other affinities could safely run in parallel. For example, if the Volume Logical affinity was running, then its Stripe affinities were excluded along with its parent Volume, Aggregate, and Serial affinities. Other affinities, such as Volume VBN, were allowed to run. This design ensured that no two messages with conflicting data accesses ran concurrently, because they ran in affinities that excluded one another. The Aggregate, Volume, Stripe, and Range affinity types had multiple instances under a single parent affinity, which allowed parallel execution of messages operating on disjoint data, such as any two operations in different aggregates, FlexVol volumes, or regions of blocks in a file. Additional details on Waffinity and the data managed by each affinity are available in earlier work [2]. Section IV-B2 discusses how write allocation was parallelized by leveraging this architecture.

In spite of the efforts to parallelize WAFL, single-threaded write allocation had become a major scalability bottleneck on many-core systems, as described in detail in Section V. To allow Data ONTAP to scale performance to larger core counts, it was necessary to parallelize this critical component. It would have been possible to move inode cleaning into Hierarchical Waffinity. However, that would have introduced extra complexity, because WAFL write allocation involves assigning both a VBN and a Virtual VBN and updating file system metadata within both the aggregate and FlexVol volume, and therefore does not fit neatly into any single affinity. Further, keeping inode cleaning in parallel with the rest of WAFL maximizes cleaning parallelism because it bypasses the hierarchical exclusion in Waffinity that limits when affinities can run; instead, a cleaner thread can run at any time.

IV. The White Alligator Architecture

In this section, we present the White Alligator architecture that allows significantly increased parallelization of inode cleaning. This model shipped alongside Hierarchical Waffinity in Data ONTAP 8.1 in 2011. White Alligator achieves increased parallelism in two primary ways: (1) it introduces multiple inode cleaner threads than can execute concurrently and (2) it allows the parallelization of metadata updates by migrating such accesses to the Hierarchical Waffinity model.

To achieve these new levels of parallelism, White Alligator defines a new write allocator *infrastructure* component that builds collections of unused VBNs called *buckets* and defines a set of operations on them to allow cleaner threads to use the VBNs. The infrastructure runs as messages in Waffinity to read and update metafiles, thereby allowing that sophisticated multiprocessor architecture to manage concurrent metadata accesses just as it does other data and facilitating parallelism in metafile accesses. A set of cleaner threads now use a well-defined API exported by the infrastructure to obtain and allocate unused VBNs by operating on a limited set of lock-protected in-memory structures, and they do not directly perform any metafile accesses. By greatly simplifying the work done by cleaner threads and providing an MP-safe API, the White Alligator architecture makes it possible for multiple cleaner threads to operate concurrently on different inodes or different regions of a single inode. Using buckets of VBNs amortizes the overhead associated with each VBN allocation.

A. The White Alligator API

As mentioned earlier, the White Alligator architecture creates a well-defined API that is provided by the infrastructure and is consumed by a set of parallel cleaner threads. Such a separation in WAFL write allocation allows the infrastructure and cleaner threads to be independently parallelizable, and therefore to scale with newer platforms that have ever-more processors. Further, this architecture brings important software development advantages in that the division of code across a well-defined API makes the code more maintainable and allows the data structures to have simple MP-safety rules. Here we provide an overview of the API; details of each component are discussed in the following sections.

Figure 2 presents the separation of the infrastructure and the cleaner threads across an API composed of GET, USE, and PUT operations that execute in the context of cleaner threads. The infrastructure processes allocation metafiles to find available VBNs that meet the write allocator's objectives and uses them to construct a set of

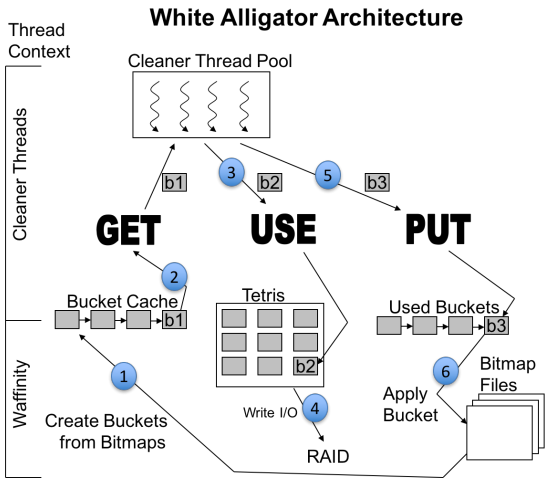

Fig. 2: The separation of cleaner threads and the infrastructure via the White Alligator API.

buckets. These buckets are then enqueued in step 1 to a lock-protected list of available buckets called the *bucket cache* that is filled by the infrastructure and consumed by the cleaner threads. Each cleaner thread begins by assembling a list of dirty buffers from its assigned inode. The cleaner thread then uses the GET operation in step 2 to acquire a bucket of VBNs from the bucket cache for write allocating the dirty buffers to persistent storage. Once the cleaner thread has acquired a bucket of VBNs, it iterates over its list of dirty buffers in step 3 and the USE operation assigns one VBN from the bucket to each buffer in the list and marks the VBN as consumed in the bucket metadata. This step also enqueues each buffer to a per-RAID group data structure called a *tetris* (described in detail in Section IV-E) that accumulates buffers to be written to persistent storage. The cleaner thread continues to insert buffers into the tetris until a sufficient number have been added, at which time the entire tetris of buffers is sent to persistent storage in step 4. Once the cleaner thread has either consumed all free VBNs in a bucket or run out of dirty buffers to clean, it returns the bucket to the infrastructure. To do so, the cleaner thread uses the PUT operation in step 5 to enqueue the bucket onto the *used bucket queue*, a lock-protected staging queue that stores buckets while they await processing by the infrastructure. Finally, in step 6, the infrastructure removes buckets from the used bucket queue and updates the file system allocation metafiles to reflect the VBNs that have been used by a cleaner thread. After that, the bucket begins the cycle again and the bucket is refilled with VBNs.

A similar, though simpler, process (not shown in the figure) occurs for overwritten blocks whose VBNs must be freed in the file system. The cleaner thread stores the freed VBNs to a structure called a *stage*, which is analogous to a bucket. When a stage is full, the cleaner thread sends a message to the infrastructure to commit those frees to the metafiles. Details on WAFL free space reclamation are available in other work [10].

B. Parallelism in White Alligator

1) Parallel Cleaner Threads: The White Alligator architecture allows cleaner threads to use a simple API to interact with the infrastructure and assign VBNs to dirty buffers rather than accessing and updating metafiles directly. As a result, the operations performed in cleaner threads are greatly simplified, which allows multiple cleaner threads to run concurrently on different inodes or even on different regions of a single inode. Synchronization is required only on the bucket cache, the tetris data structures, and the used bucket list, and the use of buckets of VBNs amortizes this overhead. The remaining updates performed by the cleaner thread happen to its local bucket and to metadata that tracks the on-disk location of a file's data.

The previous architecture for inode cleaning could theoretically have been parallelized, but this would have involved adding a considerable amount of synchronization. In particular, races between multiple cleaner threads would need to be prevented when updating either metafiles or write allocation global state while selecting a new VBN for each dirty buffer. White Alligator instead minimizes the required synchronization, which is mandatory in building a scalable system.

2) Infrastructure Parallelism: To support the bucket-based API in Section IV-A, the infrastructure must: (1) read allocation bitmap files to find free VBNs with which to fill buckets and (2) write to allocation bitmap files to reflect VBN allocations and frees performed by cleaner threads. White Alligator moves all metafile processing out of the cleaner threads into the infrastructure running in Waffinity. Beyond simplifying the cleaner thread work and facilitating parallelization of inode cleaning, Waffinity-based management allows scalable parallelization of metafile operations. As an additional benefit, always accessing metafiles in Waffinity greatly simplifies the MP safety rules around metafile accesses, which carries long-term advantages in maintainability.

White Alligator provides data partitioned access to metafiles within Hierarchical Waffinity by including Volume VBN and Aggregate VBN affinities to coordinate concurrent accesses to metadata (so named because their files are typically indexed by VBN). Parallelism is facilitated in four ways. First, allocation bitmaps in each aggregate (representing VBNs) and FlexVol volume (representing Virtual VBNs) map to different Aggregate

VBN and Volume VBN affinities, respectively. Thus, accesses to metafiles in different aggregates and volumes are parallelized in Waffinity because threads running in parallel on different cores can read and write to metafiles without explicit synchronization. In contrast, any two messages operating on the same metafile blocks run in affinities that serialize each other, and the Waffinity scheduler prevents the messages from executing in parallel. Second, Waffinity provides a set of Range affinities under each Volume VBN and Aggregate VBN affinity in order to allow parallel accesses to different blocks in metafiles of a single volume or aggregate. Each Range affinity provides access to a range of blocks within each metafile. The most expensive infrastructure operations were optimized to run in these Range affinities to allow operations within a single volume or aggregate to run in parallel. Third, client-facing data is mapped to the Volume Logical affinity, which can run in parallel with write allocation work in the Volume VBN affinity (see Figure 1). This allows parallelism between client-facing operations and write allocation infrastructure tasks within a single volume. Fourth, write allocation work in Waffinity runs in parallel with cleaner threads performing inode cleaning.

C. Bucket Details

Buckets are the basic units of allocation that allow the design of a simple MP-safe model for the cleaner threads. A bucket is simply a set of contiguous VBNs on each drive that is defined by a starting VBN and a length, with additional metadata to track which VBNs have already been used. The number of VBNs in a bucket is determined by the *chunk* size, which is the number of consecutive blocks to which the system write allocates file blocks to maximize sequential read performance. It is typically a multiple of 64 blocks.

It is possible to allocate VBNs one at a time by using the White Alligator API (i.e., a bucket size of one). However, using buckets with a chunk of VBNs has several concrete advantages. First, buckets of VBNs amortize the overhead of finding free VBNs in the infrastructure. Second, buckets amortize the cleaner thread synchronization that is required at the bucket granularity, such as when GETing a new bucket, and this reduces contention as a result. Third, using buckets ensures that cleaner threads acquire a set of contiguous VBNs, which is not possible when allocating one at a time in a multithreaded environment. Allocating file blocks contiguously on persistent storage improves performance when the blocks are subsequently read sequentially.

D. Filling Buckets with VBNs

In providing VBNs for use by cleaner threads, the infrastructure must meet three primary objectives: (1) minimize reads required for RAID parity computation; (2) place consecutive file blocks contiguously on a single drive to improve future sequential read performance; and (3) maximize the available write throughput of the aggregate. As discussed next, White Alligator satisfies these goals by guaranteeing equal progress down each drive in an aggregate and focusing allocations to the emptiest regions of storage.

A *stripe* is a set of blocks belonging to the data drives of a RAID group, one per drive, which share the same parity block on the parity drive. An *Allocation Area* (AA) is a contiguous set of stripes. The infrastructure selects the Allocation Area in each RAID group that contains the most free blocks and walks the allocation bitmaps to find free VBNs on each drive from the corresponding regions. Each data drive in an aggregate contributes one bucket that is filled starting from the top of the AA. By using AAs to find empty regions of disk, WAFL increases the probability of full stripe writes that do not require RAID parity reads (objective 1) and also make it possible to allocate consecutive blocks of a file to consecutive blocks (objective 2).

Once a cleaner thread consumes all VBNs within a bucket, it is placed on the *used buckets* queue using the PUT operation. Returned buckets are then refilled with the next chunk-worth of VBNs from the corresponding drive as selected by the infrastructure. White Alligator maintains a lock-protected set of buckets called a *bucket cache* and keeps this list non-empty to ensure that the GET operation does not block. Only after the buckets from all drives in an aggregate have been used and refilled with VBNs are they collectively put back into the bucket cache for allocation by cleaner threads. This synchronized insertion process ensures equal progress on each drive and exploits the bandwidth provided by each drive (objective 3). When all VBNs from an AA have been used, a new AA is selected from the same RAID group. Figure 3 shows a simplified example aggregate with two RAID groups and five data drives. A version of this infrastructure is reused to write allocate Virtual VBNs within FlexVol volumes.

E. Tetris Processing

A *tetris* is the unit of write I/O in WAFL. Logically, it is a collection of blocks whose width is equal to the number of drives in the RAID group and whose depth is the desired write I/O size per drive. In other words, it is a contiguous collection of stripes. White Alligator maintains a corresponding in-core tetris data structure

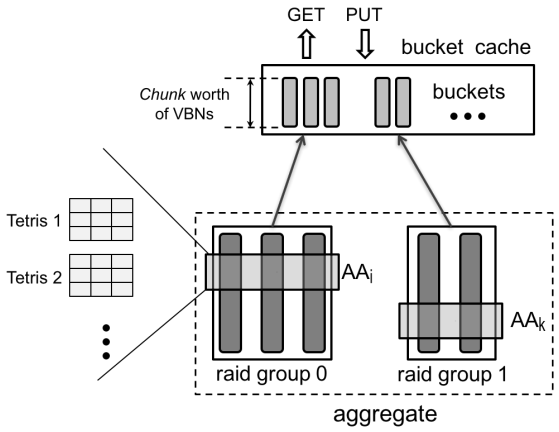

Fig. 3: An Allocation Area (AA) from each RAID group is divided up to refill per-drive buckets. The buckets from all RAID groups of an aggregate are placed in the bucket cache. The tetris structure gets filled with blocks as the buckets are used by the write allocator.

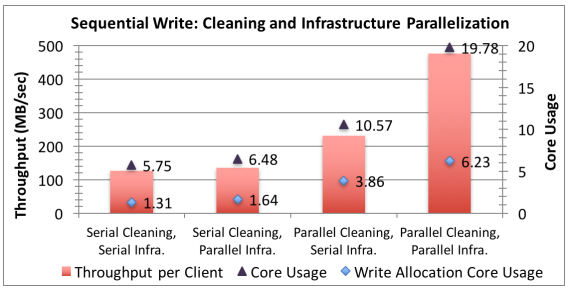

Fig. 4: Throughput per client and core usage (write allocation work and total) with each combination of parallel cleaner threads and infrastructure being enabled and disabled.

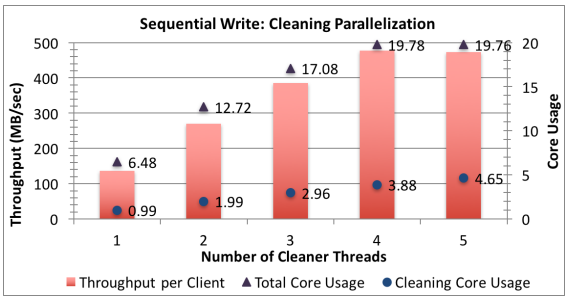

Fig. 5: Throughput per client and core usage (cleaner threads and total) as the number of cleaner threads is increased.

to track when the I/O is ready to be sent, i.e., when all free VBNs from the tetris have been assigned to buffers. As discussed in Section IV-A, each buffer that is write allocated by a cleaner thread is enqueued to the tetris by the USE operation. The tetris structure tracks lists of recently cleaned buffers on a per-drive basis. Locking is not required when enqueuing buffers to the tetris because the cleaner thread that owns a bucket has exclusive access to the corresponding drive in the current tetris at that instant. Each tetris also maintains a reference count of its outstanding buckets that is atomically decremented by the USE operation. When this reference count drops to zero, an I/O is constructed and sent to RAID for writing to persistent storage.

V. Performance Analysis

This section presents a quantitative analysis of the White Alligator architecture to show how it scales in various dimensions. The benefits of the parallelized write allocation are evaluated using benchmarks on storage servers running Data ONTAP version 9.2, which shipped in 2017. Although this release contains several improvements, the fundamental White Alligator design has remained the same.

A. Core Scalability

A major advantage of the new architecture is the independent parallelization of the infrastructure and the cleaner threads. To demonstrate the scalability achieved by White Alligator, we analyze the results of write-intensive workloads (sequential write and random write) from Fibre Channel clients on a current mid-range system, an Intel Ivy Bridge dual-socket platform with 10 cores per socket and all SSD drives. Write workloads produce the maximum load on the write allocation infrastructure to find new VBNs, since all writes require VBN allocations, and to free VBNs of overwritten blocks. In these experiments, we used an instrumented kernel with serialized cleaner threads and/or infrastructure to be able to isolate the impact of parallelization.

1) Sequential Write: Figure 4 presents the results of a sequential write workload with four permutations of cleaner thread and infrastructure parallelization enabled. The figure shows that parallelizing the infrastructure and cleaner threads each *individually* demonstrate 7% and 82% gains in I/Os per second throughput, respectively. Parallelizing both achieves 274% higher throughput and saturates all cores in the system, with 6.23 cores performing write allocation work. Clearly, sequential write is more limited by VBN assignment load in cleaner threads, but both components represent major bottlenecks. Write allocation core usage caps around six cores (i.e., 2.35 infrastructure + 3.88 cleaner threads) because Data ONTAP employs pipelined parallelization and cores are used in proportion to write allocation load. That is, using a total of six cores for write allocation on this system is the sweet spot, beyond which cores would be taken away from other important tasks.

Figure 5 presents the impact of increasing numbers

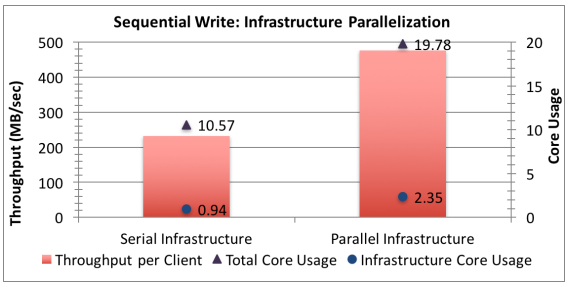

Fig. 6: Throughput per client and core usage (infrastructure and total) with and without infrastructure parallelization.

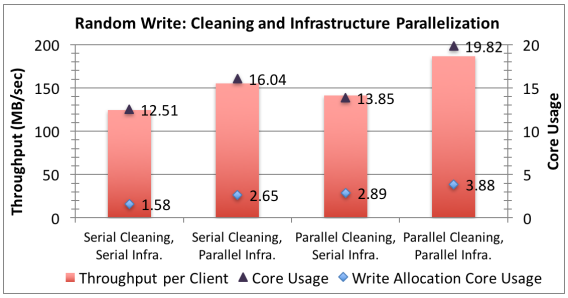

Fig. 7: Throughput per client and core usage (write allocation work and total) with each combination of parallel cleaner threads and infrastructure being enabled and disabled.

of cleaner threads in the presence of a parallelized infrastructure. In the sequential write workload, adding cleaner threads yields a nearly linear increase in system throughput up to the point when system CPUs are saturated and can absorb no additional work. Core usage within the infrastructure goes from 0.94 cores to 2.35 cores as a result of infrastructure parallelization, as shown in Figure 6. This boost in computational bandwidth signifies an increased ability to process allocation metafiles and translates into an increased capacity to handle client load, yielding a 106% rise in throughput.

2) Random Write: The experiment just described was repeated on the same platform with a random write workload, which places different pressure on the use of infrastructure and cleaner threads, as shown in Figure 7. The figure shows that parallelizing the infrastructure in this case carried the greater advantage with a 25% benefit in throughput, compared to a 14% improvement from parallelizing cleaner threads. This inverted result reveals that random write is more limited by the processing in the infrastructure. In particular, a random write workload results in block frees that are randomly distributed in the VBN space. Since allocation metafiles are indexed by VBN, this randomness causes a higher ratio of metafile block updates than does sequential write, in which the updates are concentrated within fewer metafile blocks. This places increased pressure on

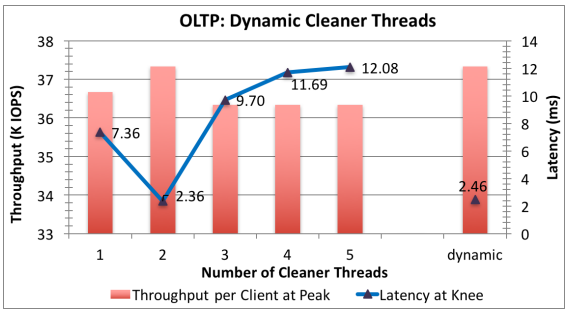

Fig. 8: Throughput per client at the peak load and latency at a lower load that represents the "knee" of the scalability curve.

the infrastructure to keep up with the load of block frees. In contrast, block frees during sequential write are more efficient and create more pressure on inode cleaning, as seen in the earlier results. Collectively, parallelization of both components yields a gain of 50%.

In both workloads, White Alligator was able to scale to many cores. In each case, all cores in the system were saturated, which indicates that CPU availability had become the bottleneck and suggests that White Alligator will be able to scale even further on future platforms with more cores.

B. Dynamic Number of Cleaner Threads

While workloads like sequential write with heavy inode cleaning demands benefit from several cleaner threads, less write allocation intensive workloads may see a performance degradation with too many cleaner threads. More threads come with additional lock contention accessing shared data structures (e.g., bucket caches), increased thread management overhead, and excessive CPU consumption that takes resources away from other work. Because no single number of threads is best in all cases, WAFL dynamically tunes the number of cleaner threads in use based on the observed workload patterns. Additional threads are activated when cleaner thread utilization exceeds some threshold and are deactivated below another (e.g., 90% and 50%).

Figure 8 shows the throughput achieved at peak load and the latency achieved at off-peak load for an internal OLTP benchmark from clients connected via Fibre Channel with increasing numbers of cleaner threads. Specifically, we look at the latency at the "knee" of the scalability curve, beyond which increases in load cause disproportional increases in latency [11]. Considering performance at off-peak load is important because many customers operate their systems below saturation in order to: (1) achieve lower latencies and (2) leave headroom to absorb load from a high-availability partner system in the case of a failure. The testbed was a 20-

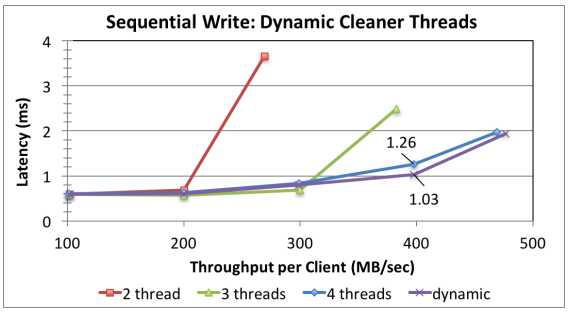

Fig. 9: Throughput per client vs. latency at increasing levels of client load. Lower and to the right is better.

core system with a combination of SSD and hard drives, using NetApp Flash Pool® technology. It is apparent that single-threaded cleaning is unable to keep up with cleaning load, because adding a second thread increases peak throughput and decreases off-peak latency. However, using more than two threads increases latency and reduces throughput by 3% as a result of added overhead. The dynamical tuning of the number of threads strikes the optimal balance.

Dynamic optimization occurs every 50ms in order to quickly respond to changes in workload. Such a fine granularity allows the system to react quickly to changes in the incoming client load and progress of the cleaning work. Figure 9 plots throughput versus latency at increasing levels of load using different numbers of cleaner threads for sequential write configuration described earlier. While peak throughput is achieved with four threads, lower latency is achieved at off-peak load with only three threads. Dynamic tuning achieves lower latency at 300MB/sec per client compared to four threads and exceeds the performance of any static number of threads at higher load by using fewer threads for short intervals with reduced cleaning work.

C. Batched Inode Cleaning

Batched inode cleaning allows multiple inodes to be associated with a single message in cases when the dirty inodes each has few dirty buffers, in order to reduce the message processing overhead. This allows large numbers of inodes to be cleaned with reduced overhead. We ran an internal benchmark with a mix of NFSv3 operations that includes reads, writes, and metadata operations across a large number of inodes on a 20-core storage server with SAS hard drives, with and without batching enabled. The use of batching simultaneously improves throughput from 21.2K ops/s to 22.0K ops/s per client and reduces latency from 6.7ms to 6.5ms. As discussed earlier, we also handle the opposite scenario wherein many writes happen to a small number of files by allowing individual inodes to be processed in parallel by multiple cleaner threads. We do not present these results due to space limitations.

VI. Related Work

Operating system scalability for multicore systems has been the subject of extensive research. Much of this work has emphasized minimizing the use of shared memory in the operating system in favor of message passing between cores that are dedicated to specific functionality [12], [13], [14], [15], [16]. Such designs allow scaling to many-core systems; however, their new designs cannot be easily adopted in legacy systems.

A recent investigation [6] found that write-intensive workloads scale poorly on modern file systems due to contention for shared objects and coordinated accesses to file system metadata, such as allocation bitmaps. Min, et al. [4] analyzed the scalability of five production file systems and found many bottlenecks, including some that may require core design changes. Another study [3] demonstrated scalability bottlenecks in tracking reference counts to mounted file system objects. To circumvent this issue, they proposed "sloppy counters" to allow reference counts to be taken per-core rather than synchronizing them globally. This proposal is very similar to our approach to "loose accounting" that batches global counter updates, which is also similar to distributed objects [17]. All of these investigations find major scalability issues, but White Alligator is able to scale write allocation to many cores.

Other scalability work leverages the fact that operating systems scale well to small core counts by running multiple OS instances within virtual machines in a many-core system to mitigate contention for shared data structures within each OS instance [18], [19]. Cerberus [19] gives each VM a dedicated storage partition with a private file system.

In a similar way, MultiLanes [5] and SpanFS [6] create independent virtualized storage devices to eliminate contention for shared resources in the storage stack. MultiLanes instantiates a collection of VMs and each VM runs its own file system on a virtualized storage device with a dedicated I/O stack. Disk writes from each VM bypass the host file system, which eliminates contention on global kernel data structures and locks. SpanFS partitions files and directories into "domains," each with dedicated file system services to allow concurrent access without contending for globally shared resources. Each domain has independent persistent structures, in-memory data structures, and kernel services so that writes to different domains are parallelized. Their method also parallelizes buffer cache operations,

which is something we have previously explored [20]. IceFS [7] creates partitions (called "cubes") in the file system to house different files and directories. Their main objective is to provide fault tolerance, but they also log transactions per cube to enable parallel transaction commits. Because we support multiple cleaner threads per file, our work increases parallelism in single-file workloads compared to mapping files to partitions. Silicon Graphic's XFS [21] divides the available storage into multiple allocation groups that are managed individually to increase parallelism. The Fast File System [22] divides media into cylinder groups to increase performance, but the resulting layout naturally supports concurrency. NOVA [23] is a log-structured file system designed to exploit nonvolatile memories that allows synchronization-free concurrency on different files.

VII. Conclusion

In this paper, we have presented the White Alligator architecture for scalable write allocation in the WAFL file system. White Alligator facilitates scaling to many cores by creating a clear division between the write allocation infrastructure that reads and manipulates file system metadata and a set of inode cleaner threads that perform the actual assignment of VBNs to dirty data. White Alligator leverages buckets of VBNs to amortize the overhead of VBN allocation and minimize the synchronization overhead. The resulting system significantly improves CPU scaling and demonstrates performance improvements of up to 274% over serialized write allocation. Further, this paper offers insight into the internal workings of the WAFL file system and its write allocation technology and presents the evolution of parallelism within this subsystem. This work is evidence that, contrary to conventional wisdom, a traditional file system design can scale to many cores, with the right architecture in place.

References

[1] D. Hitz, J. Lau, and M. Malcolm, "File system design for an NFS file server appliance," in *Proceedings of USENIX Winter Technical Conference*, 1994.

[2] M. Curtis-Maury, V. Devadas, V. Fang, and A. Kulkarni, "To waffinity and beyond: A scalable architecture for incremental parallelization of file system code," in *Proceeding of Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[3] S. Boyd-Wickizer, A. T. Clemens, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, "An analysis of Linux scalability to many cores," in *Proceedings of the Symposium on Operating System Design and Implementation (OSDI)*, 2010.

[4] C. Min, S. Kashyap, S. Maass, W. Kang, and T. Kim, "Understanding manycore scalability of file systems," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2016.

[5] J. Kang, B. Zhang, T. Wo, C. Hu, and J. Huai, "MultiLanes: Providing virtualized storage for OS-level virtualization on many cores," in *Proceedings of Conference on File and Storage Technologies (FAST)*, 2014.

[6] J. Kang, B. Zhang, T. Wo, W. Yun, L. Du, S. Ma, and J. Huai, "SpanFS: A scalable file system on fast storage devices," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2015.

[7] L. Lu, Y. Zhang, T. Do, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Physical disentanglement in a container-based file system," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[8] J. K. Edwards, D. Ellard, C. Everhart, R. Fair, E. Hamilton, A. Kahn, A. Kanevsky, J. Lentini, A. Prakash, K. A. Smith, and E. Zayas, "FlexVol: flexible, efficient file volume virtualization in WAFL," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, Jun 2008, pp. 129–142.

[9] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems*, vol. 10, pp. 1–15, 1992.

[10] R. Kesavan, R. Singh, T. Grusecki, and Y. Patel, "Algorithms and data structures for efficient free space reclamation in WAFL," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*, 2017.

[11] N. M. Patel, "Half-latency rule for finding the knee of the latency curve," *SIGMETRICS Perform. Eval. Rev.*, vol. 43, no. 2, Sep. 2015.

[12] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schupbach, and A. Singhania, "The multikernel: A new OS architecture for scalable multicore systems," in *Proceedings of the Symposium on Operating System Principles (SOSP)*, 2009.

[13] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: An operating system for many cores," in *Proceedings of the Symposium on Operating System Design and Implementation (OSDI)*, 2008.

[14] D. A. Holland and M. I. Seltzer, "Multicore OSes: Looking forward from 1991, er, 2011," in *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2011.

[15] M. Li, S. S. Vazhkudai, A. R. Butt, F. Meng, X. Ma, Y. Kim, C. Engelmann, and G. Shipman, "Functional partitioning to optimize end-to-end performance on many-core architectures," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2010.

[16] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): The case for a scalable operating system for multicores," *Operating Systems Review*, vol. 43(2), 2009.

[17] J. Appavoo, D. D. Silva, O. Krieger, M. Auslander, M. Ostrowski, B. Rosenburg, A. Waterland, R. W. Wisniewski, J. Xenidis, M. Stumm, and L. Soares, "Experience distributing objects in an SMMP OS," *ACM Transaction on Computer Systems*, vol. 25(3), 2007.

[18] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum, "Disco: Running commodity operating systems on scalable multiprocessors," *ACM Transaction on Computer Systems*, vol. 15(4), 1997.

[19] X. Song, H. Chen, R. Chen, Y. Wang, and B. Zang, "A case for scaling applications to many-core with OS clustering," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2011.

[20] P. Denz, M. Curtis-Maury, and V. Devadas, "Think global, act local: A buffer cache design for global ordering and parallel processing in the WAFL file system," in *Proceedings of the Internal Conference on Parallel Processing (ICPP)*, 2016.

[21] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the XFS file system." in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 1996.

[22] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, "A fast file system for UNIX," *Transactions on Computer Systems*, vol. 2, no. 3, pp. 181–197, 1984.

[23] J. Xu and S. Swanson, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proceedings of Conference on File and Storage Technologies (FAST)*, 2016.