# Hybrid Aggregates: Combining SSDs and HDDs in a single storage pool

John D. Strunk
NetApp, Inc.
john.strunk@netapp.com

## ABSTRACT

Relative to traditional hard disk drives (HDDs), solid state drives (SSDs) provide a very large number of I/Os per second, but they have limited capacity. From a cost-effectiveness perspective, SSDs provide significantly better random I/O throughput per dollar than a typical disk, but the capacity provided per dollar spent on SSDs limits them to the most demanding of datasets.

Traditionally, Data ONTAP® storage aggregates have been provisioned using a single type of disk. This restriction limits the cost-effectiveness of the storage pool to that of the underlying disks. The Hybrid Aggregates project within the Advanced Technology Group (ATG) explored the potential to combine multiple disk types within a single aggregate. One of the primary goals of the project was to determine whether a hybrid aggregate, composed of SSDs (for their cost-effective performance) and Serial-ATA (SATA) disks (for their cost-effective capacity), could simultaneously provide better cost/performance and cost/throughput ratios than an all Fibre-Channel (FC) solution.

The project has taken a two-pronged approach to building a prototype system capable of supporting hybrid aggregates. The first part of the project investigated the changes necessary for Data ONTAP RAID and WAFL® layers to support a hybrid aggregate. This included propagating disk-type information to WAFL, modifying WAFL to support the allocation of blocks from a particular storage class (i.e., disk type), and repurposing the existing write-after-read and segment-cleaning infrastructure to support the movement of data between storage classes. The second part of the project examined potential policies for allocating and moving data between storage classes within a hybrid aggregate. Through proper policies, it is possible to automatically segregate the data within the aggregate such that the SSD-backed portion of the aggregate absorbs a large fraction of the I/O requests, leaving the SATA disks to contribute capacity for colder data.

This paper describes the implementation of the Hybrid Aggregates prototype and the policies for automatic data placement and movement that have been evaluated. It also presents some performance results from the prototype system.

## Categories and Subject Descriptors

D.4.2 [**Storage Management**]: Storage hierarchies.



## General Terms

Algorithms, Management, Performance

## 1. INTRODUCTION

The appearance of NAND flash devices in enterprise storage systems presents an opportunity to cost-effectively provide a large number of IOPS for stored data. There are several locations within the storage stack where flash might be deployed. The approach used in this project is to use flash in the form of solid-state drives (SSDs) that are a physical drop-in replacement for traditional hard disk drives. SSDs can service I/O requests at a much higher rate than traditional disk drives because they have no moving parts, making the data access latency very small.

SSDs' low latency provides the potential for large improvements in storage system performance. An SSD not only provides lower latency and higher I/Os per second than a traditional disk in absolute terms, it does so using a dollar-cost metric as well. On a dollars-per-I/O-per-second metric, SSDs are approximately ten times less expensive than traditional disk-based storage. Unfortunately, their capacity is rather limited. They are approximately ten times more expensive than disks in terms of dollars per GB.

The high cost of SSD-based storage on a capacity basis severely limits the situations where it can be cost-effectively applied by itself. Only the most demanding applications, with very high I/O rates or low latency requirements would benefit from being placed solely on SSD media.

The majority of datasets contain a spectrum of access characteristics. There are small portions that are frequently accessed while large portions are rarely touched. In these configurations, it would be advantageous to be able to use the cheap performance of SSDs to satisfy the I/O requirements and use the cheap capacity of HDDs to hold the bulk of the data.

The Hybrid Aggregates ATG project was initiated to investigate the feasibility of automatically tiering SSDs and high capacity (SATA) HDDs within Data ONTAP to create a storage solution that is more cost-effective than a traditional, all-FC disk solution.

## 2. BACKGROUND

This work was carried out in the 2008 – 2009 timeframe as an investigation into one of the many ways that NetApp could integrate flash-based storage into its products. At the time, the perception was that enterprise-grade SSDs were very expensive and consumer-grade SSDs had poor performance. As a result, the ability to provide cost-effective solutions with SSDs in an enterprise environment was still an open question. The results of this project fed into the development effort that yielded Flash Pools [11] in Data ONTAP 8.1.1.

The notion of moving data between tiers of storage based on access characteristics has a long history [3, 4]. The AutoRAID project [13] dynamically moved data between different RAID levels depending on access characteristics. The Conquest system [12]

used persistent RAM plus disk to create a file system. It statically allocated files into a tier based on size.

Around the time the Hybrid Aggregates project was ongoing, Narayanan, et al. [10] showed that SSDs were not an obvious replacement for disks and that care must be taken in how they are deployed in order for them to provide a cost-effective solution. The work highlighted the need to be selective about what data should be placed on SSDs, providing evidence that good allocation and movement policies would be the key to creating cost-effective solutions.

The work by Guerra, et al. [8] examined dynamically assigning data to HDD-based or SSD-based storage depending on the workload's characteristics. Their work examined both an initial provisioning scenario as well as optimization in a running system. It used 64 MB extents as the granularity of data movement to minimize the amount of metadata overhead. The hybrid aggregate prototype is able to use 4 kB blocks as the unit of data movement without adding additional metadata since it integrates with the WAFL file system and can re-use those existing structures.

The Hystor system [2] is probably the most similar to Hybrid Aggregates in its choice of policies for data migration. Hystor uses a metric for moving data where the desire to place data onto the SSD is proportional to its access frequency, and inversely proportional to the I/O size. Their reasoning in creating this metric was very similar to ours—a desire to service as many I/Os as possible from the SSD but tempered by the diminished benefit of SSDs (from a cost-performance perspective) as the I/O size increases.

# 3. DESIGN AND IMPLEMENTATION

High capacity HDDs and SSDs have very different performance characteristics and cost structure. The HDDs excel at providing inexpensive capacity, but they have relatively poor random I/O capabilities. SSDs provide the complement to this—good random I/O performance—but the capacity that they provide is expensive. Capturing the potential value from an HDD/SSD hybrid aggregate relies on being able to partition the data and workload, directing each to the proper class of storage.

The Hybrid Aggregates project seeks to build a single storage aggregate that is composed of a combination of HDD and SSD storage. Both types of storage can serve as the permanent location for data (i.e., the capacity of both the HDDs and SSDs contribute to the physical space of the aggregate). The two classes of storage are separated at the granularity of a RAID group. Therefore, a hybrid aggregate has at least two RAID groups, with at least one constructed from HDDs and at least one from SSDs.

The main contributions of the project come from three sources. First, it was necessary to determine how the storage class information (i.e., the type of device) would be passed to WAFL. Natively, the SSD looks just like a disk drive, but WAFL needs to know the difference so that allocations can be properly directed. Second, it was necessary to modify existing mechanisms for data movement (write after read and segment cleaning) and placement (write allocation) to allow the migration and initial placement of data onto a particular class of storage. Third, policies had to be developed to determine when and what data should be moved between classes.

## 3.1 Design choices

In constructing a hybrid aggregate prototype, it was desirable to minimize the number and size of changes to existing WAFL [6, 9] and Data ONTAP code. To this end, a number of key design choices guided the prototyping effort.

- Use existing mechanisms as much as possible for data movement and placement.
- Avoid changes to on-disk structures.
- Allow a wide spectrum of relative SSD to HDD capacities, but primarily target configurations with SSD capacity that is less than 10% of the total aggregate capacity.
- Avoid data structures that grow with the storage capacity.
- Permit flexible policies for data placement.

The first two objectives are a direct result of wanting to minimize the changes to WAFL and Data ONTAP as well as to increase the likelihood that it will be accepted for productization. The third and fourth objectives stem from the belief that the capacity of SSDs will increase significantly with subsequent generations. Additionally, it should be possible to tailor the ratio of SSD and HDD capacity to fit a customer's dataset and workload since this will be critical to making hybrid aggregates a cost-effective alternative to high performance disks. The final objective is an acknowledgement that creating a good data placement policy is likely to be difficult and that it may need to be tailored to the needs of specific customers' workloads.

## 3.2 Changes necessary to support a hybrid aggregate

The Hybrid Aggregates prototype is a modified version of Data ONTAP, taken from the development codeline of Data ONTAP 8. The changes required to create a hybrid aggregate focused on making the device class (SSD or HDD) information available to WAFL as well as allowing WAFL to direct writes to a particular class of storage.

Since SSDs are a plug-in replacement to HDDs, they are recognized at the disk and RAID level in the same way as traditional disks. To designate a device as an SSD, a special disk RPM was defined, and the RAID code was modified to recognize devices with this RPM as SSDs. An SSD designation was added as a flag to the disk and RAID group structures, allowing an individual RAID group to be recognized as an SSD or HDD RAID group, based on its constituent devices.

When WAFL needs information about the disks that comprise an aggregate, it acquires this information via a RAID topology map that is constructed based on the RAID groups. This topology map was also augmented with an SSD flag, allowing WAFL to determine the underlying storage class of a given physical block. The ability to classify blocks forms the foundation of hybrid aggregates in WAFL.

During write allocation, the class of the current write disk (the disk whose blocks are being allocated) is compared to the desired class of storage for the buffer being allocated. If these do not match, a new current write disk is chosen. The process of choosing a new disk for write allocation begins with the selection of a RAID group, providing the opportunity to switch to a different type of RAID group. The RAID group selection uses the previously mentioned flags to preferentially choose a RAID group of the block's desired class. If that is not possible, the selection will fall back to any available RAID group. In this way, WAFL's existing write allocation code can be used to direct buffers to either SSD or HDD-based storage. A side-effect of this approach is that when switching from one class to the other and back again (e.g., HDD to SSD and back), the allocator will not (typically) resume

allocating from the previous disk, hurting the layout's sequentiality.

There is an ongoing effort to redesign WAFL's write allocation code to increase its modularity and allow improved write allocation policies. The new design works on the concept that an allocation policy would request a "bucket" of free blocks of a particular type and assign buffers to those buckets. This new model fits nicely with the needs of Hybrid Aggregates by potentially allowing an allocation policy to hold two different buckets, one of SSD-based storage and the other HDD-based. In this way, the choice of storage class for a buffer is nicely partitioned from the RAID group selection code whose original purpose was to efficiently spread writes across all of the disks in the system.

## 3.3 Data movement mechanisms

Effective implementation of a hybrid aggregate requires the ability to move data between the two classes of storage. Moving data during write allocation (i.e., moving dirty data) is accomplished by using the modified write allocator, as described previously, to choose a new physical block that resides on the target class of storage. It is desirable to be able to move data not just when it is being written by a client but also when it is being read. Additionally, to manage the capacity constraints of the SSDs, it is necessary to be able to move cold data (i.e., data that is not being accessed at all) to the HDDs. This section discusses how a hybrid aggregate uses existing data movement mechanisms within WAFL to migrate data between SSDs and HDDs.

### *3.3.1 Data migration to SSD*

To get the most benefit from a hybrid aggregate, it is expected that data placement policies will attempt to place frequently accessed data onto the SSDs. In the case where this data is being written by the foreground workload, the write allocation path can be used to move the data as it is cleaned during a Consistency Point (CP) [9]. For data that is being read but not modified, another technique is required. To move data that is being read, a hybrid aggregate uses a write-after-read (WAR) operation, wherein a block is marked dirty when it is read, causing it to be scheduled for write allocation during a subsequent CP. The ability to perform a WAR operation is a pre-existing capability of WAFL that has primarily been used for extent reallocation (data defragmentation).

### *3.3.2 Data eviction from SSD*

The SSD portion of the aggregate has a limited capacity, and steps must be taken to ensure that there is sufficient free capacity in the SSD RAID groups to accept new block writes. To increase the amount of free space, cold data is migrated from the SSDs to the HDDs as necessary. This migration is primarily triggered by the capacity utilization of the SSDs reaching some predefined threshold. Additionally, the data that is to be moved is presumed to be cold. This makes the standard move at write-time or WAR operations unsuitable for this task. Instead, segment cleaning, another existing WAFL mechanism, is used to migrate the data.

There are several benefits to using segment cleaning. First, it works at a block level. Specific blocks (i.e., the blocks that cover the SSD portion of the aggregate) can be directly targeted, and it is not necessary to know the contents of the blocks that are being moved. The segment cleaner identifies them, as necessary. Second, segment cleaning works on a group of disk blocks that compose a number of full RAID stripes. This creates a region of free space within the SSD RAID group that will be efficient to allocate in the future because RAID will have the ability to perform full-stripe writes.

## 3.4 Data movement policies

The previous section focused on the mechanisms used to place data into, or to move data between, particular classes of storage. This section discusses the data movement policies that determine which blocks to move and when to move them.

### *3.4.1 Policy API*

Separating the choice of data placement policy from the mechanisms that implement it allows multiple policies to co-exist within the system. This versatility has benefits not just for easy development, but it also allows the particular policy to be matched to a customer's workload requirements and access pattern. Since policies are concerned only with the movement of data, they are not required to maintain any state about the current data distribution, and they are also not directly involved in serving data. This independence allows policies to be stopped, started, and switched "on the fly" in the system with little effect on the client workload.

Although the current prototype permits only one active policy for the entire system, having one policy per aggregate would be a relatively minor extension. Since the SSDs are shared across all data within an aggregate, having one policy per aggregate would better match the policy granularity with the granularity of resource sharing. This allows the policy to remain as the central resource arbiter. Within a given policy, however, blocks from different volumes or files may be treated differently. For example, a policy may allow an administrator to "lock" a volume into the SSD and have the blocks from other volumes to dynamically move in and out automatically.

Each policy has three opportunities to manipulate (e.g., move) blocks.

- First, the policy module is notified each time a block is read from storage. Currently, this is limited to non-speculative I/O requests. When notified of a read, the policy module has the opportunity to examine the buffer and related data structures (e.g., the buffer header, inode, etc.) and determine if it should be moved via a WAR operation.
- Second, during write allocation, the policy has the opportunity to examine each buffer to determine where it should be placed.
- Third, the policy module controls the invocation of the segment cleaner to move data out of SSD. It has the choice of which segment to move as well as when and how aggressively to move data.

To communicate policy decisions to the code that implements the data placement and movement, an additional field was added to the in-memory buffer header structure. This new field indicates the desired class of storage for the data. It can have three main values: "unknown", "SSD", and "HDD". The "SSD" and "HDD" values indicate that the block should be placed on that class of storage, while the "unknown" value implies that a decision has not been made about the block or that no action is requested by the policy module.

### *3.4.2 Metadata in SSD*

The most basic policy examined for Hybrid Aggregates was to place all metadata in the SSDs and leave normal data on the HDD-based storage. The motivation for this policy is that metadata I/O is typically small and random, and it is usually accessed in the critical path of performing I/O on other blocks. These characteristics imply that it is a good candidate for placement in SSD-based storage.

The "MD-in-SSD" policy is implemented using only the policy module's hooks into the write path. When a block is being write allocated, the buffer is examined to determine whether it is a file system indirect block or whether it belongs to one of the metafiles (e.g., the inode file or block allocation bitmap). In the case of a positive result, it is assigned to the SSD portion of the aggregate.

This policy has the benefit of being extremely simple, but as the benchmarks will show, its benefits are severely limited. The first drawback of the MD-in-SSD policy is that there is a fixed set of blocks that are assigned to the SSDs. Any extra capacity goes unused. The other important drawback is that this policy tends to significantly under-utilize the I/O capabilities of the SSDs. Only in very extreme situations is the presented I/O load a reasonable fraction of what the SSDs can support. This discrepancy means that a significant fraction of the capabilities of the SSDs are wasted with this policy.

Although this policy is useful as a baseline against which to compare, it is also useful as a building block for other policies. Having metadata resident in the SSDs aids segment cleaning by speeding the accesses necessary to update the block pointers during redirection. It also naturally handles the case where metadata (typically the first level of indirect blocks) does not fit in the filer's DRAM cache, and it accomplishes this using relatively little capacity.

### *3.4.3 Hot, random read detection*

SSDs excel at providing fast access for small, random I/O requests. Their main advantage comes from the lack of initial positioning time for each new I/O. Sequential transfer is approximately the same as traditional HDDs, making the HDDs considerably less expensive when looking at a dollars-per-MB/s comparison for reasonably sized requests. To get the most benefit from the SSDs, this policy attempts to place data that is frequently, randomly read into the SSDs. Assuming the presence of temporal locality in the disk access stream, it would be reasonable to conclude that a block that has been randomly read "recently" will be randomly read in the near future, and that the more accesses that have been seen to that block in some window of time, the more likely there will be future requests to that block. The "hot, random read detection" policy is designed to track random read accesses to blocks and move the block to the SSD portion of the aggregate once some threshold number of accesses has been met. Setting the threshold higher decreases the rate at which blocks are moved into the SSDs with the intent of only moving the hottest blocks. There are different thresholds for moving a block at read time versus at write time (although the algorithm still only counts read accesses to reach the threshold). The reason for having a separate threshold to move a block at write time is that the movement is less expensive since it must be written anyway. This implies a "move on write" threshold that is less than the "move on read" threshold. Typical values for these thresholds are two accesses for write and three for read. This implies that once a block has been read twice, the next access (read or write) will cause it to be moved to the SSDs.

An efficient data structure is needed to count the number of accesses to blocks that are read. For this task, the policy uses a Spectral Bloom filter [5]. Bloom filters [1] are a memory-efficient data structure designed to probabilistically track set membership (the set of blocks that have been randomly read), and they function by hashing each item to a set of bits within a large bit vector. To insert an entry, the corresponding bits are set. An item is considered a member of the set only if all of its corresponding bits are set. Counting Bloom filters [7] are an extension that allows the multiplicity of members to be tracked. Multiplicity is tracked by maintaining a counter instead of a single bit at each location within the vector. The multiplicity can then be estimated as the minimum value of the counters to which the item hashes. Spectral Bloom filters are a further enhancement that allows membership to be tracked by its frequency of insertion. This is accomplished by decaying the multiplicity of set members periodically (i.e., all counters are periodically decreased, typically via a division by 2). The result is a data structure that tracks the number of times that a block has been accessed (from HDD) over a decaying window of time with some small expected rate of error (the block may appear to have been accessed more frequently than it has). The filter allows trading off the memory consumption of the data structure, the period of decay (the window size over which to count accesses), and the expected error rate. A typical set of parameters used in the prototype consumes 30 MB of memory, has a window size of 14 million HDD reads, and has an expected error rate of 1.3%. This provides the ability to track up to three accesses for a given block, and amounts to using 2.1 bytes of RAM per I/O that is tracked. In the experiments, this policy is referred to as "SBloom" after its implementation.

### *3.4.4 Tiering via run-length*

A third policy that was implemented is closely related to the hot, random read policy previously described. This policy, "run-length," begins by ensuring all metadata is placed in the SSDs; then it places regular data into the SSDs based on the size of the request. All reads of length less than some threshold number of blocks are moved via WAR to SSD. For this policy, the run-length of a request is determined by the number of consecutive blocks in the file block number (FBN) space that comprise the I/O request. The FBN run-length does not imply disk layout sequentiality. This policy looks for randomness in the FBN space because blocks that merely have a poor layout on disk (sequential FBN but random physical block number) could be optimized by existing mechanisms such as read reallocation (defragmentation).

Since the majority of the benefit from having a block in SSD is the elimination of the initial positioning latency, the run-length of an I/O is a reasonable approximation of the per-access benefit of placing it in SSD. In more concrete terms, assuming that streaming bandwidth from HDD and SSD are approximately the same, the entire response time benefit is captured at the start of the I/O. If this is examined as the average latency saved per block read, it is possible to see that the benefit from SSD drops off as the inverse of the I/O size. A two block read benefits half as much as a single block read. An alternate interpretation is to examine the SSD capacity consumed to "capture" an I/O. The motivation for this view is that the policy should be directing as many I/Os to the SSDs as possible, lightening the load on the HDD portion of the aggregate. Here again, it can be seen that two-block I/Os require twice the capacity to satisfy a given number of I/O requests.

The run-length policy moves data on its first access (that is below the given threshold). As a consequence, it does not need to maintain state like the "SBloom" policy does. Additionally, it adapts much more quickly because it does not need to wait until the block has been seen multiple times. Although this approach of moving data on first access raises the possibility of polluting the SSD capacity with blocks that are accessed only once, the rate of pollution is considerably limited by the poor random I/O performance of the HDDs—to move a block to SSD, it must be read from HDD, and a low run-length threshold ensures these requests amount to small, random I/Os.

### 3.4.5 Capacity-based data eviction

The policies discussed above are concerned with the movement of data into the SSD portion of the aggregate. Because the SSD capacity is a finite resource, the flow of blocks into SSD must be balanced by the movement of data from SSD to HDD. The Hybrid Aggregates prototype implements capacity-based data eviction from the SSDs, targeting a specific capacity utilization for the SSD RAID group.

The policy module tracks the percent utilization of the SSD RAID groups and triggers segment cleaning as necessary to ensure adequate free space is maintained. The rate of segment cleaning is controlled by a proportional control system that modulates the number of outstanding segment cleaning requests as a function of a target capacity utilization and a gain parameter. These parameters are typically set to begin cleaning when the SSDs reach 85% utilization and increase the cleaning rate up to a maximum of five outstanding segment requests once the utilization has climbed to 95%. Using this linear control, the effect on the foreground workload is kept to a minimum by only issuing as many cleaning requests as necessary to keep the utilization constant.

Complementary to the rate is the choice of which segments to clean. The current prototype chooses SSD segments randomly. This random data eviction policy was chosen for its ease of implementation, low resource usage, and robustness. One of the guiding principles of the design was to avoid data structures that grow in proportion to the size of the storage. Implementing a more traditional eviction policy such as an LRU or clock algorithm would have violated this principle.

Even with fairly aggressive segment cleaning, it is still possible to move data into the SSDs faster than it can be cleaned. To address this, the policy module also throttles HDD to SSD data movement based on the capacity utilization. The prototype probabilistically rejects data migration requests as the capacity utilization climbs from 85% to 95%. This, again, is implemented using linear, proportional control.

# 4. EVALUATION

This section examines the performance of the hybrid aggregate prototype by comparing the performance of an all HDD aggregate to that of a hybrid aggregate. The experiments were performed using a NetApp FAS3070. The root volume was stored on an additional aggregate that was not used for testing. The aggregate used for testing was composed of 14 500 GB SATA HDDs in a DS14mk2-AT shelf configured as two RAID-4 groups of seven disks each. The experiments that used a hybrid aggregate added either three or five 146 GB Fibre Channel SSDs configured as a single RAID-4 group in a DS14mk4 shelf.

## 4.1 SPC-1-like workload

The SPC-1-like workload benchmark simulates a transaction processing database workload. The benchmark dataset has three components (ASUs), two of which are accessed (mostly) randomly, and the third is only accessed via sequential writes. The random access to the first two ASUs is concentrated to a relatively small area, making this benchmark a prime candidate for a hybrid aggregate.

In this test, an all HDD aggregate is compared to a 3 SSD hybrid aggregate using various data placement policies. The test uses a 3 TB dataset spread across 100 FlexVol® volumes in the single aggregate, and the workload is generated using a Java™-based workload generator. The benchmark was run for 36 hours to allow the hybrid aggregate placement algorithms time to detect and move hot data. Data access was provided via NFS to a single Linux client. Once this initial sustained load was finished, the average latency was measured for different I/O loads.

Figure 1 shows the result of running the benchmark on the hybrid aggregate prototype. The three previously discussed data placement/movement policies are compared against the HDD-only configuration. From this graph, it can be seen that the run-length policy has the best performance of the candidates, both at providing low latency at low loads and at maximizing the potential throughput. At the 10% load level, the run-length policy had an average latency of 1.9 ms, while the HDD-only configuration's latency was 4.9 ms. Additionally, the maximum throughput for the hybrid aggregate was 2.5 times that of the HDD-only configuration.

Examining the fraction of disk block requests that are directed to each class of storage provides an indication of how the performance improvement was achieved. As shown in Table 1, the run-length policy is able to serve 73% of disk block reads from the SSD portion of the aggregate. Additionally, the SSDs capture 51% of the write traffic. The policy also makes full use of the SSDs' available capacity, up to the cleaning threshold of 85%.

## 4.2 Exponentially distributed I/O

In typical workloads, I/O requests are not distributed evenly across the entire dataset. Instead, some small fraction of the blocks receive a large portion of accesses. An effective hybrid aggregate policy is able to detect which blocks are frequently accessed (i.e., hot) and move those to SSD for good performance while leaving the cold data on the less expensive ($/GB) HDD-based storage.

This set of tests used filersio (a workload generation tool that is built into Data ONTAP) to generate a workload of random 4 kB I/O requests using a 50/50 read/write mix and concurrency level of 30 simultaneous requests. The I/O requests were skewed, using filersio's built-in exponential workload such that a large fraction of the requests went to a small region of the test dataset. For this experiment, 14 500 GB SATA HDDs were used in combination with five 146 GB SSDs. A 4000 GB FlexVol was used to contain a single 2500 GB test file.

Figure 2 shows the system throughput over time as the policies adapt to an I/O access pattern where 95% of the I/O requests access 5% of the dataset. The all HDD case is shown as a baseline which produces a relatively constant 2000 IO/s. The MD-in-SSD policy shows minimal improvement because the metadata for the frequently accessed portion of the dataset fits in the system's cache. The Spectral Bloom filter-based policy adapts slowly because it requires several accesses to each block. The best policy is the run-length policy, that achieves approximately 10000 IO/s— a 5x improvement over the base case.

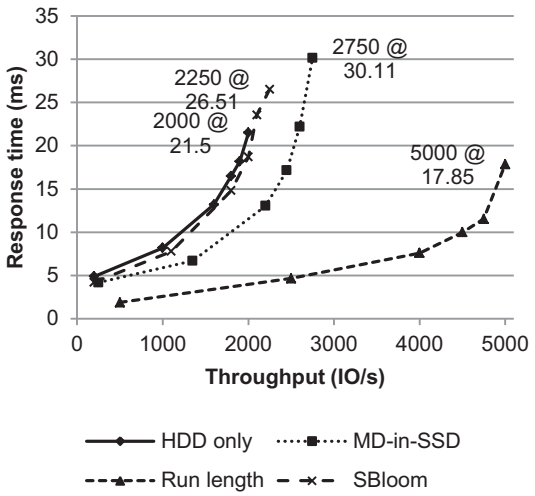


**Figure 1 – Throughput vs. response time for an SPC-1-like workload.** This graph compares the response times as a function of throughput for different hybrid aggregate data placement policies. Table 1 shows the percentage of reads and writes that were serviced by the SSD portion of the aggregate as well as the fraction of the SSD capacity that was used.

**Table 1 – Fraction of storage read and write requests serviced by the SSD portion of the aggregate including fraction of the SSDs' capacity used during the 95/5 exponential benchmark.**

| | Read requests | Write requests | Capacity utilization |
|---|---|---|---|
| MD-in-SSD | 16% | 36% | 14% |
| Run length | 73% | 51% | 87% |
| SBloom | 13% | 11% | 7% |

Changing the access distribution to one that is less concentrated (i.e., 80% of I/O requests to 20% of the dataset), results in less improvement because of the lower SSD hit rate. These results are shown in Figure 3. In this case, the MD-in-SSD policy does show some improvement because the number of indirect blocks that are accessed is larger, making them less cacheable. The Spectral Bloom filter performs poorly because the window of accesses in which it looks for frequently accessed blocks is not large enough to effectively detect the hot blocks. The run-length policy is, again, the best performing policy, showing a throughput improvement of approximately 2x.

To further investigate the performance of the run length policy, this 80/20 exponential workload was run for an extended period of time. Figure 4 shows the policy's performance over approximately 9 days. This graph also shows the rate at which blocks are moved between the SSD and HDD portion of the aggregate. At the start of the benchmark, only the metadata resides in the SSDs.

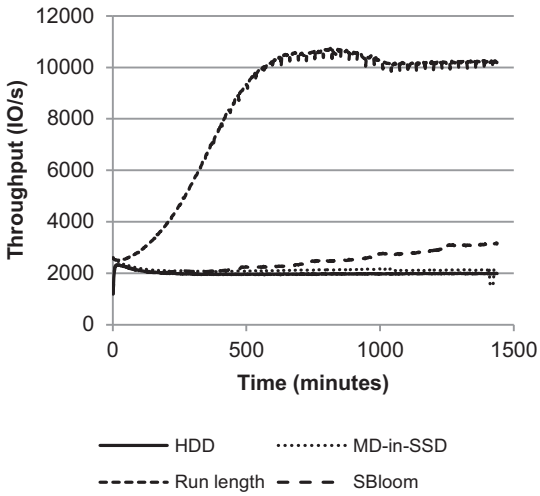


**Figure 2 – Filersio with 95/5 exponential distribution**. This graph shows the throughput achieved by filersio as a function of time for different hybrid aggregate policies with 95% of I/O concentrated in 5% of the dataset.

Initially, data is quickly moved to the SSDs until they reach their target capacity utilization. At approximately 29 hours, the SSDs reached capacity (85% utilization) and data began to be migrated back to the HDDs to maintain this level. For the remainder of the benchmark, the rate of movement into and out of the SSDs was approximately equal in order to hold the capacity utilization constant. The percent of read and write I/O requests that were satisfied by the SSDs is also shown in the graph. Both the read and write SSD hit rates stabilize near 85%.

# 5. CONCLUSION

Tiering storage via a hybrid aggregate shows promise for increasing the cost-effectiveness of a storage system. For workloads with significant hot spots, SSDs can satisfy the I/O demands while using cheaper HDDs for the bulk of the data. The correct choice of placement policy is critical to achieving the cost benefits of the SSDs, and this work identified a relatively simple policy based on the run-length of requests that is able to automatically take advantage of the SSDs' capabilities.

The changes required to Data ONTAP to support the hybrid configuration were relatively minimal because it was possible to leverage existing data placement and movement mechanisms. Using the existing metadata structures of WAFL avoided introducing any additional structures that consume large amounts of memory. Additionally, since the flash-based data is stored as part of the normal physical space and tracked by WAFL, it naturally survives filer crashes, reboots, and takeover in the same way as an all HDD-based aggregate.

# 6. REFERENCES

[1] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 130 (7):0 422-426, July 1970. ISSN 0001-0782. 10.1145/362686.362692.

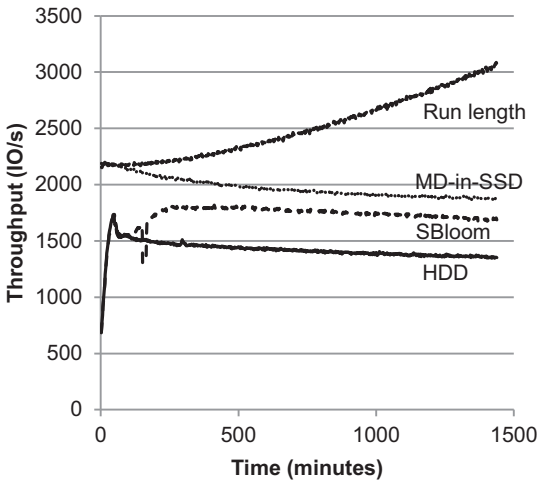


**Figure 3 – Filersio with 80/20 exponential distribution.** This graph shows the throughput achieved by filersio as a function of time for different hybrid aggregate policies when 80% of the I/O is concentrated in 20% of the dataset.

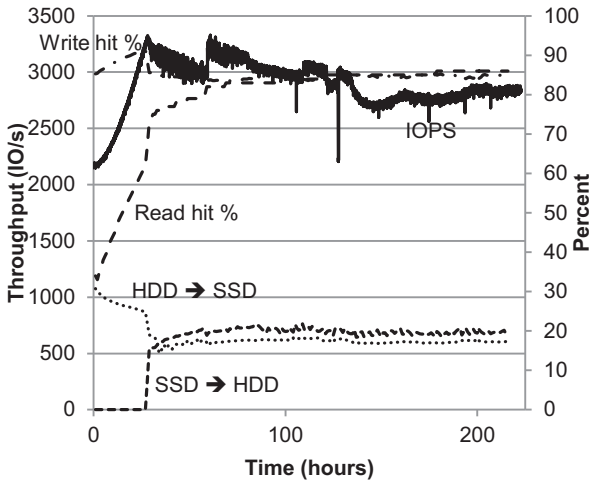


**Figure 4 – Extended run of 80/20 filersio with run-length policy.** This graph shows a longer term picture of how the run-length policy adapts to the workload.


[2] Feng Chen, David Koufaty, and Xiaodong Zhang. Hystor: Making the best use of solid state drives in high performance storage systems. In *International Conference on Supercomputing (ICS)*, pages 22-32. ACM Press, 2011. ISBN 978-1-4503-0102-2. 10.1145/1995896.1995902.

[3] Peter P. S. Chen. Optimal file allocation in multi-level storage systems. In *National Computer Conference and Exposition*, pages 277-282. ACM Press, 1973. 10.1145/1499586.1499662.

[4] E. I. Cohen, G. M. King, and J. T. Brady. Storage hierarchies. *IBM Systems Journal*, 280 (1):0 62-76, March 1989. ISSN 0018-8670. 10.1147/sj.281.0062.

[5] Saar Cohen and Yossi Matias. Spectral bloom filters. In *International Conference on Management of Data (SIGMOD)*, pages 241-252. ACM Press, 2003. ISBN 1-58113-634-X. 10.1145/872757.872787.

[6] 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*, pages 129-142. USENIX Association, 2008.

[7] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary Cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 80 (3):0 281-293, June 2000. ISSN 1063-6692. 10.1109/90.851975.

[8] Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. Cost effective storage using extent based dynamic tiering. In *Conference on File and Storage Technologies (FAST)*. USENIX Association, 2011.

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

[10] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating server storage to SSDs: Analysis of tradeoffs. In *European Systems Conference (EuroSys)*, pages 145-158. ACM Press, 2009. ISBN 978-1-60558-482-9. 10.1145/1519065.1519081.

[11] Paul Updike. Flash Pool design and implementation guide. Technical Report TR-4070, NetApp Inc., May 2012. URL http://media.netapp.com/documents/tr-4070.pdf.

[12] An-I Andy Wang, Geoff Kuenning, Peter Reiher, and Gerald Popek. The Conquest file system: Better performance through a disk/persistent-RAM hybrid design. *ACM Transactions on Storage*, 20 (3):0 309-348, August 2006. ISSN 1553-3077. 10.1145/1168910.1168914.

[13] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The *HP* AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 140 (1):0 108-136, February 1996.