To Waffinity and Beyond: A Scalable Architecture for Incremental Parallelization of File System Code

Matthew Curtis-Maury, Vinay Devadas, Vania Fang, and Aditya Kulkarni, *NetApp, Inc.*

https://www.usenix.org/conference/osdi16/technical-sessions/presentation/curtis-maury

This paper is included in the Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16).

November 2–4, 2016 • Savannah, GA, USA

ISBN 978-1-931971-33-1

Open access to the Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

To Waffinity and Beyond: A Scalable Architecture for Incremental Parallelization of File System Code

Matthew Curtis-Maury, Vinay Devadas, Vania Fang, and Aditya Kulkarni
NetApp, Inc.
{mcm,vdevadas,vania,adityak}@netapp.com

Abstract

In order to achieve higher I/O throughput and better overall system performance, it is necessary for commercial storage systems to fully exploit the increasing core counts on modern systems. At the same time, legacy systems with millions of lines of code cannot simply be rewritten for improved scalability. In this paper, we describe the evolution of the multiprocessor software architecture (MP model) employed by the Netapp® Data ONTAP® WAFL® file system as a case study in incrementally scaling a production storage system.

The initial model is based on small-scale data partitioning, whereby user-file reads and writes to disjoint file regions are parallelized. This model is then extended with hierarchical data partitioning to manage concurrent accesses to important file system objects, thus benefiting additional workloads. Finally, we discuss a fine-grained lock-based MP model within the existing data-partitioned architecture to support workloads where data accesses do not map neatly to the predefined partitions. In these data partitioning and lock-based MP models, we have facilitated incremental advances in parallelism without a large-scale code rewrite, a major advantage in the multi-million line WAFL codebase. Our results show that we are able to increase CPU utilization by as much as 104% on a 20-core system, resulting in throughput gains of up to 130%. These results demonstrate the success of the proposed MP models in delivering scalable performance while balancing time-to-market requirements. The models presented can also inform scalable system redesign in other domains.

1 Introduction

To maintain a competitive advantage in the storage market, it is imperative for companies to provide cutting-edge platforms and software to maximize the returns from such systems. Recent technological trends have made this prospect more difficult, as increases in CPU clock speed have been abandoned in favor of increasing core counts. Thus, to achieve continuing performance gains, it has become necessary for storage systems to scale to an ever-higher number of cores. As one of the primary computational bottlenecks in storage systems, the file system itself must be designed for scalable processing.

Due to time-to-market objectives, it is simply not feasible to rewrite the entire code base of a production file system to use a new multiprocessor (MP) model. Reimplementing such a system to use explicit fine-grained locking would require massive code inspection and changes and would also carry with it the potential for introducing races, performance issues caused by locking overhead and contention, and the risk of deadlocks. In this paper, we present a series of techniques that have allowed us to simultaneously meet scalability and schedule requirements in the WAFL file system over the course of a decade and to minimize the code changes required for parallelization. In particular, all of our approaches emphasize *incremental parallelization* whereby common code paths can be optimized without having to make changes in less critical code paths. Although we evaluate these techniques in a file system, the approaches are not inherently limited to that context.

The first technique we discuss—referred to as *Classical Waffinity*—applied data partitioning to fixed-size regions of user files. This approach provided a mechanism to allow read and write operations to different ranges of user files to occur in parallel without requiring substantial code rewrite, because the use of data partitioning minimized the need for explicit locking. Extending this model, *Hierarchical Waffinity* further parallelized operations that modify systemwide data structures or metafiles, such as the creation and deletion of files, by implementing a hierarchical data partitioning infrastructure. Compared to Classical Waffinity, Hierarchical Waffinity improves core usage by up to 38% and achieves 95% average utilization across a range of critical workloads.

Finally, we have extended Hierarchical Waffinity to handle workloads that do not map neatly to these partitions by adding a fine-grained lock-based MP model *within* the existing data-partitioned architecture. This innovative model—called *Hybrid Waffinity*—provides sig-

nificant parallelism benefits for previously problematic access patterns while not requiring any code changes for workloads where Hierarchical Waffinity already excelled. That is, the hybrid model introduces minimal explicit locking in narrowly defined cases to overcome specific scalability limitations in Hierarchical Waffinity to increase core usage by up to 104% and improve throughput by as much as 130%. Using these techniques has allowed WAFL to scale to the highest-end Data ONTAP platforms of their time (up to 20 cores) while constraining modifications to the code base.

The primary contributions of this paper are:

- We present a set of multiprocessor software architectures that facilitate incremental parallelization of large legacy code bases.
- We discuss the application of these techniques within a high-performance, commercial file system.
- We evaluate each of our approaches in the context of a real production storage system running a variety of realistic benchmarks.

Next, we present a short background on WAFL. Sections 3 through 5 present the evolution of the WAFL multiprocessor model through the various steps outlined above, and Section 6 evaluates each of the new models. Section 7 presents related work, and we conclude in Section 8.

2 Background on the WAFL File System

WAFL implements the core file system functionality 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* and handles file system operations to them. In WAFL, all metadata and user data (including logical units in SAN protocols) is stored in files, called metafiles and user files, respectively. The file system is organized as a tree rooted at the super block. File system operations are dispatched to the WAFL subsystem in the form of messages, with payloads containing pertinent parameters. For detailed descriptions of WAFL, see Hitz et al. [17] and Edwards et al. [13].

Data ONTAP itself was first parallelized by dividing each subsystem into a private *domain*, where only a single thread from a given domain could execute at a time. For example, domains were created for RAID, networking, storage, file system (WAFL), and the protocols. Communication between domains used message passing. Domains were intentionally defined such that data sharing was rare between the threads of different domains, so this approach allowed scaling to multiple cores with minimal code rewrite, because little locking was required. The file system module executed on a dedicated set of threads in a single domain, such that only a single thread could run at a time. This simplistic model provided sufficient performance because systems at the time had very few cores (e.g., four), so parallelism within the file system was not important. Over time, each of these domains has become parallel, but in this paper we focus on the approaches used to parallelize WAFL.

3 Classical Waffinity

As core counts increased, serialized execution of file system operations became a scalability bottleneck because such operations represented a large fraction of the computational requirements in our system. To provide the initial parallelism in the file system, we implemented a multiprocessor model called *Waffinity* (for WAFL affinity), 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 then partitioned into *file stripes* that corresponded to a contiguous range of blocks in the file (approximately 2MB in size), 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 threads executing on different cores. In contrast, any two messages that were operating on the same region of a file would be enqueued within the same affinity and would therefore execute sequentially. 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 and the development cost of parallelizing the file system. Some locking was still required to protect shared global data structures that could be accessed by multiple affinities.

In Waffinity, we introduced a set of threads to execute the messages in each affinity, and we allowed the thread scheduler to simultaneously run multiple Waffinity threads. The Waffinity scheduler maintained a FIFO list of affinities with work; that is, affinities that had been sent messages that operated within that partition. Any running thread dynamically called into the Waffinity scheduler to be assigned an affinity from which to execute messages. The number of Stripe affinities was

empirically tuned to be 12 and the number of Waffinity threads was defined per platform to scale linearly with the number of cores. NetApp storage systems at the time maxed out at 8 cores, which ensured more affinities than threads. Having more affinities than threads and creating a dynamic association between them decreased the likelihood of any thread being unable to find work.

The benefit of this model comes from the fact that most performance-critical messages at the time, such as user file reads and writes, could be executed in Stripe affinities, because they operated within a single user-file stripe. However, reads and writes across file stripes or operations such as Open and Close that touch file system metadata could not be performed in parallel from the Stripe affinities. To handle such cases, we provided a single-threaded execution context—called the Serial affinity—such that when it was scheduled, no Stripe affinities were scheduled and vice versa. This approach is analogous to a Reader-Writer Lock, where the Serial affinity behaves like a writer with exclusive file system access and the Stripe affinities behave like readers with shared access. Therefore, any messages requiring access to data that was unsafe from the Stripe affinities could be assured exclusive access to all file system data structures by executing in the Serial affinity. Use of the Serial affinity excessively serialized many operations; however, it allowed us to incrementally optimize the file system by parallelizing only those messages found to be performance critical. This approach also provided an option whereby unsafe code paths could be dynamically re-sent to the Serial affinity (called a *slowpath*).

Classical Waffinity exposed sufficient concurrency to exploit high-end NetApp platform core counts at the time, i.e., 8 cores. That is, further parallelizing file system operations would not have had a major impact on overall performance because the cores were already well used. However, as the number of cores increased, limitations in the partitioning provided by the model led to scalability bottlenecks. For example, an SFS2008 benchmark running on 12 cores saw the Serial affinity in use 48% of the time, as a result of the metadata operations such as Setattr, Create, and Remove. This serialization resulted in considerable core idle time, as evaluated in Section 6.1.1, which reduced potential performance. Further, most of the work remaining could not be moved to a Stripe affinity due to the strict rules of what can run there (i.e., operating within a single user file stripe). With higher core counts, serialized execution had a larger impact on performance, because the speedup achieved through parallelism was limited by the serial time, in accordance with Amdahl's law. Thus, although sufficient at the time, Classical Waffinity as first designed was simply unable to provide the required performance going forward.

Figure 1: The structure of Classical Waffinity with the Serial affinity on top and the twelve Stripe affinities below.

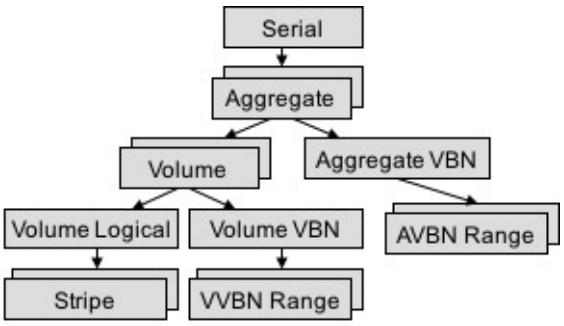

Figure 2: The Hierarchical Waffinity hierarchy rooted at the Serial affinity.

4 Hierarchical Waffinity

Hierarchical Waffinity builds on top of the Classical Waffinity model to enable increased levels of parallelism. This model, which first shipped with Data ONTAP 8.1 in 2011, greatly reduces the increasingly critical single-threaded path by providing a way to parallelize additional work. Further, the model offers insight into protecting hierarchically structured systems in other domains [22] by using data partitioning.

4.1 The Affinity Hierarchy

An alternative view of the Serial and Stripe affinities of Classical Waffinity is as a hierarchy, as shown in Figure 1, where a node is mutually exclusive with its children but can run in parallel with other nodes at the same level. Hierarchical Waffinity facilitates additional parallelism by extending this simple 2-level hierarchy to efficiently coordinate concurrent accesses to other fundamental file system objects beyond data blocks of user files.

Each affinity is associated with certain permissions, such as access to metadata files, that serialized execution in the file system in Classical Waffinity (Figure 2 and Table 1). The new affinity scheduler enforces execution exclusivity between a given affinity and its children, so Hierarchical Waffinity only restricts the execution of an affinity's *ancestors* and *descendants* (hierarchy parents and children, respectively); all other affinities can safely run in parallel. For example, if the Volume Logical affinity is running, then its Stripe affinities are excluded along with its parent Volume, Aggregate, and Serial affinities.

This design ensures that no two messages with conflicting data accesses run concurrently, because they would run in affinities that exclude one another. Hierarchical Waffinity is analogous to a hierarchy of Reader-Writer Locks, where running in any affinity acquires the lock as a writer and thus prevents any readers (i.e., descendants) from running concurrently, and vice versa.

The WAFL file system is itself hierarchical (i.e., buffers within inodes within FlexVol volumes within aggregates), making Hierarchical Waffinity a natural fit. Knowledge of the specific data access patterns that are most common in WAFL informed the decision of affinity layout in the hierarchy and the mapping of specific object accesses to those affinities.

As a general rule, client-facing data, such as user files, logical units, and directories, is mapped to the Volume Logical branch of the hierarchy and internal metadata is mapped to Volume VBN—so named because it is typically indexed by volume block number (VBN). This allows parallelism between client-facing and internal operations within a single volume. The Aggregate, Volume, Stripe, and Range affinity types have multiple *instances*, which allows 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. Each file system object is mapped to a particular instance in the affinity hierarchy, based on its location in the file system. The number of instances of each affinity as well as the mapping of objects to instances can be adapted to the observed workload to maximize performance. Further, new affinity types can be (and have been) added to the hierarchy over time in response to new workloads and data access patterns.

4.2 Affinity Access Rules

The affinity permissions required by a message are determined by the type of objects being accessed and the access types. We used knowledge of the system to derive affinity permissions that allowed performance-critical operations to run with the most parallelism. In WAFL, any access is associated with a specific affinity, using the rules shown in Table 1. *Exclusive* access to an object ensures that no concurrently running affinities can access that object. The fundamental objects that are protected by data partitioning in Waffinity are buffers, files, FlexVol volumes, and aggregates. Accesses to other object types are infrequent but are protected with fine-grained locking, and deadlock is prevented through the use of lock hierarchies.

Each FlexVol volume and aggregate is mapped to a Volume and Aggregate affinity when it comes online. In the WAFL file system, files are represented by *inodes*, which are mapped to an affinity within the hierarchy of the volume or aggregate in which they reside. Inodes can be accessed in either *exclusive* or *shared* mode. In exclusive mode, only one message has access to the inode and consequently can change any inode property or free the inode. In shared mode, the inode's fundamental properties remain read-only, but the majority of an inode's fields can be modified and are protected from concurrent accesses by fine-grained locking. Similar distinctions are in place for FlexVol volumes and aggregates.

Blocks are represented in memory by a buffer header and a 4KB payload. Details of the WAFL buffer cache architecture have previously been published [11]. Accesses to buffers fall into the following four categories:

- Insert: A new buffer object is allocated and the payload is read in from disk. The buffer becomes associated with the corresponding inode.
- Read: The payload of an in-memory buffer is read. No disk I/O is required in this case.
- Write: The payload of the in-memory buffer is modified in memory. The buffer header is also modified to indicate that it is dirty.
- Eject: The buffer is evicted from memory.

Each of these access modes is mapped to a specific affinity instance for a given buffer. For Write, Insert, and Eject accesses, our data partitioning model requires that only a single operation perform any of these accesses at a time. Thus, operations must run in the designated affinity for a given access mode or its ancestor. On the other hand, read accesses are safe in parallel with each other, but it is necessary to ensure that the buffer will not be ejected underneath it, so that it can run in any ancestor or descendant of the Eject affinity. Typically, Write and Eject access are equivalent, which similarly prevents concurrent reads and writes. The affinity mappings for buffers are chosen to allow maximum parallelism, while ensuring access to the buffers from all necessary affinities. For example, user file reads and writes run in the Stripe affinities because the relevant buffers map to these affinities.

4.3 Mapping Operations to Affinities

Messages are sent to a predetermined affinity based on the required permissions of the operation, as identified by the software developer. If a message requires the permissions assigned to multiple affinities, then it is directed to an affinity that is an ancestor of each. For example, an operation that requires privileges assigned to a particular

Affinity Name	Access Privileges Provided
Stripe	Provides exclusive access to a predefined, distinct set of blocks. Enables concurrent access to sub-file-level user data.
Volume Logical	Provides exclusive access to most client-visible files (and directories) within a volume and to certain file system internal metadata.
Volume VBN	Provides exclusive access to most file system metadata, such as those files that track block usage. Such files are typically indexed by volume block number (VBN).
Volume VBN Range	Provides access to specific ranges of blocks within files belonging to Volume VBN.
Volume	Provides exclusive access to all files within a volume as well as per-volume metadata that lives in the containing aggregate.
Aggregate VBN	Similarly to Volume VBN, provides access to most file system metadata in an aggregate.
Aggregate VBN Range	Provides access to specific ranges of blocks within files belonging to Aggregate VBN.
Aggregate	Provides exclusive access to files in an aggregate.
Serial	Inherits the access rights of all other affinities and provides access to other global data.

Table 1: The affinities and the access rights that they provide.

Stripe affinity and to a Volume VBN affinity could safely execute in the Volume affinity. As in Classical Waffinity, if a message is routed to a particular affinity and later determines that it requires additional permissions, it may be dynamically re-sent (i.e., slowpath) to a *coarser* (i.e., less parallel) affinity that provides the necessary access rights. Object accesses are achieved through a limited set of APIs that we have updated to enforce the required affinity rules, including slowpathing if necessary. Thus, the programming model helps ensure code correctness. The more access rights required by a message, the higher in the affinity hierarchy it must run and the more it limits its parallelism by excluding a larger number of affinities from executing. Thus, it is preferable to run in as *fine* an affinity as possible. For this reason, we have selected data mappings that allow the most common operations to be mapped to fine affinities, and objects that are frequently accessed together are given similar mappings.

Figure 3 shows several example affinity mappings of operations under a single Volume affinity. For simplicity, we assume a file stripe size of 100 contiguous user blocks, or 10 blocks in metafiles. Reads and writes within a file stripe map to a Stripe affinity. However, a user file deletion ("Remove: A") runs in the Volume Logical affinity, because it requires exclusive access to that user file, as does a write operation that spans multiple file stripes ("W: A[100..200]"). Running in this affinity prevents the execution of any reads or writes to blocks in that file. However, reads and writes to files in other volumes are not impacted, nor are operations on file system metadata within the same volume. Note that "W: A, MD" must run in Volume affinity to write to both a user file and metafile. As noted earlier, key message parameters are tracked in the message payload. When a message is sent into WAFL, its payload is inspected to determine the type of the message and other data from which the destination affinity is calculated. Effectively, each message exposes the details of its data accesses to the scheduler so that MP safety can be enforced at this level, similar to some language constructs for task-based systems [3, 34]. For example, a write message exposes the offsets being written and thus the required affinity for execution.

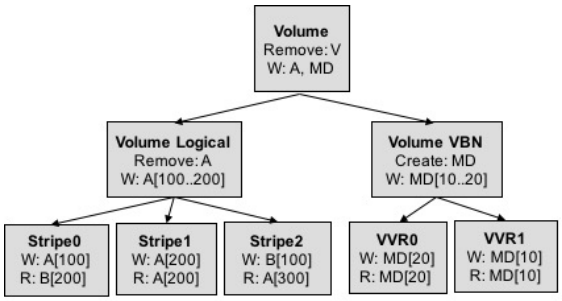

Figure 3: Example affinity mappings of Read ("R"), Write ("W"), Remove, and Create operations to user files A and B, metadata file MD, and FlexVol V. A block offset of 100 with file A is denoted as A[100].

4.4 Development Experience with Hierarchical Waffinity

Hierarchical Waffinity allows parallelization at the granularity of a message type, running all other message types in the Serial affinity, allowing *incremental optimization* over time. Thus, parallelization effort scales

with the size of the message, rather than requiring the entire code base to be updated at once. A typical message handler is on the order of *hundreds* or *thousands* of lines of code, whereas the file system is on the order of *millions*. Further, a message can first be parallelized into one affinity and later moved to a finer affinity when need arises and the extra development cost can be justified.

Messages often require only minimal code changes during parallelization because data access guarantees are provided by the model. In such cases, after a detailed line-by-line code inspection to evaluate multiprocessor safety, messages can be moved into fine affinities just by changing the routing logic that computes the target affinity. For example, parallelizing the Link operation to run in the Volume Logical affinity required fewer than 20 lines of code to be written, none of which were explicit synchronization. In other cases, major changes are required to safely operate within a single affinity, for example by restructuring data accesses, thus requiring potentially thousands of lines of code changes. In WAFL, the underlying infrastructure required to implement the affinities, scheduling, and rule enforcement amounted to approximately 22K lines of code. Compared to the alternative of migrating the whole file system to fine-grained locking, which would involve inspecting and updating a large fraction of the millions of lines of code, these costs are relatively small.

Software systems in many domains employ hierarchical data structures (such as linear algebra [12] and computational electromagnetics [14]), and a variety of techniques have been developed to provide multiprocessor safety in such cases [15, 22]. Hierarchical Waffinity offers an alternative architecture that is capable of incremental parallelization. In applying this approach to other systems, the types of affinities to create, the number of instances of each type, and the mapping of objects to affinities would be based on domain-specific knowledge of the data access patterns. In practice, this approach applies most naturally to message passing systems where a subset of message handlers could be parallelized while leaving others serialized, or alternatively to task-based systems such as Cilk++ [26].

4.5 Hierarchical Scheduler

The Hierarchical Waffinity scheduler is an extension of the Classical Waffinity scheduler that maps the now greater set of runnable affinities to the Waffinity threads for execution while enforcing the hierarchical exclusion rules. As before, Waffinity threads are exposed to the general CPU scheduler, and when they are selected for execution, they begin by calling into the Waffinity scheduler for work. A running thread then begins processing the messages queued up to that affinity for the duration of an assigned quantum, after which the thread calls back into the scheduler to request another affinity to run.

As messages are sent into WAFL and processed by the Waffinity threads, the Waffinity scheduler tracks the sets of affinities that are runnable (i.e., with work and not excluded), running, or excluded by other executing affinities. When threads request work, the scheduler selects an affinity from the runnable list, assigns it to the thread, and updates the scheduler state to reflect the newly running affinity. In particular, the scheduler maintains a queue of affinities that is walked in FIFO order to find an affinity that is not excluded. An excess of work in coarse affinities manifests in the scheduler as a shortage of runnable affinities, resulting in situations where available threads cannot find work to do and must sit idle, which in turn results in wasted CPU cycles. We also track the number of runnable affinities and ensure that the optimal number of threads are in flight any time an affinity begins or ends execution. To prevent the starvation of coarse affinities, we periodically drain all running affinities and ensure that all affinities run with some regularity. Figure 4 shows a sample scheduler state. In this example, there are seven affinities currently running and five more that can be selected for execution by the affinity scheduler.

4.6 Waffinity Limitations and Alternatives

Fundamental in the Waffinity architecture is a mapping of file system objects to a finite set of affinities, often causing independent operations to unnecessarily become serialized. For example, any two operations that require exclusive access to two user files in the same volume (such as deletion) are serialized. As long as sufficient parallelism is found to exploit available cores on the target platforms, this limitation is acceptable in the sense that increasing available parallelism will not result in increased performance. Two other scenarios that can result in significant performance loss are 1) when frequently accessed objects directly map to a coarse affinity; and 2) when two objects mapped to different affinities must be accessed by the same operation. These scenarios are not well handled in Hierarchical Waffinity and result in poor scaling in several important workloads, as shown in Section 6.

An alternative to the Waffinity architecture would be to use fine-grained locking to provide MP safety. In such an approach, all file system objects would be protected by using traditional locking, and no limits need to be imposed on which operations can be executed in parallel. This would provide additional flexibility in the programming model; however, it carries many drawbacks

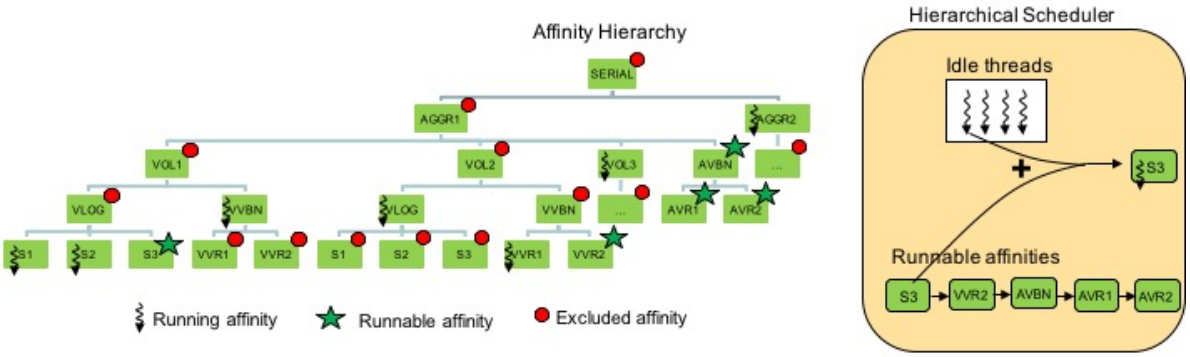

Figure 4: Sample hierarchical affinity scheduler state.

that led to the decision to implement Classical and Hierarchical Waffinity. Reimplementing the file system to exclusively use fine-grained locks would require a massive code rewrite and would carry with it the potential for introducing races, performance issues caused by locking overhead and contention, and the risk of deadlocks. Instead, our approach reduces the overall amount of locking required, and even allows messages to run in coarse affinities without locking.

5 Hybrid Waffinity

The next step in the multiprocessor evolution of WAFL is *Hybrid Waffinity*, which shipped with Data ONTAP 9.0 in 2016. This model is designed to scale workloads for which Hierarchical Waffinity is not well suited, due to a poor mapping of data accesses to affinities, as discussed in Section 4.6. This model supports fine-grained locking *within* the existing hierarchical data partitioned architecture to protect particular objects when accesses do not map neatly to fine partitions. At the same time, we continue to use partitioning where it already excels, such as for user file reads and writes. Overall, Hybrid Waffinity leverages both fine-grained locking *and* data partitioning in cases where each approach excels. Although this may seem like an about-face from the data partitioned models, in fact it is merely an acknowledgement that in some cases fine-grained locking is required for effective scaling. Retaining partition-based protection in most cases is critical so that only the code that scales poorly with data partitioning needs to be updated.

In Hybrid Waffinity, we allow buffers in a few select metafiles to be protected by locking, while the vast majority of buffers, as well as all other file system objects, continue to use data partitioning for protection. The result is a *hybrid* model of MP-safety where different buffer types have different protection mechanisms. The use of locking allows these buffers to be accessed from finer affinities; however, we continue to allow *lock-free* access from a coarser affinity. That is, because all object accesses occur within some affinity subtree, a message running in the root of that subtree can safely access the object without locking.

5.1 Hybrid-Insert

With Hierarchical Waffinity, each buffer is associated with a specific Insert affinity that protects the steps involved in inserting that buffer. This mandates that all messages working on inserting the buffer will run in the same affinity to be serialized. This design is effective in the common case where a single buffer is accessed or multiple buffers with similar affinity mappings are accessed. However, in cases where a message accesses multiple buffers in different partitions, the message must run in a coarser affinity that provides all of the necessary permissions. Figure 5 illustrates a scenario in which both User file buffers and Metafile buffers must be accessed, which happens when replicating a FlexVol volume (discussed in Section 6.2.3). This operation must run in the AGGR1 affinity, rather than the more parallel S1 or AVR1 affinity. In such cases, parallelism in the system is reduced, because time spent running in coarse affinities limits the available affinities that can be run concurrently, potentially starving threads and cores of work.

We overcome these shortcomings with Insert by allowing *Hybrid-Insert* access to certain buffers from *multiple* fine affinities. Only a few buffer types are frequently accessed in tandem with other buffers, and we apply Hybrid-Insert only in such cases. Thus, a message accessing two buffers now runs in the traditional (fine) Insert affinity of one buffer and protects the second buffer by using Hybrid-Insert, rather than in a coarse affinity with Insert access to both buffers. Allowing multiple affinities to insert a buffer means that two messages can simultaneously insert the same buffer, but Hybrid-Insert resolves such races and synchronizes all callers of any

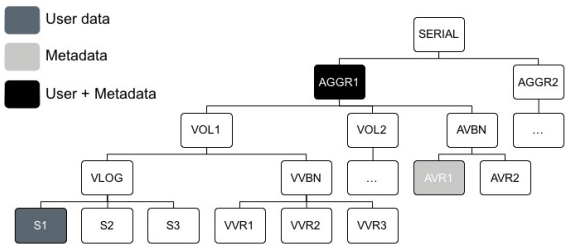

Figure 5: Hierarchical Waffinity model with User data access in S1 and Metadata access in AVR1.

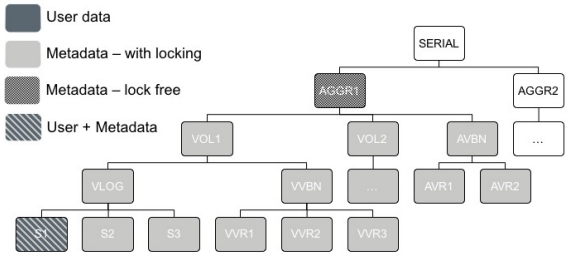

Figure 6: Hybrid Waffinity model where User data access continues to be in S1, but Metadata access is permitted from any descendant of AGGR1.

insert code path. Referring back to Figure 5, the User data can continue to be protected with partitioning in S1 and the Metadata can now be accessed from the S1 affinity (for example) with explicit synchronization, as shown in Figure 6. Further, noncritical messages that operate on Metadata can run in AGGR1 without being rewritten, because the scheduler will not run any other messages that access this data.

For explicit synchronization, Hybrid-Insert uses what we refer to as *MP-barriers*. MP-barriers employ a set of spin-locks that are hashed based on buffer properties. The insert process consists of a series of critical sections of varying lengths. Short critical sections can simply hold the spin-lock for their duration. However, longer critical sections avoid holding the lock for long periods by instead stamping the buffer with an *in-progress* flag under the lock at the beginning of the critical section and clearing it at the end. Other messages that encounter the in-progress flag can then block, knowing that a message is already moving this buffer toward insertion.

5.2 Hybrid-Write

The next buffer access mode we consider is Write, which involves changing the contents of a buffer and updating associated metadata. Hierarchical Waffinity serializes all readers and writers to the same buffer by mapping all such operations to the same affinity. Thus, Write access made it safe for writers to modify the buffer without locking and for readers to know that no writes were happening concurrently to the buffer. However, as with Insert access, messages requiring access to multiple buffers with different affinity mappings needed to run in a coarse affinity, thereby limiting parallelism.

Hybrid-Write allows writes to certain types of buffers from multiple affinities so that messages routed to an affinity for Write access to one buffer will also be able to access a different buffer by using Hybrid-Write, again as in Figure 6. Thus, readers and writers must synchronize by using fine-grained locking to ensure MP-safety and data consistency. In the new model, we have retained the traditional Write affinity in which an operation can run without locking. Read access has been redefined for Hybrid-Write buffer types such that it now maps to the traditional Write affinity, thereby providing (slow) read access to the buffer without any locking. New access modes called *Shared-Write* and *Shared-Read* have been added to provide access from finer affinities, and only by explicitly using these access modes—and thus implicitly agreeing to add the necessary locking—is any additional parallelism achieved. Thus, Hybrid-Write uses an *opt-in* model wherein legacy code remains correct by default until it is manually optimized.

Hybrid-Write uses spin-locks to protect buffer data and metadata. Spin-locks are sufficient here because the critical sections for reads and writes are typically small. The introduction of fine-grained locking increases complexity; however, it can be done incrementally as required by specific messages without a large-scale code rewrite. As an example of the increased complexity, buffer state observed at any time in Hierarchical Waffinity could always be trusted because the entire message execution was under the implicit locking of the scheduler. In contrast, buffer state observed inside an explicit critical section can no longer be trusted once the lock is released.

5.3 Hybrid-Eject

The final buffer access mode is Eject, which provides exclusive access and allows arbitrary updates to the buffer, including evicting the buffer from memory. Thus, Eject access maps to an affinity that excludes all affinities with access to this buffer. *Hybrid-Eject* instead uses fine-grained locking to provide exclusive access from a finer affinity. Unlike Hybrid-Insert and Hybrid-Write, we compute a *single* Hybrid-Eject affinity for each buffer to serialize all code paths. While Hybrid-Eject could always be used in place of Hybrid-Write, the semantics of Hybrid-Eject are more restrictive and would limit perfor-

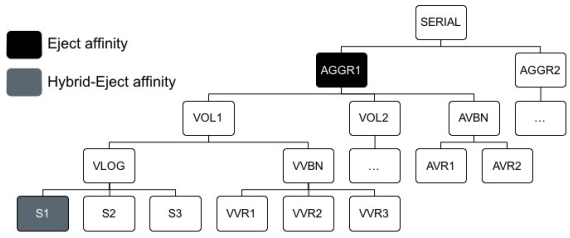

Figure 7: Waffinity model where traditional Eject access maps to AGGR1, but Hybrid-Eject maps to a specific fine affinity S1.

mance. We have also retained the traditional Eject affinity to minimize required code changes. Figure 7 shows a sample hierarchy with Eject affinity in AGGR1 and Hybrid-Eject in S1.

Simply protecting each buffer with a spin-lock would not be feasible for Hybrid-Eject because messages can read many buffers, each of which must be protected from ejection. Instead, we track a global serialization count that is incremented at periodic serialization points within the file system. Whenever a reference to a buffer is taken, it is stamped with the current serialization count under a spin-lock and is said to have an *active stamp*. Buffers with active stamps cannot be evicted and are implicitly unlocked when the serialization count is next incremented. Because message execution cannot span serialization points, buffers with stale stamps can be safely ejected. Preventing the ejection of a buffer for this duration is excessive but practical, since only 0.002% of buffers considered for ejection had an active stamp during an experiment with heavy load on a high-end platform.

To extend this infrastructure beyond buffer ejection, we also define an *exclusive stamp* that prevents any concurrent access at all. A message requiring exclusive access can simply put this value on the buffer and all subsequent accesses to the buffer spin until the exclusive stamp has been cleared. Spinning is acceptable in practice, since only 0.007% of exclusive stamp attempts encountered an active stamp in an experiment with heavy load.

5.4 Development Experience with Hybrid Waffinity

Adopting Hybrid Waffinity within WAFL involved creating the underlying infrastructure and then parallelizing individual messages to take advantage of it. For each of the access modes, we required approximately 3K lines of code changes, which included extensive rule enforcement and checking. As in the case of Hierarchical Waffinity, the effort involved in each specific message parallelization varied widely. Leveraging Hybrid-Insert and Hybrid-Eject requires few code changes because the infrastructure is primarily embedded within existing APIs. Hybrid-Write, on the other hand, requires more code changes due to the addition of fine-grained locking throughout the message handler. For example, all three messages that we optimized using Hybrid-Eject required fewer than 20 lines of code changes. In contrast, two messages parallelized using Hybrid-Insert and Hybrid-Write required a few thousand lines of changes.

We have already begun to apply this technique to other objects within WAFL. In particular, a project is under way to further parallelize access to certain inodes in WAFL by using Hybrid Waffinity, and we have found the code changes to be relatively modest. We are optimistic that the ease with which this technique was applied to inodes will translate to other software systems. The techniques of Hybrid Waffinity can potentially be applied in any data partitioned system, not only those that are hierarchically arranged. Prior work has discussed the difficulty in operating on data from different partitions [21, 38], and our approaches can be used in such cases to improve parallelism. For example, consider a scientific code operating on two arrays. Tasks could be divided up based on a partitioning of one array, and access to the other array could be protected through fine-grained locking.

6 Performance Analysis

6.1 Hierarchical Waffinity Evaluation

To highlight the improvements provided by Hierarchical Waffinity, we chose two performance benchmarks that emphasize the limitations of Classical Waffinity. Hierarchical Waffinity was released in 2011 as part of Data ONTAP 8.1, and we used this software to evaluate its benefits. In this section, the benchmarks were run on a 12-core experimental platform that was the highest-end Data ONTAP platform available at the time this feature shipped. Many changes were made in Data ONTAP between releases, so we cannot directly compare the approaches. Instead, we used an instrumented kernel that runs messages in the same affinities as would Classical Waffinity for our baseline to isolate the impact of our changes. We used multiple FlexVol volumes in these experiments because this is representative of the majority of customer setups.

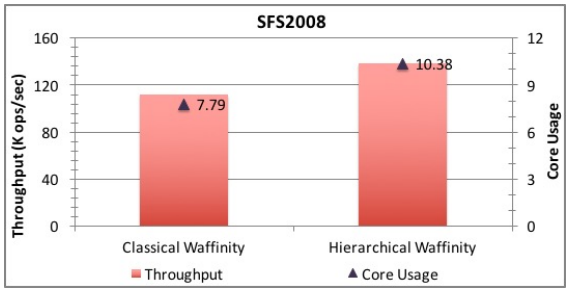

Figure 8: Throughput and core usage improvements with Hierarchical Waffinity compared to Classical Waffinity on a Spec SFS2008 workload.

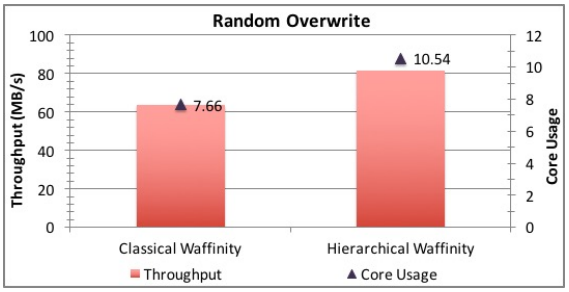

Figure 9: Throughput and core usage improvements with Hierarchical Waffinity compared to Classical Waffinity on a random overwrite workload.

6.1.1 Spec SFS2008

We first measured performance using the Spec SFS2008 benchmark [35] using NFSv3 on 64 FlexVol volumes. This workload generates mixed workloads that simulate a "typical" file server, including Read, Write, Getattr, Lookup, Readdir, Create, Remove, and Setattr operations. Several of these operations involve modification of file attributes, so the corresponding messages were forced to run in the Serial affinity in the Classical Waffinity model. Hierarchical Waffinity allowed us to move Create and Setattr into the Volume Logical affinity and Remove to the Volume affinity. Since there are eight Volume affinities, up to eight Create/Setattr/Remove operations can now run in parallel with each other and can also run in parallel with the remaining client operations in Stripe affinities.

Figure 8 shows the throughput and core usage improvements of Hierarchical Waffinity over Classical Waffinity for SFS2008. As noted, in Classical Waffinity many operations ran in the Serial affinity, thereby serializing all file system operations and resulting in idle cores. In the baseline, the Serial affinity was busy 48% of the time, thus limiting parallel execution to only 52% of the time. In contrast, by providing additional levels of parallelism, the hierarchical model was able to reduce Serial affinity usage to 9%. Alleviating this significant scalability bottleneck increased the system-wide core usage by 2.59 out of 12 cores, showing that parallelism is significantly improved through the ability to process operations on metadata in parallel with each other and with reads and writes. Most importantly, the additional core usage successfully translated into a 23% increase in throughput. That is, Hierarchical Waffinity effectively exploits the additional processing bandwidth to improve performance.

6.1.2 Random Overwrite

We next evaluate the benefit of Hierarchical Waffinity on a 64 FlexVol volume random overwrite workload. Block overwrites in the file system are particularly interesting because WAFL always writes data to new blocks on disk. Thus, for each block overwritten, the previously used block on disk must be freed and the corresponding file system metadata tracking block usage must be updated. It is not the write operation itself that is of interest in this experiment, because that was parallelized even in Classical Waffinity. This benchmark instead demonstrates the gains from parallelizing block free operations that used to run in the Serial affinity, because they involve updating file system metadata. With Hierarchical Waffinity, these messages can now run in the Volume VBN and Aggregate VBN affinities (when freeing in a FlexVol volume and aggregate, respectively) because these affinities provide access to all of the required metafiles. Thus, the changes provided in Hierarchical Waffinity 1) allow block free messages to run concurrently with each other on different FlexVol volumes because each volume has its own Volume VBN affinity; and 2) allow block free messages to run in parallel with client operations in the Stripe affinities.

Here again, Hierarchical Waffinity demonstrates a significant reduction in Serial affinity usage, from 27% to 7%, as a result of parallelizing the block free operations. This reduction in serialization made it possible for the same workload to scale to an additional 2.88 cores compared to Classical Waffinity, as shown in Figure 9. This extra core usage allowed an overall improvement in benchmark throughput of 28%. The random write workload is interesting because it demonstrates the benefits of running internally generated metadata operations in Volume VBN affinity in parallel with front-end client traffic in the Stripe affinities, which was not possible in Classical Waffinity.

6.1.3 Overall CPU Scaling

The above analysis of Hierarchical Waffinity showed a substantial improvement in the number of cores used, but 1.5 cores were still idle. These benchmarks were se-

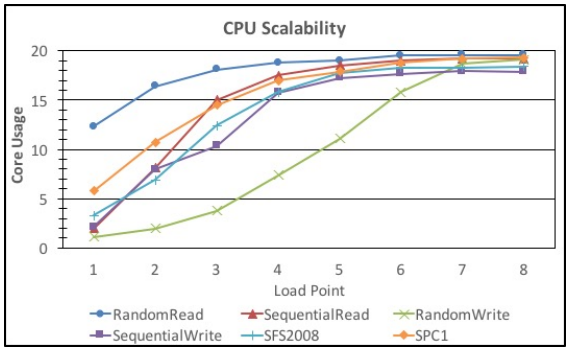

Figure 10: Core usage with Hierarchical Waffinity on a variety of workloads under increasing levels of load.

lected to illustrate the benefits of Hierarchical Waffinity *compared to* Classical Waffinity, not maximum utilization. Idle cores could have been driven down still further by parallelizing even the remaining 9% of Serial execution in SFS2008 and 7% in random write, but this was not required to meet performance and scaling objectives at the time.

In subsequent releases, we have further parallelized many operations, resulting in the improved CPU scalability shown in Figure 10. These parallelization efforts have included both reducing Serial affinity usage and moving already parallelized work into still finer affinities. In particular, the graph shows the achieved core usage at increasing levels of load (i.e., load points) for a variety of workloads on 64 FlexVol volumes on a 20-core storage server running Data ONTAP 9.0. The key takeaways are the low core usage that occurs at low load and the high core usage achieved in the presence of high load. In particular, four of the six benchmarks achieve a utilization of 19+ cores, with all benchmarks reaching at least 18 cores. This data demonstrates that Hierarchical Waffinity is able take advantage of computational bandwidth up to 20 cores in a broad spectrum of important workloads, and cores are not starved for work by an excess of computation in coarse affinities. In Section 6.3, we discuss the issue of continued scaling on future platforms.

6.2 Hybrid Waffinity Evaluation

Although the previous section demonstrated the success of Hierarchical Waffinity across a wide range of workloads, there are also cases where its scalability is limited. Thus, we next evaluate the benefits of the Hybrid Waffinity model by considering benchmarks that emphasize cases where the hierarchical model falls short. We focus on single FlexVol volume scenarios because any coarse affinity utilization significantly limits parallelism; however, we also consider the scalability with multiple FlexVol volumes where Hierarchical Waffinity itself is typically very effective already. Single-volume configurations are less common, but they are still a very important customer setup. This section describes analysis done with Data ONTAP 9.0 on a 20-core platform, the highest-end system available in 2016.

6.2.1 Sequential Overwrite

We first look at the benefits of Hybrid Waffinity on a sequential overwrite workload. As discussed above, block overwrites are interesting in WAFL because they result in block frees. In Hierarchical Waffinity, block free work runs in the Volume VBN affinity, because it requires updating various file system metadata files, so it already runs in parallel with front-end traffic in the Stripe affinities. However, if the system is unable to keep up with the block free work being created, then client operations must perform part of the block free work, which hurts performance. This problem is exacerbated on single-volume configurations where all block free work in the system must go through the single active Volume VBN affinity, which can become a major bottleneck.

Increasing the parallelism of this workload requires running block free operations in Volume VBN Range (or simply *Range*) affinities. Certain metafile buffers required for tracking free blocks can be mapped to specific Range affinities, but other metafiles need to be updated from any Range affinity, so the operation must run in an affinity that excludes both relevant affinities. Fortunately, this is an ideal scenario for *Hybrid-Insert* and *Hybrid-Write*, in that buffers from certain metafiles can be mapped to a specific Range affinity and others can be protected by using locking from any Range affinity. Using a combination of partitioning and fine-grained locking to protect its buffer accesses, Hybrid Waffinity allows the block free operation to run in 1) a finer affinity and 2) an affinity of which there are multiple instances per FlexVol volume.

We evaluate a single-volume sequential overwrite benchmark on a 20-core platform with all flash drives. Figure 11 shows the throughput achieved and core usage at increasing levels of sustained load from a set of clients (i.e., the load point). Comparing peak load points shows a 62.8% improvement in throughput from the use of 4.8 additional cores. Under sufficient load, the Hierarchical Waffinity performance falls off, because block free work is unable to keep pace with the client traffic generating the frees. Hybrid Waffinity prevents this from happening by increasing the computational bandwidth that can be applied to block free work on a single FlexVol volume. This experiment demonstrates scalability up to 13.2 cores; however, repeating the test with 64 volumes

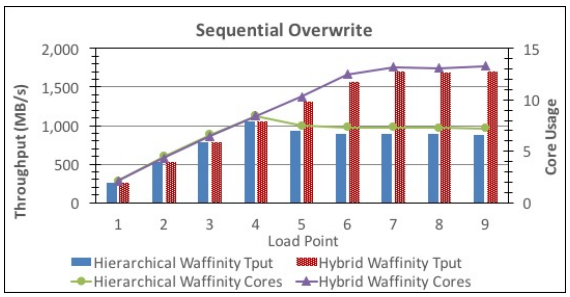

Figure 11: Throughput and core usage improvements at various levels of server load with Hybrid Waffinity compared to Hierarchical Waffinity for a sequential overwrite workload.

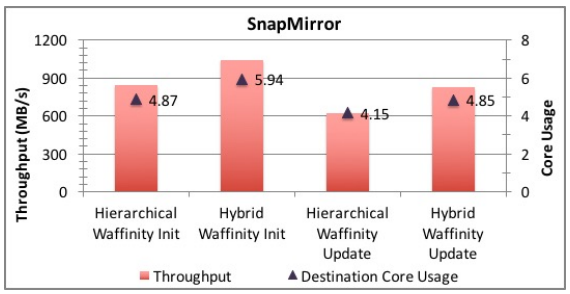

Figure 12: Throughput and core usage improvements with Hybrid Waffinity compared to Hierarchical Waffinity for a SnapMirror workload. Core usage is out of 20 available cores.

shows the system scaling further to 16.6 cores due to the activation of additional Volume affinity hierarchies. Hybrid Waffinity benefits throughput by only 7.5% and core usage by 1.6 cores in the multi-volume case because Hierarchical Waffinity is already so effective.

6.2.2 NetApp SnapMirror

The next workload we consider is NetApp SnapMirror®, a technology that replicates the contents of a FlexVol volume to a remote Data ONTAP storage server for data protection [33]. During the "init" phase, the entire contents of the volume are transferred to the destination and subsequent "update" transfers replicate data that has changed since the last transfer. In particular, SnapMirror operates by loading the blocks of a metadata file representing the entire contents of the volume on the source, sending the data to a remote storage server, and writing to the same metafile on the destination volume. Exclusive access to this file's buffers belongs to the Volume affinity, which results in substantial serialization, because such work serializes all processing within the volume. *Hybrid-Eject* allows these buffers to instead be processed in the Stripe affinities.

Figure 12 shows the benefits of Hybrid-Eject on a single-FlexVol SnapMirror transfer. The transfer is destination-limited, so we evaluate core usage on the destination. During the init phase, core usage goes up by 1.1 cores and the throughput is improved by 24.2%. Similarly, the update phase uses an additional 0.7 cores, resulting in a 32.4% gain in throughput. Despite the benefit, core usage is low in the single-volume case; however, an experiment replicating 24 volumes scales to 12.5 cores (init) and 9.8 cores (update), at which point the workload becomes bottlenecked elsewhere and Hybrid Waffinity provides no benefit beyond the hierarchical model alone.

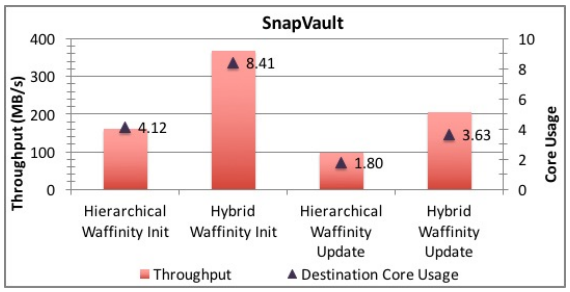

Figure 13: Throughput and core usage improvements with Hybrid Waffinity compared to Hierarchical Waffinity for a SnapVault workload. Core usage is out of 20 available cores.

6.2.3 NetApp SnapVault

We conclude our analysis by looking at the performance benefits of *all three hybrid access modes* together. Another Data ONTAP technology for replicating FlexVol volumes, called SnapVault®, writes to both user files *and* metafiles on the destination server. In Hierarchical Waffinity, the operations run in a coarse affinity (such as Volume or Volume Logical) with access to both types of buffers. With Hybrid Waffinity, user file buffers remain mapped to Stripe affinities, but Hybrid-Write and Hybrid-Insert allow the metafile buffers to be accessed by any child of the Volume affinity. Thus, SnapVault operations can run in the Stripe affinity of their user file accesses and use locking to protect metadata accesses. Further, the metafile buffers map to Volume Logical for Eject access, which can be optimized by using Hybrid-Eject to facilitate processing in the Stripe affinities.

Figure 13 presents the improvements in throughput and core usage of Hybrid Waffinity on a single-volume SnapVault transfer. The new model facilitated a throughput gain of 130% in the init phase on an extra 4.29 cores used out of 20 available. Update performance is similarly improved by 112% on a core usage increase of 1.83 cores. Increasing the transfer to 8 volumes increases the total core usage to 17.7 cores (init) and 10.6 cores (up-

date), at which point the primary bottlenecks move to other subsystems. Even here, the hybrid model improves scalability by 0.5 cores and 3.1 cores and throughput by 20.2% and 18.1%, for init and update, respectively.

In summary, Hybrid Waffinity is able to greatly improve performance in single-volume scenarios where Hierarchical Waffinity struggles most, and even improves scaling in certain multi-volume workloads.

6.3 Discussion of Future Scaling

The analysis described in this paper was conducted on the highest-end platforms available at the time each feature was released. These platforms drive the scalability investment that is made, because scaling beyond available cores does not add customer value. New platforms with higher core counts will continue to enter the market in the future and our requirements will continue to increase as a result. We expect that the infrastructure now in place will continue to pay rich dividends as future parallelization investments focus on utilizing the techniques discussed in this paper to greater degrees rather than inventing new ones. That is, the bottlenecks on the horizon are not a limitation of the architecture itself. Internal systems with more cores are currently undergoing extensive tuning, and the techniques discussed in this paper have already allowed scaling well beyond 30 cores.

7 Related Work

Operating system scalability for multicore systems has been the subject of extensive research. Recent 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 [2, 4, 18, 27, 40]. Such designs allow scaling to many-core systems; however, their new designs cannot be easily adopted in legacy systems because they require the re-architecting of major kernel components and probably are best suited for new operating systems. So although such research is crucial to the OS community, our approaches for incremental scaling are also required in practice. A recent study [5] investigated the scalability of the Linux operating system and found that traditional OS kernel designs, such as that of Data ONTAP, can scale effectively for near-term core counts, despite the presence of specific scalability bugs in the CPU scheduler [29]. In contrast, Min, et al. [31] analyze the scalability of five production file systems and find many bottlenecks, including some that may require core design changes.

Recently, many file system and operating system designs have been offered to improve scalability. NOVA [41] is a log-structured file system designed to exploit non-volatile memories that allows synchronization-free concurrency on different files. Hare [19] implements a scalable file system designed for non-cache-coherent multicore processors. The work most similar to our own mitigates contention for shared data structures by running multiple OS instances within virtual machines [7, 36]. In a similar way, MultiLanes [23] and SpanFS [24] create independent virtualized storage devices to eliminate contention for shared resources in the storage stack. Other approaches to OS scaling on multicore systems include reducing OS overhead by collectively managing "many-core" processes [25], tuning the scheduler to optimize use of on-chip memory [6], and even exposing vector interfaces within the OS to more efficiently use parallel hardware [39].

One obvious alternative to data partitioning is the use of fine-grained synchronization, to which many optimizations have been applied. Read-copy update is an approach to improve the performance of shared access to data structures, in particular within the Linux kernel [30]. Both flat combining [16] and remote core locking [28] improve the efficiency of synchronization by assigning particular threads the role of executing all critical sections for a given lock. Specifically in the context of hierarchical data structures, intention locks [15] synchronize access to one branch of a hierarchy, and DomLock [22] makes such locking more efficient. Our approach provides lock-free access to hierarchical structures in the common case, although the locking introduced by Hybrid Waffinity certainly stands to benefit from some of these optimizations. Parallel execution can also be provided via runtime systems that infer task data-independence without explicit data partitions [3, 34].

The database community has long used data partitioning to facilitate parallel and distributed processing of transactions. Many algorithms exist for deploying a scalable database partitioning [1]. The process of defining partitions for optimal performance can also be done on the fly while monitoring workload patterns [20]. The Dora [32] and H-Store [37] models provide data partitions such that operations in a partition can be performed without requiring fine-grained locking. Other work [21, 38] seeks to address the problem of accesses to multiple partitions in a partitioned database. In these proposals, operations with a partition are serialized (and therefore lock-free), but transactions applied to multiple partitions are facilitated through use of a two-phase commit protocol (or similar). In our Hierarchical Waffinity model, we instead superimpose a locking model on top of the partitioned data such that the operation can run safely from within a single partition.

Hierarchical data partitioning has also been explored. In

most cases, the partitioning model is optimized around the presence of a hierarchical computational substrate [8, 9, 10, 12]. In contrast, our work focuses on the hierarchy inherent in the data itself in order to provide ample lock-free parallelism on traditional multicore systems. Similar work has been done to optimize scientific applications by using a hierarchical partitioning of the input data [14].

8 Conclusion

In this paper, we have presented the evolution of the multiprocessor model in WAFL, a high-performance production file system. At each step along the way we have allowed continued multiprocessor scaling without requiring significant changes to the massive and complicated code base. Through this work we have 1) provided a simple data partitioning model to parallelize the majority of file system operations; 2) removed excessive serialization constraints imposed by Classical Waffinity on certain workloads by using hierarchical data partitioning; and 3) implemented a hybrid model based on the targeted use of fine-grained locking within a larger data-partitioned architecture. Our work has resulted in substantial scalability and performance improvements on a variety of critical workloads, while meeting aggressive product release deadlines, and it offers an avenue for continued scaling in the future. We also believe that the techniques discussed in this paper can influence other systems, because the hierarchical model is relevant to any hierarchically structured system and the hybrid model can be employed in insufficiently scalable systems based on partitioning.

References

[1] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the Internal Conference on Management of Data (SIGMOD)*, 2004.

[2] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schupbach, and Akhilesh Signhania. The multikernel: A new OS architecture for scalable multicore systems. In *Proceedings of the Symposium on Operating System Principles (SOSP)*, 2009.

[3] Micah J. Best, Share Mottishaw, Craig Mustard, Mark Roth, Alexandra Federova, and Andrew Brownsword. Synchronization via scheduling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[4] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M. Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In *Proceedings of the Symposium on Operating System Design and Implementation (OSDI)*, 2008.

[5] Silas Boyd-Wickizer, Austin T. Clemens, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nikolai Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the Symposium on Operating System Design and Implementation (OSDI)*, 2010.

[6] Silas Boyd-Wickizer, Robert Morris, and M. Frans Kaashoek. Reinventing scheduling for multicore systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2012.

[7] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transaction on Computer Systems*, 15(4), 1997.

[8] M. Chu, K. Fan, and S. Mahlke. Region-based hierarchical operation partitioning for multicluster processors. In *ACM SIGPLAN Notices*, 2003.

[9] M. Chu, R. Ravindra, and S. Mahlke. Data access partitioning for fine-grain parallelism on multicore architectures. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2007.

[10] D. Clarke, A. Ilic, A. Lastovetsky, and L. Sousa. Hierarchical partitioning algorithm for scientific computing on highly heterogeneous CPU+GPU clusters. In *Proceedings of the European Conference on Parallel and Distributed Computing (Euro-Par)*, 2012.

[11] Peter Denz, Matthew Curtis-Maury, and Vinay 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.

[12] H. Dutta, F. Hannig, and J. Teich. Hierarchical partitioning for piecewise linear algorithms. In *Parallel Computing in Electrical Engineering*, 2006.

[13] John K. Edwards, Daniel Ellard, Craig Everhart, Robert Fair, Eric Hamilton, Andy Kahn, Arkady Kanevsky, James Lentini, Ashish Prakash, Keith A. Smith, and Edward Zayas. FlexVol: flexible, efficient file volume virtualization in WAFL. In *USENIX Annual Technical Conference (ATC)*,

2008.

[14] O. Ergul and L. Gurel. A hierarchical partitioning strategy for an efficient parallelization of the multilevel fast multipole algorithm. In *IEEE Transactions on the and Propagation*, 2009.

[15] J. N. Gray, R. A. Lorie, and G. R. Putzolu. Granularity of locks in a shared data base. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1975.

[16] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2010.

[17] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *USENIX Winter Technical Conference*, 1994.

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

[19] Charles Gruenwald III, Filippo Sironi, M. Frans Kaashoek, and Nickolai Zeldovich. Hare: a file system for non-cache-coherent multicores. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2015.

[20] A. Jindal and J. Dittrich. Relax and let the database do the partitioning online. In *Enabling Real-Time Business Intelligence*, 2012.

[21] Evan P. C. Jones, Daniel J. Abadi, and Sameul Madden. Low overhead concurrency control for partitioned main memory databases. In *Proceedings of the Internal Conference on Management of Data (SIGMOD)*, 2010.

[22] Saurabh Kalikar and Rupesh Nasre. DomLock: A new multi-granularity locking technique for hierarchies. In *Proceedings of the Symposium on Principles and Practices of Parallel Programming (PPoPP)*, 2016.

[23] Junbin Kang, Benlong Zhang, Tianyu Wo, Chunming Hu, and Jinpeng Huai. MultiLanes: Providing virtualized storage for OS-level virtualization on many cores. In *Proceedings of Conference on File and Storage Technologies (FAST)*, 2014.

[24] Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yun, Lian Du, Shuai Ma, and Jinpeng Huai. SpanFS: A scalable file system on fast storage devices. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2015.

[25] Kevin Klues, Barret Rhoden, Andrew Waterman, David Zhu, and Eric Brewer. Processes and resource management in a scalable many-core OS. In *Proceedings of the Workshop on Hot Topics in Parallelism (HotPar)*, 2010.

[26] Charles E. Leiserson. The Cilk++ concurrency platform. In *Proceedings of the Design Automation Conference (DAC)*, 2009.

[27] Min Li, Sudharshan S. Vazhkudai, Ali R. Butt, Fei Meng, Xiaosong Ma, Youngjae Kim, Christian Engelmann, and Galen 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.

[28] Jean-Pierre Lozi, Florian David, Gael Thomas, Julia Lawall, and Gilles Muller. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2012.

[29] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quema, and Alexandra Federova. The Linux scheduler: a decade of wasted cores. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2016.

[30] Paul E. McKenney, Dipankar Sarma, Andrea Arcangeli, Andi Kleen, Orran Krieger, and Rusty Russell. Read copy update. In *Proceedings of the Ottawa Linux Symposium*, 2002.

[31] Changwoo Min, Sanidhya Kashyap, Steffen Maass, Woonhak Kang, and Taesoo Kim. Understanding manycore scalability of file systems. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2016.

[32] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. In *Proceedings of the VLDB Endowment (PVLDB)*, 2010.

[33] Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Stever Kleiman, and Shane Owara. SnapMirror: File system based asynchronous mirroring for disaster recovery. In *Proceedings of Conference on File and Storage Technologies (FAST)*, 2002.

[34] Martin C. Rinard and Monica S. Lam. The design, implementation, and evaluation of Jade. *ACM Transaction on Programming Languages and Systems*, 20(1), 1998.

[35] SPEC SFS (System File Server) benchmark. `www.spec.org/sfs2008`. 2014.

[36] Xiang Song, Haibo Chen, Rong Chen, Yuanxuan Wang, and Binyu Zang. A case for scaling applications to many-core with OS clustering. In *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2011.

[37] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2007.

[38] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the Internal Conference on Management of Data (SIGMOD)*, 2012.

[39] Vijay Vasudevan, David G. Andersen, and Michael Kaminsky. The case for VOS: The vector operating system. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, 2011.

[40] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): The case for a scalable operating system for multicores. *Operating Systems Review*, 43(2), 2009.

[41] Jian Xu and Steven 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.

Copyright notice