Spring 2015

Improving capacity-performance tradeoffs in the storage tier

Eric P. Villasenor
Purdue University - Main Campus

Graduate School Form 30
(Revised 08/14)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Eric Villasenor

Entitled
Improving Capacity-Performance Tradeoffs in the Storage Tier

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

MITHUNA S. THOTTETHODI

SAMUEL P. MIDKIFF

T. N. VIJAYKUMAR

VIJAY S. PAI

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification/Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

MITHUNA S. THOTTETHODI

Approved by Major Professor(s): _____

Approved by: Michael R. Melloch 04/30/2015

Head of the Department Graduate Program Date

IMPROVING CAPACITY-PERFORMANCE TRADEOFFS IN THE STORAGE TIER

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Eric Villaseñor

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2015

Purdue University

West Lafayette, Indiana

Dedicated to my grandmother Ofelia Tijerina, and all the other members of my family who supported me through this journey.

ACKNOWLEDGMENTS

I'd like to acknowledge my major professor Mithuna Thottethodi for all of his support and guidance throughout my stay in graduate school.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABBREVIATIONS

GB	Gigabytes
TB	Terabytes
PB	Petabytes
EB	exabytes
FS	File system
HDFS	Hadoop Distributed File System
DFS	Distributed File System
JBOD	Just a Bunch Of Disks
sn	shared nothing

GLOSSARY

node	a machine with computational and storage resources
cluster	group of nodes

ABSTRACT

Villaseñor, Eric Ph.D., Purdue University, May 2015. Improving Capacity-Performance Tradeoffs in the Storage Tier. Major Professor: Mithuna Thottethodi.

Data-set sizes are growing. New techniques are emerging to organize and analyze these data-sets. There is a key access pattern emerging with these new techniques, large sequential file accesses. The trend toward bigger files exists to help amortize the cost of data accesses from the storage layer, as many workloads are recognized to be I/O bound. The storage layer is widely recognized as the slowest layer in the system. This work focuses on the tradeoff one can make with that storage capacity to improve system performance.

Capacity can be leveraged for improved availability or improved performance. This tradeoff is key in the storage layer, as this allows for data loss prevention and bandwidth aggregation. Typically these tradeoffs do not allow much choice with regard to capacity use. This work will leverage replication as the enabling mechanism to improve the capacity-performance tradeoff in the storage tier, while still providing for availability.

This capacity-performance tradeoff can be made at both the local and distributed file system level. I propose two techniques that allow for an improved tradeoff of capacity. The local file system can be employed on scale-out or scale-up infrastructures to improve performance. The distributed file system is targeted at distributed frameworks, such as MapReduce, to improve the cluster performance. The local file system design is MorphStore, and the distributed file system is BoostDFS.

MorphStore is a file system that significantly improves performance when accessing large files by using two innovations. MorphStore combines (a) load-adaptive I/O access scheduling to dynamically optimize throughput (aggregation), and (b) utility-

driven replication to best use capacity for performance. Additionally, adaptive-access scheduling can be utilized to optimize scheduling of requests (for throughput) on systems with a large number of storage devices. Replication is utilized to make available high utility files and then optimize throughput of these high utility files based on system load.

BoostDFS is a distributed file system that allows a better capacity-performance tradeoff via inter-node file replication. BoostDFS is built on the observation that distributed file systems currently inter-node replication for availability, but provide no mechanism to further improve performance. Replication for availability provides diminishing returns on performance, this is due to saturation of locality. BoostDFS exploits the common by improving I/O performance of these local tasks. This is done via intra-node replication by leveraging MorphStore as the local file system. This technique allows for capacity to be traded for availability as well as performance, with a small capacity overhead under constant availability.

Both MorphStore and BoostDFS utilize replication. Replication allows for both bandwidth aggregation and availability, This work primarily focuses on the performance utility of replication, but does not sacrifice availability in the process. These techniques provide an improved capacity-performance tradeoff while allowing the desired level of availability.

1. INTRODUCTION

Data-sets are growing in size. Cloud computing has information constantly being pushed to the data center where it is accessible for use and analysis. The term "Big Data" has been coined from this trend. This is the idea that there is a wealth of information stored within these data-sets, if only one could analyze them to discover that information. The need to gather such information has led to the introduction of frameworks and organization techniques to accomplish exactly this analysis on large data-set workloads.

Storage is commonly acknowledged as the slowest tier of a system [1]. This tier is important, not only because of its persistence, but mainly because of the capacity. One can store quite a lot of data in this tier, and this is cheaper than any of the other tiers for sheer capacity. So, while this may be the slowest tier, just about every use for a system will at some point in time end up in this tier. That makes storage a very integral part of system performance.

There are two main goals in the storage tier, availability and performance. Both of these goals make use of the capacity in the storage tier. Yet, currently most designs favor availability in the tradeoff of capacity, and the side effect is performance. I do not propose to degrade availability, but to allow a choice in the tradeoff of capacity. This choice improves performance in return for capacity.

Performance, at the storage level is device oriented. A single device can only perform up to its limits. However, one can pool devices and their aggregate performance can exceed a single device. This technique is common knowledge and is used by many systems.

Availability, this is the idea that information stored on the storage tier can later be accessed regularly, no interruptions (i.e., it can be found). Systems require a certain

level of availability to ensure operation, this requirement is predominantly placed on the storage tier. This is the primary capacity tradeoff most designs target.

Large transfers amortize costs. Overhead amortization by increasing the amount of useful work; this is a basic tennent of system architecture. One can effectively utilize a high percentage of storage throughput by selecting a large enough block size to transfer. This provides an access pattern used throughout large data-set workloads. The access pattern is large sequential reads of files.

This trend toward large files enables the use of a mechanism to improve the capacity-performance tradeoff, without sacrificing availability. The mechanism to which I refer is replication. Replication controlled at the file system level provides the flexibility to maintain availability and provide performance. This work introduces two designs that capitalize upon this goal of a improved capacity-performance tradeoff. MorphStore, a local file system, and BoostDFS, a distributed file system.

I will first expand upon the file systems that enable these frameworks, both local and distributed. Then I will discuss the frameworks that would be applicable to the proposed designs. Then I will outline the goals and contributions of this work and detail the organization of the remainder of this document.

1.1 Local File Systems

The local file system layer interfaces with the underlying attached storage devices. The local file system is responsible for allocating space for files and tracking file metadata. The local file system ensures data consistency when updates to files occur. Typical local file systems do not manage the underlying storage devices.

Popular local file systems, such as ext2 [2], are responsible for ensuring data is placed on the underlying storage device. The block layer manages these storage devices and provides either a performance or reliability configuration for the file system.

1.2 Distributed File Systems

There are many varieties of distributed file systems for large data-set applications [3–8]. Distributed file systems are typically built on top of local file systems. The storage system and its performance for accessing large files is of great importance to large data-set applications as demands increase. The frameworks that manage the data center resources have a distributed file system (DFS) component that manages the storage resources for each node in the cluster while providing the abstraction of a unified, large store.

The use of large data-sets for computation necessitates the use of a distributed file system (DFS). Currently, there are frameworks that use such file systems such as MapReduce [9, 10], Dryad [11], and Spark [12]. Presto [13] and Hive [14], are other frameworks which do not use a distributed file system, but rather a simple storage abstraction, this can be used on top of a DFS. Presto is used for Structured Query Language type commands in distributed analytics. All of these frameworks start by loading data from the storage tier in order to do computation tasks. They are all working are large data-sets which have the sequential read heavy characteristic [15].

Typically distributed file systems are designed for fault-tolerance [16]. The primary motivation for replication is reliability. Should one node malfunction and the data stored at that node become unavailable, any task that requires the data can not execute as the necessary piece of data is unavailable. For example, distributed file systems use inter-node replication of files to increase availability. The goal is to make effective use of the nodes in the cluster, by placing data at strategic locations. Inter-node replication of files ensures availability in such scenarios. This helps maximize the use of network, storage, and compute resources. In addition, such replication is also leveraged for performance. A "shared-nothing" cluster – the standard design of a scale-out cluster – ensures that each node is self sufficient. This means each node has its own compute and storage resources although other remote resources may be

available. Replicating the file increases the available nodes that can execute the task locally, allowing better utilization of cluster resources and availability of data.

1.3 Goals and Contributions

The goal of this thesis is to develop techniques to improve the performance of file systems that access large files (64MB-1GB+). The contributions of this work span both the local file system and the distributed file system layers. There are the main components of this work.

- A utility driven replication technique that improves utilization of storage device capacity at the local file system. The key insight of my technique is careful allocation of replication capacity. Capacity is provided to files that can extract the maximum utility (typically popular, read-mostly files),

- A load adaptive scheduler that improves the throughput of the underlying storage devices at the local file system layer for small-scale storage systems (i.e., nodes with 4 or fewer storage devices). The key insight of my technique is that dynamic load-aware scheduling switches between striping across storage devices (at low loads) and single storage device access (at high loads) for best case performance. My collaborator on the `MorphStore` project also pursued an alternative access scheduling approach [17]. It employs integer linear programming (ILP) to compute optimal access schedules. It uses offline computation of schedules and online lookup in a table (which contains the schedule). The work presented within does not use ILP-based scheduling, rather a simpler scheme based on load history.

- A capacity-performance tradeoff for the distributed file system that maximizes node utilization by capitalizing on common case local tasks. The key insight is that inter-node replication provides diminishing returns via the saturation of locality. Instead local task performance can be improved by intra-node replica-

tion of blocks. This provides improved throughput for the common case. This approach allows capacity to be traded for performance while maintaining the desired level of availability.

These contributions allow one to trade capacity for performance at the local or distributed file system layer. The use of replication allows the effective use of storage device bandwidth. These techniques improve the overall performance of the system.

1.4 Organization

This dissertation consists of the following chapters. First, in Chapter 2, I will discuss the background of techniques used in storage systems. In Chapter 3, I will discusses throughput improvement techniques that increase performance for local file systems. Next, in Chapter 4, I will discusses a distributed file system that leverages intra-node replication to offer a capacity-performance choice for I/O bound workloads. The last chapter, Chapter 5, discusses the impact of this work.

2. BACKGROUND

I will discuss the basic configurations of storage devices. These are static and do not allow the flexibility to tradeoff capacity for availability or performance. I will also discuss the tiered storage systems that use multiple configurations, or a caching configuration to provide static configurations that do not allow for a capacity tradeoff.

2.1 Redundant Arrays of Inexpensive Disks

Redundant Arrays of Inexpensive Disks (RAID), have been around for many years [1] and are in use throughout systems today. There are three basic types of array configurations, Figure 2.2 illustrates these configurations. Others exist, but for the most part they are a mixture of one of these basic types and parity.

- `Striping` (RAID-0) is the configuration where one places a chunk of data on individual storage devices then accesses the data as a stripe across all storage devices. This configuration will aggregate bandwidth across disks.

- `Mirroring` (RAID-1) is the configuration where one places a copy of the data on another storage device. One can then steer accesses to a storage device that is available to serve the data. This configuration will utilize single-disk bandwidth of each disk in the configuration, effectively aggregating the disks bandwidth.

- `Concatenation` (JBOD) is the configuration where one linearly concatenates storage devices. All devices are able to store individual files, no copies of data are made on this configuration. This configuration will aggregate bandwidth of disks based on file system block allocation which spaces files to lower fragmentation.

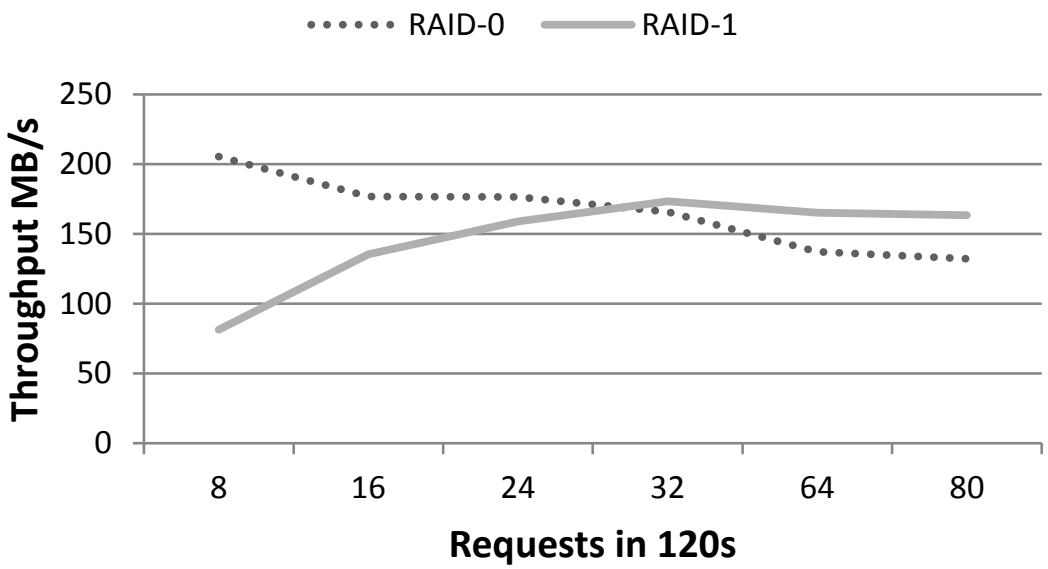


Figure 2.1.: RAID Under Load (Low to High)

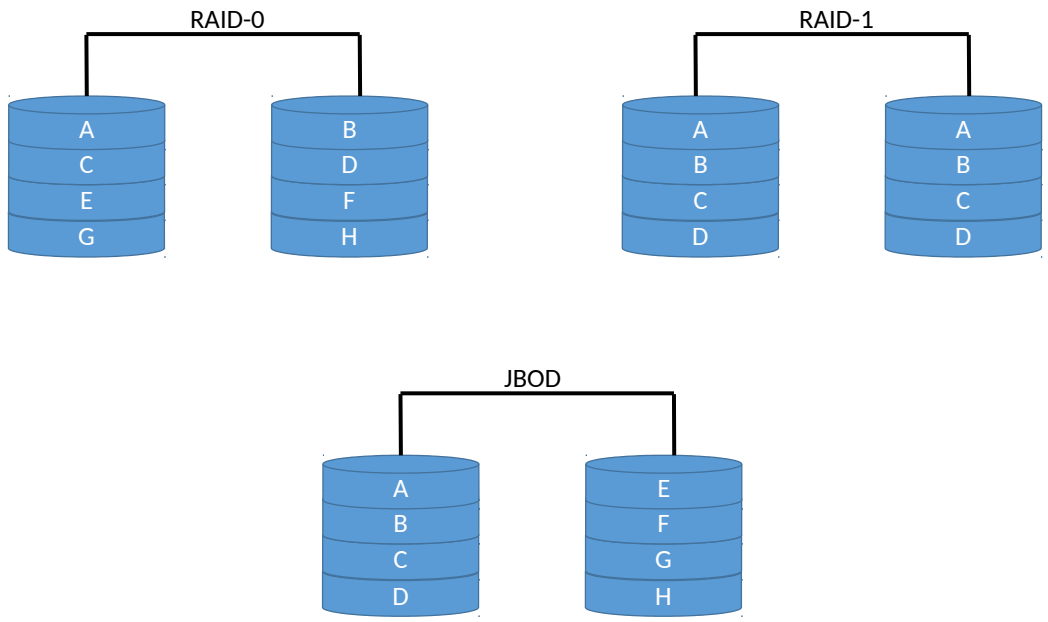


Figure 2.2.: RAID Configurations

These two techniques are the foundation for throughput aggregation. Striping is beneficial when only a few large sequential access occur. Mirroring is beneficial when there are a lot of large sequential accesses. This behavior can be seen in Figure 2.1.

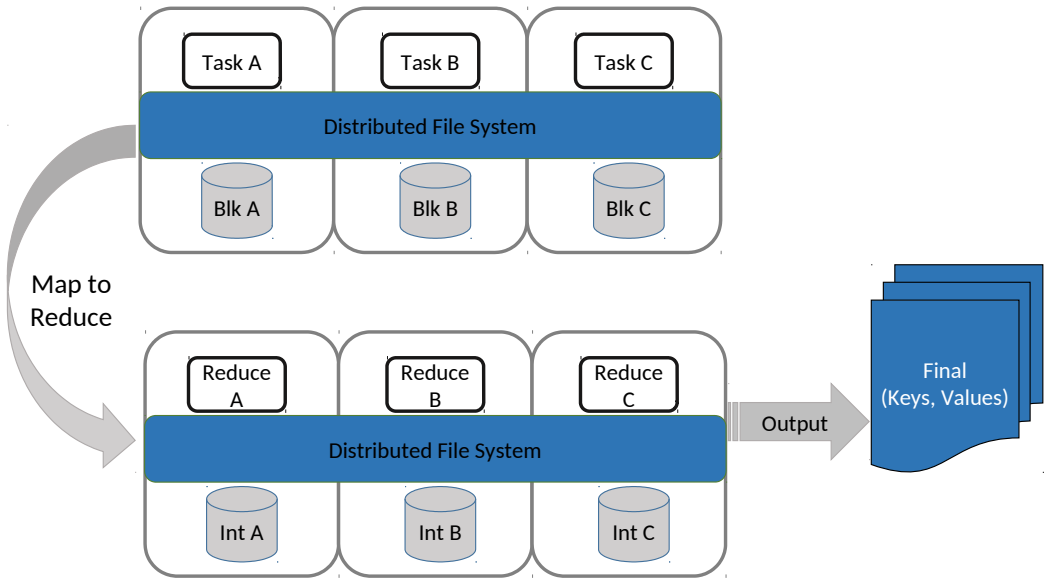


Figure 2.3.: MapReduce Framework Functionality

These configurations suffer from static allotment of capacity. Even those that attempt to provide a mix of two static configurations [18] must migrate data between those static configurations and do not allow a capacity performance tradeoff.

2.2 Tiered Storage Layers

A tiered storage layer provides a fast medium (cache) to serve data and a slow medium to store the data long term. This can be either a faster storage device [19], such as a solid state drive, or a location in memory to hold data [20]. These tiered storage systems work well for workloads that iterate over a working set, as that working set will be located in the faster storage medium. However caching is ineffective when the characteristic is sequential large files accesses.

2.3 MapReduce Framework

Throughout this work the Hadoop MapReduce framework [10] will be discussed. This framework is one of a few that provides ease of programmability to analyze large data-sets. I will discuss the overall functionality of this framework in this section.

Figure 2.3 illustrates the MapReduce [9] framework flow. MapReduce executes tasks on nodes in a cluster. These nodes hold data that the tasks use to compute an intermediate (key, value) pair. Once the intermediate (key, value) pair are computed, this data is then sent through a reduction to combine keys and output the final (key, value) pair.

The distributed file system that connects all the nodes in the cluster facilitates this style of computation. Tasks are assigned to nodes which contain a local copy of the block the task will read as input. This is called a local task. Tasks, are either local or remote. A remote task must fetch the data it wishes to read off another node. Local tasks are preferable as they better utilize system and network resources.

Nodes are prone to failure, thus to assure availability, blocks are replicated by a specified factor. These replicated blocks are placed on different nodes such that if one node fails, that block can be available from a different node.

3. A LOCAL FILE SYSTEM FOR BIG DATA WITH UTILITY-DRIVEN REPLICATION AND LOAD-ADAPTIVE ACCESS SCHEDULING

3.1 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 [21]. Well-known disk performance optimizations such as striping, replication and combinations thereof have targeted this bottleneck [18, 22–24]. 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 multi-disk 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 mirroring. 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 in the file system. `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 this 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. A replication strategy is developed that is guided by access statistics (from prior profile runs) to answer both of the questions. This profile-guided, utility-driven replication strategy is based on the assumption that the profile access pattern which is used is a reasonable expectation of future access patterns (capitalizing upon historys periodicity). In general, `MorphStore` may use any other source of profile data 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, the profile calibration is solely in the purview of the administrator. Once the degree of replication is established, `MorphStore` 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.

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

`MorphStore` is evaluated using a combination of micro- and macro-benchmarks. The microbenchmark-based evaluations highlight the clear advantage of `MorphStore`'s load-adaptivity; `MorphStore` is consistently closer to the best of either mirror-only or striping-only strategies. The macro-benchmark based evaluations show that `MorphStore` uses significantly less replication capacity than RAID-1 (mirroring) while still achieving 12% average performance improvements on a video server workload and 8% average performance improvement on a NoSQL workload.

In summary, the major contributions are:

- The development of a storage architecture – `MorphStore`– that achieves load-adaptive performance improvements for disk-bound applications.
- `MorphStore` uses a profile/expectation-guided 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 mirroring/striping strategies.

The remainder of this chapter is organized as follows: Section 3.2 offers a brief background on disk and file system organization. Section 3.3 and Section 3.4 describe the design and implementation of `MorphStore`, respectively. Section 4.5 describes the experimental methodology. Section 4.7 presents experimental results. Finally, Section 4.9 summarizes the impact of `MorphStore`.

3.2 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 [1], 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, Specifically, I consider JBOD, RAID-0 and RAID-1 organizations. Other levels of RAID are based on RAID-0 and RAID-1 with the addition of parity for reliability. I focus on the performance impact of disk array organization, rather than reliability, and thus the consideration of only the base configurations. Let us consider the following as the key tradeoffs of these base configurations. 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 by this configuration. 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 (i.e., performance is degraded). One may reduce the relative fraction of seek overhead for larger files by using larger block sizes. However, the larger the block size, the greater the possibility of extended disk unavailability because block accesses cannot be preempted. In practice, I limit RAID-0 block sizes to 2MB which can keep disks busy for up to 50ms. (Larger block sizes can make the sytem unresponsive to critical events such as page-faults, network events and user interactions.)

`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, which negates the necessity to configure disk resources as a combination of striped and mirrored.

Local vs. Global file systems Distributed file systems like Google File System (GFS) [25] and the Hadoop Distributed File System (HDFS) [3] are used in big data analysis. However, such file systems are typically overlaid on top of local file systems such as ext2/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.

3.3 `MorphStore` Design Overview

High-level design choices are made to facilitate the design discussion. First, `MorphStore` is implemented at the file system level. One may wonder why, as competing replication techniques (e.g., RAID-0 and RAID-1) are implemented at the block/device layer, so this design choice needs some justification. The design goals for `MorphStore` are two-fold. First, it is necessary to manage replication and access scheduling for large files based on load levels, popularity, and read-write ratios. 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 this state. This also allows flexible capacity allotment for replication, without the need to transition between configurations. Second, it is also necessary 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, two key components of `MorphStore` follow: load-adaptive access scheduling and utility-driven replication.

3.3.1 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). It is necessary for the adaptive mechanism to react to load levels, therefore, the number of open inodes in a recent (tunable) window of time is tracked. 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 is set to a factor of 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 3.2. `MorphStore`'s approach to exploit intra-request parallelism achieves the same degree of disk parallelism as RAID-0. However, In contrast to RAID-0, 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 3.1). On mechanical drives this requires adjustment to the next block. 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. Should other technology besides mechanical storage be used, `MorphStore` may be less impacted by the locality.

For the non-striped accesses, ideally it is desired for `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, it is 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 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, in an attempt to load balance, requests that are not yet associated with any device are steered to the most lightly loaded device.

3.3.2 Utility-driven Replication (UDR)

The second component of `MorphStore` is utility-driven replication which aims to use replication capacity to maximize performance. The model assumed 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 dy-

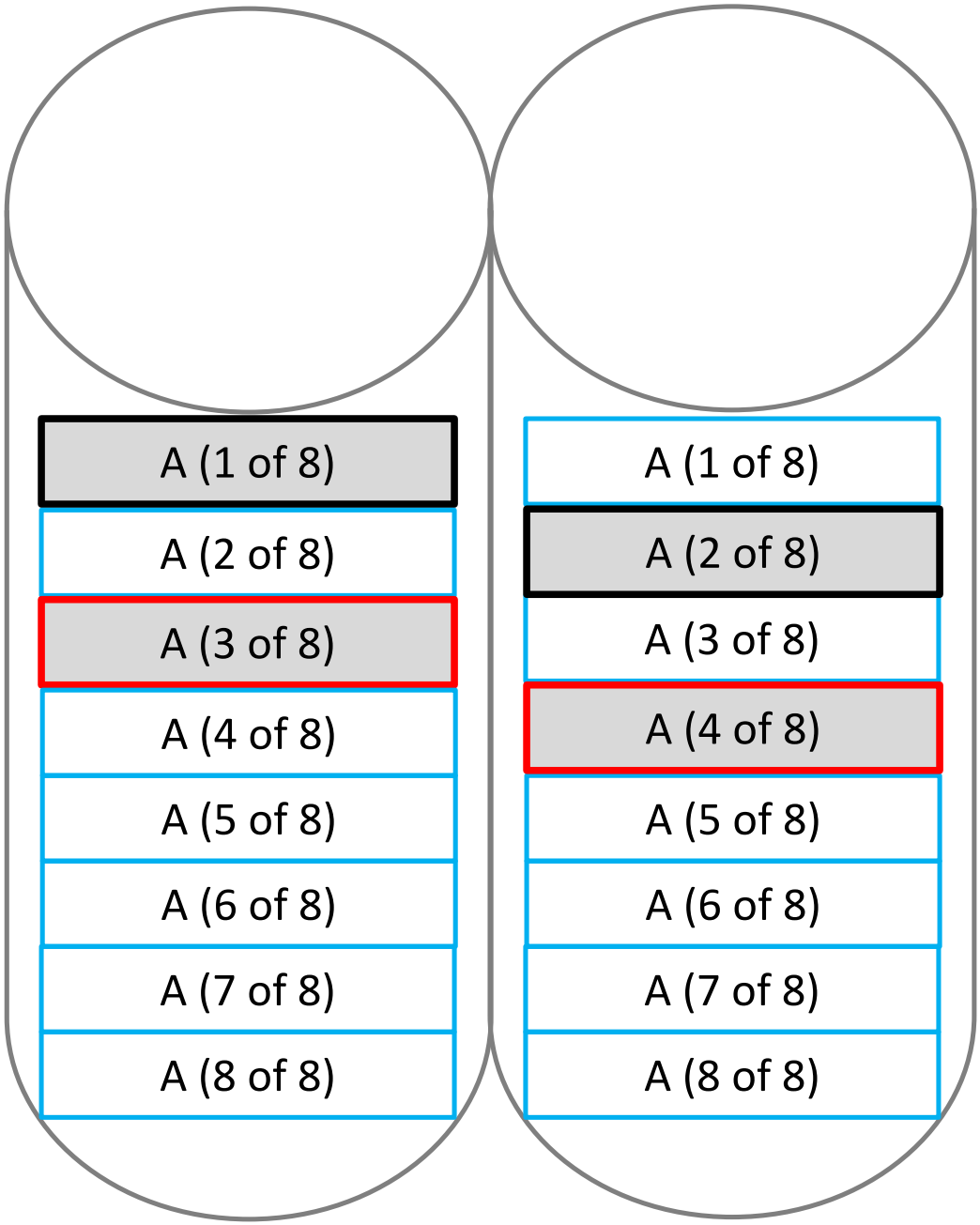


Figure 3.1.: MorphStore Stripe

namic[1] replication allows highly utilized files to be replicated on additional disks in

[1]Note, that UDR is referred to 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.

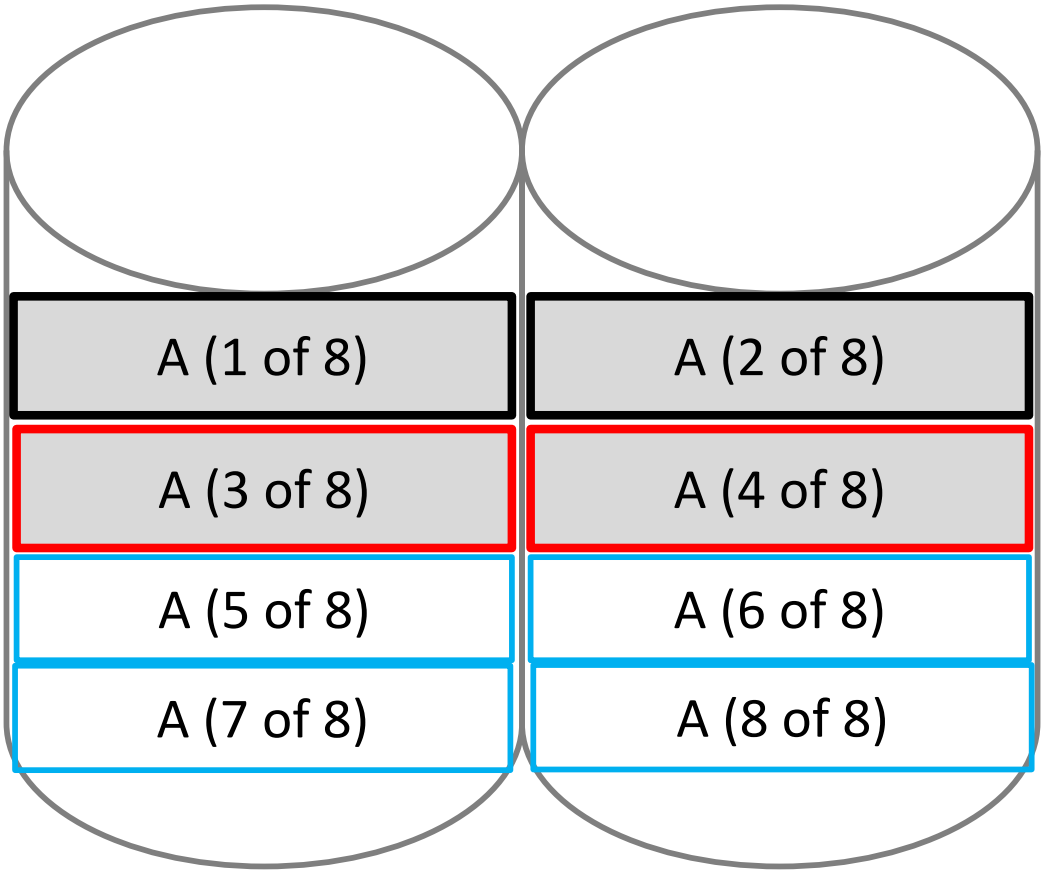


Figure 3.2.: RAID-0 Stripe

the disk array, including the originating disk. Such replicas enable the load-aware access scheduling that was 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.

3.3.3 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, one can define this cost for a given file as $file.reads/file.replicas + file.writes \times file.replicas$. With this cost-metric, the incremental utility of an additional replica is defined 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.

A priority queue Q is used 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, the replication factor for that file can be increased by 1 while decreasing the system capacity by the file size; otherwise it is marked 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.

The operation of the *REPLICATE* algorithm is illustrated with an example in Figure 3.3. 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

Algorithm 1 REPLICATE(Q)

```
while Q is not empty or capacity > 0 do
    file ← dequeue Q
    utility ← [file.reads/(file.replicas) + file.writes * (file.replicas)] −
    [file.reads/(file.replicas + 1) + file.writes * (file.replicas + 1)]
    if utility > 0 then
        file.replicas ← file.replicas + 1
        capacity ← capacity − 1
    else
        add file to done list
    end if
    if file.replicas = disks then
        add file to done list
    else
        enqueue(file)
    end if
end while
```

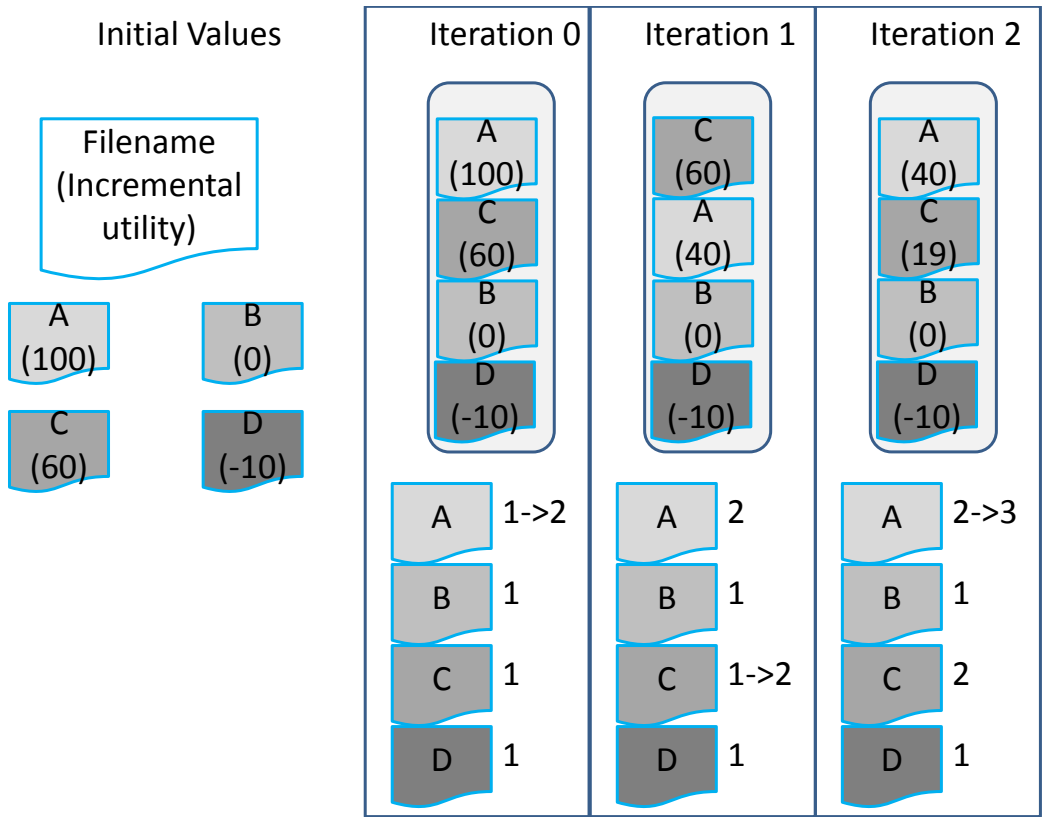


Figure 3.3.: Replication Strategy

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.

3.4 Implementation Details

The `MorphStore` design may be implemented as modifications to any file system. For this study, I implemented `MorphStore` as a modification of Linux's second extended file system `ext2` [2] 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* [26], which allow viewing the base file system and modifying its attributes on disk. The following sub-sections will discuss the implementation of `MorphStore`. First, I present a brief discussion of some implementation challenges. Second, I discuss the meta-data used to maintain the replicated data. Third, I discuss the replication operation. Finally, I will close implementation details with reads and writes to replicas (the replicated files).

3.4.1 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, the file is moved 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, an unlikely scenario – one in which a large number of small devices and serious capacity pressure – would have to be considered the common case.

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.

3.4.2 `MorphStore` Meta-data

The meta-data for `MorphStore` consists of two parts, on-disk meta-data and in-memory meta-data. The former meta-data is in the form of special directories and file attributes located on the disk, and the latter meta-data is a data structure attached to the kernel's VFS inode structure. Most frequent data 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 3.4.1 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, it leverages existing meta-data mechanisms, and augments 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.

The existing meta-data structures are used, and augmented 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 subsequently not replicated, and no reads/writes occur to replicas explicitly (i.e. via file system API); they are only accessed as a byproduct of accessing the master file.

3.4.3 `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. This information is used to create the 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. This provides an opportunity to again update the extended attributes where `MorphStore` stores file read/write statistics. Piggybacking on this update affords the minimization of reads and writes to the device for meta-data. The tradeoff is that should the system crash the updated meta-data will be lost (as it has not yet been written to the backing store). 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 request is simplified. The

Table 3.1.: Test System Configuration

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

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 3.3.1.

`MorphStore` handles writes in a similar fashion to reads. Writes to a file occur to all copies of the file (a necessity of maintaining replicas). 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 (mirrored).

3.5 Evaluation Methodology

Two tools are used: *Filebench* [27], and *IOStat* [28] to gather information from the file system level and the device IO level, respectively. The test system is configured as shown in Table 3.1, with four devices configured in either JBOD (linear personality), RAID-0 (striped personality), or RAID-1 (mirrored personality). Each configuration is then formatted with the ext2 file system to test performance, and `MorphStore` is loaded, as the file system driver (in place of ext2), additionally when using the JBOD configuration to test the performance of `MorphStore`.

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, all of these parameters are tunable by the system administrator.

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 Two workloads are used 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. A benchmark that mimics the IO access patterns of MongoDB is used, in lieu of an actual database. 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 the ability to scale between low and high load scenarios with these workloads. They both also use large (GB) files for IO accesses.

Filebench is used to generate a workload from the video server template, with an average file size of 1 GB, This workload is scaled from a low load to a high load, which increases the number of requests sent to the device array. Flowops are also defined 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`.

3.6 Results

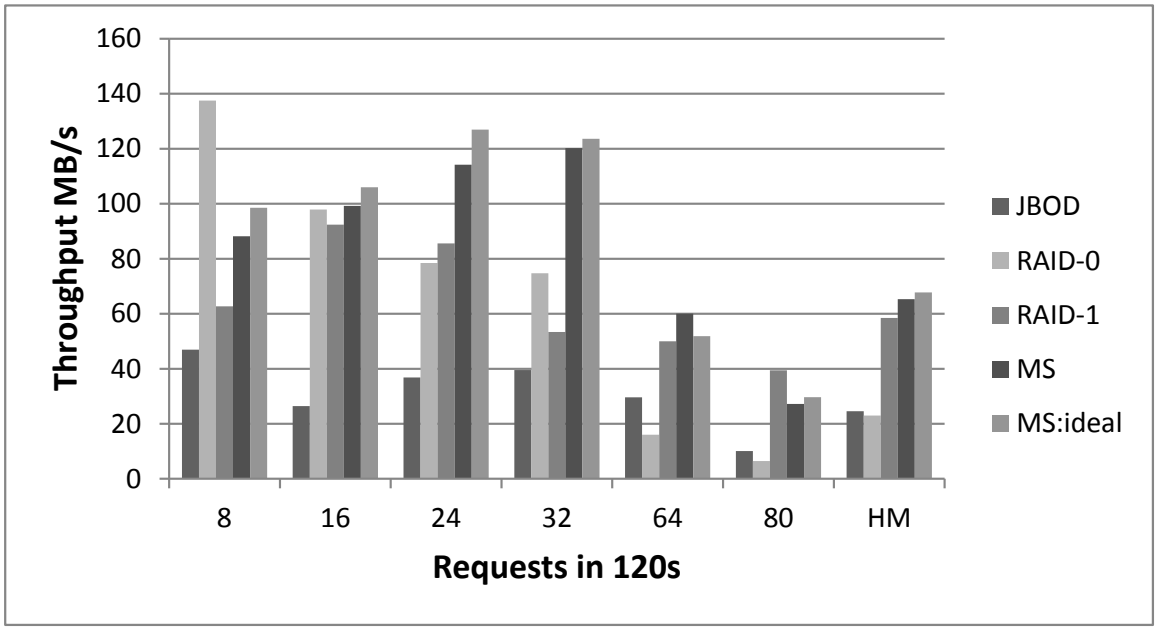
The key findings reveal that `MorphStore` achieves significantly higher performance. These findings also validate the intuition behind each of the proposed techniques.

- In both workloads video-server and MongoDB, `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 (i.e. capped by capacity). While this does diminish the gains, results reveal that significant gains can be achieved with minimal space overhead.
- Isolating the load-adaptive access scheduling technique, it is observed 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 to within 4% of performance as an ideal (but impractical) replication scheme that has perfect knowledge of file access patterns.

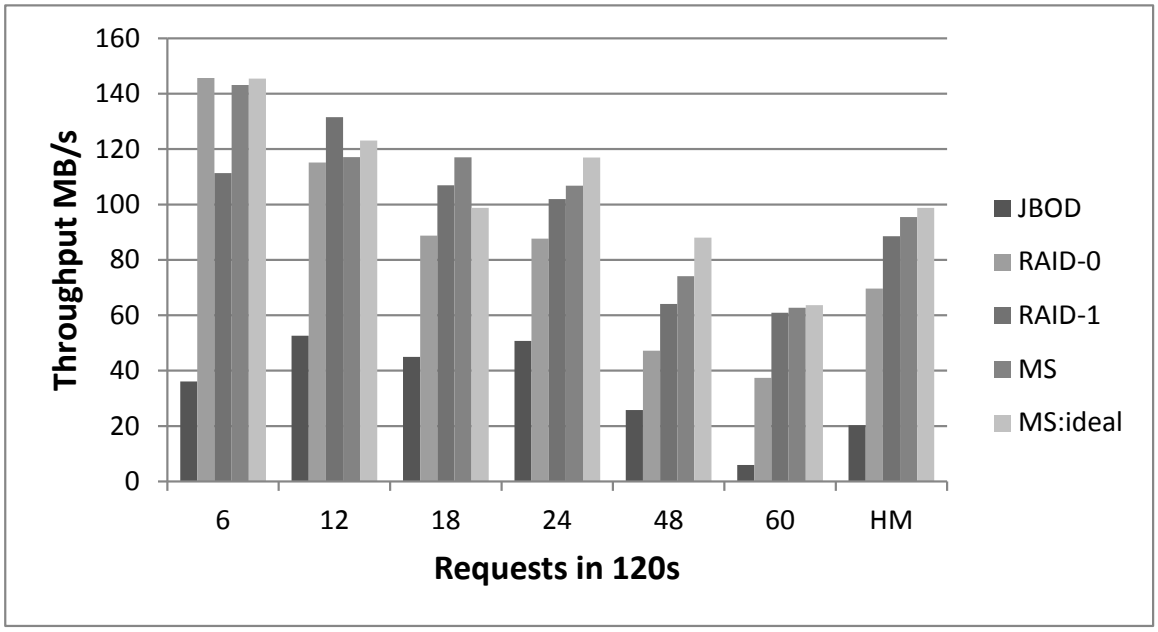
The remainder of this section elaborates on each of the above results.

3.6.1 `MorphStore` for Video Server and MongoDB Access Patterns

Figure 3.4 plots the file system data throughput (Y-axis, MB/s) for a range of load levels (X-axis, requests/second) for the two workloads (two graphs). For each load level (group of bars), the graphs show the performance of JBOD, RAID-0, RAID-1 and `MorphStore` (MS). An additional bar is included in each group for an Ideal-MS which achieves ideal replication. Finally, the harmonic mean (HM) throughput is also included for each configuration in the final (rightmost) set of bars.



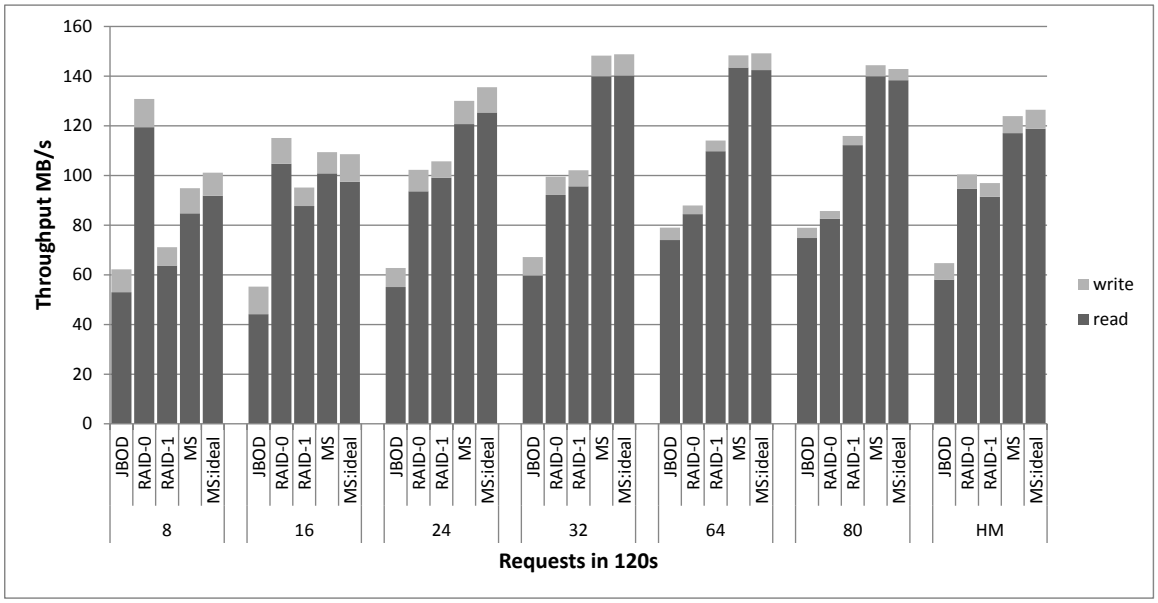
(a) Video Server



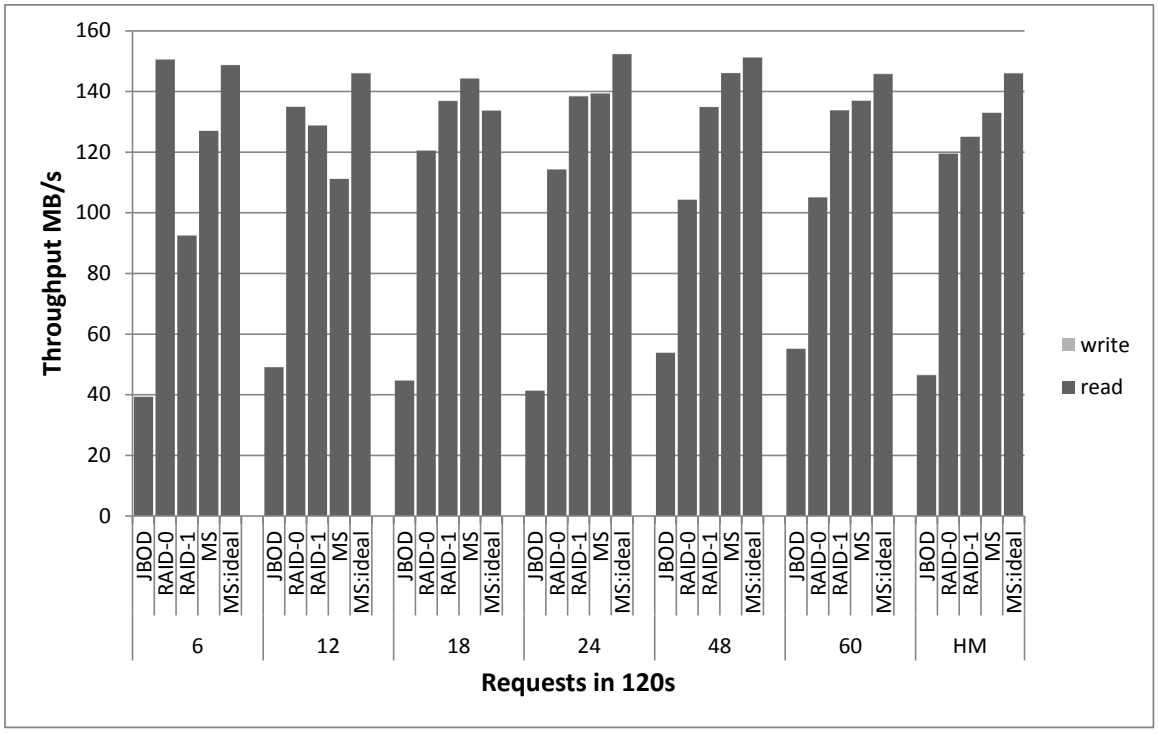
(b) MongoDB

Figure 3.4.: File System Throughput

Figure 3.4 leads to four key observations common to both workloads. First, as expected, RAID-0 achieves the best performance at low loads and it significantly



(a) Video Server



(b) MongoDB

Figure 3.5.: Device (disk-array) Throughput (IOStat)

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 (sequential access). 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 3.4(a) for the video server workload. The corresponding speedups over RAID-0 and RAID-1 for the MongoDB,Figure 3.4(b), workload are 37% and 8%, respectively. Finally, it is observed 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), the throughput of the disk array (i.e., from IOStat [28]) is also measured. Figure 3.5 uses similar axes and grouping as the earlier Figure 3.4 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.

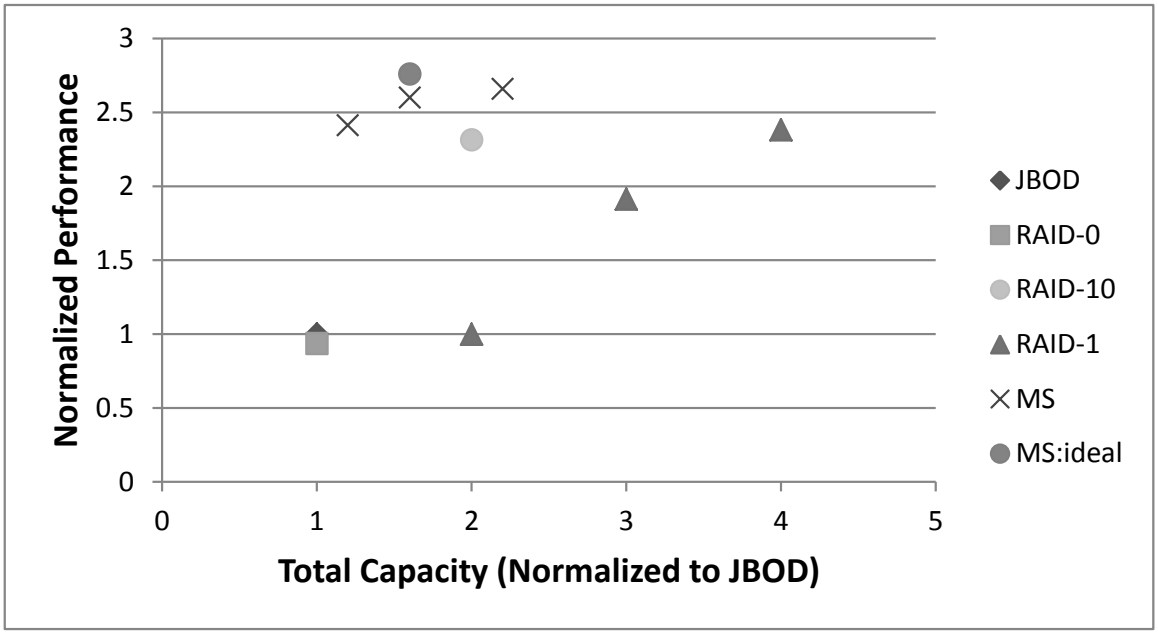
The trends remain unchanged for RAID-0; RAID-0 is best at low loads. RAID-1's throughput saturates at high loads (unlike the file system throughput which degrades

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 high read/write ratio. This is not surprising as the read-heavy nature of datastores is widely reported [29–31].

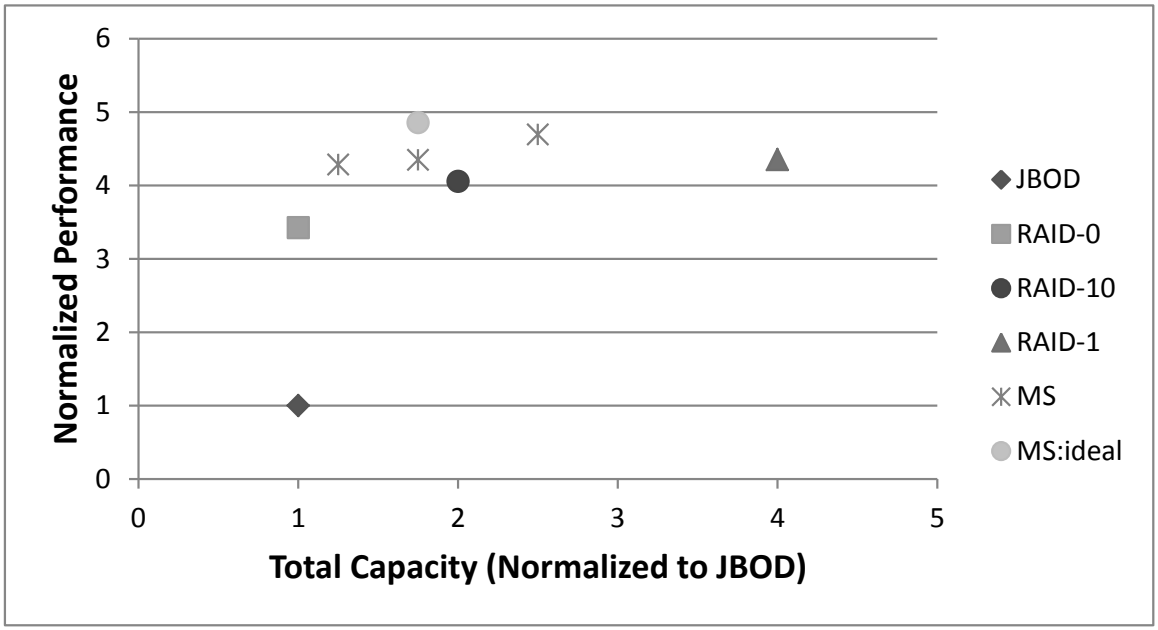
3.6.2 `MorphStore` Under Capacity Constraints

In the previous experiments, it is assumed that `MorphStore`'s UDR replicated all files until the system ran out of disks (i.e., there are as many replicas as disks) or the system ran out of files with positive utility. No artificial constraints were imposed 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. In this section, `MorphStore` is examined under various capacity constraints for performance.

Figure 3.6 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. The performance of `MorphStore` is plotted against varying total capacity. For the video workload the total capacity is varied in multiples of 1.2X, 1.6X, and 2.2X of the baseline JBOD capacity. Similarly, for the MongoDB workload, multiples of 1.25X, 1.75X, 2.5X for the baseline JBOD file system capacity are used. 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.



(a) Video Server



(b) MongoDB

Figure 3.6.: Capacity vs Performance: The Pareto Frontier

For the baseline systems, the X-axis effectively shows the inherent replication capacity used by such systems. For example, when a RAID-1 configuration is used

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, 2-way and 3-way mirrored RAID-1 configurations are also included; 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, a RAID-10 configuration is included 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, it is desired that systems be in the top-left corner of the space because this will maximum performance (higher) with the minimal replication (toward the left). The points on the upward-left facing frontier represent this 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 3.6 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.

One may think that with 4x the number of disk devices, the performance will also be at 4x. This is true for MongoDB, Figure 3.6(b), as RAID-1 does achieve slightly better than 4x, as well as RAID-10 (which also has 4 devices). RAID-10 however, only requires 2x the replication capacity where as RAID-1 uses 4x, yet performs just as well. This is due to the mix of striped and mirrored accesses, which help aggregate bandwidth, as well as steer accesses. Notice that for video server, Figure 3.6(a), RAID-1 performance is at around 2.5x, this can be attributed to the updates (writes, as seen in Figure 3.5(a)) in the video server workload which occupy all devices for each write.

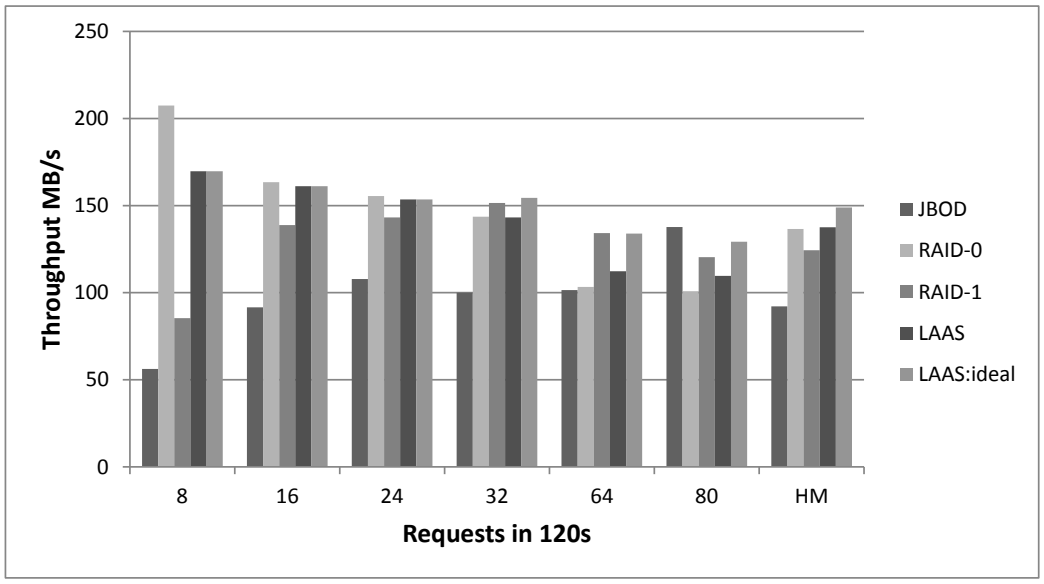


Figure 3.7.: Filebench LAAS RO Throughput

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 more and more replication capacity is used. (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.

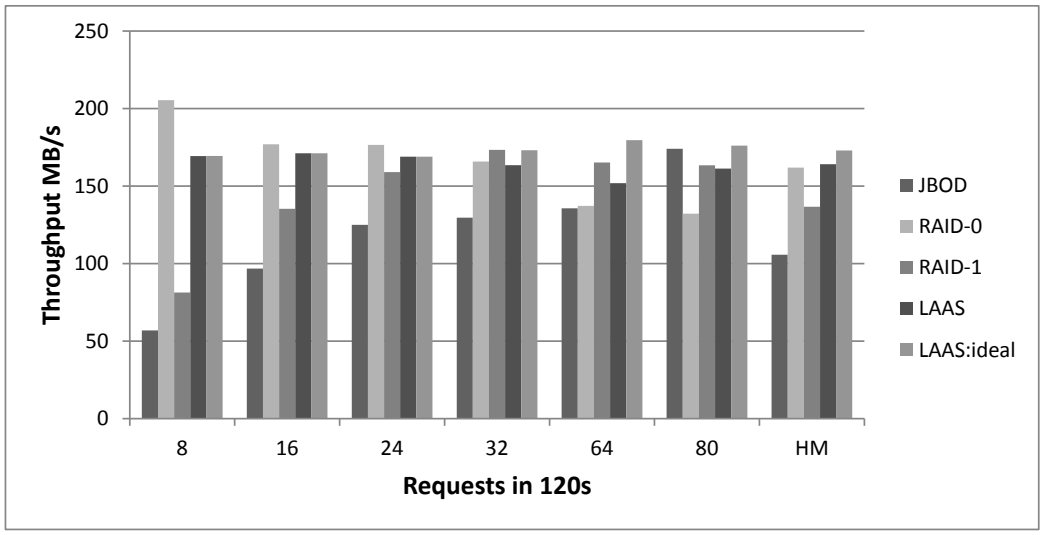


Figure 3.8.: IOStat LAAS RO Throughput

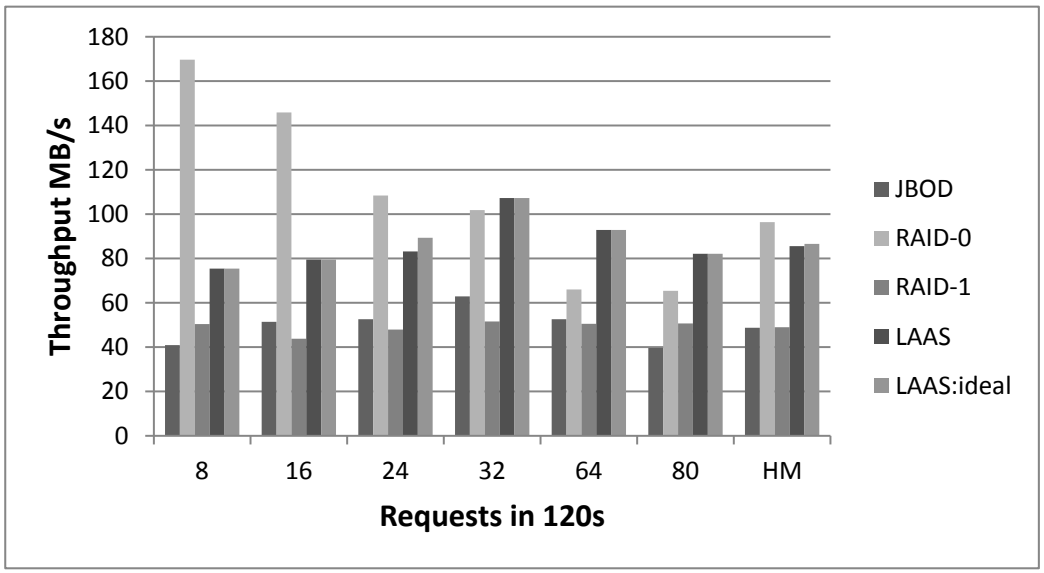


Figure 3.9.: Filebench LAAS Replication Cost Throughput

3.6.3 Isolating the Effect of LAAS

To isolate the impact of LAAS from UDR, let us revert to using RAID-1 style full mirroring with `MorphStore`. For this subset of results, evaluation is limited to the video server workload.

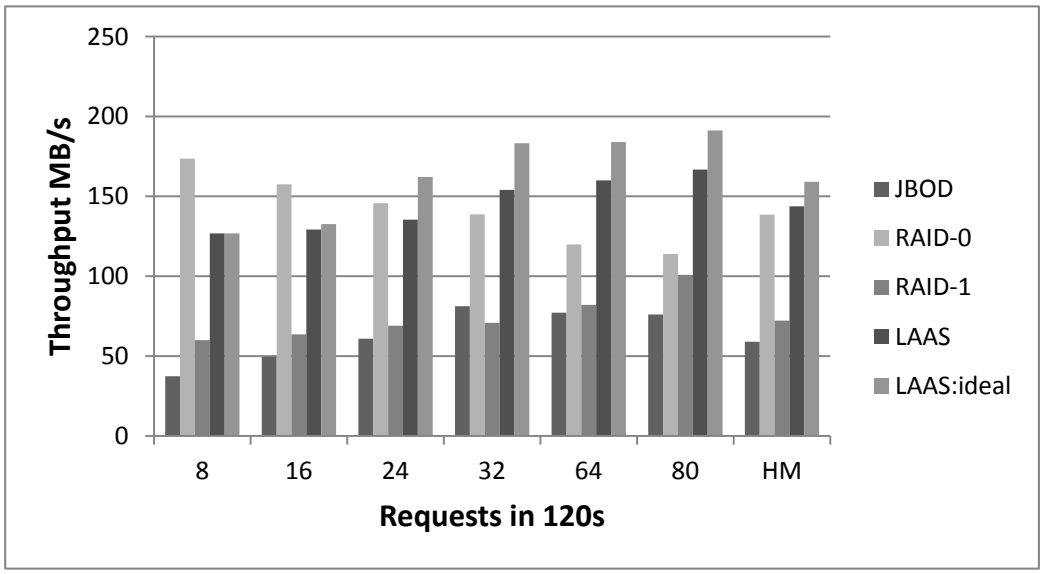


Figure 3.10.: IOStat LAAS Replication Cost Throughput

The performance of such a LAAS-only design is captured in Figure 3.9 and Figure 3.10. Because the disk requests include writes, and because writes interact poorly with replication, one sees a sharp decline in RAID-1, as the devices are utilized for mirroring and must also verify writes as redundancy 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, a similar configuration but with only reads is employed.

Figure 3.7 contains the file system characteristics of the following techniques: JBOD, RAID-0, RAID-1, and `MorphStore`. Figure 3.8 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` out performs 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` out performs RAID-1 by 21% and RAID-0 by 2% in Figure 3.8.

3.7 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 chapter focuses on the challenge of providing high performance for big file manipulation on local file systems.

My 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 mirroring via dynamic replication to advance the Pareto frontier and provide dynamic design alternatives to existing static techniques.

4. LEVERAGING INTRA-NODE REPLICATION FOR A BETTER BARGAIN WHEN TRADING CAPACITY FOR PERFORMANCE IN MAPREDUCE

4.1 Introduction

MapReduce [9] and other data analysis frameworks such as Dryad [11] and Spark [12] have emerged as a powerful tools that offer the twin benefits of easy programmability and automatic orchestration of large scale computation over large collections of servers. Such frameworks focus on computations that span a very broad range of behavior; some reorganize data (e.g., sort, reverse index), some summarize data (e.g., word-count, grep). Although my technique is broadly applicable to these frameworks, I focus on the MapReduce framework for the remainder of this chapter for ease of exposition.

MapReduce uses an underlying global file system (e.g., GFS, HDFS) which is built on top of the local file systems on individual servers. The global file system offers two key benefits that are important for MapReduce. First, the global file system allows for transparent replication of files across multiple nodes' local file systems. Such replication offers the twin benefits of improved data availability on server failure *and* improved performance by maximizing the opportunity to co-locate computation with data. Second, the global file system ensures that all files are accessible from all nodes which enables remote execution of map-tasks. Remote execution of tasks, although uncommon, is important because it maximizes server utilization by avoiding the situation wherein a task must wait to be executed even though a node is idle.

The interaction of MapReduce computation with the underlying global file system (e.g., GFS, or HDFS) creates an interesting tradeoff between replication/capacity on the one hand, and performance and availability, on the other. Using a higher degree of

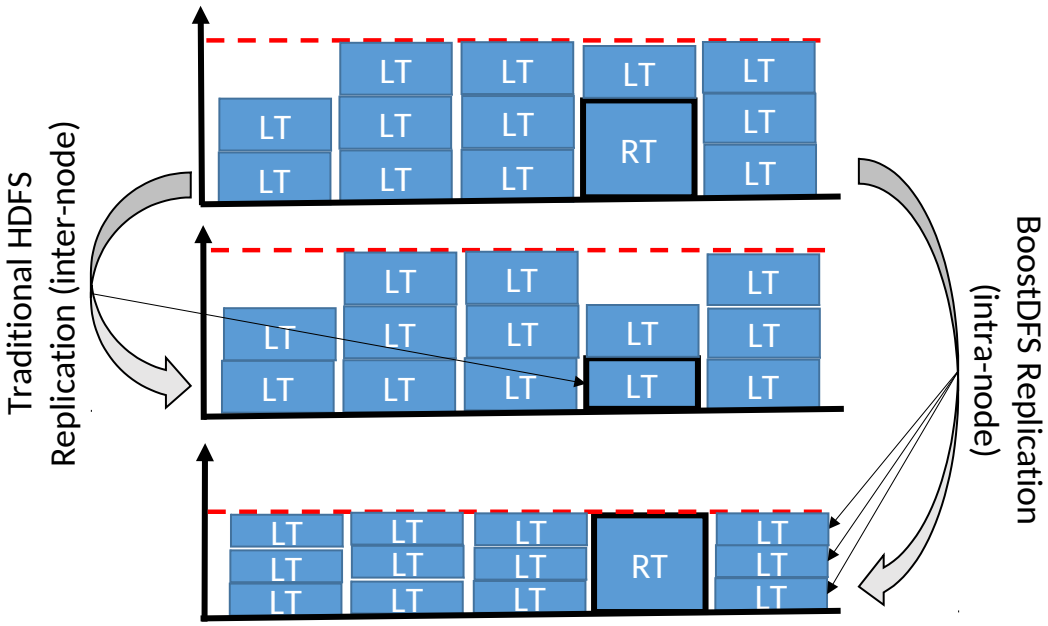


Figure 4.1.: Inter-node vs. Intra-node Replication (Deep Stack of Map Tasks on Each Node)

inter-node replication in HDFS offers two benefits: (1) performance benefits, because it reduces the probability of (slower) remote tasks, and (2) availability of data when servers fail, because data is present on additional servers. Both these benefits exhibit a pattern of diminishing returns because high task locality and high availability can be achieved with a modest number of inter-node replicas (e.g., 3 replicas achieves 99+% local task execution). Such diminishing returns reduces the incentive to tradeoff storage capacity for increased performance and reliability.

Consider the challenge of performance and reliability separately. For performance, the key challenge with the above tradeoff is that it expends storage resources to improve the performance of the uncommon case – i.e., remote tasks are a small fraction of the total number of tasks. Large fractions of map tasks are typically local tasks. In this chapter, I detail a design to overcome this drawback by using intra-node replication (instead of HDFS' inter-node replication) to target and boost the performance of the common case – i.e., the performance of tasks that were already

local. `BoostDFS` leverages the additional intra-node replicas to achieve higher disk bandwidth, which baseline systems (e.g., based on JBOD) cannot fully capture.

Figure 4.1 illustrates the above tradeoff. The top graph in Figure 4.1 shows the map task execution times (Y-axis) for a set of five machines (X-axis) for a baseline Hadoop run. For each machine, *local tasks (LTs)* – tasks that execute on a node that holds the data they operates on, and *remote tasks (RTs)* – a task that must fetch data from a remote node before it can begin map computation are shown as stacked boxes. Figure 4.1 includes the impact on performance due to traditional HDFS replication (middle graph) and our `BoostDFS` replication (bottom graph). Even though the traditional replication scheme may reduce the execution time of the remote map task by leveraging the additional replicas to achieve local execution, the overall improvement is minimal because only a small fraction of execution time is reduced. In contrast, `BoostDFS` reduces the execution time of a large fraction of LTs. Even though `BoostDFS` does not decrease the number of remote tasks, it achieves better overall performance because the common case performance is optimized.

One may think that the traditional approach of targeting remote tasks is appropriate because it targets the slowest map tasks. As shown in Figure 4.2, if one considers a tail-latency effect wherein the overall map completion time is determined by the remote tasks, indeed speeding up the local tasks does not result in any improvement in overall map time. However, if Hadoop is configured with an adequate number of map tasks (e.g., the original MapReduce paper [9] recommends 100 times as many map tasks as number of compute nodes), combined with dynamic load balancing inherent in Hadoop, speeding up local tasks reduces overall execution time.

To that end, I propose `BoostDFS`– a distributed file system organization with an intra-node replication strategy to enable higher performance for local tasks which are the common case. The first contribution is an analytical model that quantifies the nature of the capacity/performance tradeoff. This model provides intuition that guides the `BoostDFS` design. `BoostDFS` uses intra-node replication to boost the available disk bandwidth for local tasks which are the common case. This increase in disk

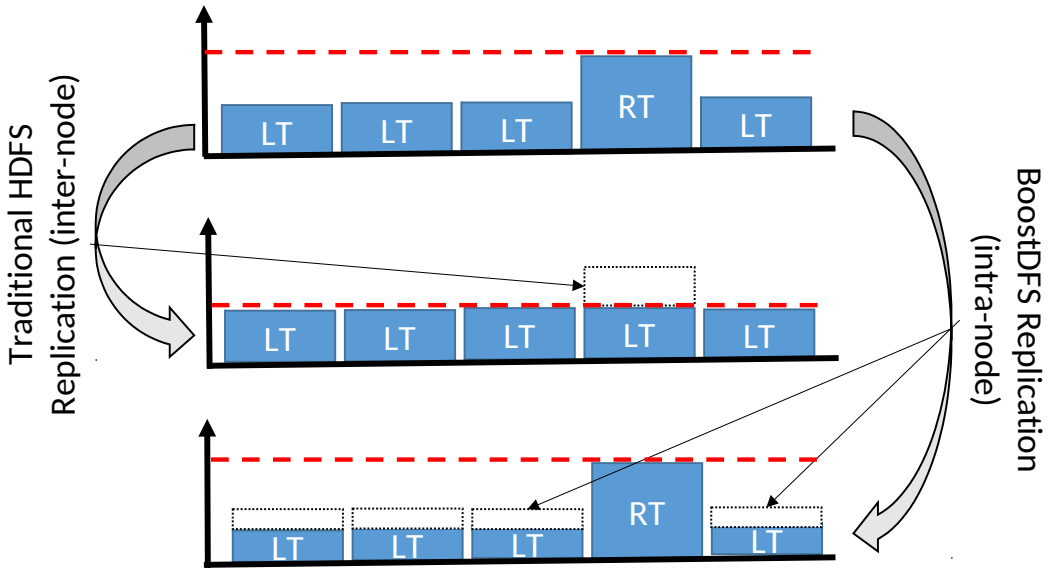


Figure 4.2.: Inter-node vs. Intra-node Replication (Shallow Wavefront of Map Tasks)

bandwidth reduces the execution time of disk-bound map tasks. Because `BoostDFS` focuses primarily on boosting local disk bandwidth, it does not offer any improvement for compute-bound map tasks.

Recall that reliability was the other metric affected by replication. Any replication/redundancy offers protection against certain classes of failures. For example, intra-node replication (e.g., RAID 1) protects against disk failures whereas and HDFS's inter-node replication protects against server failure. Intra-node replication offers no protection against data-loss under server failure. However, one can interpret `BoostDFS` as enabling the option of improving performance by trading off replication capacity (for intra-node replication) even under identical availability.

Evaluations using the PUMA benchmarks [32] on a small testbed reveals that `BoostDFS` achieves 14% performance improvement for disk-bound benchmarks and does not hurt the performance of compute-bound benchmarks. Disk-bandwidth measurements confirm that `BoostDFS`'s key advantage is the improved bandwidth it offers for a large fraction of map tasks.

The rest of the chapter is organized as follows. Section 4.3 describes an analytical model to reason about the diminishing performance returns for Hadoop applications from increasing inter-node replication in HDFS. Section 4.4 describes the basic architecture of `BoostDFS` and extends the analysis from Section 4.3 to provide intuition as to why `BoostDFS` improves performance beyond what is possible by inter-node replication alone. Section 4.4.2 explains the implementation details. I present my evaluation methodology and experimental results in Section 4.5 and Section 4.7, respectively. Section 4.8 discusses related work. Finally, I conclude this chapter with Section 4.9 which discusses the impact of `BoostDFS`.

4.2 Background

Disk-bound map reductions MapReduce computation computes over large sets of data. Broadly, they may be classified into two classes: those that *summarize* (e.g., computing/counting frequency of patterns, distributions, maximums) and those that *reorganize* (e.g., sorting, reverse indexing).

Hadoop/HDFS operation and replication/performance tradeoffs First, I offer a brief background of Hadoop operation and the key replication, availability,and performance tradeoffs to anchor the discussion in the remainder of the paper.

Consider the basic HDFS and Hadoop architecture shown in Figure 4.3, which illustrates a small cluster of three nodes, each with its own local file system. The local file system may include multiple disk drives; Figure 4.3 shows 2 disk drives at each node.

The HDFS provides a shared file system abstraction across all nodes. For ease of exposition, it is assumed that the dataset includes three files – A, B, and C – that must each be processed by a map task. Further, it is assumed that HDFS uses a replication factor of 2 because of which, each file is replicated on exactly two randomly chosen nodes in the system (e.g., file A is present on Node 0 and Node 1). Note, because of random file placement, the files may not be evenly distributed

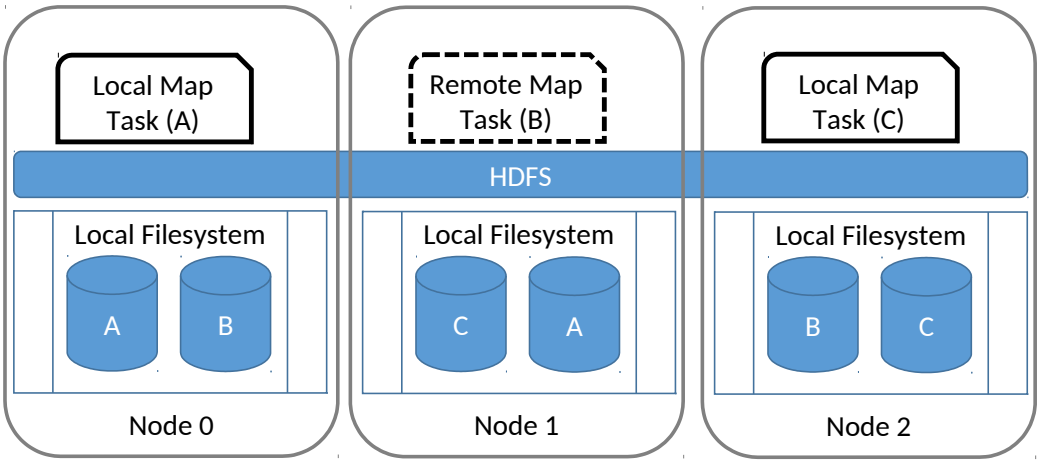


Figure 4.3.: HDFS with 2-way Replication

even though our example shows even distribution. Given the above file placement, the map task assignment shown in Figure 4.3 results in two map tasks enjoying local file access (A and C). Unfortunately, the map task processing file B must access its data remotely.

Now, consider an alternative configuration where the HDFS replication is increased to 3 (i.e., each file is now on 3 nodes, see Figure 4.4). In general, the additional replica increases the probability of local map task execution. In this small example, (with three nodes and three replicas), it guarantees that all map tasks are local as shown in Figure 4.4.

The above examples leads to two observations.

- Increasing the replication leads to improved performance because of fewer remote tasks. However, increased replication incurs two overheads. First, it incurs capacity cost for storing additional replicas, which may not result in any increased capital expenditure if there were spare disk drive capacity. Second, it also incurs increased write cost to produce the additional replica. (This additional write cost is incurred by the upstream process which produces the dataset.)

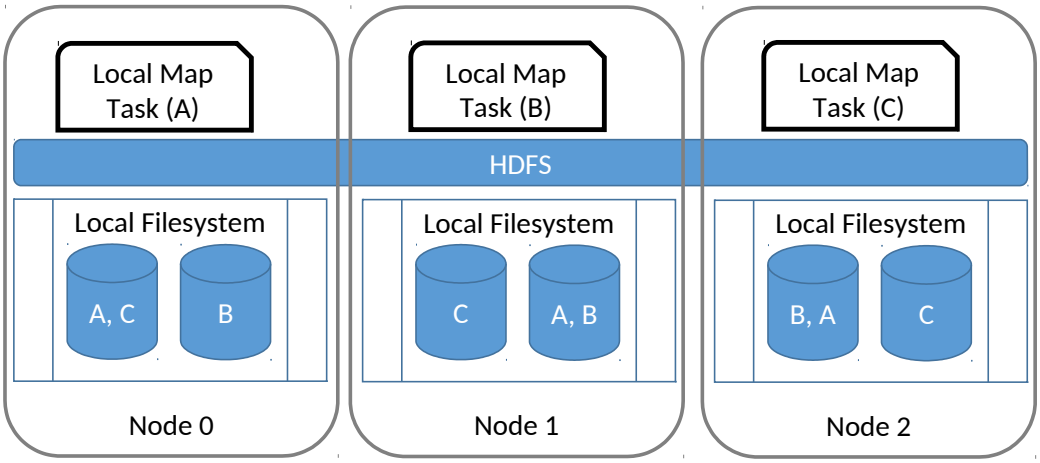


Figure 4.4.: HDFS with 3-way Replication

- In general, additional replicas result in improved data availability under server failure, which is assumed to be independent. With two replicas, both servers must fail for a file to become unavailable. Clearly, the probability of n independent failures diminishes with increasing n ($\prod_{i=1}^{n} P(fail_i)$). No attempt is made to quantify data availability under server failure. Instead, replication is used as an experimental control to equalize availability. For example two configurations where data is present on an identical number of servers is said to achieve equivalent data availability.

4.3 Hadoop Performance Analysis and Design Insight

Let us first develop an analytical model that accounts for the tradeoff between replication and map task performance. The insights gleaned from this model are used to guide the `BoostDFS` design, which aims to make the common case fast.

4.3.1 Modeling the Remote Map Task Fraction (RMTF)

The key bottleneck to address is map task execution. Several simplifying assumptions are made in developing an analytical model for the remote map task fraction.

Table 4.1.: Parameters for Model of Remote Task Execution Fraction

Param.	Description
f	Number of files in input dataset
N	Number of compute nodes in Hadoop cluster
k	Ratio of files to servers (f/N). [9] recommends $k = 100$.
r	Degree of replication in HDFS ($r = 1$ implies single copies with no additional replicas)
RMTF	Remote map tasks (as fraction of total map tasks)

Specifically, no attempt is made to model the variable map task latency and instead assume that each map task takes a unit time to complete. Second, it is assumed static and perfect load balance of task execution (realistic random file placement is modeled). In practice, with a large number of map tasks, and dynamic load balancing, the load balance is close to perfect with any imbalance being limited to the last wavefront of map tasks.

Let k be the ratio of files to servers, and thus k is the number of files each server will be required to serve. In the absence of replication ($r = 1$), the *shortfall* is defined at a server; as the difference between k and the number of files actually stored at that server. Because it is assumed that file placement is random, the distribution of files is not even. The shortfall is a random variable for each server.

When shortfall at a server is positive, with the assumption of perfect load balance of map tasks, the shortfall is a lower bound on the number of tasks that must be accessed remotely by the map tasks that run on that server. When summed over servers with positive shortfall, the total shortfall is the number of map tasks that

must be served remotely. The total shortfall gives a lower bound on the remote service. Remote service will actually be more than this lower bound due to imperfect choices in assigning files for remote service, especially with a low value for k and without replication.

The closed form of this can be derived via the intuition in Appendix A. It models the shortfall of the expected value of blocks per node in the cluster. This analysis simplifies to the following equation:

$$1/\sqrt{2k\pi} \tag{4.1}$$

When one generalizes Equation 4.1 to include r, the model can be represented as:

$$1/\sqrt{2rk\pi} \tag{4.2}$$

The model represented by Equation 4.2 is validated using a 20 node cluster with a value of $k = 32$ and falls within 1% - 4%, and averaged over all runs falls within 1% of the model, dependent on the initial distribution of files on the cluster. A Monte Carlo simulations with randomized file (block) placement is also used to further validate the model. Map tasks were schedule with a priority for local execution and were assigned for remote tasks only if nodes were idle and tasks were available (similar to Hadoop's scheduling policy). These simulations help confirm that the remote-served fraction does not depend on the number of servers n, but only on the ratio k and the replication factor r as shown in the analysis above. To do this, the number of nodes are varied between 10 and 5000 while keeping k and r constant. Figure 4.5 compares the model-predicted RMTF to one obtained by Monte Carlo simulation of Hadoop task assignment. For varying values of k (curve groups along X-axis), Figure 4.5 plots the percentage of local tasks (Y-axis) for various replication factors (X-axis).

From this analysis come four key observations from Figure 4.5. First, in the region of interest ($r > 1$ and $k > 50$, which is typical for MapReduce), the analytical model is within 2% of simulations. Second, it tends to underestimate the local map task fraction. Third, even in the case of $r = 1$, the number of local map tasks is above

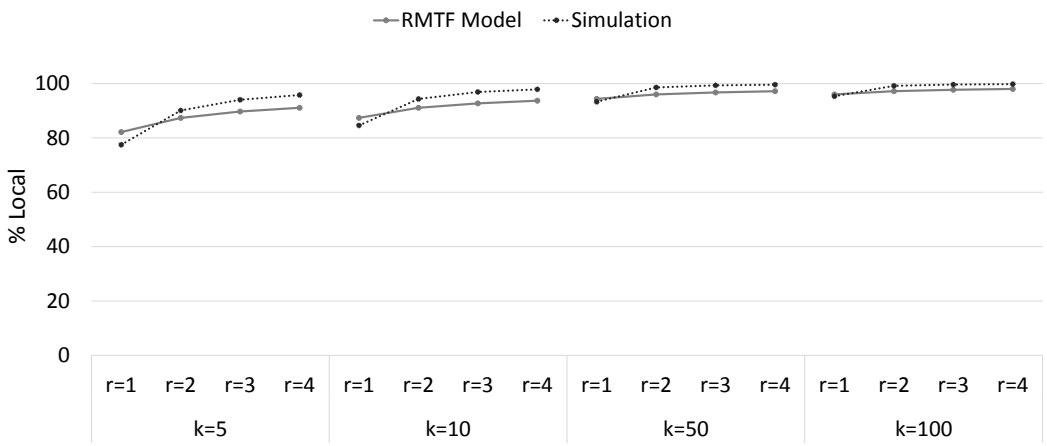


Figure 4.5.: Model Predictions vs. Simulation

90%. While $r = 1$ is not typical for Hadoop, this property will be exploited in the design of `BoostDFS`. Finally, note that the reduction in local map task fraction because of increasing replication is fairly modest, especially for the typical configuration of $k \geq 50$. This confirms the earlier claim of diminishing locality returns from Hadoop replication.

4.3.2 Modeling Map Task Performance by Leveraging the RMTF (ρ) Model

The RMTF model enables estimation of the number of remote map tasks. However, it offers no guidance as to what the slowdowns for remote tasks are. There is a wide range of slowdowns reported in the literature from scheduling [33–35] to slot contention [36] to spills [37,38] and shuffles [39]. Some have also reported asymmetric slowdowns wherein the source of the remote data incurs the slowdown than the map task that consumes the remote data [40]. In practice, the slowdown due to map task is hard to predict as it may depend on multiple factors such as disk utilization and network bisection. To avoid a dependence on map task slow downs, the slowdown ratio is varied as an independent parameter in this model. For a given replication

ratio, the map task slowdown is computed as a weighted average of local tasks with unit speedup and remote tasks with parametric slowdown, as shown below.

Assuming unit completion times for local map tasks and a slowdown factor of s_{remote} for remote tasks, the overall execution time (relative to an imaginary run in which all tasks are local tasks is a weighted $\rho s_{remote} + (1 - \rho)$. This formulation lends itself to Amdahl's law analysis given that ρ is typically under 3% for MapReduce's region of operation ($k > 50, r > 2$). This implies that attempting to improve performance by increasing inter-node replication is not an attractive tradeoff.

4.4 BoostDFS Design

`BoostDFS` proposes the use of intra-node replication as the basic mechanism to boost performance of MapReduce. Consider the advantages of intra-node replication as shown in Figure 4.6. File/Block A is present on two different disk spindles on Node 0. Such mirroring can be used to achieve higher disk bandwidth. Beyond the simple illustrative example, it is shown that with a single dataset replica; 80+% of map task locality is able to be captured. Consequently, one could speedup the common case of local tasks by making them faster than local — or "superlocal".

Effectively, among the r copies of r-way replication, where HDFS would spread the copies on r different nodes, `BoostDFS` spreads them on $r - 1$ servers resulting in at least one server enjoying an additional intra-node replica.

One may think that a baseline JBOD-based (or any other concatenation) system, which also uses multiple devices, can fully utilize the bandwidth of its individual disks. However, there remains some unexploited bandwidth because of JBOD's fundamental constraints. First, JBOD cannot guarantee parallel access across the two disks because the two files being accessed are on the same device. (In contrast, `BoostDFS`'s replication offers the guarantee of parallel disk access.) Second, because of seek minimization, files may be clustered (e.g., after disk defragmentation) which can limit disk parallelism when files from the same cluster are accessed. In contrast, `BoostDFS`

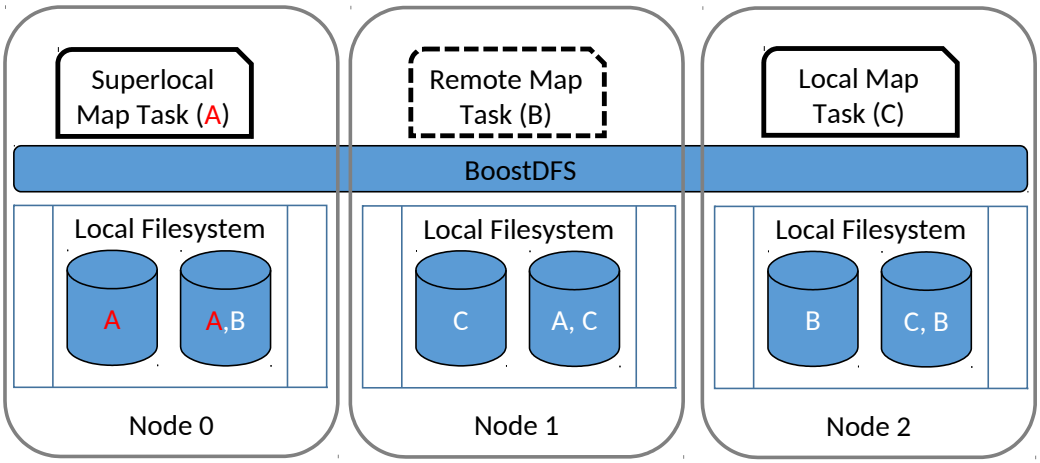


Figure 4.6.: `BoostDFS` with 3-way Replication (Compare to Figure 4.4)

enables us to simultaneously support clustered layout for seek minimization via the local file system *and* exploit all available disk parallelism via `BoostDFS`'s intra-node replication.

To ensure that `BoostDFS`'s bandwidth advantage is exploited by the vast majority of map tasks, it is necessary to alter the map task scheduling policy in `BoostDFS`. `BoostDFS` treats the map tasks assigned to files on the intra-node replicated file system as *superlocal*. With one replica in the replicated local file system, and with the Hadoop modification that prefers superlocal copies over local copies, the number of superlocal tasks is expected to be the same as indicated by the RMTF analysis for one replica.

For the above design, assuming a superlocal speedup of $s_{superlocal}$ and a remote slowdown of s_{remote}, the following three term expression can be derived for overall execution time corresponding to superlocal tasks, local tasks, and remote tasks respectively (ρ_r is used to mean the RMTF for $r = k$).

$$(1 - \rho_1)/s_{superlocal} + (\rho_l - \rho_r) + \rho_r s_{remote} \tag{4.3}$$

Figure 4.8 plots the speedup predicted by the model expressed by Equation 4.2 of `BoostDFS` over a regular HDFS-based Hadoop for various k and r values. This model proves the opportunity by showing 12% to 19% speedups from `BoostDFS` for the same capacity utilization. Figure 4.9 plots the speedup predicted by the model of

`BoostDFS` over HDFS-based Hadoop, similar to above, except at similar availability (i.e., incurs a capacity overhead).

4.4.1 Capacity Costs and Headroom

It would be unreasonable to assume that a cluster is provisioned with no spare capacity. This would lead to the inability to generate intermediate data, or output a result, or even grow the input data-set. Consider the total capacity as a multiple (say C) of baseline replicated input data size. We refer to this multiplicative factor (C) as *capacity headroom*. Capacity costs from intra-node replication will be paid from this capacity headroom.

The capacity overhead cost, under same availability, for a superlocal copy $r \rightarrow r+1$ is a fraction $1/r$ of the replicated input data size. Further, from our definition of capacity headroom, the overhead relative to total capacity can be modeled as follows:

$$1/(C_{headroom} r) \tag{4.4}$$

For example, assuming $10x$ capacity headroom over the baseline replicated input size, at $r = 3$ that would be about a 3% capacity overhead for an intra-node replica using Equation 4.4. Figure 4.7 illustrates the cost of this extra intra-node replica for r values of 2 and 3, by varying the capacity headroom between 5× and 50×. As one would expect, the more capacity headroom that exists in the system, the less overhead cost replicas incur. The overhead can range from just over 6% on a cluster with very minimal headroom, to under 1% on a cluster that is provisioned with extra headroom for an availability of $r = 2$.

BoostDFS creates the superlocal copies statically. Alternately, one may configure a system where a superlocal copy is dynamically created on a node when there exists sufficient disk bandwidth (i.e., the task has some computational overhead). In the disk-bound PUMA benchmarks, however, there was no bandwidth slack to facilitate such on-the-fly replica creation. However, if other applications/contexts have bandwidth slack, one could use this alternative model to dynamically create/destroy

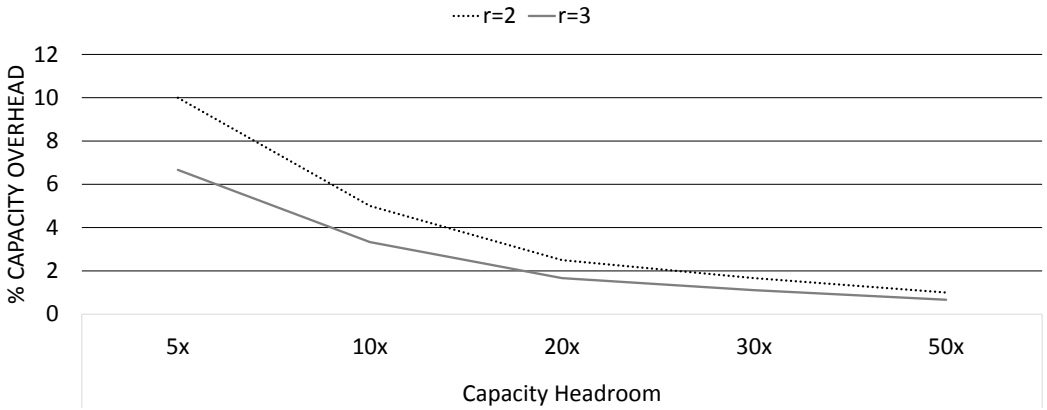


Figure 4.7.: Capacity Overhead Same Availability

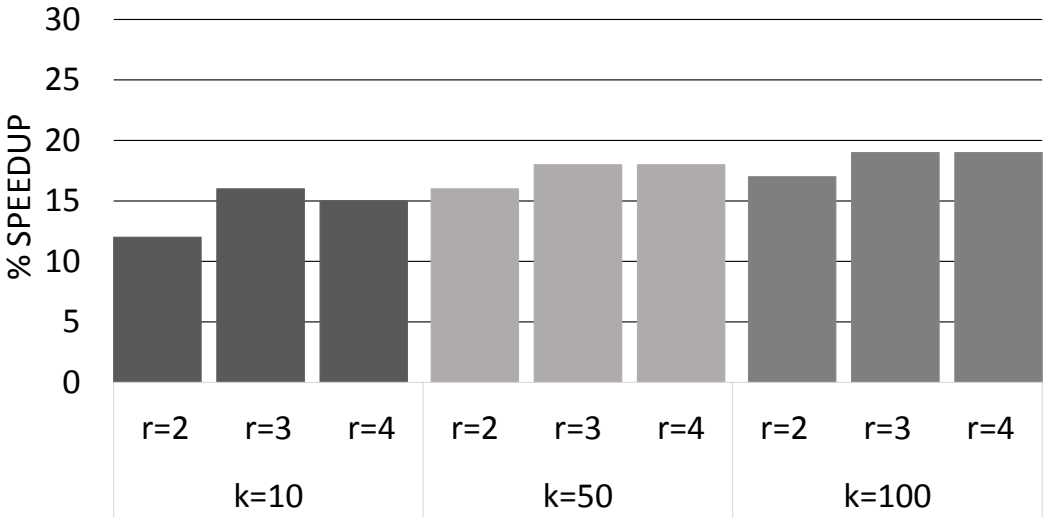


Figure 4.8.: Model-predicted `BoostDFS` Speedup Over HDFS Same Capacity

replicas. Note, such dynamic replicas may be deleted at will if space is needed. In the worst case, if a job arrives assuming superlocal performance, but the intra-node replica has been deleted, it will run as a regular local copy.

4.4.2 BoostDFS Implementation

`BoostDFS` is designed around the Hadoop [10] MapReduce framework. Care is taken to minimally modify the existing design, this is done to limit the impact of

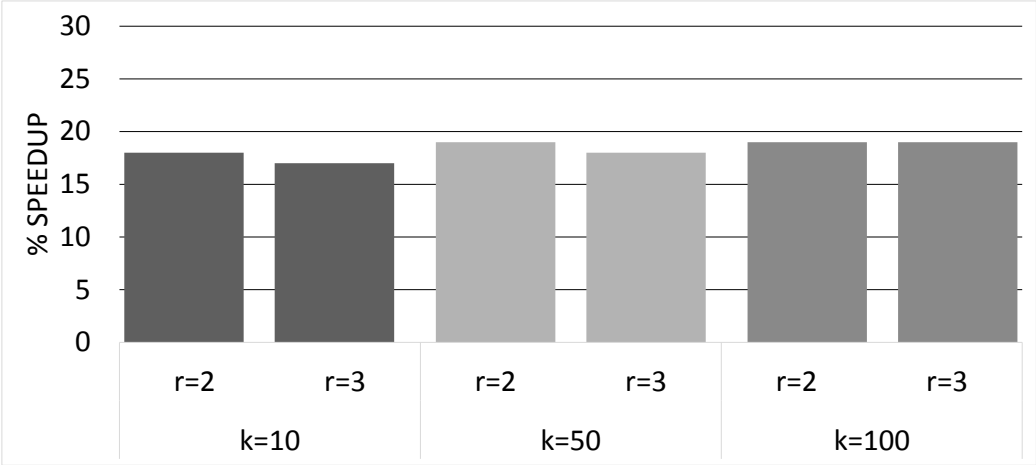


Figure 4.9.: Model-predicted `BoostDFS` Speedup Over HDFS Same Availability

the modifications. Primarily, I focus on the distributed file system and the locality scheduler. The two design goals of this implementation are (1) to provide transparent intra-node replication of storage to support high bandwidth access, and (2) to provide a mechanism to enable replication-aware scheduling of map tasks.

4.4.3 Replicated Storage Types

The Hadoop Distributed File System (HDFS) [3] supports custom-defined storage policies and storage types. While the main purpose is to facilitate hierarchical storage, I leverage these storage types and policies to augment HDFS with the ability to a support a new `REPLICATED` storage type. The `REPLICATED` storage type has the following behavior. The storage type is defined over two disk spindles with a Morphstore [41] file system that supports automatic replication on two disks. The original Morphstore uses a load-adaptive policy that dynamically chooses striped or mirrored accesses at low and high loads, respectively. In practice, because MapReduce runs see a heavy load on the disks, this implementation did not use dynamic load-adaptation.

To ensure that Hadoop preserves the total number of replicas in the system, I ensure that each block placed on a `REPLICATED` storage type counts as two replicas as far as Hadoop's replication is concerned. However, when it comes to scheduling, Hadoop treats the replicated storage type as a single high-bandwidth copy rather than as two independent replicas. I extend the capabilities of the existing map task scheduling to support such intra-node replicated blocks. The existing implementation tracks the presence of blocks on hosts. Specifically, when a block is accessed, the hostnames of nodes which contain local copies are communicated along with the block information. To support replication-aware scheduling, I further bifurcate the set of hosts containing local copies into two sets. Let us define a superlocal copy as a copy which resides on a `REPLICATED` disk. A subset of the hosts is created with copies on `REPLICATED` disks; these are called superlocal hosts. HDFS is augmented to communicate this list of superlocal hosts and the local hosts along with the block information. This allows other components of the MapReduce framework to interact with local or superlocal blocks, appropriately.

Superlocal Storage Policy

It is also necessary to create a storage policy called `SUPERLOCAL` which helps automate the placement based on storage types, and to accurately account for the replication factor. To that end, it must be ensured that `REPLICATED` storage is supported only when the replication factor exceeds 1. Moreover, the replication factor is reduced by 1 when a block is placed on a `REPLICATED` storage type to account for the transparent mirroring within the storage type. This study is limited to using two intra-node replicas at most. In general, one could communicate the maximum replication factor of the `REPLICATED` disk to the DFS so that it may modify the distributed replication factor accordingly.

Map Task Locality Scheduling

In the baseline Hadoop implementation, when a map task is scheduled, the resource allocation checks if the map task can be scheduled in a slot on any node which contains a local copy of that block. Failing to find an available node with a local copy, the same rack is checked for an available slot, and failing this the task is scheduled in a any available free slot. The resource scheduler for the MapReduce framework gathers this information based on a list of hostnames which contain a local copy of the map block (obtained from the DFS).

For `BoostDFS`, as mentioned in section Section 4.4.3 I augment this information to include the superlocal hosts as well. Because superlocal data hosts are a subset of local data hosts, no extra work is required to include super local hosts in the scheduling flow. The assignment of a map task to a node is modified to ensure that the assignment to a super local node is attempted before the block is assigned to a local node. The new scheduling proceeds as follows: First, the super local nodes are checked for available slots, failing to find one the local nodes are checked. Should both of those fail, rack local nodes are checked for a free slot, and finally the map task is placed in any free slot.

4.5 Evaluation Methodology

I use the open source MapReduce framework, `Hadoop` [10], as the base platform for evaluating `BoostDFS`. The 2.6.0 version of `Hadoop` is used as both the basis for this evaluation and comparison. Because this evaluation requires intrusive changes to the servers (custom file system, custom disk drive organization with three spindles – two for data and one for the OS partition), the evaluation platform is limited in scale. The testbed MapReduce cluster consists of four datanodes and a namenode server. The network is gigabit ethernet, with all nodes connected to a single switch.

Table 4.2.: Data Node Resources

Memory	12GB DDR2 1639MHz (6GB swap)
Processor	2x Dual-Core AMD Opteron(tm) Processor 2222
Disk	3 7200RPM SATA HDDs (457GB total capacity)
Network	Intel 82571EB Gigabit Ethernet Controller

4.5.1 Machine Configuration

The machines in the cluster share identical hardware configurations, as shown in in Table 4.2. All the machines run Linux with a 3.3 kernel with `iostat` version 10.2.1 [42] installed. The `iostat` tool is used to gather disk bandwidth statistics for the results. The disks are configured in a 'Just a Bunch of Disks' (JBOD) configuration with the kernel software raid. The baseline configuration uses an `ext2` file system on the JBOD drive, while the experimental configuration uses an approach similar to the `MorphStore` [41] local file system with two way replication tuned for heavy load (additional details of MorphStore are discussed in Chapter 3). Note, I omit RAID-0 because of its known performance limitations at high loads (see Section 3.2) – MapReduce computations typically operate in the high-load region.

4.5.2 Hadoop Configuration

The cluster is configured with `Hadoop` version 2.6.0 with 'high availability'[1] disabled. I configured HDFS, Yarn, and Map/Reduce tasks for the cluster. First, the distributed file system configuration,Table 4.5, has two configurations: (1) the base

[1]High availability in Hadoop uses a redundant namenode to prevent the single namenode from becoming a single point of failure. Because it has nothing to do with data replication – the focus of this work, this feature is disabled.

Table 4.3.: MapReduce Configuration

Framework	yarn
Slowstart Completed Maps	0.90
Shuffle Parallel Copies	16
Reduce Input Buffer	0.70
Merge Threshold	0
Sort Memory(MB)	400
Map Memory(MB)	1024
Map Java-opts	-Xmx820m
Reduce Memory(MB)	3072
Reduce Java-opts	-Xmx2457m
AppMaster Memory(MB)	2048
AppMaster Command-opts	-Xmx1638m
AppMaster CPU-vcores	2

Table 4.4.: Yarn Configuration

CPU-vcores	8
Maximum Allocation-vcores	8
Minimum Allocation-vcores	1
Memory(MB)	8192
Maximum Allocation(MB)	8192
Minimum Allocation(MB)	1024

HDFS configuration, and (2) the `BoostDFS` configuration. I define two storage types and policies to ensure comparable behavior. The first storage type – `REPLICATED`– is used for the `BoostDFS` configurations and it ensures that blocks placed on this storage type are replicated within the same node. The base configuration uses the second –

Table 4.5.: HDFS Configuration

Replication Factor	1
Block Size(MB)	128
Base Config	NONREPLICATED, HDD
Boost Config	REPLICATED, HDD

`NONREPLICATED` – storage type to ensure there is at most one copy per node. It also asserts that the JBOD drive will not carry a local replica of the blocks placed on it and that the replication factor will be used normally. `BoostDFS`, however will have a replica of blocks locally on the MorphStore drive.

Yarn [43] is the resource management framework used to coordinate hardware resources among the different jobs and tasks. I configured Yarn,Table 4.4, based on the hardware resources available to the cluster, Table 4.2. The number of virtual cores is set to $2 * physicalCores$ and 4GB of memory is reserved for OS/system use. The minimum allocation size is set to 1GB, as this is the smallest amount of memory a task will request. The maximum is set at the total resources available for the framework, 8GB, this allows a container to be allocated anywhere from 1GB to 8GB of memory [44, 45].

The MapReduce parameters are shown in Table 4.3, these apply to the Map, Shuffle, and Reduce phases of the MapReduce framework. The framework is set to Yarn, this specifies that the new MapReduce 2 framework (this primarily deals with resource management) is what will be used to for these tests. Based on the hardware resources, Table 4.2, I tune the MapReduce parameters for a reasonable configuration. The intermediate data which each map outputs needs to be aggregated on a host; the `Slowstart Completed Maps` parameter will allow this data to begin copying after 90% of the map tasks have completed, this prevents copy from interfering with map data reads on local and remote hosts. Similarly the `Shuffle Parallel Copies` parameter limits the threads for aggregating intermediate data, as to not

Table 4.6.: Shuffle Light Benchmarks

Benchmark	Input Size	Dataset
classification	8GB	Movie Ratings
grep	8GB	Wikipedia
histogrammovies	8GB	Movie Ratings
histogramratings	8GB	Movie Ratings
wordcount	8GB	Wikipedia

overburden the system. I make an effort to utilize as much of memory, as would a production system, to limit spills to disk. Memory pressure is allowed to dictate when map outputs are merged, and the amount of memory for sorting is increased, as this reduces spills. I also allow reduce tasks to keep a percentage of the map outputs in memory, otherwise all data would be spilled to disk and need to be read again for the reduce task. Map and reduce tasks are specifically sized for the cluster. I set the memory size of map tasks to 1GB, reduce tasks to 3GB and the application master to 2GB. The size of the heap is increased to 80% of the allocated memory size for each task. This is done to increase the use of memory and again, avoid spills as much as possible. Finally, a separate server is used as the namenode, this is to allow the datanodes to focus entirely on compute, and not necessitate the memory pressure the namenode services require.

The default resource calculator used when allocating node resources does not take into consideration the number of vcores. It solely provisions based on the memory constraints. This is switched to the dominant resource calculator to better manage resources [35].

Table 4.7.: Shuffle Heavy Benchmarks

Benchmark	Input Size	Dataset
adjacencylist (adjlist)	8GB	Synthetic
invertedindex	8GB	Wikipedia
kmeans	8GB	Movie Ratings
selfjoin	8GB	Synthetic
terasort	8GB	Random
termvector	8GB	Wikipedia

4.5.3 Benchmarks and Data Sets

The workloads used to evaluate `BoostDFS` are from the PUMA Benchmark Suite [32, 39]. These are benchmarks which represent realistic classes of MapReduce workloads. These are listed in Table 4.6 and Table 4.7, as well as the dataset and input size for each benchmark. The shuffle heavy workloads have a reduce time that is significantly larger than the map time.

Input sizes of 8GB and 50GB (selectively) are used, as this will maintain the high block to node locality ratio of as discussed in Section 4.3. the larger input is used to verify the benchmarks, and show the sensitivity of input sizes, as these two dataset points encompass the bounds of our model.

4.5.4 Evaluation

`BoostDFS` is evaluated with the benchmarks listed in Table 4.6 and Table 4.7. Input sizes of 16x and 100x (as input sensitivity) are select, with the smaller of the two used as the primary input size, this allows the evaluation of the model with a modest cluster configuration.

This evaluation occurs with a real system and is thus subject to artifacts such as OS, page cache, network stack, and run time fluctuations. To mimic the effects of

large data sets that do not fit in caches, the caches are dropped between executions of a benchmark, as well as after moving data to the cluster. To alleviate the effects of other artifacts, benchmarks are run for 3 iterations and averaged. To keep execution times manageable on this modestly sized cluster, benchmarks are only simulated at the 16x input size. Replication is determined by setting the MorphStore drive to default to a replication factor of 2 and setting the default Hadoop replication factor to 2 and increasing that to a total replication factor of 4. The base case is run from a replication factor of 2 to 4 as well. This is done to compare MapReduce with MorphStore. The base case replication factor of 1 is used to normalize speedups and verify the model in Section 4.3.1. Before each workload is run, it is ensured that the storage policy of the input directory is set to `SUPERLOCAL` so that only one copy of the block will get placed on a device with storage type `REPLICATED`. The `BoostDFS` configuration provides each datanode with one `REPLICATED` disk.

4.6 PUMA Benchmarks

The PUMA benchmark suite is available for an older version of Hadoop. This suite has been updated to the new MapReduce v2 API. This allows for current work to continue using the latest version of the Hadoop MapReduce framework. The benchmarks were extensively evaluated to ensure that the update modifications did not alter the benchmark algorithm. This new version will be available online, and possibly merged into the Hadoop project as part of the base example suite. At the very least, a downloadable package will be distributed online so that others may use this benchmark to evaluate their work.

4.7 Results

The key findings of this paper are as follows:

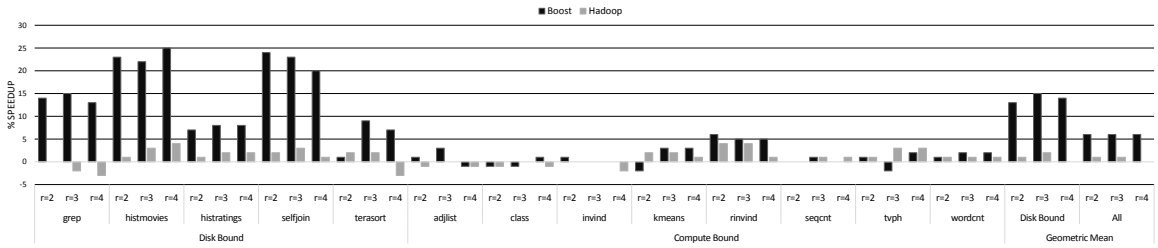


Figure 4.10.: Overall Performance Same Capacity

- It is shown that under identical total replication, `BoostDFS` outperforms baseline Hadoop by 14%, on average (geometric mean), for disk-bound map reductions. The geometric mean across all benchmarks is 6%. Conversely, under identical availability, `BoostDFS` achieves 14% higher performance for disk-bound-applications at the cost of an additional replica. This corresponds to about a 3% capacity overhead at an availability of 2 inter-node replicas and one super-local replica.

- `BoostDFS` does not hurt the performance of compute-bound benchmarks.

- It is confirmed through direct file IO measurement that `BoostDFS` significantly increases available bandwidth, which results in superior map task performance.

The above results are discussed at detail in the remainder of this section.

4.7.1 Overall Performance

Figure 4.10 illustrates the speedup (relative to HDFS with a single copy of the dataset, Y-axis) of various benchmarks (groups of bars on the X-axis) using the same Capacity. The benchmarks are sorted with disk-bound benchmarks on the left and compute-bound benchmarks on the right. For each benchmark, Figure 4.10 shows the performance of six configurations. These are the `BoostDFS` and HDFS configurations with 2, 3, and 4 replicas. In addition to the bars for individual benchmarks, two

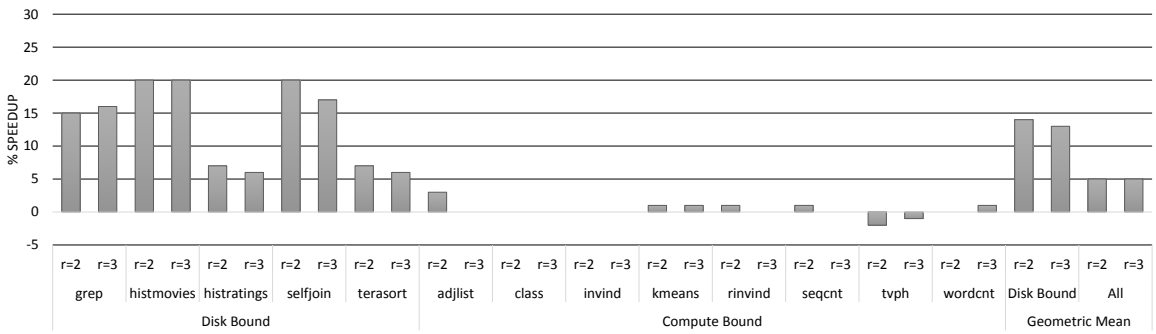


Figure 4.11.: Overall Performance Same Availability

set of bars are included that show the geometric mean (GM) of speedup across (1) the disk-bound benchmarks, and (2) all benchmarks. For some benchmarks one will notice that increasing replication factors are not necessarily monotonically increasing. These perturbations are run-to-run variations which are caused by changes in the numbers and placement of remote map tasks, and other system jitter. Such run-to-run variations are within 3% of total execution time, and do not reflect any systemic performance issues.

First, consider the disk-bound benchmarks. Comparing pairs of `BoostDFS` and HDFS bars with the same replication factor, it is observed that `BoostDFS` achieves higher performance than HDFS for disk-bound benchmarks. This comparison isolates the availability-performance tradeoff under the same replication factor (Capacity) because `BoostDFS` achieves higher performance by concentrating the r replicas on $r-1$ nodes. The speedups vary between 5% and 23%, on average, for the disk-bound applications. On the other hand by comparing `BoostDFS` with r replication against HDFS with $r-1$ replication, it can also be observed that `BoostDFS` trades off the capacity cost of the additional replica for better performance under the same availability. To facilitate an understanding of this comparison, Figure 4.11, illustrates the speedup (relative to HDFS with availability r, Y-axis) of the same puma benchmarks, this time with availability r. This means Hadoop has r inter-node replicas, and `BoostDFS` has r inter-node and 1 intra-node replica (i.e., incurs a capacity overhead). The

speedup for disk-bound applications averages around 14%, and 5% on average for all applications.

Effectively, the recommended use case for `BoostDFS` is when system designers know that the availability requirements are satisfied with a minimum replication factor of (say) r. If the system has spare additional capacity beyond the r copies, one may use it to add a `BoostDFS` replica which provides a better bargain in the capacity/performance tradeoff than the default tradeoff of increasing the number of HDFS replicas.

Finally, `BoostDFS` is within 2% of the performance of compute-bound benchmarks. This is not surprising because compute-bound applications are insensitive to the bandwidth penalty for remote map tasks. The combined geometric mean speedup is 6% even after averaging in the compute-bound benchmarks.

These experiments also revealed that the fraction of superlocal tasks for all but one benchmark was higher than 90%. The remote map task fraction is 1% at $r > 2$. The sole exception was terasort for which the number of superlocal tasks were 80%. Note, the number of superlocal tasks is higher than the model predictions because each node is capable of executing multiple map tasks in these runs, this allows scheduling flexibility for each node.

4.7.2 Disk Bandwidth Utilization

Figure 4.12 illustrates a single server's time-varying disk bandwidth utilization (Y-axis, in MB/s as measured by `iostat`) for a node running a map-only Hadoop microbenchmark for two different configurations (two curves). It is ensured that all map tasks run in superlocal mode for one of the configurations and in local mode for the other. Each configuration uses 16 map tasks per node.

The first 60 seconds reveal little activity as the Hadoop runtime attends to one-time application startup issues. Disk activity ramps up beginning at approximately the 60 second mark. The superlocal configuration is able to achieve a significantly

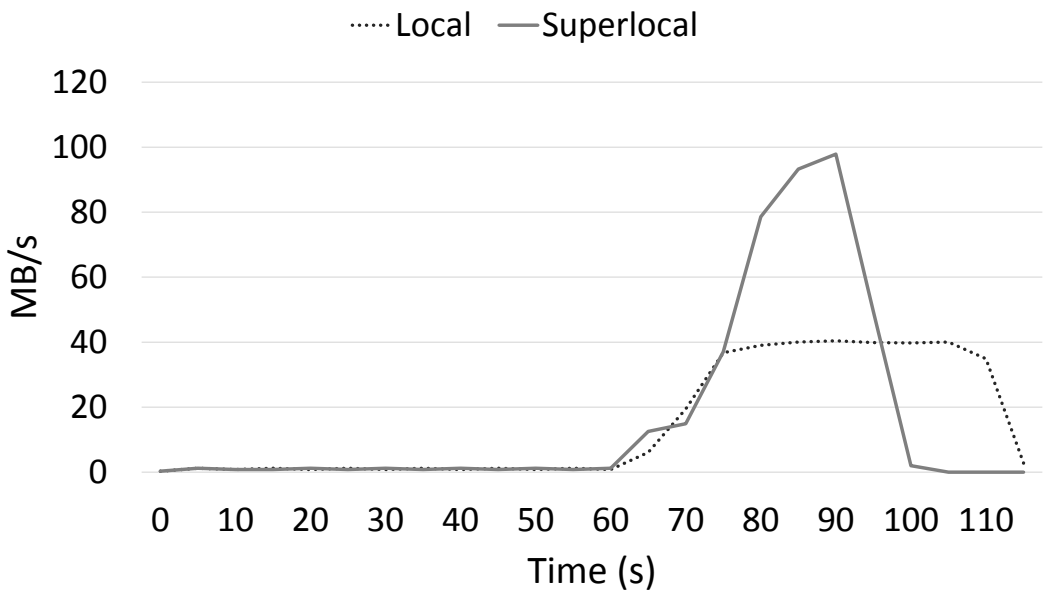


Figure 4.12.: Disk Bandwidth Utilization for Local and Superlocal Map Tasks

higher bandwidth which results in quicker completion. In contrast, the local-only configuration saturates the achieved bandwidth which causes the map task data reading to continue over a longer stretch of time. This is precisely `BoostDFS`'s main advantage; `BoostDFS` offers such high disk bandwidth to a large fraction of map tasks.

4.7.3 Input Size Sensitivity

The main results were measured with an 8GB dataset. To confirm that `BoostDFS`'s performance advantage is more broadly applicable, the performance is verified using two benchmarks – grep and word-count – with a larger (50GB) dataset. The two benchmarks were chosen as representatives of the disk-bound and compute-bound class of benchmarks. Figure 4.13 plots the performance of `BoostDFS` and HDFS for the seven configurations for 8GB and 50GB dataset sizes. The performance of each benchmark at each dataset size is normalized to that of the same benchmark running on HDFS with 1-way replication (i.e., single copy). The two key conclusions continue to hold: `BoostDFS` improves the performance of disk-bound map-reductions

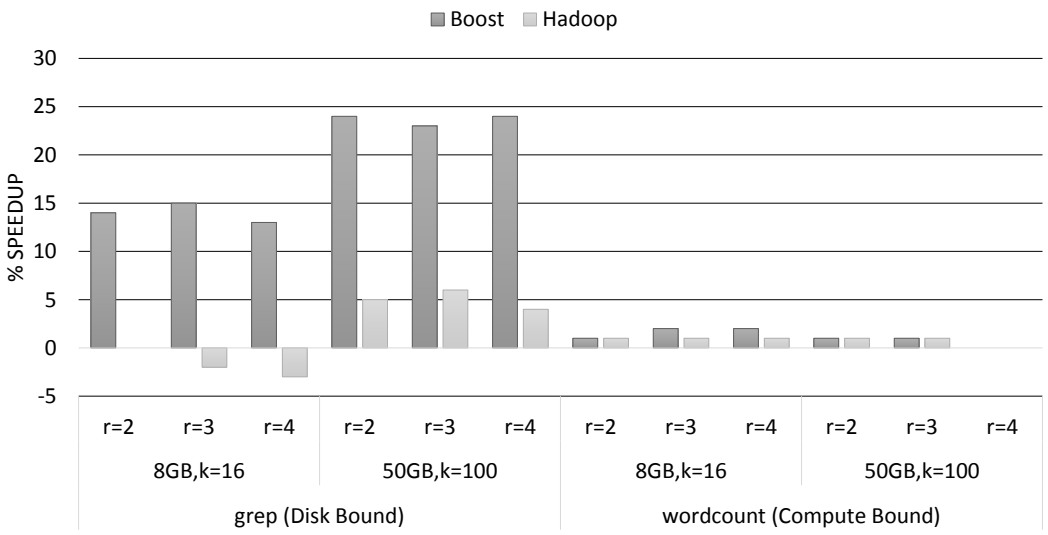


Figure 4.13.: Input Size Sensitivity

and `BoostDFS` does not hurt the performance of compute-bound map-reductions. Further, from this limited sensitivity study, one can see that the benefits are further magnified at larger dataset sizes (the speedup of the disk bound benchmark increases from 14%, 15%, and 13% to 24%, 23% and 24%, respectively. I conjecture that the growth in speedup is because of some remaining bandwidth headroom in `BoostDFS` that the 8GB datasets did not fully utilize. However, because this is a single datapoint, it may require broader validation with the full suite of benchmarks.

4.8 Related Work

To the best of my knowledge, this work is the first to propose replacing some inter-node replication with intra-node replication to achieve higher bandwidth disk access. There is a large body of MapReduce performance optimizations that target various opportunities other than storage (e.g., heterogeneous clusters [46–49], overlapping shuffle with computation [39], better handling of stragglers [36, 50, 51], multi-tenancy [52], cache-aware scheduling for small map reductions [53]). Such schemes that are unrelated to storage performance are orthogonal to `BoostDFS`. There is additional work

that has looked at using heterogeneous storage architectures with solid-state drives for MapReduce [19]. While one could use SSDs as a form of "superlocal" storage, that tradeoff is more complicated as the hardware costs are different (because of SSDs) from that of the baseline Hadoop. In contrast, `BoostDFS` uses the same hardware as baseline Hadoop with the only assumption being that there are multiple disk spindles per node and that there is spare capacity. Others [20, 54] propose using DRAM as the storage medium for data copy placement and storage. These techniques offer improvements to performance, but workloads are limited by the amount of RAM, as well as system expense.

4.9 Conclusion

The traditional use of replication, in the distributed file systems used in MapReduce (e.g., GFS [25], HDFS [3]), offers diminishing returns in both performance and reliability with increased inter-node replication. The key reason for the diminishing returns on performance is that added inter-node replication helps reduce the number of remote map tasks – which is an uncommon event. In contrast, this design – BoostDFS – targets the common case and further boosts the disk bandwidth available to a large fraction of local map tasks via intra-node replication.

BoostDFS uses well-known mirroring techniques to boost disk bandwidth [41, 55]. But the innovation is to make this visible to Hadoop through our DFS such that Hadoop's job scheduler can aggressively seek to maximize *super-local* map tasks – tasks that read their inputs from the replicated file system.

BoostDFS improves the replication/performance/availability tradeoffs in the following ways. At the same replication factor, BoostDFS can boost the performance of Hadoop as it gains more performance by exploiting higher disk bandwidth on a large fraction map tasks than the traditional approach, which achieves marginal gains. However, to achieve such a performance gain, BoostDFS trades off availability under server failure as BoostDFS concentrates the same number of replicas as Hadoop on

fewer servers. An alternate view of BoostDFS is that it outperforms baseline Hadoop under the same availability by using more replication. In this view, the number of servers with replicas is the same in both BoostDFS and traditional Hadoop (which leads to identical availability under server failure); however BoostDFS uses additional replicas on the same servers to boost performance. In this view, BoostDFS trades off spare disk capacity to improve performance. Note all the above tradeoffs are with identical hardware; i.e., the comparisons are iso-cost.

5. SUMMARY

I present two techniques that improve performance of file systems, both local and distributed, by using replication as a mechanism for an improved capacity-performance tradeoff. At the local file system, throughput of the underlying storage devices is improved by adapting to the load level of the system, and identifying files that provide the most utility. An improvement in the scheduler is also proposed that provides a mix of striping and steering based on the accesses to replicated files. The distributed file system benefits from using changing an inter-node replica to an intra-node replica and improving performance of the common case.

All of these techniques culminate in the improvement of large file accesses for systems that work over large data-sets. The impact is that while the storage layer may be the lowest and slowest in the hierarchy, providing performance improvements aids the other layers and the overall system performance. These techniques use replication to provide better aggregation of throughput for the storage devices, thus providing administrators a capacity tradeoff that can be used for availability and performance alike, with minimal overhead costs. All of the techniques discussed in this work are agnostic to any specific file system, local or distributed, and can be incorporated into any variant of the a framework that is targeted for large file accesses.

REFERENCES

REFERENCES

[1] 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

[2] 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.

[3] 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

[4] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922

[5] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *In Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006, pp. 307–320.

[6] F. Schmuck and R. Haskin, "Gpfs: A shared-disk file system for large computing clusters," in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, ser. FAST '02. Berkeley, CA, USA: USENIX Association, 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=1083323.1083349

[7] P. Schwan, "Lustre: Building a file system for 1,000-node clusters," in *PROCEEDINGS OF THE LINUX SYMPOSIUM*, 2003, p. 9.

[8] *MapR*. [Online]. Available: http://www.mapr.com

[9] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327492

[10] A. S. Foundation, *The Apache Hadoop project*, 2011 (accessed April 13, 2015). [Online]. Available: http://hadoop.apache.org

[11] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 59–72, Mar. 2007. [Online]. Available: http://doi.acm.org/10.1145/1272998.1273005

[12] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[13] Facebook, *Presto: Distributed SQL Query Engine for Big Data*, 2013. [Online]. Available: http://prestodb.io

[14] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive- a warehousing solution over a map-reduce framework," in *IN VLDB '09: PROCEEDINGS OF THE VLDB ENDOWMENT*, 2009, pp. 1626–1629.

[15] Facebook, *Presto: Interacting with petabytes of data at Facebook*, 2013. [Online]. Available: https://www.facebook.com/notes/facebook-engineering/presto-interacting-with-petabytes-of-data-at-facebook/10151786197628920

[16] Apache, "The hadoop distributed file system: Architecture and design," Apache Software Foundation, Tech. Rep., 2007.

[17] J. M. Dyaberi, "Networking and storage support for video-on-demand data delivery," p. 113, 2011, copyright - Copyright ProQuest, UMI Dissertations Publishing 2011; Last updated - 2014-01-22; First page - n/a. [Online]. Available: http://search.proquest.com/docview/1015155426?accountid=13360

[18] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, "The hp autoraid hierarchical storage system," *ACM Trans. Comput. Syst.*, vol. 14, no. 1, pp. 108–136, Feb. 1996. [Online]. Available: http://doi.acm.org/10.1145/225535.225539

[19] D. Zhao and I. Raicu, "Hycache: A user-level caching middleware for distributed file systems," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, ser. IPDPSW '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1997–2006. [Online]. Available: http://dx.doi.org/10.1109/IPDPSW.2013.83

[20] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. New York, NY, USA: ACM, 2014, pp. 6:1–6:15. [Online]. Available: http://doi.acm.org/10.1145/2670979.2670985

[21] 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.

[22] 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

[23] 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.

[24] 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.

[25] 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

[26] T. Ts'o, *E2fsprogs: Ext2/3/4 Filesystem Utilities*, e2fsprogs.sourceforge.net, December 2010.

[27] O. S. Community, *Filebench*, December 2010. [Online]. Available: http://www.solarisinternals.com/wiki/index.php/FileBench

[28] N. Garfield, *Iostat*, 2012. [Online]. Available: http://books.google.com/books?id=qAusMQEACAAJ

[29] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, pp. 385–398. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala

[30] 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.

[31] 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

[32] F. Ahmad, S. Lee, M. Thottethodi, and T. N. Vijaykumar, "Puma: Purdue mapreduce benchmarks suite," 2012. [Online]. Available: http://docs.lib.purdue.edu/ecetr/437

[33] Q. Chen, D. Zhang, M. Guo, Q. Deng, and S. Guo, "Samr: A self-adaptive mapreduce scheduling algorithm in heterogeneous environment," in *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, June 2010, pp. 2736–2743.

[34] X. Sun, C. He, and Y. Lu, "Esamr: An enhanced self-adaptive mapreduce scheduling algorithm," in *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*, Dec 2012, pp. 148–155.

[35] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 323–336. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972490

[36] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: Coping with skewed content popularity in mapreduce clusters," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11. New York, NY, USA: ACM, 2011, pp. 287–300. [Online]. Available: http://doi.acm.org/10.1145/1966445.1966472

[37] S. Ahmed and D. Loguinov, "On the performance of mapreduce: A stochastic approach," in *Big Data (Big Data), 2014 IEEE International Conference on*, Oct 2014, pp. 49–54.

[38] K. Elmeleegy, C. Olston, and B. Reed, "Spongefiles: Mitigating data skew in mapreduce using distributed memory," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '14. New York, NY, USA: ACM, 2014, pp. 551–562. [Online]. Available: http://doi.acm.org/10.1145/2588555.2595634

[39] F. Ahmad, S. Lee, M. Thottethodi, and T. N. Vijaykumar, "Mapreduce with communication overlap (marco)," *J. Parallel Distrib. Comput.*, vol. 73, no. 5, pp. 608–620, 2013. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2012.12.012

[40] Z. Guo, G. Fox, and M. Zhou, "Investigation of data locality in mapreduce," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, ser. CCGRID '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 419–426. [Online]. Available: http://dx.doi.org/10.1109/CCGrid.2012.42

[41] E. P. Villasenor, T. Pritchett, J. M. Dyaberi, V. S. Pai, and M. Thottethodi, "Morphstore: A local file system for big data with utility-driven replication and load-adaptive access scheduling," in *IEEE 30th Symposium on Mass Storage Systems and Technologies, MSST 2014, Santa Clara, CA, USA, June 2-6, 2014*, 2014, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/MSST.2014.6855537

[42] S. Godard, *Linux SYSSTAT Utilities*. [Online]. Available: http://sebastien.godard.pagesperso-orange.fr

[43] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 5:1–5:16. [Online]. Available: http://doi.acm.org/10.1145/2523616.2523633

[44] Hortonworks, *Hortonworks Data Platform 2.2.4 Documentation*. [Online]. Available: http://docs.hortonworks.com/HDPDocuments/HDP2/HDP-2.2.4/index.html

[45] M. Sousa, *Tuning the heap size of map and reduce tasks*, 2014. [Online]. Available: https://www.ibm.com/developerworks/community/wikis

[46] B. Palanisamy, A. Singh, L. Liu, and B. Jain, "Purlieus: Locality-aware resource allocation for mapreduce in a cloud," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 58:1–58:11. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063462

[47] X. Bu, J. Rao, and C.-z. Xu, "Interference and locality-aware task scheduling for mapreduce applications in virtual clusters," in *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '13. New York, NY, USA: ACM, 2013, pp. 227–238. [Online]. Available: http://doi.acm.org/10.1145/2462902.2462904

[48] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar, "Tarazu: Optimizing mapreduce on heterogeneous clusters," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 61–74. [Online]. Available: http://doi.acm.org/10.1145/2150976.2150984

[49] S. Khalil, S. A. Salem, S. Nassar, and E. M. Saad, "Mapreduce performance in heterogeneous environments: a review," *International Journal of Scientific & Engineering Research*, vol. 4, no. 4, pp. 410–416, 2013.

[50] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu, "Grass: Trimming stragglers in approximation analytics," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 289–302. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616448.2616475

[51] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 29–42. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855744

[52] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar, "Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 1–13. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ahmad

[53] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely, Jr., and J. Emer, "Pacman: Prefetch-aware cache management for high performance caching," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 442–453. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155672

[54] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The case for ramclouds: Scalable high-performance storage entirely in dram," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 4, pp. 92–105, Jan. 2010. [Online]. Available: http://doi.acm.org/10.1145/1713254.1713276

[55] 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

[56] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein *et al.*, *Introduction to algorithms*. MIT press Cambridge, 2001, vol. 2.

APPENDICES

A. REMOTE MAP TASK FRACTION CLOSED FORM

The closed form of the remote map task fraction model is provided by Prof. Robert Givan.

A.1 Closed Form Intuition

The expected total shortfall can be computed theoretically. Shortfall at a particular server is a zero-mean random variable that is well approximated by a normal distribution for reasonable numbers of files and servers, by the central limit theorem. Given the linearity of expectation, summing the positive shortfall values must give exactly the same magnitude as summing the negative shortfall values. Thus, summing the absolute-value of the shortfall variables gives exactly twice the desired total shortfall. Consequently, the expected total shortfall is half the expected absolute-value of the sum of the shortfall variables. The absolute-value of a zero-mean normal random variable is given by $\sqrt{2\sigma^2/\pi}$. Here, the variance σ^2 can be computed to be approximately k (exactly $k(1-1/n)$ for n servers). A detailed explanation is omitted as the result is a well-known identity that is used to analyze the complexity of bucket sort [56]. The expected remote service fraction is thus lower-bounded by approximately $\sqrt{2k/\pi}/(2k)$, dividing by 2 because it is expected that at least half the absolute value, as stated above, is remote-served, and by k as the target number of files to serve.

Equation 4.1 can be generalized in this analysis to replication values of r greater than 1 by observing that by symmetry, a given replica of a file has a probability $1/r$ of being the one served to the consuming map task. Thus, it is necessary to compute the expected shortfall at each server after each file located there is removed with probability $(r-1)/r$. This expectation can be computed with a very similar

approach to the $r = 1$ analysis. The total shortfall of placing r replicas of kn files on n servers is computed by :

$$\sum_{i=1}^{n}(|rk - files_i|/r) \tag{A.1}$$

Equation A.1 is a summation of half of the absolute value of the difference between rk and the number of files stored locally for each server, and then dividing by r to reflect the probability that each stored file is served elsewhere. The total shortfall again depends on the variance, which when placing rkn files on n servers is $\sqrt{2rk/\pi}$, which is derived again by an analysis like that of bucket sort. The expected remote fraction is thus lower-bounded by approximately $\sqrt{2rk/\pi}/(2rk)$, dividing by 2 and k as for $r = 1$ above, and by r to account for replicas served elsewhere as just discussed.

VITA

VITA

Eric Villasenor is a Ph.D. student in Electrical and Computer Engineering at Purdue University. His research interests include Heterogeneous Systems, Storage Systems, and Programming Models. He received his M.S from Purdue University in 2007 and B.S. from Purdue University in 2005.