

MorphStore: A Local File System for Big Data with Utility-driven Replication and Load-adaptive Access Scheduling

Eric Villaseñor, Timothy Pritchett, Jagadeesh M. Dyaberi*, Vijay S. Pai, Mithuna Thottethodi

*Electrical and Computer Engineering
Purdue University West Lafayette, IN
{evillase, tpritch, vpai, mithuna}@purdue.edu*

**Saavn, LLC
Mountain View, CA
jd@saavn.com*

Abstract—File system performance is critical for overall performance of Big Data workloads. Typically, Big Data file systems consist of dual layers; a local node-level file system and a global file system. This paper presents the design and implementation of MorphStore, a local file system design that significantly improves performance when accessing large files by using two key innovations. First, MorphStore uses a load-adaptive I/O access scheduling technique that dynamically achieves the benefits of striping at low load and the throughput benefits of replication at high loads. Second, MorphStore uses a utility-driven replication to maximize the utility of replication capacity by allocating replication capacity to popular read-mostly files. Experiments reveal that MorphStore achieves 8% to 12% higher throughput while using significantly less replication for workloads that access large files. If we consider the performance-capacity tradeoff of file systems built on static techniques such as JBOD, RAID-0 and RAID-1, MorphStore extends the Pareto frontier to achieve better performance at the same replication capacity.

Keywords—Big Data; File-systems; Replication;

I. INTRODUCTION

Disk performance remains a key performance bottleneck for a large and important class of applications. While disk capacity has increased tremendously, disk access latency has only been improving at about 15% per year [1]. Well-known disk performance optimizations such as striping, replication and combinations thereof have targeted this bottleneck [2], [3], [4], [5]. Unfortunately, existing approaches employ a static one-size-fits-all approach wherein entire storage systems are statically striped and/or replicated. Because striping and replication target specific and disjoint types of parallelism, such a static “one-size-fits-all” approach has drawbacks. For example, while replication is beneficial for workloads that are disk-read-intensive, replication does not result in any performance improvement under low loads because disk accesses do not benefit from replication. Further, replication results in poor write performance, especially at high loads. On the other hand, while striping is useful at low

disk loads, striping may be counter-productive at high disk loads since striped disk-accesses (which occupy all disks) have to be serialized. In a non-striped multidisk environment, it may be possible to achieve better performance accessing disks in parallel (depending on file placement).

From these observations, ideal performance requires (a) striped disk accesses at low loads (b) replication of files that are accessed in a read-intensive way, especially at high-loads and (c) non-replication of write-intensive files. This paper presents MorphStore – a file system architecture that automatically achieves all the above design goals.

There are two key innovations in MorphStore.

First, MorphStore achieves load-adaptive disk access scheduling to achieve the best of both striping and replication. MorphStore maximizes throughput at high loads by exploiting inter-request parallelism by assigning requests to parallel disks, thus mimicking replication. However, at low loads, MorphStore uses replicated data to achieve striping (i.e., intra-request parallelism) by exploiting the fact that large disk transfers may be broken down to smaller transfers from different disks. Such a strategy allows an arbitrary striping degree that is limited only by the number of replicas. MorphStore uses a simple, yet highly effective, history-based load prediction mechanism that directly drives the choice of striping vs single-disk access.

Second, MorphStore employs a utility-based replication technique that treats replication capacity as a resource that must be used to maximize its utility. At a high-level our replication strategy is based on the conventional wisdom that replicating read-intensive files has positive utility (i.e., benefits performance) and that replicating write-intensive files has negative utility (i.e., hurts performance). However, the conventional wisdom does not address several concrete questions such as (a) which files to replicate, (b) how many replicas are to be created for each file. We develop a replication strategy that is guided by access statistics (from prior profile runs) to answer both of the questions. We use profile-guided, utility-driven replication assuming

that the profile access pattern is a reasonable expectation of future access patterns. In general, MorphStore may use any other source and is not limited to the use of the most recent profile (or any profile for that matter). Note, MorphStore only concerns itself with the degree of replication. It assumes random placement of replicas for load balancing. Because randomization reduces the probability of pathological problems (e.g., correlated files being placed on the same disk), MorphStore does not attempt to address the replica placement issue.

We have implemented the MorphStore architecture in a Linux environment with the ext2 file system serving as the code-base on which MorphStore is implemented. In our implementation, MorphStore maintains replication meta-state in the inode’s extended attributes.

We evaluate MorphStore using a combination of micro- and macro-benchmarks. Our microbenchmark-based evaluations highlight the clear advantage of MorphStore’s load-adaptivity; MorphStore is consistently closer to the best of either replication-only or striping-only strategies. Our macro-benchmark based evaluations show that MorphStore uses significantly less replication capacity than RAID-1 while still achieving 12% average performance improvements on a videosever workload and 8% average performance improvement on a NoSQL workload.

In summary, the major contributions of this paper are:

- We develop a storage architecture – MorphStore – that achieves load-adaptive performance improvements for disk-bound applications.
- MorphStore uses a profile/expectation-guided optimal utility-based replication strategy that maximizes performance by selectively replicating data within a given amount of replication capacity.
- MorphStore performs closer to the best of static replication/striping strategies.

The remainder of this paper is organized as follows: Section II offers a brief background on disk and file system organization. Section III and Section IV describe the design and implementation of MorphStore, respectively. Section V describes our experimental methodology. Section VI presents our experimental results. Finally, Section VII concludes this paper.

II. BACKGROUND AND RELATED WORK

Block-level disk-array organization: Block-level devices can be organized as a unit via a technique known as Redundant Arrays of Inexpensive Disks [6], otherwise known as RAID. There exist a number of RAID variants that target different levels of the reliability/performance tradeoff. While reliability is one of the key motivations behind RAID organizations, we focus on the performance impact of disk array organization. Specifically, we consider JBOD, RAID-0 and RAID-1 organizations. The key tradeoffs are as follows. In the presence of inter-request parallelism (i.e., abundant

requests) JBOD and RAID-1 are likely to perform well because independent requests can be handled independently on separate disks. While JBOD may be subject to disk conflicts (two independent accesses that happen to access blocks on the same disk), which result in request serialization, RAID-1 is more reliably able to exploit inter-request parallelism of read requests because all files are present on all disk-mirrors. Note, RAID-1’s advantage has an associated cost in terms of (1) capacity overhead of mirroring, and (2) the performance overhead of writing to all mirrored copies on each write.

RAID-0, on the other hand focuses solely on intra-request parallelism by using striping. Inter-request parallelism is not exploited. At low loads, when inter-request parallelism is also low, RAID-0 outperforms JBOD and RAID-1. However, RAID-0 incurs a performance penalty at high loads because it incurs the cost of intra-request parallelism (i.e., striping incurs seek overhead on all disks, which increases disk occupancy) and ignores the abundant inter-request parallelism.

MorphStore differs from the above static mechanisms in two key ways. First, MorphStore moves away from the one-size-fits-all policies; instead relying on (1) per-file utility-driven replication strategy, and (2) a load-sensitive access scheduling policy that targets inter-request parallelism at high loads and intra-request parallelism at low loads. Second, MorphStore is implemented purely at the file system level.

Local vs. Global file systems: Distributed file systems like GFS [7] and HDFS [8] are used in big data analysis. However, such file systems are typically overlaid on top of local file systems such as ext3 and ext4 (rather than operating directly on the block-level interface). Any improvement in local file systems’ performance will be reflected at the global level as well.

Finally, the I/O stack is designed with a file size distribution in mind. For example, the Linux inode structure is optimized for small files. Some of those assumptions must be questioned when the same local file systems are used in emerging domains where large files are routinely manipulated. MorphStore addresses this gap.

III. MorphStore DESIGN OVERVIEW

We make some high-level design choices to facilitate the design discussion. First, we choose to implement MorphStore at the file system level. Because competing replication techniques (e.g., RAID-0 and RAID-1) are implemented at the block/device layer, this design choice needs some justification. Our design goals are two-fold. First, we wish to manage replication and access scheduling for large files based on load levels, popularity, and read-write ratio. Given that per-file meta-data can be conveniently tracked using existing file system structures such as inodes, file systems are a natural candidate for tracking such state. Second, we wish to manage replication at a suitable granularity without excessive overheads. In the file

system layer, replicating data at the file granularity is natural because the file system supports primitives to manipulate (read/write/create/delete) replicas. One may think that finer granularity tracking may be more helpful in being more selective by replicating only hot pages rather than entire files. However, it will add significant complexity to track finer grain meta data.

Given the above design decisions, we now describe the two key components of MorphStore: load-adaptive access scheduling and utility-driven replication.

A. Load-adaptive access scheduling(LAAS)

A key component of MorphStore is the load-aware access scheduling mechanism that decides if requests are issued to exploit striping across replicas (i.e., exploiting intra-request parallelism like RAID-0) or to a single disk (i.e., exploiting inter-request parallelism like RAID-1). Because we want the adaptive mechanism to react to load levels, we track the number of open inodes in a recent (tunable) window of time. If the measured load exceeds a threshold, MorphStore switches to a high load configuration where requests are issued to specific disks. In contrast, under low load operation (i.e., when the measured load is below the threshold) the requests are striped over available replicas. The threshold value is a configurable value which we set to the number of devices in the storage array based on empirical observations.

The striped requests are split over the available replicas as evenly as possible. While MorphStore’s striping resembles RAID-0’s striping at first glance, there exists an important difference between the striping methods. Striped access in the RAID-0 system results in contiguous reads; because of this, reads may be merged easily with subsequent reads as shown in Figure 1(b). MorphStore’s approach to exploit intra-request parallelism achieves the same degree of disk parallelism. In contrast, the reads are non-contiguous (and hence non-mergeable) because the sub-blocks being read from different disks are effectively selected from full replicas (see Figure 1(a)). An alternate way of looking at the above distinction is to say that RAID-0 organization achieves both parallelism and locality whereas MorphStore achieves only parallelism.

For the non-striped accesses, ideally we want MorphStore to mimic RAID-1. However, RAID-1 (because it operates at the block level) has information on disk scheduling that MorphStore does not have (because MorphStore operates at the file level). Specifically, RAID-1 can schedule block requests to the replica with the nearest disk-head. To overcome this handicap, we observed that in practice, RAID-1’s disk level knowledge results in the characteristic that most block accesses of a single file are sent to the same disk. As such, in MorphStore the non-striped requests are sent to a single device such that further accesses to the same file (i.e., for other blocks) are

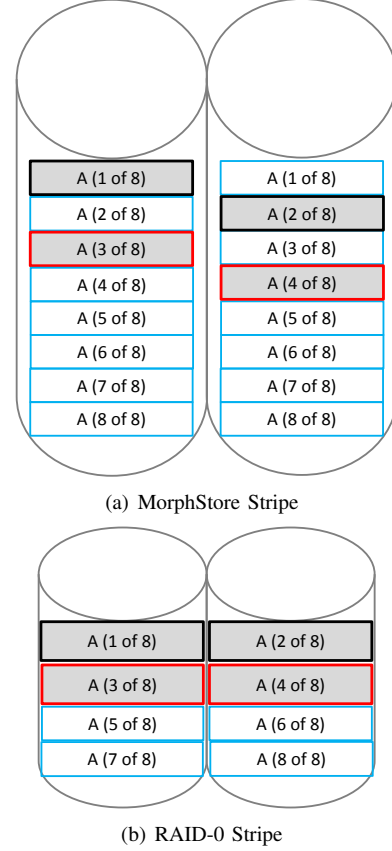


Figure 1. Stripe Comparison

bound to that device for the duration of the file access. This has the benefit of aiding reference locality. MorphStore accomplishes this by associating the master inode with a device while in high load mode. When the inode is no longer in use the device association is eliminated. Finally, we steer requests that are not yet associated with any device to the most lightly loaded device to achieve load balancing.

B. Utility-driven Replication (UDR)

The second component of MorphStore is our utility-driven replication that aims to use replication capacity to maximize performance. The model we assume is as follows. The file system tracks read/write accesses over fairly long periods of times (say hours/days). File access statistics are analyzed infrequently (e.g., once every few days) during the low-load periods (e.g., late at night when the diurnal load cycle is typically low) to (a) determine which files will yield the highest opportunity for replication, and (b) copying files over to achieve the desired replication. Such dynamic¹ replication allows highly utilized files to be replicated on

¹Note, that we refer to UDR as dynamic replication because the replication strategy changes at runtime, although the change is rather infrequent (i.e., once every few days). The contrast is with static replication strategies like RAID-1 where the replication strategy is invariant.

additional disks in the disk array, including the originating disk. Such replicas enable the load-aware access scheduling that we previously described. Dynamic replication also provides space-saving over static replication by selectively replicating only those files which will benefit the most by the strategy outlined below.

Replication is facilitated by use of extended attributes which allows the replication information to be fed into the file system. The information consists of the number of times the system should replicate this file.

C. Replication strategy

The number of replicas to be generated for each file is decided based on the notion of utility, which is determined by the number of read and write accesses to that particular file as described in the algorithm below. The key idea is a notional cost-metric in which replicas help read-costs (because reads may be distributed over the replicas) and hurt write-costs (because writes must be duplicated for each replica). Assuming that the *file* data structure holds the number of reads, number of writes, replications and notional cost for each file, we define this cost for a given file as $file.reads / file.replicas + file.writes \times file.replicas$. With this cost-metric, we can define the incremental utility of an additional replica as the difference in notional costs that a file may incur if the total number of replicas of that file is increased by 1 (see line 3 in Algorithm 1). The *incremental utility* captures the benefit (or loss) associated with the additional replica.

Algorithm 1 REPLICATE(Q)

```

1: while Q is not empty or capacity > 0 do
2:   file  $\leftarrow$  dequeue Q
3:   utility  $\leftarrow$   $[file.reads / (file.replicas) +$ 
    $file.writes * (file.replicas)] -$ 
    $[file.reads / (file.replicas + 1) + file.writes * (file.replicas + 1)]$ 
4:   if utility > 0 then
5:     file.replicas  $\leftarrow$  file.replicas + 1
6:     capacity  $\leftarrow$  capacity - 1
7:   else
8:     add file to done list
9:   end if
10:  if file.replicas = disks then
11:    add file to done list
12:  else
13:    enqueue(file)
14:  end if
15: end while

```

We use a priority queue *Q* to hold all the files with *incremental utility* of an additional replica as the priority value. Initially the number of replicas of each file is set to 1. As long as the incremental utility of creating an additional

replica of the file is positive, we increase the replication factor for that file by 1 while decreasing the system capacity by the file size; otherwise we mark it as done (lines 4 – 9). Depending on the total number of replicas for the file and the number of disks in the system, the file is either marked as done or enqueued back in the priority queue (lines 10 – 14). The algorithm terminates if there is no more capacity left in the system to store the replicas or if there remains no positive utility for further replication.

We illustrate the operation of the *REPLICATE* algorithm with an example in Figure 2. The example uses four files (A, B, C, and D) with the initial utility values as shown in the left of the figure. The figure then illustrates the progress of the algorithm for the first three iterations. On the first iteration (numbered iteration 0), the algorithm recognizes that file A has the highest incremental utility (100) and hence is most favorable for replication. After increasing the number of replicas of file A to 2, the incremental utility is recalculated and inserted back into the priority queue. The new incremental utility of file A (40) is less than the incremental utility of file C (60) and hence file C is now considered by the algorithm for replication. The same process of increasing the replication factor and calculating the new incremental utility is performed and the file inserted back into the queue. In the third iteration, file A re-emerges at the head of the queue and is hence re-replicated to bring the number of copies to 3. Some files like B and D have very low incremental utility, so they are never considered for replication.

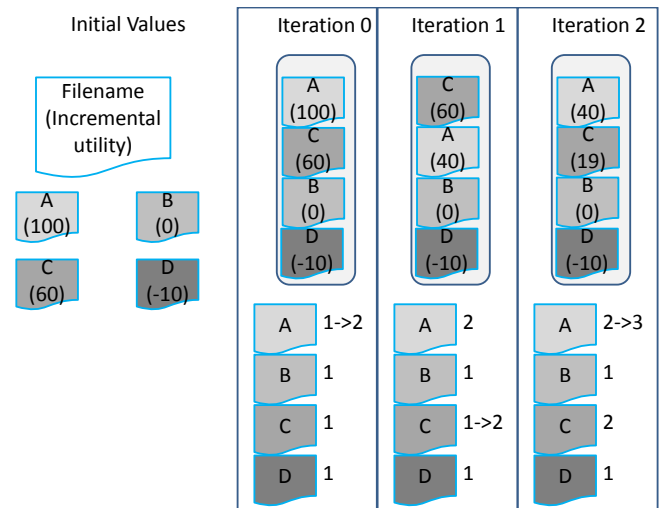


Figure 2. Replication strategy

IV. IMPLEMENTATION DETAILS

MorphStore may be implemented as modifications to any file system. For this study, we implemented MorphStore as a modification of Linux's second extended

file system `ext2` [9] because it is a widely used file system that has a relatively small and clean code base. Moreover, there are mature tools to aid usage and development for this file system such as *e2fsprogs* [10], which allow viewing the base file system and modifying its attributes. The following sub-sections will discuss the implementation of MorphStore. First, we present a brief discussion of some implementation challenges. Second, we discuss the meta-data used to maintain the replicated data. Third, we discuss the replication operation. Finally, reads and writes to replicas (the replicated files) will be discussed.

A. Concatenation of disks

A normal file system need not concern itself with the number and size of the disks used as its backing store. However, in order to ensure that replication occurs on specific devices MorphStore must know two characteristics of the underlying device array, both of which are contained in the VFS (Virtual File System) super block structure.

First, the size of each device must be known. This is to ensure that files are placed within the boundaries of the device. This information is used in *file allocation* to ensure that a replica is located on the intended device. Normal files need not be remanded to a specific device; they may be placed on any device with sufficient room. The one exception is that normal files may not cross disk boundaries. File allocation uses device boundary information to ensure maintenance of this property. The necessity for the file system to be aware of boundaries when allocating files impacts contiguous file placement; should the file allocation begin near a boundary. In such cases, we move the file beyond the boundary to ensure contiguous placement on a single device. However, this is not a serious limitation because (a) it only affects one file at each device-to-device boundary, and (b) to have any serious impact, we would have to consider a large number of small devices and serious capacity pressure – an unlikely scenario.

Second, the number of devices must be known, as MorphStore requires more than one device as a backing store. The use of the JBOD RAID level lends itself well to this restriction, as this minimum number of devices is the only restriction for MorphStore to function. The number of devices in the array determines the maximum number of replicas that can exist within the system.

B. MorphStore Meta-data

The meta-data for MorphStore consists of two parts, on-disk meta-data and in-memory meta-data. The former is in the form of special directories and file attributes located on the disk, and the latter is a data structure attached to the kernel's VFS inode structure. Most of our frequent updates occur on the in-memory meta-data which is inexpensive to modify; a significantly smaller fraction of changes trickle

down to the on-disk meta-data which is more expensive to modify.

The initial structures of MorphStore are created upon first mounting of the file system; these are the replicated file directories. These are special directories that only contain replicated files. Section IV-A discussed the necessity to partition the file system via device boundaries. These directories reside on specific devices and hold replicated files for those devices.

Since MorphStore is implemented at the file system layer, we leverage existing meta-data mechanisms, and augment them to track replicated data. This is accomplished through the use of VFS inode structures, not to be confused with the `ext2` inode structure. The VFS inode is augmented with a data structure that maintains the following list of information.

- Replica list: holds replication information for the associated file.
- Replica mask: bit mask to easily select and test existence of replicas.
- Master location: device location of original file.
- File statistics: read/write usage for file.

We use existing meta-data structures, and augment them with 42 bytes of extra meta-data (assuming 8 byte pointers). This is on a per master file basis. Replicated files do not need to hold meta-data, as they are not replicated, and no reads/writes occur to replicas explicitly; they are only accessed as a byproduct of accessing the master file.

C. MorphStore Replica operations

When a file is opened the VFS inode associated to the newly opened file is populated with information from the `ext2` inode. We make use of this to create our replica meta-data structure and attach it to the file's VFS inode. In addition, a *find replica* routine is called to search the special replication directories for existing replicas of the file. If replicas are found, they are opened and their VFS inode information is held in the master's replica meta-data structure. Also upon file open, the stored attributes are read from disk to populate the mask and file read/write statistics.

As file operations occur on the open file, updates to meta-data are stored in the in-memory inode structure. When the file is closed, that structure is used to update the `ext2` inode structure and it is written back to disk. We use this opportunity to again update the extended attributes where we store file read/write statistics. This allows us to minimize reads and writes to the device for meta-data. The tradeoff is that should the system crash the updated meta-data will be lost. This is an acceptable tradeoff as the next open will resume from the previous state loaded from disk.

MorphStore uses the replica mask to either split the read (number of pages) over replicas (RAID-0), or send all pages to a particular replica (RAID-1). Remember the replicas are opened with the master, so sending the block

CPU	Itanium 2
RAM	3 GB
Page Size	64 KB
File System Block Size	4 KB
Drives	4x500 GB

Table I
TEST SYSTEM CONFIGURATION

request is simplified. The request is done via the normal file system read routine. MorphStore uses the meta-data to select which replica (including the master) will supply the data. Striping is handled as discussed earlier in Section III-A.

MorphStore handles writes in a similar fashion to reads. Writes to a file occur to all copies of the file. This is accomplished easily through the file system write routine. Again, the meta-data is used to select all replicas when sending the write data. Writes will use all disks where replicas exist, similar to RAID-1.

V. EVALUATION METHODOLOGY

We use two tools: *Filebench* [11], and *IOStat* [12] to gather information from the file system level and the device IO level, respectively. The test system is configured as shown in Table I, with four devices configured in either JBOD (linear personality), RAID-0 (striped personality), or RAID-1 (mirrored personality). The ext2 file system is then placed on top of each configuration to test performance, and MorphStore is placed upon JBOD to test performance.

When the system is configured as RAID-0 the stripe size is set at 512 KB; this helps amortize the seek overhead over large transfers. The RAID-0, RAID-1, and JBOD configurations deal in file system blocks which are 4 KB, The system reads in pages from disk which are 64 KB in size. MorphStore is not limited to a specific configuration.

The read ahead size is set at 32 MB for each run of the respective configurations, JBOD, RAID-0, and RAID-1. The tool mke2fs configures the ext2 file system with the information of the underlying device array, so no extra configuration is needed. Statistics are collected from seven 120 second runs of Filebench with IOSTat collecting from the devices every 5 seconds while Filebench is running.

Workloads: We use two workloads based on *filebench* benchmarks. The first is a video server workload, which emulates large media distribution servers. The second is a database workload based on popular NoSQL database MongoDB.

Video server workload emulates media distribution providers. These providers serve large files for consumption, typically multimedia video files. This workload has been modified to include content updates to the server.

MongoDB is a NoSQL open-source database. NoSQL databases have widespread adoption in many large scale-out datacenters. We use a benchmark that mimics the IO

access patterns of MongoDB. This benchmark emulates an access pattern with random appends to files and whole file reads.

Both of these workloads have been augmented to scale up the number of threads performing IO accesses. This allows us to scale between low and high load scenarios with these workloads. They both also use large files for IO accesses.

Filebench is used to generate a work load from the video server template, with an average file size of 1 GB, This work load is scaled from a low load to a high load, which scales up the number of requests sent to the device array. We also defined flowops to introduce popularity for files in the file sets. The file sets typically take up 40% of the device array for each configuration, this leaves room for replication on MorphStore.

VI. RESULTS

The key findings reveal that MorphStore achieves significantly higher performance. These findings also validate the intuition behind each of our techniques.

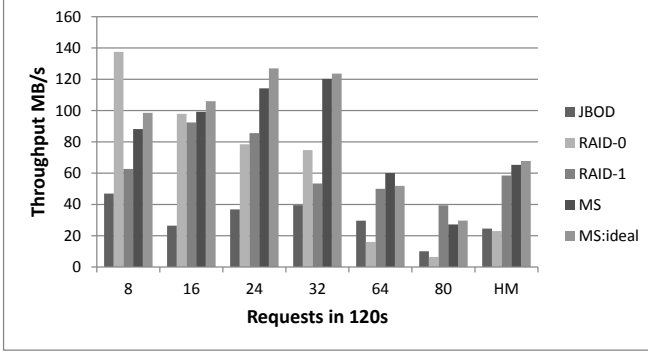
- In both our video-server and MongoDB workloads, MorphStore achieves significant improvements relative to both RAID-0 and RAID-1 when averaged (harmonic mean) across high and low load levels.
- MorphStore can respond to limits on replication capacity by implementing utility driven replication to the extent possible. While this does diminish the gains, our results reveal that significant gains can be achieved with minimal space overhead.
- Isolating the load-adaptive access scheduling technique, we observe that MorphStore tracks the better of either RAID-0 or RAID-1. MorphStore’s simple load-prediction based on recent-history is nearly as effective (within 8%) as *a priori* knowledge of load.
- Similarly isolating the utility-driven replication, MorphStore achieves within 4% of performance as an ideal (but impractical) replication scheme that has perfect knowledge of file access patterns.

We elaborate on each of the above results in the remainder of this section.

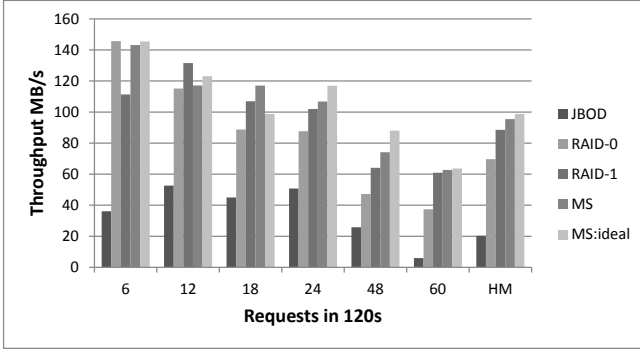
A. MorphStore for video server and MongoDB access patterns

Figure 3 plots the file system data throughput (Y-axis, MB/s) for a range of load levels (X-axis, requests/second) for our two workloads (the two graphs). For each load level (group of bars), we show the performance of JBOD, RAID-0, RAID-1 and MorphStore(MS). We include an additional bar in each group for an Ideal-MS which achieves ideal replication. Finally, we include the harmonic mean (HM) throughput of each configuration in the final (right-most) set of bars.

Figure 3 leads us to four key observations common to both workloads. First, as expected, RAID-0 achieves the best



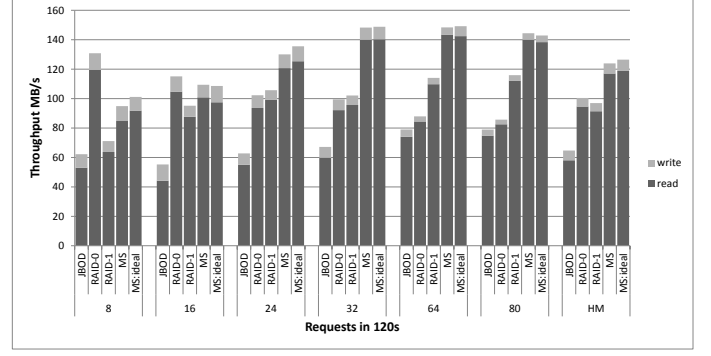
(a) Video Server



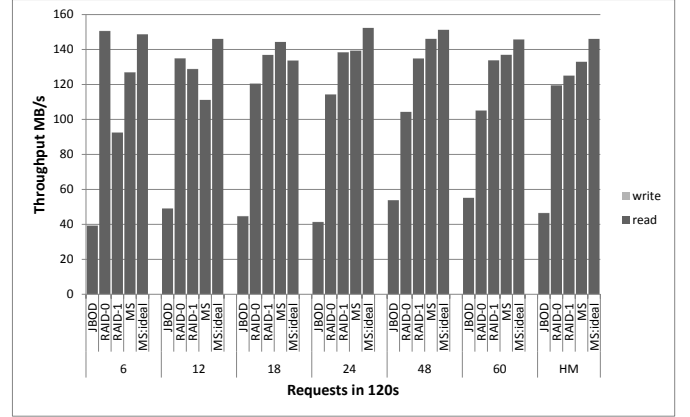
(b) MongoDB

Figure 3. File System Throughput

performance at low loads and it significantly outperforms other configurations. However, its performance degrades with increasing load and becomes the worst-performing configuration at high loads. Second, RAID-1's trend is a composite of two factors. On the one hand, at high loads, queuing delays have the effect of reducing throughput at the file system level. On the other hand, RAID-1 is better able to exploit inter-request parallelism at high loads. The combination of these two factors results in a "sweet spot" in performance for RAID-1 because queuing delays hurt performance at high loads and lack of intra-request parallelism hurts at low loads. Note, the queuing delays at extremely high loads reduce the effective bandwidth for all configurations. Third, MorphStore achieves similar to RAID-0 performance at low loads although there remains a significant gap. That gap is due to the fact that RAID-0 achieves perfectly contiguous reads on all disks when requests merge. However, because MorphStore uses full file replication and not true striping, MorphStore incurs more seek overheads which diminish performance. At high loads, while overall bandwidth reduces because of high queuing delays, MorphStore remains closer to RAID-1 and much better than RAID-0. The harmonic mean (HM) shows MorphStore performing 2.84x better than RAID-0 and 12% better than RAID-1 in Figure 3(a) for the video



(a) Video Server



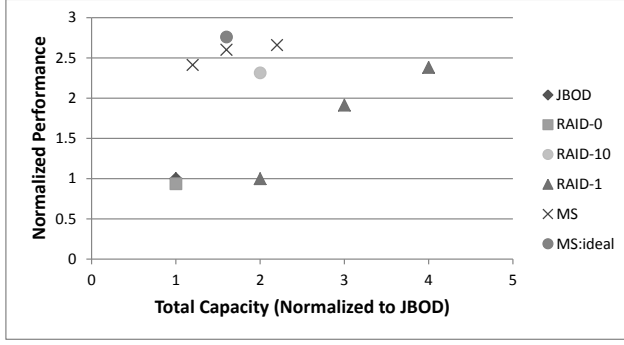
(b) MongoDB

Figure 4. Device (disk-array) Throughput (IOStat)

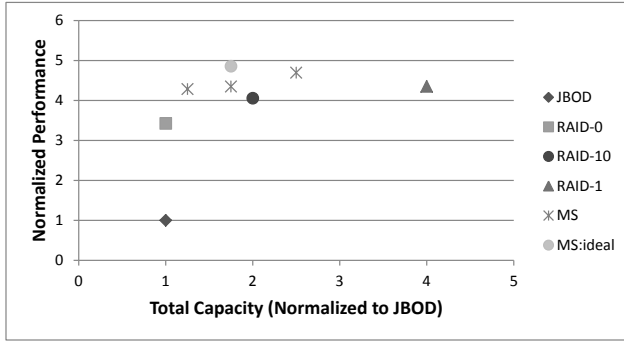
server workload. The corresponding speedups over RAID-0 and RAID-1 for the MongoDB workload are 37% and 8%, respectively. Finally, we observe that MorphStore's profile-based UDR is only marginally worse than the Ideal-MS which has *a priori* knowledge of popularity and read/write ratios.

To isolate the disk-array bandwidth from the file system bandwidth (effectively to hide the effects of file system queuing delays), we also measure the throughput of the disk array (i.e., from Iostat [12]). Figure 4 uses similar axes and grouping as the earlier Figure 3 with two key differences. First, the Y-axis shows device throughput rather than file system throughput. Next, each bar separates out the read bandwidth from the write bandwidth.

The trends remain unchanged for RAID-0; RAID-0 is best at low loads. RAID-1's throughput saturates at high loads (unlike the file system throughput which degrades because of queuing delays). MorphStore outperforms both RAID-0 and RAID-1 by significant margins. The harmonic mean shows MorphStore performing 24% better than RAID-0, and 26% better than RAID-1, on average for the video server and 11% better than RAID-0 and 6% better than RAID-1, on average for the MongoDB workload. Note, MongoDB has a



(a) Video Server



(b) MongoDB

Figure 5. Capacity vs Performance: the Pareto Frontier

high read/write ratio. This is not surprising as the read-heavy nature of datastores is widely reported [13], [14], [15].

B. MorphStore under capacity constraints

In our previous experiments, we assume that MorphStore’s UDR replicated all files until we ran out of disks (i.e., we have as many replicas as disks) or we ran out of files with positive utility. We did not impose any artificial constraints on replication capacity. Even without any such limits, MorphStore utilized only 2.2X additional capacity for the video server workload and 2.5X additional capacity for the MongoDB workload (compared to 4X capacity used by RAID-1) while achieving higher performance than RAID-1. We examine the performance of MorphStore under various capacity constraints in this section.

Figure 5 shows the performance of MorphStore when the replication capacity is arbitrarily limited; and compares MorphStore’s performance and replication overhead to that of the baseline JBOD/RAID configurations, for each of the two workloads. We plot the performance of MorphStore by varying the total capacity. For the video workload we vary the total capacity in multiples of 1.2X, 1.6X, and 2.2X of the baseline JBOD capacity. Similarly, for the MongoDB workload, we use the multiples 1.25X, 1.75X, 2.5X of the baseline JBOD file system capacity. The maximum multiples (2.2X of video server and 2.5X for

MongoDB) are chosen because that is the maximum capacity needed by MorphStore. Beyond that capacity, there are no files with positive incremental utility that UDR would replicate further. The other capacity multiples are chosen arbitrarily to explore the space between no-replication case and the maximum replication cases.

For the baseline systems, the X-axis effectively shows the inherent replication capacity used by such systems. For example, when we use a RAID-1 configuration with 4-way mirroring, it effectively means that RAID-1 uses 4X the capacity of the JBOD baseline. Correspondingly, the RAID-1 data point for 4-way mirroring is plotted with an X-axis value of 4. For the video server, we also include 2-way and 3-way mirrored RAID-1 configurations where it is clear that these configurations are not on the Pareto frontier. As such, we omit the 2-way and 3-way mirrored RAID-1 configurations from the MongoDB workloads.

Similarly, We also include a RAID-10 configuration that uses the same four disk devices in a 2-way mirrored, 2-way striped (within each mirror) configuration. The RAID-10 configuration is plotted at an X-axis value of 2 because of the 2-way mirroring. Note, RAID-0 does not replicate any data; consequently, its capacity is identical to that of JBOD (which is effectively a capacity multiplier of 1).

Ideally, we want systems to be in the top-left corner of the space because we want maximum performance (higher) with minimal replication (toward the left). The points on the upward-left facing frontier represent that Pareto-frontier of the replication-performance tradeoff. The normalized performance (mean file system throughput) of all systems is shown on the Y-axis. In addition to the practical configurations, Figure 5 includes a data-point for the *ideal* variant of MorphStore which is an impractical version of MorphStore with *a priori* knowledge of read/write frequencies of files.

UDR’s greedy nature can be observed in the fact that replicating the most beneficial files to take up 20% of replication capacity results in 2.2x performance improvement for the video server workload. The curve settles at 2.5x performance gains relative to JBOD at 2.2x capacity. This shows that there are diminishing returns on performance as we use more and more replication capacity. (The trends for the MongoDB workload are similar. They saturate at approximately 4.7X of JBOD at a total capacity of 2.5X. As with the video server, nearly all of the opportunity is greedily captured even with a capacity of only 1.25 where MorphStore achieves approximately 4.2X of JBOD’s performance.)

MorphStore outperforms both RAID-0 and RAID-1 by 2.84X and 12% for the video server workload. The corresponding improvements are 37% and 8% for the MongoDB workload. Finally, note that MorphStore comes close to the ideal performance in both workloads.

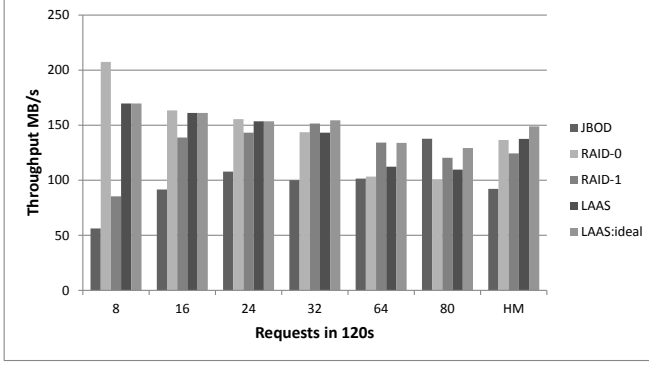


Figure 6. Filebench LAAS RO Throughput

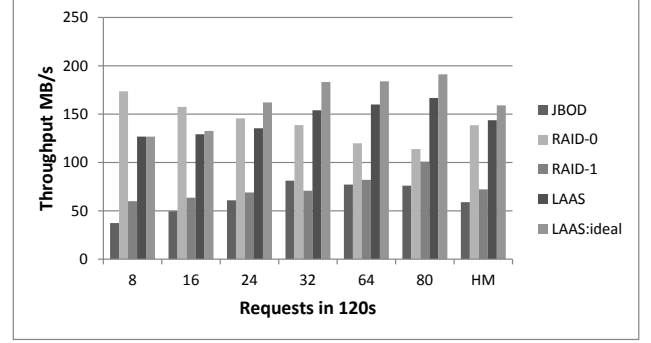


Figure 9. IOSTat LAAS Replication Cost Throughput

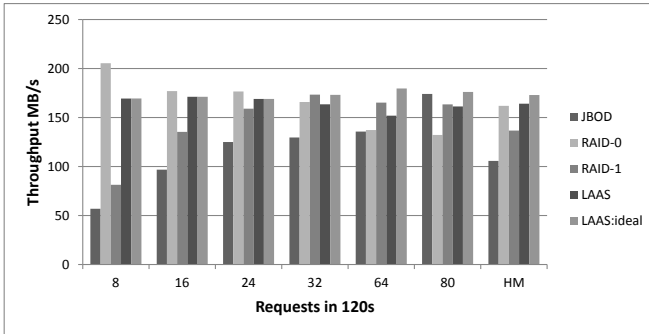


Figure 7. IOSTat LAAS RO Throughput

C. Isolating the effect of LAAS

To isolate the impact of LAAS from UDR, we revert to using RAID-1 like full mirroring with MorphStore. For this subset of results, we limit our evaluation to the video server workload.

The performance of such a LAAS-only design is captured in Figure 8 and Figure 9. Because the disk requests include writes, and because writes interact poorly with replication, we see a sharp decline in RAID-1, as the devices are utilized for mirroring and must also verify writes as redundancy

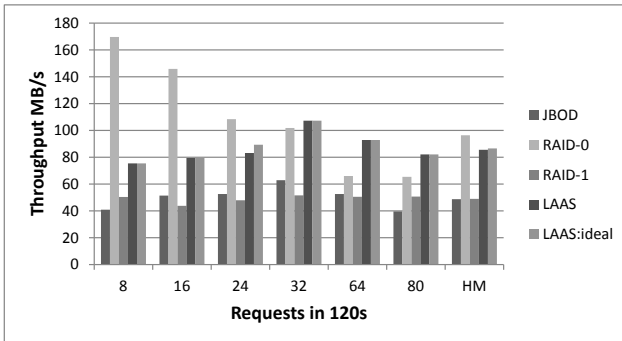


Figure 8. Filebench LAAS Replication Cost Throughput

is the design point of RAID-1. RAID-0 however, weathers the writes much better as writes are not redundant. MorphStore maintains much better performance than RAID-1 by 71%, and only slightly less than RAID-0 by 14%.

To further eliminate the effect of writes, we employ a similar configuration but with only reads.

Figure 6 contains the file system characteristics of the following techniques: JBOD, RAID-0, RAID-1, and MorphStore. Figure 7 contains the characteristics at the device level of the same systems. There are a few key observations that should be understood from this Figure. Consider JBOD first, one would expect JBOD to perform better under low load and worse under high load, but clearly JBOD benefits from a higher load. This is attributed to the *ext2* file system *block allocation* algorithm, which attempts to distribute data across the block groups of the file system. JBOD is able to service more requests at higher load as the block groups span multiple disks.

RAID-0 does exceptionally well under low load and has a sharp decrease as load is slightly increased, one might expect RAID-0 to gradually decline as the load increases, but low load is a special case. The RAID-0 system seeks minimally for requests under low load, thus accesses resemble a sequential access and nearly full throughput can be seen from the RAID-0 system. One can expect RAID-0 to decrease as load increases due to the striped reads and the necessity of the disk system proceed as one unit. RAID-1 behaves as one would expect, at low load there exists minimal opportunity to utilize mirrored resources, while at high load requests can be serviced from the mirror disks. While under medium load one can observe the change of performance of RAID-0 and RAID-1.

The key observation is that RAID-1 overtakes RAID-0 as load increases, and the opposite (RAID-0 overtakes RAID-1) as load decreases. Finally, MorphStore demonstrates the ability to track load levels and adapt to use the better of RAID-0 or RAID-1 access strategies. The harmonic mean shows the overall performance of MorphStore with

respect to RAID-0 and RAID-1, in which MorphStore outperforms RAID-1 by 12% and performs equivalently to RAID-0 (RAID-0 is an outlier in the low load case). The load adaptive access scheduling MorphStore outperforms RAID-1 by 21% and RAID-0 by 2% in Figure 7.

VII. CONCLUSION

Big-data analysis is emerging as an important tool and file system performance when manipulating large files is a critical aspect of performance for big-data analysis. Such analysis typically occurs over large collections of data on distributed file systems. However, such distributed file systems are overlaid on underlying single-node local file systems. This paper focuses on the challenge of providing high performance for big file manipulation on local file systems.

Our design — MorphStore — uses two key innovations to improve performance over static one-size fits all approaches to replication and access scheduling. First, MorphStore uses load-aware access scheduling (LAAS) to dynamically capture the benefits of striping at low loads while also capturing read-parallelism across replicas at high loads. Second, MorphStore customizes replication on a per-file basis based on the expected utility. Our utility measure reflects the intuitive notion that popularly read blocks benefit (positive utility) from replication and that heavily written blocks have a cost (negative utility) due to replication. For any given replication capacity, MorphStore’s utility-driven replication (UDR) strategy maximizes the benefits by greedily selecting files for replication that yield the most utility.

In combination, the two features enable MorphStore to extend the Pareto frontier in the replication-capacity vs. performance trade-off; implying that MorphStore can offer higher performance at lower replication cost than prior designs. For example, for a video-server workload, MorphStore with 2.2X replication cost achieves 12% and 2.84x higher file system throughput than RAID-1 and RAID-0, respectively. In conclusion, MorphStore combines the benefits of both striping and replication to advance the Pareto frontier and provide dynamic design alternatives to existing static techniques.

REFERENCES

- [1] E. Grochowski and R. Halem, “Technological impact of magnetic hard disk drives on storage systems,” *IBM Systems Journal*, vol. 42, no. 2, pp. 338–346, 2003.
- [2] D. A. Patterson, G. Gibson, and R. H. Katz, “A Case for Redundant Arrays of Inexpensive Disks (RAID),” in *Proceedings ACM SIGMOD Conference*, Chicago, Illinois, Jun. 1988.
- [3] P. Sanders, S. Egner, and J. Korst, “Fast concurrent access to parallel disks,” in *soda*, vol. 11, San Francisco, Jan. 2000, pp. 849–858. [Online]. Available: <http://theory.lcs.mit.edu/~dmjones/SODA/soda.bib>
- [4] S. Berson, S. Ghandeharizadeh, R. Muntz, and X. Ju, “Staggered striping in multimedia information systems,” in *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*. ACM New York, NY, USA, 1994, pp. 79–90.
- [5] Y. R. Choe and V. S. Pai, “Achieving reliable parallel performance in a vod storage server using randomization and replication,” in *IPDPS*. IEEE, 2007, pp. 1–10.
- [6] D. A. Patterson, G. Gibson, and R. H. Katz, “A case for redundant arrays of inexpensive disks (raid),” in *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’88. New York, NY, USA: ACM, 1988, pp. 109–116. [Online]. Available: <http://doi.acm.org/10.1145/50202.50214>
- [7] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The google file system,” in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. SOSP ’03. New York, NY, USA: ACM, 2003, pp. 29–43. [Online]. Available: <http://doi.acm.org/10.1145/945445.945450>
- [8] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The hadoop distributed file system,” in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. [Online]. Available: <http://dx.doi.org/10.1109/MSST.2010.5496972>
- [9] S. T. Remy Card, Theodore Ts’o, “Design and implementation of the second extended filesystem,” in *Proceedings of the 1st Dutch International Symposium on Linux*, 1994.
- [10] T. Ts’o, “E2fsprogs: Ext2/3/4 filesystem utilities,” e2fsprogs.sourceforge.net, December 2010.
- [11] O. S. Community, “Filebench,” <http://www.solarisinternals.com/wiki/index.php/FileBench>, December 2010.
- [12] N. Garfield, *Iostat*. Anim Publishing, 2012. [Online]. Available: <http://books.google.com/books?id=qAusMQEACAAJ>
- [13] “Facebook’s nsdi’13 presentation,” <https://www.usenix.org/conference/nsdi13/scaling-memcache-facebook>.
- [14] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, “Workload analysis of a large-scale key-value store,” in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, 2012, pp. 53–64.
- [15] B. Debnath, S. Sengupta, and J. Li, “SkimpyStash: RAM space skimpy key-value store on flash-based storage,” in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, 2011, pp. 25–36. [Online]. Available: <http://doi.acm.org/10.1145/1989323.1989327>