An Active and Hybrid Storage System for Data-intensive Applications

by

Zhiyang Ding

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
Aug 6th, 2011

Keywords: Active Storage, Parallel and Distributed Computing, Storage Systems

Approved by

Xiao Qin, Chair, Associate Professor of Computer Science and Software Engineering
Kai-Hsiung Chang, Chair and Professor of Computer Science and Software Engineering
David A. Umphress, Associate Professor of Computer Science and Software Engineering

Abstract

Since large-scale and data-intensive applications have been widely deployed, there is a growing demand for high-performance storage systems to support data-intensive applications. Compared with traditional storage systems, next-generation systems will embrace dedicated processor to reduce computational load of host machines and will have hybrid combinations of different storage devices. We present a new architecture of active storage system, which leverage the computational power of the dedicated processor, and show how it utilizes the multi-core processor and offloads the computation from the host machine. We then solve the challenge of applying the active storage node to cooperate with the other nodes in the cluster environment by design a pipeline-parallel processing pattern and report the effectiveness of the mechanism. In order to evaluate the design, an open-source bioinformatics application is extended based on the pipeline-parallel mechanism. We also explore the hybrid configuration of storage devices within the active storage. The advent of flash-memory-based solid state disk has become a critical role in revolutionizing the storage world. However, instead of simply replacing the traditional magnetic hard disk with the solid state disk, researchers believe that finding a complementary approach to corporate both of them is more challenging and attractive. Thus, we propose a hybrid combination of different types of disk drives for our active storage system. An simulator is designed and implemented to verify the new configuration. In summary, this dissertation explores the idea of active storage, an emerging new storage configuration, in terms of the architecture and design, the parallel processing capability, the cooperation of other machines in cluster computing environment, and the new disk configuration, the hybrid combination of different types of disk drives.

Acknowledgments

I definitely could not have survived the Ph.D process without the guidance, friendship and support of many people.

First and foremost, I will always be grateful to my advisers, Dr. Xiao Qin and Dr. Kai-Hsiung Chang, for all invaluable guidance that they have given for me during my years in Auburn University. Dr. Qin, in particular, poured uncountable hours into my development ever since I started pursing the Ph.D degree. Under his supervision, I learned how to do research in storage system arena from scratch. Without them, it would be impossible for me to finish this dissertation.

I would gratefully thank my dissertation committee member, Dr. David A. Umphress. Dr. Umphress not only provides me valuable academic advices for this dissertation, but he also help and guide me in my entire Auburn life. I can still recall all that he have done for me when I first came to US. I cannot appreciate it more. I also gratefully thank Professor Wei Wang who is the Chair and Alumni Professor of Graphic Design program for serving as the university reader.

The many hours at the office were made both bearable and enjoyable by all the members of the research group. I would like to thank all my colleagues: Xiaojun Ruan, Shu Yin, James Majors, Jiong Xie, Yun Tian, Yixian Yang, Jianguo Lu and my former colleague, Adam Manzanares. As the neighbor in the office and research partner to me, Xiaojun, in particular, help me to solve a lot of research problems. Do appreciate for all his help.

In addition, I would like to thank all my friends in Auburn, including Lei Chen, Bo Dai, Wei Wang, Tianxia Li, Tianzi Guo, Bin Xu, Jiawei Zhang, Wei Yuan, Chen Chen, Wei Huang, Chun Guo, Ying Zhu, Sihe Zhang, Rui Xu, Qiang Gu, Jingshan Wang, Lingzhao

Kong, Jingyuan Xiong and etc. I will always miss the time we spent together and value our friendship.

Last but not the least, my deepest gratitude goes to my family, especially my grandpa, Qi Ding who passed away in 2009, my grandma Lizhi Deng, and my parents Yuming Ding and Yan Bai for their years selfless support and unconditional love.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since large-scale and data-intensive applications have been widely deployed, there is a growing demand for high-performance storage systems to support data-intensive applications. Compared with traditional storage systems, next-generation data storage will embrace computing capacity to reduce computational load of host processors in computing nodes. Existing hard disks have limited processing power, which can only be used to handle on-disk scheduling and physical resource management. With the advance of processor and memory technologies, future smart storage systems are promising devices to perform complex on-disk operations [97].

Active disks (*a.k.a.*, smart disks), in which processors are embedded, leverage the computational power, available in commodity disk drives, to deal with application-level processing [87]. Recently, smart disks coupled with embedded processors have been proposed to address the needs of on-drive-data-intensive workloads. Active storage brings three key advantages. First, the amount of data moved back and forth between computing nodes and storage nodes in clusters can be significantly reduced, since large datasets can be locally processed by storage nodes before being forwarded to computing nodes. Second, data-intensive applications run faster, because active storage nodes accelerate data processing operations. If computing nodes and active storage nodes efficiently coordinate, both computing nodes and storage nodes can perform data processing in parallel. Third, network performance in clusters can be improved due to reduced amounts of data moved into and out of storage nodes.

In the past, single-core processors were integrated to smart disks to improve I/O performance of data-intensive applications by manipulating data directly on the disks [38][48][69][97].

But, architecture of energy-efficient processors on a single chip does not necessary guarantee high performance. Thus, improving utilization of advanced multi-core processors has been a thorny subject in computer systems research. We design the architecture, programming model and communication interface of a active/smart storage node. And to fully utilize multi-core processors in our active storage, we incorporated a MapReduce programming model with the active storage (see Chapter 3).

There are two main challenges in applying the active storage to support data-intensive applications on clusters. The first challenge is to partition a parallel application into computation-intensive and data-intensive tasks. If such a partition is successfully created, computing nodes will handle computation-intensive tasks whereas active storage nodes will run data-intensive tasks. The second challenge lies in the coordination between computing nodes and active storage nodes. When it comes to applications where computation-intensive tasks are independent of data-intensive tasks, computing nodes and active storage nodes are non-blocking to each other, meaning that computing and storage nodes can easily operate in parallel. However, if computing nodes have to wait for storage nodes to catch up, the blocked computing nodes could slow down data-intensive applications.

In Chapter 4, to solve the blocking problem incurred by synchronized computing and storage nodes, we developed a pipelining mechanism that exploits parallelism among data processing transactions in a sequential transaction stream. We report the effectiveness of the pipelining mechanism that leverage active storage to maximize throughput of data-intensive applications on a high-performance cluster. To demonstrate the effectiveness of the pipelining mechanism, we implemented a pipelined application called pp-mpiBLAST, which extends an open-source parallel BLAST tool, mpiBLAST [34]. We also develop an analytic model to study the scalability of pp-mpiBLAST on large-scale clusters.

The third part of this dissertation focuses on the configuration of storage devices within the active storage node. Recently, the use of NAND-flash-based Solid State Devices (hereinafter referred as SSDs) has evolved in the primary system storage of enterprise servers and

data centers. However, in those areas, users still hesitate a little bit to perform a large-scale deployment of SSDs because of their poor reliability and the limited lifespan. As the cost of NAND flash has declined with increased density, the number of erase cycles a flash cell can tolerate, on which the number of write operations depends (because of the erase-before-write characteristic), has suffered. Meanwhile, server applications such as OLTP (Online Transaction Processing) [44] normally demand a high-performance and highly reliable underlying storage system. Thus, instead of simply replacing magnetic hard disk drives with SSDs, researchers [49][68][51][80][33] believe that finding an appropriate, fittest and complementary approach to balance the performance, reliability and cost is more challenging and attractive.

In Chapter 5, we propose a hybrid combination of disk arrays designed for active storage system, which uses hard disk drives as a write buffer to cache write requests and de-duplicate the redundant requests. This chapter also introduces an enhanced version of the on-SSD buffer, which supports internal parallelism processing algorithm. Together, the goal is to minimize the writes sent to the SSD without significantly impacting the performance; by doing so, it reduces the number of erase cycles and thus extends SSDs lifetime.

In summary, this dissertation is organized into the following chapters: Chapter 3 proposes and designs the new active storage architecture, the programming interface and the explores the parallel processing potential; Chapter 4 presents a pipeline-parallel processing pattern to cooperate the active node with other machines in the cluster computing environment; and the exploration of the hybrid combination of different types of disk drives is described in Chapter 5; and, finally, Chapter 6 summarizes this dissertation.

Chapter 2

Related Work

2.1 The Next Generation Data Storage is Active/Smart

Since large-scale and data-intensive applications have been widely deployed, there is a growing demand for high-performance storage systems to support data-intensive applications. Compared with the CPU and memory, which are developing at a rapid speed, the development of storage devices is behind the times. Five primary factors have catalyzed the evolution of storage architectures: I/O-bound workloads, improved disk drive attachment technologies, increased on-drive transistor density, emergence of new interconnects, and the cost of storage systems [38]. Next-generation data storage may embrace computing capacity to offload computation from host processors and improve the overall I/O performance. The active or smart disk, which is empowered by assigning the dedicated on-board processor, is introduced to be an aspiring attempt of a new storage model.

2.1.1 Smart Disk

Active/smart disks have been implemented in various forms. For example, in an active disk model proposed by Uysal *et al.* , on-disk application processing becomes possible because of large on-disk memory [97]. Another active disk model designed by Mustafa *et al.* largely relies on stream-based programming, in which host-resident code interacts with disk-resident code using streams and a code-partitioning scheme [70]. Memik *et al.* developed a smart disks architecture, where the operation bundling concept was introduced to further optimize database query executions [69]. Chiu *et al.* investigated a fully distributed processor-embedded distributed smart disks [22]. Pushing computation workloads to memory is another offloading choice [59].

The term smart disk may be used interchangeably with other terms such as intelligent disk (IDISK) [55], SmartSTOR [48], processor-embedded disks [22], and semantically smart disks [92]. Note that IDISK uses on-disk integrated processor-in-memory to exploit emerging VLSI technologies [55]; SmartSTOR [48] consists of a processing unit coupled to one or more disks [48]; and semantically smart disks [92] contains various high-level functionalities. The interface of the object-based storage devices has been standardized recently [86]. Some storage device manufacturers, such as Seagate [89], are also developing object-based storage devices.

2.1.2 Active Storage

From the perspective of the cluster computing area, active storage can be implemented at storage node levels [71][32][91]. Current practice for data-intensive applications on high-performance clusters often result in high I/O communication overhead between computing nodes and storage nodes in the cluster. When massive amounts of data must be transferred back and forth between parallel computing nodes and storage systems, cluster computing applications' performance can suffer from network bandwidth saturation. One efficient approach to reducing network traffic caused by moving data between computing nodes and storage systems is to incorporating computing capacities into storage systems, thereby offloading some data-intensive computing tasks from clusters to their storage nodes. ASF [35] and FAWN [7] are two new examples of active storage implementations at storage node levels. Fitch *et al.* developed the Active Storage Fabrics (ASF) model to address petascale data intensive challenges [7]. ASF is aimed at transparently accelerating host workloads by close integration at the middleware data/storage boundary or directly by data-intensive applications [7]. FAWN - developed by Andersen *et al.* - couples embedded CPUs to small amounts of local flash storage [7]. Andersen *et al.* used FAWN as a building block to construct a cluster, in which computation and I/O capabilities are balanced to improve energy

efficiency of the cluster running data-intensive parallel applications. Active storage has also been explored in terms of handling unstructured data [**?**] and working in a lustre system [79].

2.2 Parallel Programming

The IT industry has improved the cost-performance of sequential computing by about 100 billion times over the past 60 years [78]. The approach, which increased transistor count and power dissipation, worked fine until it hit the power limit a chip is able to dissipate. After crashing into the power wall, the industry decided to replace the single power-inefficient processor with multiple cores processor. Although the exponential processor transistor growth still follows the prediction from Moore, a high transistor density in multi-core processors does not always guarantee great practical performance on applications due to the lack of parallelism. There are cases where a roughly 45% increase in processor transistors have translated to roughly 10-20% increase in processing power [90]. Therefore, an open issue addressed in this study is how to enable data-intensive application to exploit parallelism both in the single-node environment and the cluster computing area. We believe that well-designed parallel programming APIs and models are critical players to benefit from multiple-machines or multiple-core processors.

2.2.1 The MapReduce Programming Model

MapReduce is a parallel programming model developed by Google for simplified data processing in data-intensive applications running on large clusters [26]. Map and Reduce - two primitives in MapReduce - are brought from the idea of functional testing. When the Map function is called, user's input data is partitioned into M pieces, which is processed by one copy of the program on each node in a cluster. One of the program copy becomes a master program managing the entire execution. In each MapReduce program, the Map function first takes an input data specified by the users, and outputs a list of intermediate key/value pairs *(key, value)*. Then, the Reduce function takes all intermediate values associated with

the same key and produces a list of result key/value pairs. The Reduce function typically performs some kind of merging operation. Finally, the output pairs are sorted by their key value.

Like the Google file system, MapReduce is not open source software. Google only describes the MapReduce idea without implementation details. Thus, various open source implementations of MapReduce are available for different computing platforms like clusters [9], multi-core systems [82], multiprocessor systems [82], and graphics processing units (GPU) [46].

Haddoop [9]. Hadoop is a Java software framework implemented by Apache to support data-intensive distributed applications. Inspired by Google's MapReduce and the Google file system, the Hadoop project created its own versions of MapReduce and Hadoop Distributed File System. Hadoop applications can be deployed easily by configuring some variables–some paths and nodes; Hadoop defines one master node that manages all systems and jobs, and other worker nodes. Hadoop is so far the most complete quasi-open-source version of MapReduce in the cluster arena. **Mars [46].**MapReduce has been implemented on NVIDIA GPUs using CUDA in the Mars project. Mars takes advantage of GPUs by using the capability of massive threading. In the Mars implementation, there are a large number of physically-collocated mappers and reducers run in multiple threads. The functions in Mars are classified in two categories: runtime system functions and user define functions. Our McSD is different from Mars in the sense that McSD automatically coordinates data-processing activities between host CPUs and multi-core processors embedded in smart disks. **Phoenix [82].** Phoenix is an implementation of Google's MapReduce for shared-memory multi-core and multi-processor systems. There are two categories of functions in Phoenix: the first group is provided by the Phoenix runtime environment; the second one is defined by programmers. Ranger *et .al* concluded that Phoenix is a promising MapReduce implementation for scalable performance on multi-core and multi-processor systems. Different from other MapReduce implementations on clusters, Phoenix is independent of any parallelizing compiler. In our

study, we evaluated the suitability of the Phoenix implementation for multicore-embedded smart disks.

2.2.2 Parallel Processing for Scientific Applications

Parallel computing can improve performance of scientific applications like sequence database search tools in bioinformatic filed. For example, given a database and query sequences (e.g., DNA, amino-acid sequences), the search tools search for similarities between the query sequences and known sequences in the database. The tools enable scientists to quickly identify the function of newly discovered DNA sequences or to accurately identify species of a common ancestor [64]. Among many sequence search tools, BLAST is one of the most popular tools used on daily basis by bioinformatic researchers . MpiBlast [34] is a promising, open source, parallel implementation of the BLAST toolkit [5]. Like other bioinformatic applications, mpiBlast has to pre-process (*e.g.*, format data) databases before searching for similarities between query sequences and known sequences in the preprocessed databases. Thus, in Chapter 4 we implement a pipelined application - pp-mpiBlast - to demonstrate a way of employing active storage to improve performance of data-intensive bioinformatic applications. Our pp-mpiBlast incorporates a data-processing pipeline with mpiBlast on a high-performance cluster.

2.3 Solid State Disk and Hybrid Disk Combination

The NAND flash memory based SSD, which is used to be evolved in mobile devices and laptops, plays a critical role in revolutionizing the storage system [21][52][54][56][67][81]. *Tape is dead, disk is tape, flash is disk* [39]. They are completely built on semiconductor chips without any moving parts, which results in high random access performance and lower power consumption. And the cost of commodity NAND flash – often cited as the primary barrier to SSD deployment [73] – has dropped significantly, increasing the possibility for dynastic changes in the storage arena. However, reliability and write performance are still

two major concerns against largely deployment of SSDs. Hybrid combinations of storage devices and the enhanced Flash Translation Layer (FTL) are proposed to solve the issues.

2.3.1 Hybrid Disk Combination

San Diego Supercomputer Center (SDSC) has built a large flash-based cluster, which adopts 256TB of flash memory, called Gordon [15]. This academic project is backed by a $20 million funding from the National Science Foundation. Different from the academic case, however, enterprise users still hesitate a little bit to perform a large-scale deployment of SSDs because of their relatively higher cost, poor reliability and the limited lifespan. In terms of reliability and lifespan concerns, the challenge is each block on flash-based storage media has limited erasure times and each block has to be erased before written. Erasure operation reduces both performance, reliability and lifetime. Table5.1 shows the lifespan of the single-level cell and the multiple-level cell NAND memory in terms of P/E (Program/Erase) cycles. Based on the SSD lifetime calculator provided by Virident website [40], the lifetime of a 200GB MLC-based SSD could be only 160 days if the application write rate is 50MB/s.

Thus, HDDs are still regarded as indispensable in the storage hierarchy because of their merits of low cost, huge capacity, above-average reliability, and fast sequential access speed. Instead of simply replacing HDDs with SSDs, researchers [49][68][51][80][33] [95][102]believe that finding a appropriate, fittest and complementary approach to balance the performance, reliability and cost is more challenging and attractive. So far there are many hybrid combinations of storage devices. Chang presented a hybrid approach to large SSDs, which combines MLC flash and SLC flash [17]. Yoon and etc propose a high performance Flash/FRAM hybrid SSD architecture, in which metadata used by the flash translation layer (FTL) is maintained in a small FRAM which targets at processing small random writes [101]. Soundararajan proposed a hybrid disk architecture which uses the magnetic hard disk as the write buffer for the SSD [93].

Table 2.1: NAND Flash Memory Lifetime (P/E Cycles)

Generation	**SLC**	**MLC**	**eMLC**
5X	100,000	10,000	N/A
3X	100,000	5,000	35,000
2X	100,000	2,500	N/A

2.3.2 Extended FTL

Unlike magnetic hard disk drives, SSDs have a Flash Translation Layer (or FTL) which is implemented to emulate a hard disk drive by exposing an array of logical block addresses (LBAs) to the host. FTL averagely spreads the erasure workload on flash-based storage. A bad FTL algorithm not only reduces the SSDs performance, but also wears out SSDs storage units rapidly. Chen and Zhang proposed CAFTL: a content-aware flash translation layer enhancing the lifespan of flash memory based SSDs [20]. CAFTL removes unnecessary duplicate writes to reduce write traffic to flash memory. Gupta and Urgaonkar proposed Demand-based Flash Translation Layer (DFTL) which selectively caches page-level address mappings [41] . A journal Remapping Algorithm JFTL was presented by Choi and Park [23]. JFTL writes all the data to a new region in a out-of-place update process by using an address mapping method [23] .

Most current research attempts to design new Flash Translation Layer algorithms to improve reliability and enhance performance by utilizing the built-on-board cache. Hence, many research projects have been done for using cache as write buffer. Kang applied Non-Volatile RAM (RAM) as write buffer for SSDs to improve overall performance [53]. Kim and Ahn proposed a buffer management scheme called BPLRU for improving random writes in flash storage. BPLRU buffers writes improve performance of random writes [57]. Soundararajan and Prabhakaran presented Griffin, a hybrid storage device, to buffer large sequential writes in Hard Drives [94]. Park and Jung also presented write buffer-aware address mapping for flash memory devices [77].

Chapter 3

McSD: Multicore-Enabled Smart Storage for Clusters

3.1 Motivation

Since large-scale and data-intensive applications have been widely deployed, there is a growing demand for high-performance storage systems to support data-intensive applications. Compared with traditional storage systems, next-generation data storage will embrace computing capacity to reduce computational load of host processors in computing nodes. Existing hard disks have limited processing power, which can only be used to handle on-disk scheduling and physical resource management. With the advance of processor and memory technologies, future active storage systems are promising devices to perform complex on-disk operations [97].

Smart disks (*a.k.a.*, active disks), in which processors are embedded, leverage computational power available in commodity disk drives to deal with application-level processing [28][30][87][66][50]. Recently, active disks coupled with embedded processors have been proposed to address the needs of on-drive-data-intensive workloads. In the past, single-core processors were integrated to active disks to improve I/O performance of data-intensive applications by manipulating data directly on the disks [38][69][48][97]. Smart disks avoid moving data back and forth between storage devices and host processors. Since there is no smart disk product on the market, we decided to investigate smart storage nodes rather than smart disks in this study. The target areas to apply smart/active disks include the content-aware service [24][45], unstructured data management [84][98], offloaded virus protections [25][75], and *etc.*

For the past two decades, hardware designers have used the rapidly increasing transistor speed made possible by silicon technology advances to double performance every 18

months [11]. But the power consumption issue has been raised as a side effect [14][43][42][58]. Unfortunately, approaches to increasing transistor count and clock cycle crashed into the power wall recently; increasing transistor count hits the power limit that a chip is able to dissipate. The industry decided to replace single power-inefficient processors with multi-core processors. There is an increasing need to integrate multi-core processors with devices and peripherals. Thanks to the escalating manufacturing technology, it is feasible to embed multi-core processors into storage nodes for high-performance computing. These demand and trend motivate us to develop a programming framework and a prototype for Multicore-enabled smart storage (hereinafter referred as McSD) for clusters in general and MapReduce clusters in particular.

Architecture of energy-efficient processors on a single chip does not necessary guarantee high performance. Thus, improving utilization of advanced multi-core processors has been a thorny subject in computer systems research [60]. Phoenix–an implementation of the MapReduce model–automatically manages thread creation, dynamic task scheduling, data partitioning, and fault tolerance in multicore processor systems [82]. Phoenix includes a programming API and an efficient runtime system. Phoenix allows programmers to write functional-style code that improves the utilization of multicore processors by automatically parallelizing and scheduling. To fully utilize multi-core processors in McSD, we incorporated Phoenix into multicore-embedded smart disks. Note that MapReduce is Google's programming model for scalable parallel data processing [26]. In addition to Phoenix, there exist a wide range of MapReduce implementations tailored for various computing platforms [9][46][83].

The difference between our McSD approach and conventional smart disks is two-fold. First, the focus of McSD is smart storage nodes rather than smart disks. Second, the goal of McSD is to take performance benefits of multi-core processors not single-core processors embedded in storage nodes.

Six New Features. When architecting an McSD smart storage system in a high-performance cluster, the following six features will be implemented:

- A two-layer cluster computing architecture contains host computing nodes and smart storage nodes.
- Improved I/O performance is achieved by combining processing capabilities of both computing and storage nodes.
- The McSD storage system allows programmers to write MapReduce-like code that can automatically offload data-intensive computation to smart storage.
- Smart storage nodes can communicate with their host computing nodes via a storage interface.
- A programming framework of McSDs allows smart storage nodes to take full advantages from multi-core processors in the storage nodes.
- The APIs and runtime environment in our McSD programming framework automatically handles computation offload, data partitioning, and load balancing.

We will show how to use the McSD programming framework to implement a few real world applications like word-count, string matching, and matrix multiplication.

Main Contributions. In summary, the four major contributions of this study are:

- A prototype of next-generation multicore-enabled smart data storage.
- A programming framework, which include MapReduce-like programming APIs and a runtime environment for multicore-based smart storage in the context of clusters.
- Development of three benchmark applications to test McSD for clusters.
- Single-application experimental results and multiple-application performance evaluation.

In the following Section 3.2, we review the background information and previous related research that motivate and inspire this study. In Section 3.3, we describe the design issues

of the prototype of multicore-enabled smart storage. Section 3.4 presents implementation details of the McSD runtime environment and McSD programming APIs. Experiment results and performance evaluation are discussed in Section 3.5. Finally, Section 3.6 concludes the chapter with future research directions.

3.2 Smart Storage and MapReduce

An increasing number of large-scale data-intensive applications impose performance demands on storage systems, which are the performance bottleneck of various computing platforms. One way to boost I/O performance is to embed processors into hard disk drives, thereby offloading data-intensive computation from host CPUs to hard disks. While the processing capacity in today's disk drives is to manage on-disk scheduling and resource management, future disks can be equipped with dedicated processors to perform complicated data-intensive operations. We call disk drives in which processors are incorporated as smart disks or active disks.

3.2.1 From Smart Disks to Smart Storage

Five primary factors have catalyzed the evolution of storage architectures: I/O-bound workloads, improved disk drive attachment technologies, increased on-drive transistor density, emergence of new interconnects, and the cost of storage systems [38]. Existing smart disk prototypes consist of, from hardware perspective, an embedded processor, a disk controller, on-disk memory, local disk space, and a network interface controller (NIC). From software perspective, a smart disk is comprised of an embedded operating system, a database engine, programming APIs and the like.

Unlike stand-alone PCs, smart disks do not contain I/O components such as keyboard and display. Smart drives may directly connect to their host processors through NICs. In

our prototype, smart disks or smart storage nodes are connected to host CPUs using a file-alternation-monitor mechanism, which allows a smart storage node to communicate with its host computing node without relying on a keyboard and display unit.

Single-core embedded smart disks have been implemented in various forms. For example, in an active disk model proposed by Uysal *et al.* , on-disk application processing becomes possible because of large on-disk memory [97]. Another active disk model designed by Mustafa *et al.* largely relies on stream-based programming, in which host-resident code interacts with disk-resident code using streams and a code-partitioning scheme [70]. Memik *et al.* developed a smart disks architecture, where the operation bundling concept was introduced to further optimize database query executions [69]. Chiu *et al.* investigated a fully distributed processor-embedded distributed smart disks [22]. Pushing computation workloads to memory is another offloading choice [59].

The term smart disk may be used interchangeably with other terms such as intelligent disk (IDISK) [55], SmartSTOR [48], processor-embedded disks [22], and semantically smart disks [92]. Note that IDISK uses on-disk integrated processor-in-memory to exploit emerging VLSI technologies [55]; SmartSTOR [48] consists of a processing unit coupled to one or more disks [48]; and semantically smart disks [92] contains various high-level functionalities. The interface of the object-based storage devices has been standardized recently [86]. Some storage device manufacturers, such as Seagate [89], are also developing object-based storage devices.

Due to a combination of reasons, no out-of-shelf product of the smart/active disk is available on the market. We aimed at implementing an active/smart storage system for clusters [6] and; therefore, we decided to focus on active storage nodes rather than active/smart disks in this section.

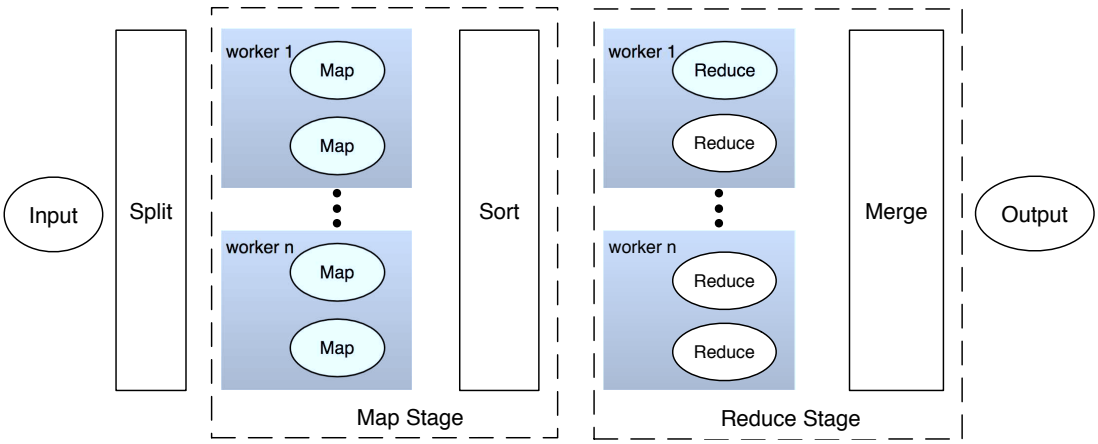


Figure 3.1: The work flow of MapReduce.

3.2.2 Parallel Programming

The IT industry has improved the cost-performance of sequential computing by about 100 billion times over the past 60 years [78]. A high transistor density in multi-core processors does not always guarantee great practical performance on applications due to the lack of parallelism. There are cases where a roughly 45% increase in processor transistors have translated to roughly 10-20% increase in processing power [90]. Therefore, an open issue addressed in this study is how to enable data-intensive application to exploit parallelism in smart disks coupled with embedded multi-core processors. We believe that well-designed parallel programming APIs must be implemented for future smart disks that can benefit from multi-core processors.

MapReduce is a programming model developed by Google for simplified data processing in data-intensive applications running on large clusters [26]. Map and Reduce - two primitives in MapReduce - are brought from the idea of functional testing. When the Map function is called, user's input data is partitioned into M pieces, which is processed by one copy of the program on each node in a cluster. One of the program copy becomes a master program managing the entire execution. In each MapReduce program, the Map function first takes an input data specified by the users, and outputs a list of intermediate key/value pairs *(key, value)*. Then, the Reduce function takes all intermediate values associated with the same

key and produces a list of result key/value pairs. The Reduce function typically performs some kind of merging operation. Finally, the output pairs are sorted by their key value.

Fig. 3.1 illustrates the work flow of the MapReduce model. The main benefit MapReduce lies in its simplicity. Programmers only provide a simple description of an algorithm that focuses on functionality and leaves actual parallelization and concurrency management to a MapReduce runtime system.

Like the Google file system, MapReduce is not open source software. Google only describes the MapReduce idea without implementation details. Thus, various open source implementations of MapReduce are available for different computing platforms like clusters [9], multi-core systems, multiprocessor systems [82] and graphics processing units (GPU) [46].

Hadoop [9] is a Java software framework implemented by Apache to support data-intensive distributed applications. Inspired by Google's MapReduce and the Google file system, the Hadoop project created its own versions of MapReduce and Hadoop Distributed File System. Hadoop applications can be deployed easily by configuring some variables– some paths and nodes; Hadoop defines one master node that manages all systems and jobs, and other worker nodes. Hadoop is so far the most complete quasi-open-source version of MapReduce in the cluster arena.

Phoenix [82] is an implementation of Google's MapReduce for shared-memory multi-core and multi-processor systems. There are two categories of functions in Phoenix: the first group is provided by the Phoenix runtime environment; the second one is defined by programmers. Ranger *et .al* concluded that Phoenix is a promising MapReduce implementation for scalable performance on multi-core and multi-processor systems. Different from other MapReduce implementations on clusters, Phoenix is independent of any parallelizing compiler. In our study, we deploy a extended Phoenix system implementation to our multicore-embedded active/smart disks.

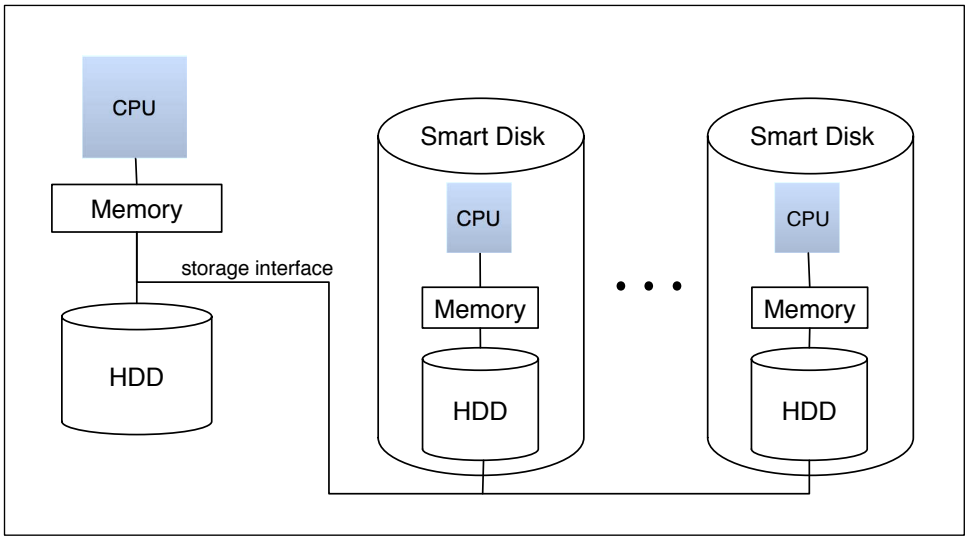


Figure 3.2: McSD - The prototype of multicore-enabled smart storage. Each smart storage node in the prototype contains memory, a SATA disk drive, and a multicore-processor.

3.3 Design Issues

A growing number of data-intensive applications coupled with advances in processors indicate that it is efficient, profitable, and feasible to offload data-intensive computations from CPUs to hard disks [87]. Our preliminary results show that we can improve performance of cluster computing applications by offloading computations from computing nodes to storage nodes.

To improve the performance of large data-intensive applications, we designed McSD - a prototype of multicore-enabled smart data storage. Different from the existing smart-disk solutions, McSD not only addresses the performance needs of data-intensive applications using multi-core processors, but also focuses on smart storage nodes rather than smart disks.

Fig. 3.2 depicts the McSD prototype, where each smart storage node contains a multicore-processor, memory, and a SATA disk drive. In what follows, let us address the following design issues.

- How to build a testbed where an McSD smart storage node is connected to a host computing node?

- How to evaluate the performance of McSD in the testbed and a cluster computing environment?
- How to fully utilize a multi-core processor available in McSD?
- What is the programming framework for McSD?
- How to pass input parameters from a host computing node to its McSD - a smart storage node?
- How to return results from the McSD storage node to the host computing node?

3.3.1 Design of the McSD Prototype

In our McSD prototype, we integrate multi-core processor, a disk controller, main memory, a local disk drive, and a network interface controller (NIC) into a smart storage node. The storage interface of existing smart-disk prototypes (see Section 3.2.1 for details on existing smart disks) is not well designed, because the existing prototypes simply represented a case where host CPUs and embedded processors are coordinated through the network interfaces or NICs in smart disks. To fully utilize the storage-interface in smart data storage, we designed a communication mechanism similar to the file alteration monitor. In our McSD prototype, a host computing node communicates with a disk drive in McSD via its storage interface rather than the NIC. In doing so, we made smart disk prototypes cost-effective since no NIC is needed. Without using NICs, the McSD prototype can adequately represent all the important features of our proposed smart data storage. The design details are described in the following two subsections.

3.3.2 A Testbed for McSD

Recall that although a few smart disk prototypes have been developed, there is no off-the-shelf commodity smart disks. Instead of simulating a smart storage system, we built a testbed for the McSD prototype. Fig. 3.3 briefly outlines the McSD testbed, where two

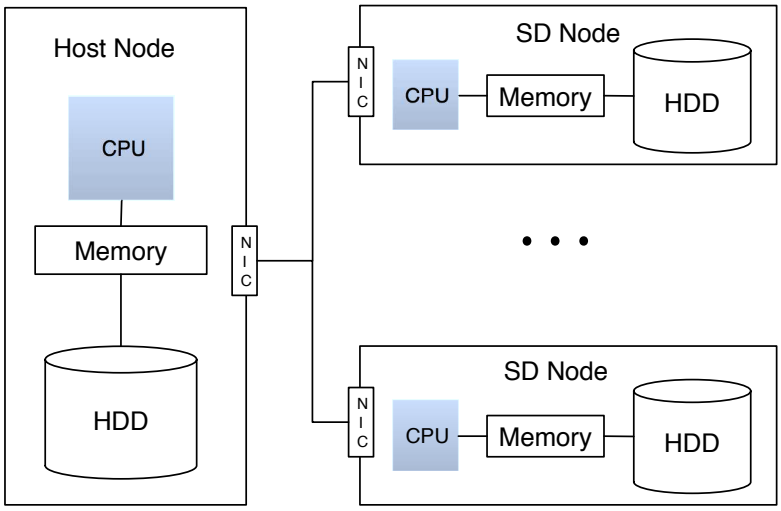


Figure 3.3: A testbed for the McSD prototype. A host computing node and an McSD storage node are connected via a fast Ethernet switch. The host node can access the disk drives in McSD through the networked file system or NFS.

computers are connected through the fast Ethernet. The first computer in the testbed plays the role of host computing node, whereas the second one performs as the McSD smart storage node. The host computing node can access the disks in the McSD node through the networked file system or NFS, which allows a client computer to access files on a remote server over a network interconnect. In our testbed the host computing node is the client computer; the McSD node is configured as an NFS server. We chose to use NFS as an efficient means of connecting the host computing node and the smart storage node, because data transfers between the host and smart storage nodes are handled by NFS.

We run three real-world applications as benchmarks on this testbed to evaluate the performance of the McSD prototype. The benchmarks considered in our experiments (see Section 3.5) include word count, string matching, and matrix-multiplication.

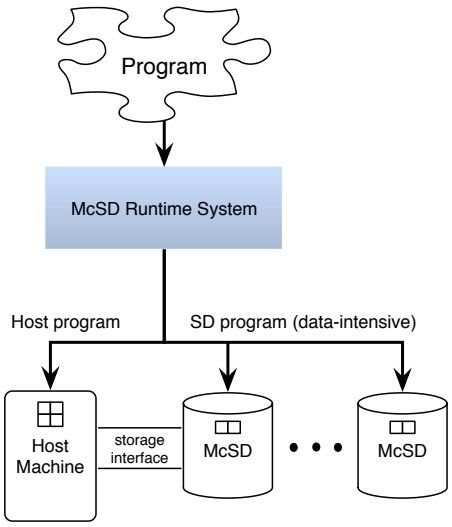


Figure 3.4: The programming framework for a host computing node supported by an McSD smart storage node.

3.3.3 A Programming Framework

Fig 3.4 shows a programming framework for a host computing node supported by an McSD smart storage node. The framework generates an optimized operation plan for data-intensive programs running on the McSD testbed (see Section 3.3.2 for the description of the testbed), where there is a host computing node and an McSD smart storage node. The framework automatically assigns general purpose operations to the host computing nodes and offloads data-intensive operations to the McSD storage node, in which Phoenix (see Section 2.2.1 for the description of Phoenix) handles parallel data processing. Although applying Phoenix in the McSD node can not increase performance for all applications running in our testbed, Phoenix can substantially boost performance of data-intensive applications (see Section 3.5). Because this programming framework provides a relatively flexible autonomy, data processing modules (e.g., word-count, sort, and other primitive operations) can be readily added into an McSD smart storage node.

To seamlessly integrate Phoenix into an McSD smart system, we addressed the issue of limited embedded memory in McSD by implementing new functions like data partitioning, which split input data files whose memory footprints exceed the memory capacity of the

McSD smart data storage. The implementation details of the new functions in the McSD programming framework can be found in Section 3.4.

3.3.4 Putting It All Together - A Cluster with McSD

We can build both stand-alone computing systems and active storage farms using McSD.

Stand-Alone Computing Systems. A set of McSD nodes can be connected together to serve as a stand-alone data-intensive computing system, where each McSD node is running data intensive applications whose data are locally stored in each storage node. The cluster of McSD can connect to back-end storage servers supporting data services such as data deduplication, data backup and recovery.

Smart Storage Farm. McSD can be used to build a smart storage farm that is connected to a front-end high-performance computing cluster. In this case, data-intensive applications are running on the front-end cluster and data processing services are supported by McSD nodes of the smart storage system. The smart McSD storage nodes can improve the front-end clusters I/O bandwidth by greatly reducing data movement between the front-cluster and the McSD storage system.

3.4 Implementation Details

3.4.1 System Workflow and Configuration

Unlike the previous network-attached smart storage, the McSD smart storage uses the SATA interface to transfer data. We implemented the McSD prototype using a host computing node and a multicore-enabled storage node (see Section 3.3.1 and 3.3.2 for the design issues of the prototype). In the prototype, the multicore storage node has no keyboard, mouse, and display unit. It is worth noting that storage nodes in other existing smart-disk prototypes have keyboard and mouse activities. Compared with the other earlier prototypes, our McSD prototype better resembles next-generation multicore-enabled smart storage systems for clusters. Smart storage nodes connected to cluster computing nodes only needs to

process on-node data-intensive operations. In other words, smart storage nodes only provide some primitive functions termed as data-intensive processing modules (or processing modules for short) in the McSD prototype.

Fig. 3.3 shows the hardware configuration of the McSD prototype where a host computing node is connected to an McSD storage node through the SATA bus interface. One of the most important implementation issues is to allow a host computing node to offload data-intensive computations to McSD. There are two general approaches to implementing computation offloading. First, each offloaded data-intensive operation or module are delivered from a host node to an McSD storage node (hereinafter refered to as McSD or McSD node) when the operation or module needs to be processed by McSD. Second, all data-intensive operations and modules are preloaded and stored in the McSD storage node. Although the first approach can handle the dynamic environment problem where data-intensive operations/modules are not predictable, the downside of the first approach lies in high communication overhead between host computing nodes and McSD storage nodes. The second approach reduces the communication overhead caused by moving data-intensive operations/modules, because the operations/modules are residing in McSD prior to the execution of the data-intensive programs.

In the process of implementing the McSD prototype, we took the second approach - preloading data-intensive modules. We believe that the preloading approach is practical for a vast variety of real-world applications, where data-intensive processing modules can be determined before the programs are executed in a host computing node accompanied by an McSD smart storage node. In our preloading approach, the program running on the computing node has to invoke the processing modules preloaded to the McSD node. An invocation mechanism, called smart-file-alternation monitor (smartFAM), was implemented to enable the host node to readily trigger the processing modules in the McSD node. The implementation issues of smartFAM are addressed in the next subsection.

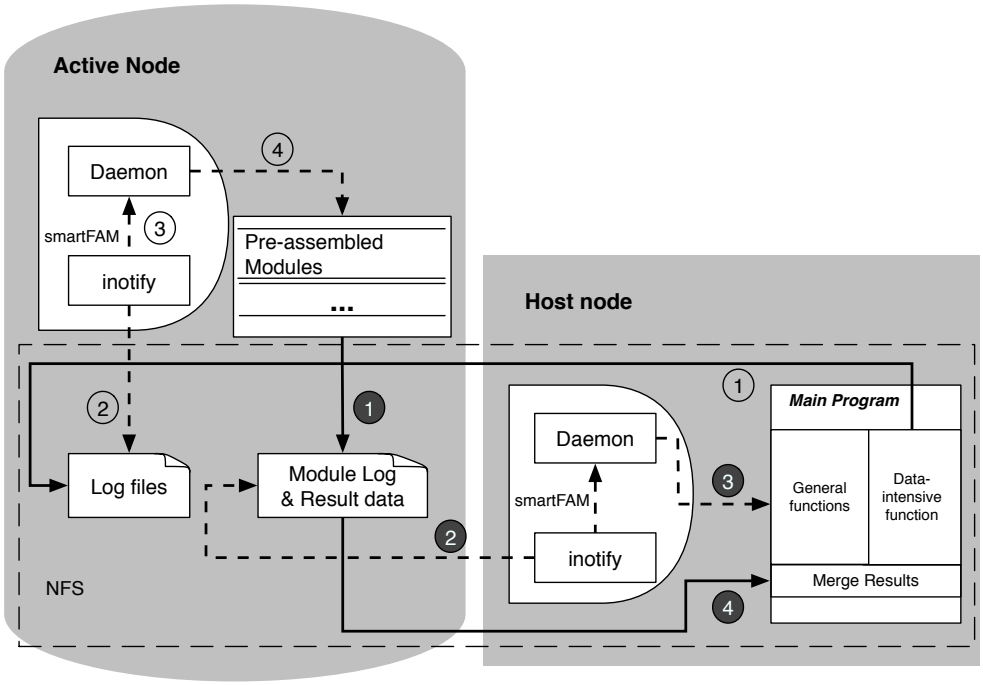


Figure 3.5: The implementation of smartFAM - an invocation mechanism that enables a host computing node to trigger data-intensive processing modules in an McSD storage node.

3.4.2 Implementation of smartFAM

Fig. 3.5 illustrates the implementation of smartFAM - an invocation mechanism that enables a host computing node to trigger data-intensive processing modules in an McSD storage node. smartFAM mainly contains two components: (1) the inotify program - a Linux kernel subsystem that provides file system event notification; and (2) a daemon program that invokes on-node data-intensive operations or modules.

To make our McSD prototype closely resemble future multicore-enabled smart storage, we connected the host node with the McSD smart-storage node using the Linux network file system or NFS. In the NFS configuration, the host node plays a client role whereas the McSD node performs as a file server. A log-file folder, created in NFS at the server side (i.e., the McSD smart-storage node), can be accessed by the host node via NFS. Each data-intensive processing module/operation has a log file in the log-file folder. Thus, when a new data-intensive module is preloaded to the McSD node, a corresponding log-file is

created. The log file of each data-intensive module is an efficient channel for the host node to communicate with the smart-storage node (McSD node). For example, let us suppose that a data-intensive module in the McSD node has input parameters. The host node can pass the input parameters to the data-intensive module residing the McSD node through the corresponding log file. Thus, the host writes the input parameters to the log file that is monitored and read by the data-intensive module. Below we address the following two questions related to usage of log files in McSD:

- (1) How to pass input parameters from a host node to an McSD storage node?
- (2) How to return results from an McSD storage node to a host?

Passing input parameters from a host node to an McSD smart-storage node. When an application running on the host node offloads data-intensive computations to the McSD node, the following five steps are performed so that the host node can invoke a data-intensive module in the smart-storage node via the module's log file (see Fig. 3.5):

Step 1: The application on the host node writes input parameters of the module to its log file on in McSD. Note that NFS handles communications between the host and McSD via log files.

Step 2: The inotify program in the McSD node monitors all the log files. When the data-intensive module's log file in McSD is changed by the host, inotify informs the Daemon program in smartFAM of McSD.

Step 3: The Daemon program opens the module's log file to retrieve the input parameters passed from the host. Note that this step is not required if no input parameter needs to be transmitted from the host to the McSD node.

Step 4: The data-intensive module is invoked by the Daemon program; the input parameters are passed from Daemon to the module.

Step 5: Go to Step 1 is more data-intensive modules in the McSD node are invoked by the application on the host.

Returning results from an McSD smart-storage node to a host node. Results produced by a data-intensive module in the McSD node must be returned to the module's caller - a calling application that invokes the module from the host node. To achieve this goal, smartFAM takes the following four steps (see Fig. 3.5):

Step 1: Results produced by the module in the McSD node are written to the module's log file.

Step 2: The inotify program in the host node monitors the log file, checking whether or not the results have been generated by McSD. After the module's log file is modified by McSD (i.e., the results are available in the log file), This inotify program informs the Daemon program in the host node.

Step 3: The Daemon program in the host notifies the calling application that the results from the McSD node are available for further process.

Step 4: The host node accesses the module's log file and obtain the results from the McSD node. Note that this step can be bypassed if no result should be returned from McSD to the host.

3.4.3 Partitioning and Merging

A second implementation issue that has not been investigated in the existing smart-storage prototypes is how to process large data sets that are too large to fit in on-node memory. In one of our experiments, we observed that the Phoenix runtime system does not support any application whose required data size exceeds approximately 60% of a computing node's memory size. This is not a critical issue for Phoenix, because Phoenix is a MapReduce framework on shared-memory multi-core processor or multiple processors systems where memory size are commonly larger than those residing in smart storage nodes. On-node memory space in smart storage nodes is typically small compared with front-end high-performance computing nodes. Thus, before we attempted to apply Phoenix in McSD

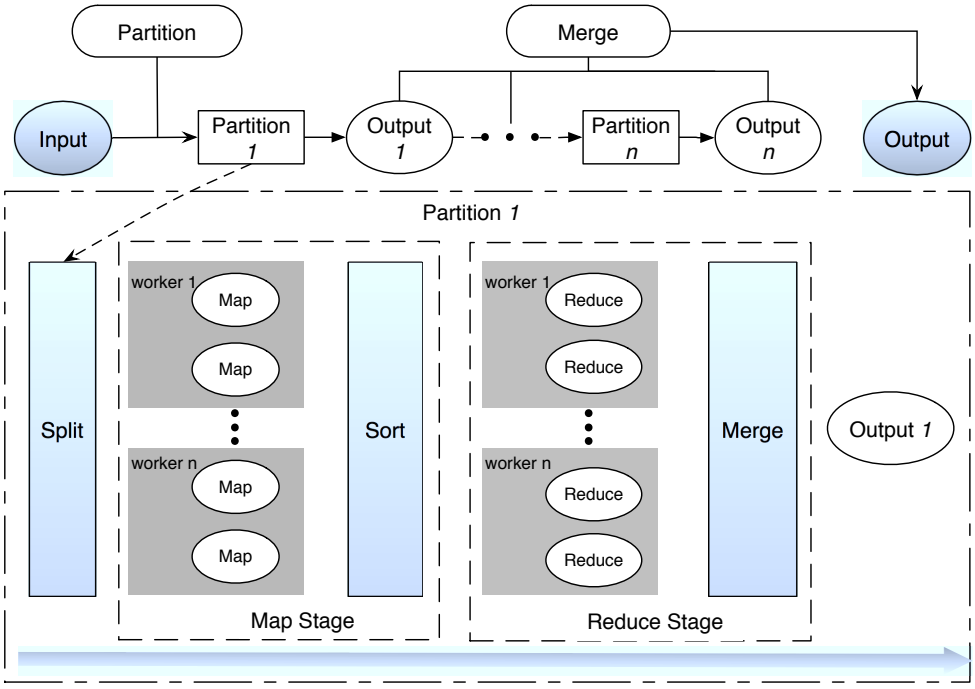


Figure 3.6: Workflow of the extended Phoenix model with partitioning and merging

smart disks, we had to address this out-of-core issue - data required for computations in McSD is too large to fit in McSD memory.

Our solution to the aforementioned out-of-core issue is to partition a large data set into a number of small fragments that can fit into on-node memory before calling a MapReduce procedure. Once a large data set is partitioned, the small fragments can be repeatedly processed by the MapReduce procedure in McSD. Intermediate results obtained in each iteration can be merged to produce a final result. Our partitioning solution has two distinct benefits:

- Supporting huge datasets whose size may exceed the memory capacity of an McSD storage node.
- Boosting performance of data-intensive applications (e.g., word-count) by improving the memory usage of McSD (see Fig. 3.9 in Section 3.5).

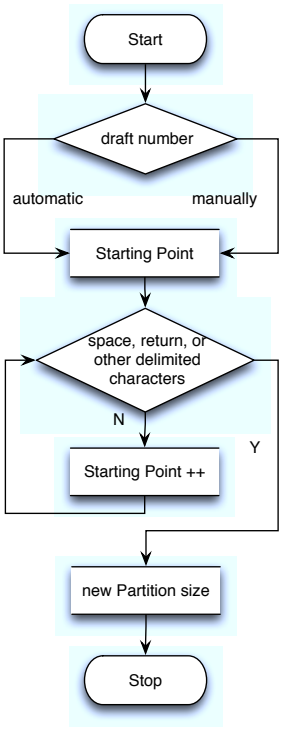


Figure 3.7: The workflow diagram of integrity checking.

Because both input data sets and emitted intermediate data are located in memory during the MapReduce stage, the memory footprint is at least twice of input data size. The partitioning solution, of course, is only applicable for data-intensive applications whose input data can be partitioned. In our experiments, we evaluated the impact of fragment size on the performance of applications. Evidence (see Fig. 3.9 in Section 3.5) shows that data partitioning can improve performance of certain data-intensive applications.

3.4.4 Incorporating the Partitioning Module into Phoenix

Fig. 3.6 depicts the work flow of the modified version of Phoenix; it can be considered as a two-stage MapReduce process. The Partition function is provided by the runtime system, while the Merge function needs to be programmed by the user to support different applications. Take an example of a Word-count command: *wordcount [data-file] [partition-size].* Fragment sizes of every new partitions are determined by (1) the number of [partition-size]

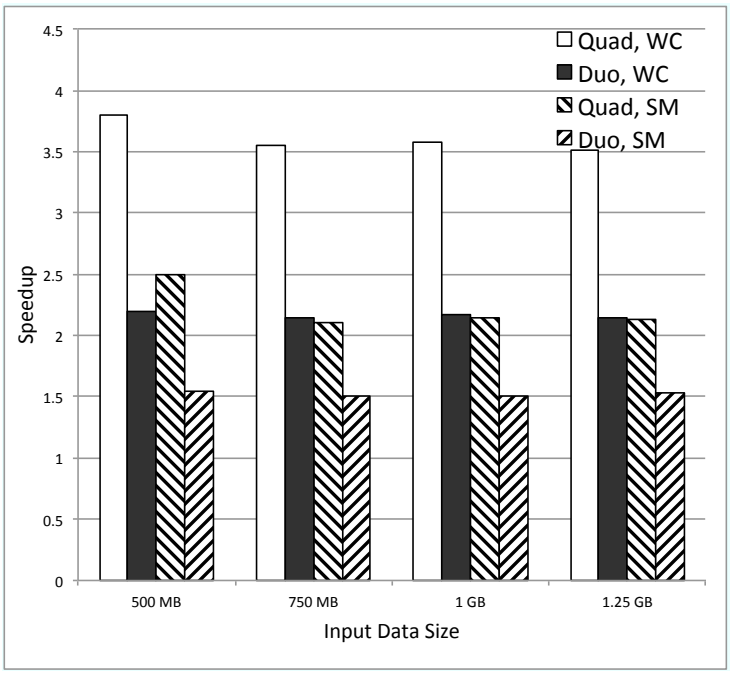
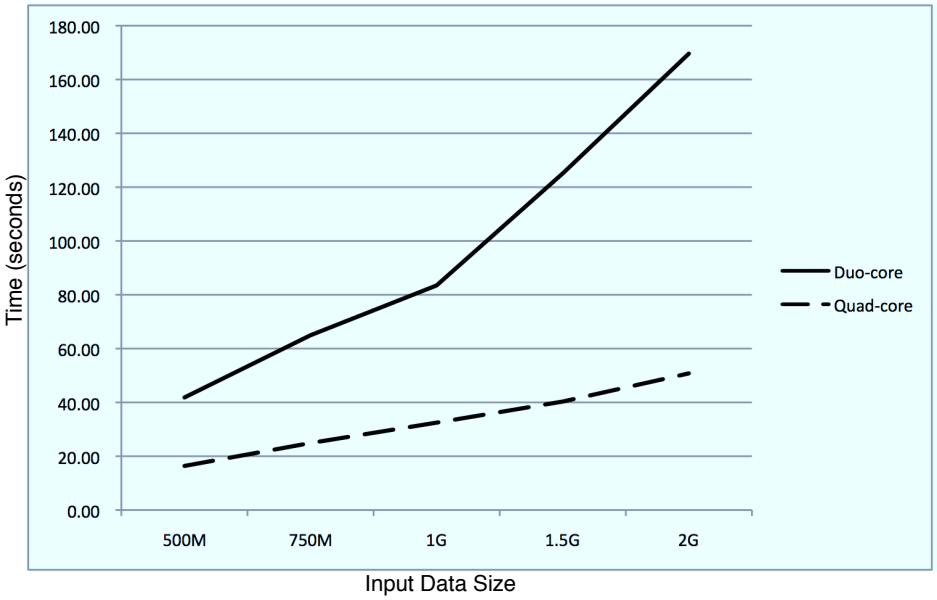
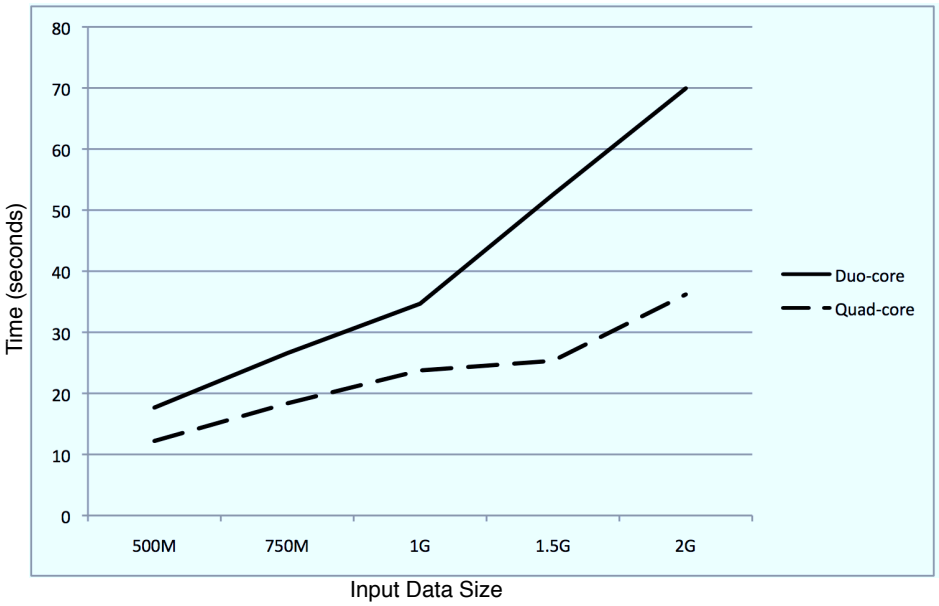


Figure 3.8: Performance Speedup. It depicts speedups of partition-enabled Phoenix vs. original Phoenix and the sequential approach on both duo-core and quad-core machines.

provided by the programmer/system and (2) the extra displacements from integrity-check function in order to make sure the new partition is ended correctly. If there is no [partition-size] parameter, the program will run in native way. Otherwise, the number of [partition-size] can be manually filled in by the programmer or automatically determined by the runtime system (e.g., Phoenix). In order to achieve a better performance, the empirical data or the details of operator may be required for the automatic approach. The integrity-check function will automatically return the extra displacements by scanning from the starting point of [partition-size] till the first space, return or the symbol defined by the programmer. The reason we involved the integrity-check procedure to the Partition function is there exists the consistency issue of partitioned data files; the content of the source data file could be broken in shatters (e.g. a word could be cut and placed into two slitted files not on purpose). Fig. 3.7 describes the integrity-check procedure.



(a) Growth curve of WC on Duo and Quad. It draws the growth curves of elapsed time on duo-core and quad-core machines in terms of Word-Count.



(b) Growth curve of SM on Duo and Quad. It draws the growth curves of elapsed time on duo-core and quad-core machines in terms of String-Match.

Figure 3.9: Single Application Performance. The data size is scaling from 500MB to 1.25GB.

3.5 Evaluations

3.5.1 Experimental Testbed

We performed our experiments on a 5-node cluster, whose configuration is outlined in Table 1. There are three types of nodes in the cluster: one of host computing node, one of smart storage nodes, and three other general purpose computing nodes. Operating system running on the cluster is Ubuntu 9.04 64-bit version. The nodes in the cluster are connected by Ethernet adapters, Ethernet cables, and one 1Gbit switch. All the general purpose computing nodes share disk space on the host node through Network File System (NFS), while the host node is sharing one folder on the McSD node. The processing modules, extended Phoenix system and SmartFAM have been set up on both the host and SD nodes. Then in order to emulate the routine work, we run the Sandia Micro Benchmark (SMB) among all the nodes except the McSD smart-storage node. We choose MPICH2-1.0.7 as our message passing interface (MPI) on the cluster. All benchmarks are compiled with gcc 4.4.1. We briefly describe the benchmarks running on our testbed in the following sub-section.

Table 3.1: The Configuration of the 5-Node Cluster

	Host	SD	Nodes ×3
CPU	Intel Core2 Quad Q9400	Intel Core2 Duo E4400	Intel Celeron 450
Memory	2GB		
OS	Ubuntu 9.04 Jaunty Jackalope 64bit version		
Kernel version	2.6.28-15-generic		
Network	1000Mbps		

- **Word Count (WC)**: It counts the frequency of occurrence for each word in a set of files. The Map tasks process different sections of the input files and return intermediate data ⟨key, value⟩ that consist of a word and a value of 1. Then the Reduce tasks add up the values for each identity word. Finally, the words are sorted and printed out in accordance with the frequency in decreasing order.

- **String Match (SM)**: Each Map searches one line in the "encrypt" file to check whether the target string from a "keys" file is in the line. Neither sort or the reduce stage is required.

- **Matrix Multiplication (MM)**: Matrix multiplication is widely applicable to analyze the relationship of two documents. Each Map computes multiplication for a set of rows of the output matrix. It outputs multiplication for a row ID and column ID as the key and the corresponding result as the value. The reduce task is just the identity function.

- **Sandia Micro Benchmark (SMB)**: It is developed by Sandia National Laboratory to evaluate and test high-performance networks and protocols. We use it in our experiment to emulate the routine work.

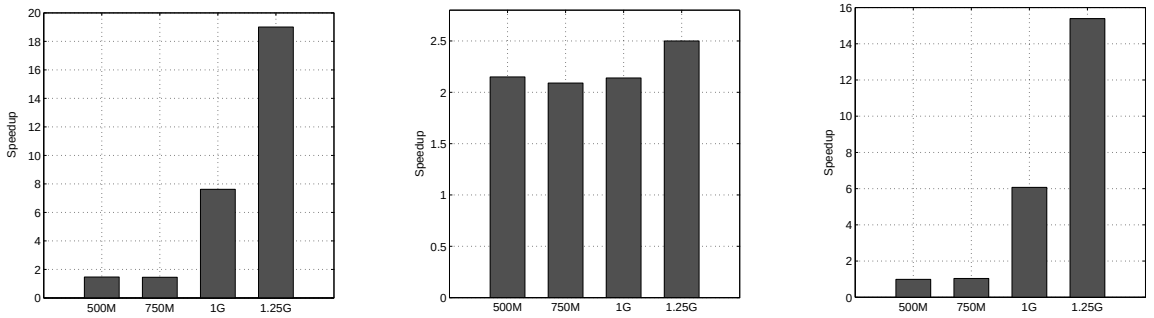
3.5.2 Single-Application Performance

Fig. 3.9 shows the speedup achieved by using the Partition-enabled programming model, relative to the no-partition version and sequential implementation, respectively. In terms of single application benchmarks, we observed that the traditional Phoenix cannot support the Word-count and the String-match for data size larger than 1.5G, because of the memory overflow. From Fig. 3.9, when the data size is in a reasonable interval (say, less than half of the memory size), the traditional parallel approach provides almost the same performance. However, in terms of Word-count, when the data size is huge (compared with the memory size), the elapsed time of Partition-enabled approach is only 1/6 of the traditional one.

When comparing with the sequential approach, both the benchmarks can achieve a 2X speedup, which proves the fully utilization of duo-core processor. Fig. 3.9(a) and Fig. 3.9(b) show the plots of the execution time versus the size of the input data file on the two SD platforms. Since we can observe that the performance curve has linear-like growth, our methodology provides scalability performance for its audience objective. We can summarize that: (1) for data-size sensitive applications, such as Word Count, the Partition procedure can not only support data size which cannot fit in the physical memory but also improve the performance; (2) for applications that are not data-intensive, the Partition model can only enhance their supportability of data-size range. Of course, all those observations are based on the assumption that the applications are partition-able; (3) the last but not the least, the use of our Partition-enabled approach can fully utilize the multicore processor in almost all subjects in this test.

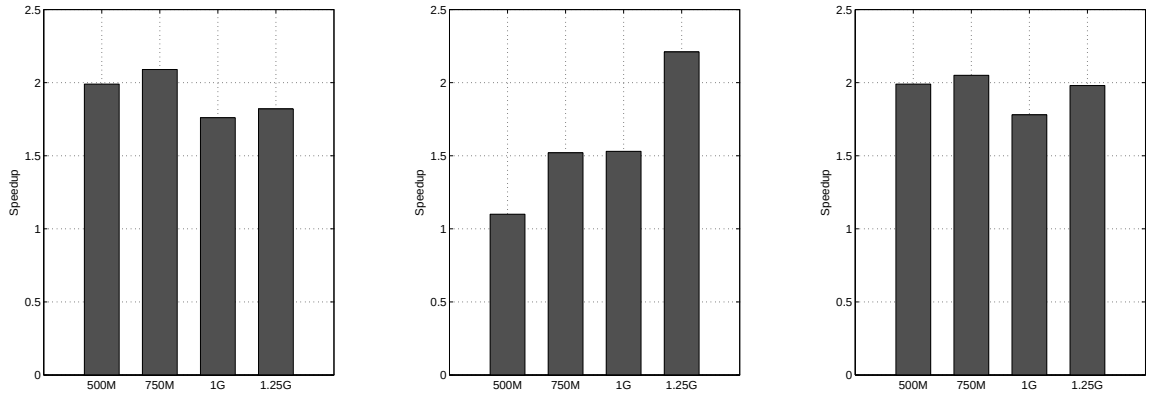
3.5.3 Multiple-Application Performance

When multiple applications are running concurrently–following the McSD framework, the system should exhibit the basic properties: (1) the system overall throughput can be increased, and (2) the overall performance of the application set can be improved. In order to evaluate our McSD execution framework, we create two multiple-application benchmarks, each of which contains : a computation-intensive function and a data-intensive one. To explore how well our system meets the performance expectations, we report two pairs of application benchmarks: Matrix-multiplicity/Word-count and Matrix-multiplicity/String-match. The first pair is very data-intensive, or memory-consuming, since the memory footprint of Word-Count is around three times of the input data size. On the other hand, the memory footprint of String-Match is around two times of the input data size. Thus, those two are representatives of two levels of data-intensive applications.



(a) Host Node Only (b) Tradtional SD (c) McSD without Partition

Figure 3.10: **Speedups of Matrix Multiplicity and Word-count.** Trad_SD - traditional smart storage (SD) with single-core processor embedded. DuoC_SD-nopar - duo-core processor embedded smart storage operating in a parallel way without the partitioning function. The benchmarks are running on the multicore host node only in the Host-only scenario. The last one, Host-part, is partitioning-enabled on the Host node. Compared with the traditional smart storage (running sequentially), our McSD improves the overall performance by 2x. With the data size increasing, the elapsed time of non-partitioned approaches (the DuoC-SD and Host-only) can cost 16 to 18 times more than that of the McSD approach.



(a) Host Node Only (b) Tradtitional SD (c) McSD without Partition

Figure 3.11: Speedups of Matrix-multiplicy and String-match. Compared with the traditional smart storage (SD) running sequentially, our McSD improves the overall performance by 1.5x. When data size is increasing, McSD improves the performance of the non-partitioning approaches (the DuoC-SD and Host-only) by 2x.

For each pair of applications, we set up four scenarios to execute the program: (1) the benchmarks running on the traditional single-core SD mode (a combination of host and single-core SD node), (2) the benchmarks running on the duo-core embedded SD mode without Partition function, (3) the programs running on the host node only, and (4) the programs follow the McSD execution framework; the host machine handles the computation-intensive part and the SD machine processes the on-node data-intensive function. Each of the solutions performs three tests: parallel processing without partition, parallel processing with partition and the sequential solution.

Fig. 3.10 and Fig. 3.11 illustrate the performance improvement of using the optimized approach, the parallel-enabled one with 600MB partition, against the other scenarios. Fig. 3.10 and Fig. 3.11 show speedups on the pair of MM/WC and MM/SM, respectively. We defined the performance speedup to be the ratio of the elapsed time without the optimization technique to that with the McSD technique. From both of figures, we observe a common point: compared with the traditional (single-core processor equipped) SD, the McSD (duo-core processor embedded) averagely improves the overall performance by 2X for both two pairs of applications. Thus it illustrates the utilization of the duo-core processor. Also, the difference between those two sets of figures is obvious. In terms of the MM/WC, the elapsed time of non-partitioned parallel approaches, host node only and McSD without Partition, increase nonlinearity. When the data size exceeds a threshold, the speedups averagely achieve 6.8X and 17.4X. On the other hand, the McSD can only make slightly improvement when the data size are 500MB and 750MB (below the threshold). In contrary, the speedups of the MM/SM, which represents less data-intensive applications, are both averagely 2X speedup.

As we can see, using our methodology gives better speedups compared with the traditional SD (averagely 2X) and parallel processing without Partition (maximum to 17X). While the SD being widely considered to be one of the heterogeneous computing platforms, the frameworks like ours will be considered to manage the system and improve the performance.

3.6 Summary

Processor-enabled smart storage can improve I/O performance of data-intensive applications by processing data directly using storage nodes, because smart nodes avoid moving data back and forth between storage and host computing nodes. Thanks to the escalating manufacturing technology, it is possible to integrate multi-core processors into smart storage nodes. In this study, we implemented a prototype called McSD for multicore-enabled smart storage that can improve performance of data-intensive applications by offloading data processing to multicore processors employed in storage nodes of computing clusters.

Our McSD system differs from conventional smart/active storage in two ways. First, McSD is a smart storage nodes rather than a smart disk. Second, McSD can leverage multi-core processors in storage nodes to improve performance of data-intensive applications running on clusters.

McSD along with its programming framework enables programmers to write MapReduce-like code that can automatically offload data-intensive computation to multicore processors residing in smart storage nodes. The McSD programming framework allows smart storage nodes to take full advantages from embedded multi-core processors. The APIs and a runtime environment in this programming framework automatically handles computation offload, data partitioning, and load balancing. The McSD prototype was implemented in a testbed–a 5-node cluster containing both host computing nodes and McSD smart-storage nodes. Our experimental results were taken by running three real-world applications on the testbed. The tested data-intensive applications include Word Count, String Matching, and Matrix Multiplication. Our multicore-enabled smart storage system - McSD - significantly reduces the execution time of the three applications. Overall, we conclude that McSD is a promising approach to improving I/O performance of data-intensive applications.

Our prototype for multcore-enabled smart storage was built in a MapReduce cluster. The performance of the benchmark applications largely depends on the testbed. Therefore, we will upgrade our testbed (e.g., replace Ethernet with Infiniband) to evaluate the impact of

fast network interconnects on McSD. Perhaps the most exciting future work lies in exploring (1) the extensibility of data-processing modules and operations (*i.e. data-intensive applications and database operations*) that are preloaded into McSD smart-disk nodes, (2) the parallelisms among multiple McSD smart disks, and (3) a mechanism in McSD to support fault tolerance and improve reliability.

Chapter 4

Using Active Storage to Improve the Bioinformatics Application Performance: A Case Study

4.1 Motivation

Processing massive amounts of data has resulted in a mushrooming of data-intensive applications like bioinformatics data processing. Evidence shows that the collective amount of genomic information doubles every 12 months [63]. Most bioinformatics applications have to deal with the I/O bottleneck issue. Processing huge datasets in a high-performance cluster normally requires copying data from storage nodes to computing nodes, thereby leading to a large number of I/O operations. Active storage is an effective technique to improve applications' end-to-end performance by offloading data processing from computing nodes to storage nodes.

Active storage brings three key advantages. First, the amount of data moved back and forth between computing nodes and storage nodes in clusters can be significantly reduced, since large datasets can be locally processed by storage nodes before being forwarded to computing nodes. Second, data-intensive applications run faster, because active storage nodes accelerate data processing operations. If computing nodes and active storage nodes efficiently coordinate, both computing nodes and storage nodes can perform data processing in parallel. Third, network performance in clusters can be improved due to reduced amounts of data moved into and out of storage nodes.

4.1.1 Challenges

There are two main challenges in applying active storage to support data-intensive applications on clusters. The first challenge is to partition a parallel application into computation-intensive and data-intensive tasks. If such a partition is successfully created, computing nodes will handle computation-intensive tasks whereas active storage nodes will run data-intensive tasks.

The second challenge lies in the coordination between computing nodes and active storage nodes. When it comes to applications where computation-intensive tasks are independent of data-intensive tasks, computing nodes and active storage nodes are non-blocking to each other, meaning that computing and storage nodes can easily operate in parallel. However, if computing nodes have to wait for storage nodes to catch up, the blocked computing nodes could slow down data-intensive applications.

4.1.2 Contributions

In this chapter, we address the partitioning and synchronization issues in the context of active storage supporting bioinformatic applications. To solve the blocking problem incurred by synchronized computing and storage nodes, we developed a pipelining mechanism that exploits parallelism among data processing transactions in a sequential transaction stream. We report the effectiveness of the pipelining mechanism that leverage active storage to maximize throughput of data-intensive applications on a high-performance cluster.

To demonstrate the effectiveness of the pipelining mechanism designed for active storage, we implemented a pipelined application called pp-mpiBLAST, which extends mpiBLAST, which is an open-source parallel BLAST tool. pp-mpiBLAST deals with a sequential data processing transactions, each of which contains a filtering/formatting task and a mpiBLAST task. The mechanism overlaps data filtering/formatting in active storage with parallel BLAST computations in computing nodes, thereby allowing clusters and their active storage nodes to perform data processing in parallel. The pp-mpiBLAST application relies on active

storage to filter unnecessary data and to format databases, which are then forwarded to the cluster running mpiBLAST.

We develop an analytic model to study the scalability of pp-mpiBLAST on large-scale clusters. This model allows us to study the performance of pp-mpiBLAST on a cluster using active storage. This performance model is ideal for application developers who have limited computing resources to test the scalability of their parallel applications using active storage. Furthermore, programmers can use the analytic model to explore the design space related to the number of computing nodes, active storage speed, data processing capacity, and input data size. The model shows the behavior of pp-mpiBLAST under different configurations of the cluster coupled with active storage.

Measurements made from a working implementation and a modeling study suggest that this method not only improves mpiBLAST's overall performance by up to 75%, but also achieves high scalability on clusters coupled with active storage.

In the Section 4.2, we review the background information and previous related research. In Section 4.3, we describe the design and implementation details of the active storage node. The analytical model is presented in Section 4.4. Experiment results and performance evaluation are discussed in Section 4.5. Finally, Section 4.6 provides conclusions and future research directions.

4.2 Background

4.2.1 Active Storage

Current practice for data-intensive applications on high-performance clusters often result in high I/O communication overhead between computing nodes and storage nodes in the cluster. When massive amounts of data must be transferred back and forth between parallel computing nodes and storage systems, cluster computing applications' performance can suffer from network bandwidth saturation. One efficient approach to reducing network

traffic caused by moving data between computing nodes and storage systems is to incorporating computing capacities into storage systems, thereby offloading some data-intensive computing tasks from clusters to their storage nodes. That inspired researchers to make the storage more "active", or "smart".

Active storage can be implemented at storage node levels [71][32][91]. ASF [35] and FAWN [7] are two new examples of active storage implementations at storage node levels. Fitch *et al.* developed the Active Storage Fabrics (ASF) model to address petascale data intensive challenges [7]. ASF is aimed at transparently accelerating host workloads by close integration at the middleware data/storage boundary or directly by data-intensive applications [7]. FAWN - developed by Andersen *et al.* - couples embedded CPUs to small amounts of local flash storage [7]. Andersen *et al.* used FAWN as a building block to construct a cluster, in which computation and I/O capabilities are balanced to improve energy efficiency of the cluster running data-intensive parallel applications. Active storage has also been explored in terms of handling unstructured data [**?**] and working in a lustre system [79]. In this study, we focus on active storage like ASF and FAWN. We implemented the pipelining technique in an active storage testbed that is similar to ASF and FAWN.

4.2.2 Parallel Bioinformatic Applications

Parallel computing can improve performance of bioinformatic applications like sequence database search tools. Given a database and query sequences (e.g., DNA, amino-acid sequences), the search tools search for similarities between the query sequences and known sequences in the database. The tools enable scientists to quickly identify the function of newly discovered DNA sequences or to accurately identify species of a common ancestor [64].

Among many sequence search tools, BLAST is one of the most popular tools used on daily basis by bioinformatic researchers . MpiBlast [34] is a promising, open source, parallel implementation of the BLAST toolkit [5]. Like other bioinformatic applications, mpiBlast

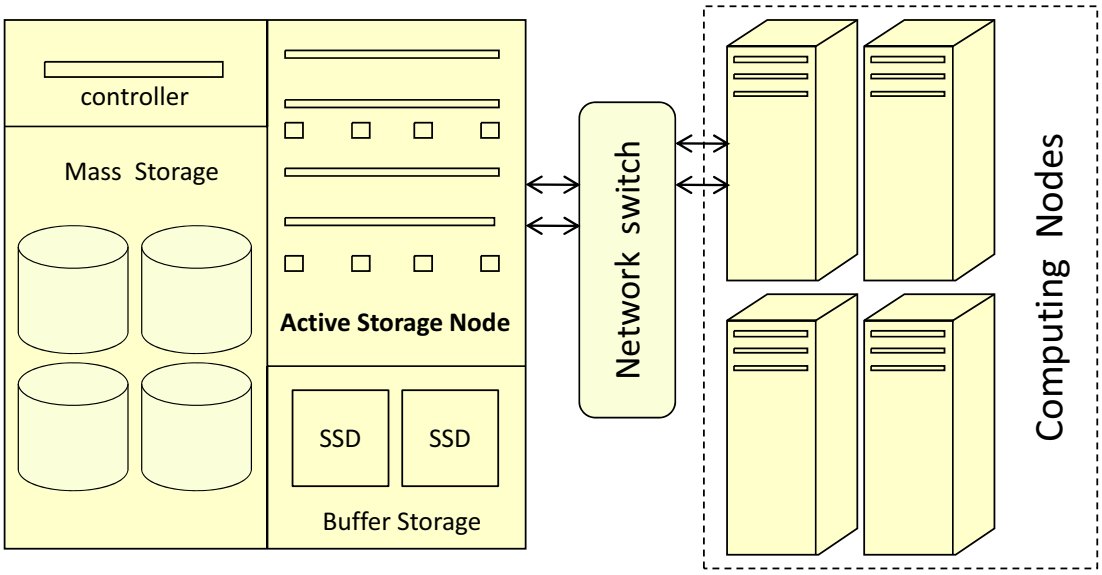


Figure 4.1: A cluster involves a collection of computing nodes and active storage nodes.

has to pre-process (*e.g.*, format data) databases before searching for similarities between query sequences and known sequences in the preprocessed databases.

We implemented a pipelined application - pp-mpiBlast - to demonstrate a way of employing active storage to improve performance of data-intensive bioinformatic applications. Our pp-mpiBlast incorporates a data-processing pipeline with mpiBlast on a high-performance cluster.

4.3 Design and Implementation

In this section, we describe the system from the top down: an overview of the system, a hybrid mix of storage devices, and the parallel pipelined processing. Hereinafter, the active storage node is refereed to as ASN for short.

4.3.1 Active Storage for Clusters

A typical high-performance cluster consists of computing nodes and storage nodes. Data-intensive applications on clusters can cause heavy I/O traffic between the computing and storage nodes in the cluster. To achieve high performance for data-intensive applications, we aim to reduce network traffic caused by moving massive amounts of data from/to storage systems in clusters. This goal can be accomplished by offloading data-intensive computations from computing nodes to active storage nodes.

Fig. 4.1 illustrates a cluster that involves a collection of computing nodes coupled with active storage nodes. Storage nodes become active if they can handle application-level data processing offloaded from computing nodes.

The storage devices can be further divided into two categories: the mass storage and the buffer storage. Benefiting from the non-volatile memory store, Solid state disk (SSD) is a new option to fill the latency gap, which is around 5-order-of-magnitude, between main memory and spinning disks [47]. Thus in our system, SSDs are used as the buffer disk drives: large-scale data is moved from mass storage to buffer drives before processing. The results of experiments in Section 4.5 show that SSDs not only speed up the I/O but also provide a better scalability performance. The advantage of using the hybrid mix of both the solid state disk and the magnetic hard disk is mutual complementarity: the fast and expensive cooperates with the large-capacity and cost-efficiency.

In this chapter, we use a commodity computer as the computing end. The "channel" formed by the computing nodes of cluster and the ASN is considered as a pipeline, or assembly line. In other words, by applying the active storage, applications containing multiple stages are capable to extend to a parallel pipelined implementation. The exploration of parallelism improve the performance data-intensive applications. As a case study, we extend the mpiBlast, a well-know parallel BLAST application, to a parallel pipelined implementation, called pp-mpiBlast.

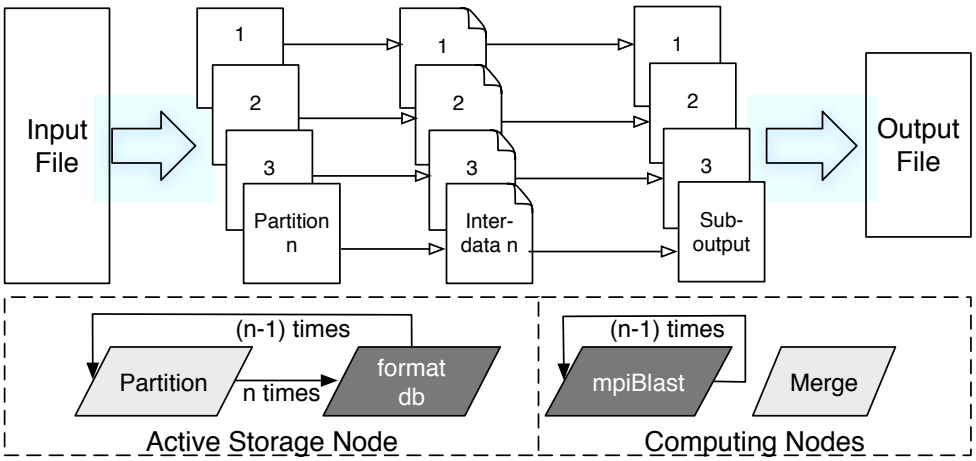


Figure 4.2: pp-mpiBlast Workflow

4.3.2 Parallel Pipelined System

Native mpiBlast application can be easily considered as two steps: format the raw database file (corresponding to the query request) and run the parallel BLAST functions to do the comparison. Thus, the pre-cook (*i.e. format*) phase and the parallel computation phase are handled by the ASN and computing nodes, respectively.

How to utilize the active storage device has always been another critical issue. In general, all cases can fall into two scenarios. The first is that tasks are independent (*i.e.* computing nodes and active storage nodes are non-blocking to each other), meaning that computing and storage nodes can easily can operate in parallel. However, if computing nodes have to wait for storage nodes to catch up, the blocked computing nodes can slow down data-intensive applications. For instance, in terms of mpiBlast, the parallel comparison step requires the formated database file from the previous step. The assembly line pattern is a one of the solutions for the second scenario.

As a case study, we extend the mpiBlast to a parallel pipeline implementation (hereinafter refereed to as pp-mpiBlast). The pp-mpiBlast system consists two tasks: 1) raw database formating, and 2) genome or protein sequences comparison. Further subdividing

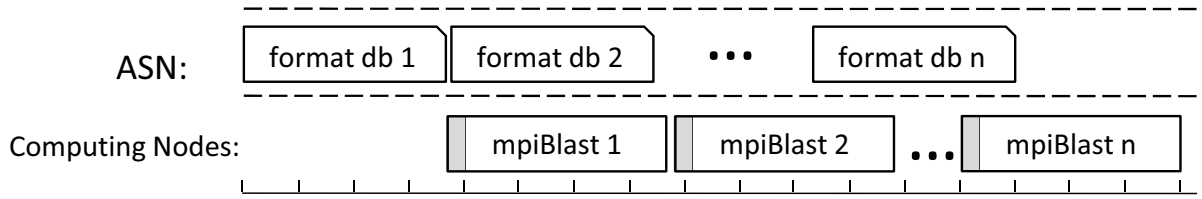


Figure 4.3: Pipeline Tasks Scheduling

the pipeline patterns, there are inter- and intra-application pipeline processing. The pp-mpiBlast is intra-application parallel processing, which means that, as the name - 'intra-' - suggests, one native sequential transaction is partitioned into multiple parallel pipelined transactions. The system performance is improved by fully exploiting the parallelism. The workflow of pp-mpiBlast is depicted in Fig. 4.2.

Intra-application Pipeline Processing

As we mentioned in the previous section, in order to extend a sequential transaction to multiple pipelined parallel transactions, both partition and merge functions are introduced in the pp-mpiBlast system. Fig. 4.3 illustrates the two-task two-stage pipeline processing workflow. The pipeline pattern no only improves the performance by exploiting the parallelism, but also can solve the out-of-core processing issue, which means required amount of data are too large to fit in the ASN's main memory. In pp-mpiBlast, partition function is implemented within *mpiformatdb* fucntion running on ASN. And the merge function is a separate one running on the front node of the cluster.

When partitioning the source data, an assistant function - the integrity-check - automatically returns the extra displacements by scanning the return or the symbol defined by the programmer. The reason we involved the integrity-check procedure to the partition function is that there exists the consistency issue of partitioned data files; the content of the source

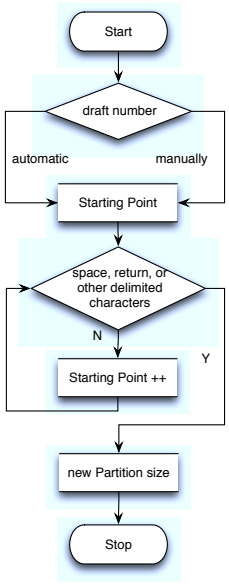


Figure 4.4: The workflow diagram of sequence-integrity checking.

data file could be broken in shatters (e.g. a sequence could be cut and placed into two slitted fragments not on purpose). Fig. 4.4 describes the integrity-check work flow.

Pipeline parallelism is an important processing pattern and we are interested in providing models and guidance for tuning the scalability and the performance using this pattern. In Section 4.4, we develop a mathematics model for analysis.

Preliminary Result

In order to prove the feasibility of the partition-based intra-application pipeline design, a preliminary test on a single-node with 2GB memory environment is performed. We extended Word-Count and String-Match benchmarks of Phoenix system [82], which is a shared-memory implementation of MapReduce, to intra-application pipeline editions using partition and merge runtime functions we developed under Phoenix system. Fig. 4.5 depicts the work flow of the extended approach. After the input data is partitioned into fragments in size of 600 MB, each of them is processed sequentially using native word-count or string-match applications. And then, the generated sub-results are merged at the end. The control

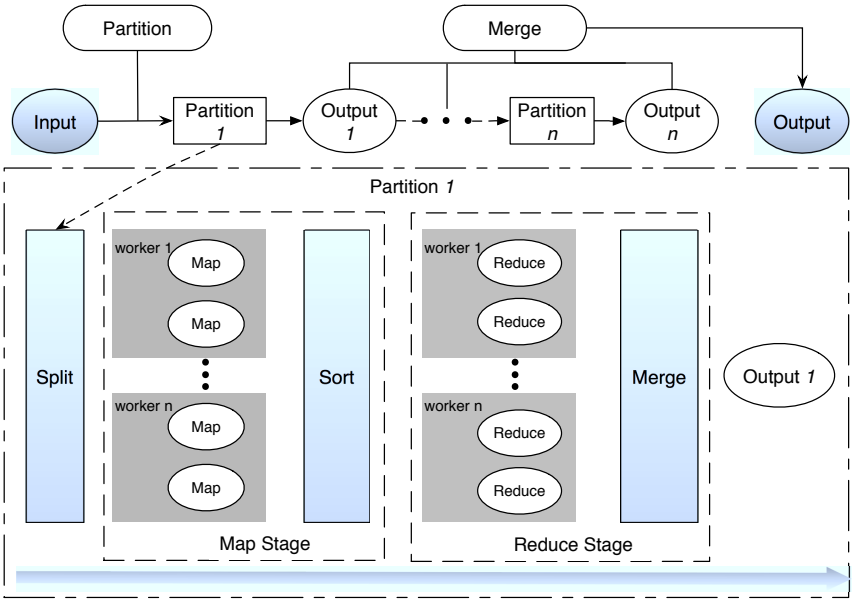


Figure 4.5: Workflow of the extended Phoenix model - intra-application pipeline

test is that we run the native applications to process the input data without the partition and merge functions.

Table 4.1 shows the results. We can observe that in terms of data-intensive, especially memory-intensive, applications, partitioning can significantly reduce the running time. For example, in terms of word-count application results, an average 2.4X speedup of time consumption can be achieved. To the contrary, partitioning does not benefit the speed of the string-match application. But, it can make the large-scale data-intensive applications running on limited memory machines. For example, when executing the string-match application without the help of partition function, the native system does not support the case that the input data size is more than two times of the local memory size. Thus, the preliminary results show that the partition-enabled design can (1) improve data-intensive applications' performance, (2) adapt the data-intensive applications to limited memory machines, or both.

Table 4.1: Running time (in seconds) of performing the Word-Count and String-Match benchmarks w/ and w/o partition function under different input data size (in GBytes) on single node. The testbed machine contains 2 GBytes main memory.

	WordCount (s)		**StringMatch (s)**	
	1 GB	**1.25 GB**	**1 GB**	**1.25 GB**
w/ partition	40.50	50.91	17.76	20.61
w/o partition	85.71	139.54	17.62	21.00

4.4 Modeling and Analysis

We develop in this section an analytic model to study performance and scalability of the parallel pipelined BLAST on large-scale cluster computing platforms. The analytic model has the following three key advantages:

- **Performance Evaluation**: The model allows us to study the performance (i.e., speedup and throughput) of parallel pipelined BLAST system with active storage.
- **Scalability Analysis**: The performance model is ideal and suitable for bioinformatics application developers who have limited local computing resources to test the scalability of their parallel applications using active storage.
- **Design-Space Exploration**: Application programmers can use the model to explore the design space related to the number of computing nodes, active storage speed, data processing capacity, and input data size. The model shows the behavior of the parallel pipelined BLAST under different configurations of a high-performance cluster coupled with active storage.

Section 4.3 describes the parallel pipelined implementation of a basic local alignment search tool or BLAST. Our pipelined BLAST has two data processing stages: (1) formatting data and (2) retrieving and processing formatted data. The first stage of the pipeline preprocess is the input of a data set before passing on to the second stage that run mpiBLAST - a parallel implementation of BLAST.

Response time, speedup, and throughput are three critical performance measures for the pipelined BLAST. Denoting T_1 and T_2 as the execution times associated with the first stage and second stage in the pipeline, we can calculate the response time $T_response$ for processing each input data set as the sum of T_1 and T_2. Thus, we have

$$T_{response} = T_1 + T_2. \tag{4.1}$$

The throughput (see Eq. 4.2) of the two-stage pipelined system is inversely proportional to the maximum of the two execution times T_1 and T_2.

$$Throughput = \frac{1}{max\,(T_1, T_2)}. \tag{4.2}$$

The speedup for the pipelined BLAST is:

$$Speedup = \frac{T_{unpipelined}}{T_{pipelined}}, \tag{4.3}$$

where $T_{unpipelined}$ is the data processing time for the unpipelined BLAST and $T_{pipelined}$ is the processing time of the pipelined BLAST. If n is the number of input data sets to be processed by a cluster with active storage, then processing time of the unpipelined BLAST is the product of n and $T_response$ (see Eq. 4.1), leading to

$$T_{unpipelined} = n \times T_{response} = n \times (T_1 + T_2). \tag{4.4}$$

The processing time for the pipelined BLAST is:

$$\begin{aligned} T_{pipelined} &= T_1 + (n-1) \times max\,(T_1, T_2) + T_2 \\ &= T_{response} + (n-1) \times max\,(T_1, T_2)\,, \end{aligned} \tag{4.5}$$

where T_1 is the processing time of stage 1 for the first data set, T_2 is the execution time of stage 2 for nth data set, $(n-1) \times max(T_1, T_2)$ is the time spent on $n-1$ data sets when the two stages are carried out is parallel. Applying Eqs. 4.4 and 4.5 to Eq. 4.3, we obtain speedup as:

$$Speedup = \frac{n}{1 + (n-1) \times \frac{max(T_1,T_2)}{T_{response}}}. \tag{4.6}$$

Now we are positioned to model execution times T_1 and T_2 for the two stages in the pipeline. The processing time of the first stage is the sum of (1) data input/output times and (2) filtering/formatting time $T_{1,comp}$. Input time is proportional to unformatted input data size s_u and inversely proportional to disk read bandwidth b_i. Similarly, output time is proportional to formatted output data size s_f and inversely proportional to disk write bandwidth b_o. This leads to:

$$T_1(s_u, s_f, b_i, b_o) = \frac{s_u}{b_i} + T_{1,comp}(s_u) + \frac{s_f}{b_o}. \tag{4.7}$$

The execution time T_2 of stage two is the sum of (1) input time of formatted data and (2) processing time $T_{2,comp}$. The input time depends on data size s_f, disk input bandwidth b_i, and the number m of computing nodes in a cluster. Assuming that the formatted data size s_f is uniformed distributed among the m computing node, we can express the input data as $\frac{s_f}{m \times b_i}$. Thus, the execution time T_2 for the second stage is given below:

$$T_2(s_f, b_i, m) = \frac{s_u}{m \times b_i} + T_{2,comp}(s_f, m), \tag{4.8}$$

where $T_{2,comp}(s_f, m)$ is affected by the formatted data size s_f and the cluster size (i.e., number of computing nodes m).

4.5 Evaluations

4.5.1 Evaluation Environment

We implemented the pp-mpiBlast in a 14-node cluster (1 node works as ASN), whose configuration is outlined in Table 4.2. The nodes in the cluster are connected by Ethernet adapters, Ethernet cables, and one 1Gbit switch. We choose an Intel X-25M 80GB solid state disk, and a SATA Raid tower with four WD50000AAKS disks as a RAID 0 array. The *MPICH2-1.0.7* is chosen as the message passing interface (MPI) in the cluster. The pp-mpiBlast is extended from mpiBlast-1.5.0, in which NCBI Blast 2.2.24 is the comparison tool. All applications are compiled with *gcc 4.4.1*.

Table 4.2: The Test Platform

	Computing Nodes	ASN
CPU	Intel Xeon X3430	Intel Q9400
Memory	2GB	
OS	Ubuntu 9.04 Jaunty Jackalope 64bit version	
Kernel version	2.6.28-15-generic	
Network	1000Mbps	

4.5.2 Individual Node Evaluation

We perform *mpiformatdb* program under different storage disk schemes: w/ SSD as the buffer disk and w/o buffer disk. Since the data pre-fetching is out of the scope of this chapter, when we test the SSD cases, we modify the program to move the data from mass storage devices to the SSD and then trigger the format function. That means the time consumption of SSD contains both the data-transfer and data-format phases. Table 4.3 shows the results. Observed from the table, the case of SSD (*e.g.*the second row) always perform better than the HDD since it benefits from large amount of random read and write when the function is reordering the sequence in a descending order based on entry length. After balancing the disk capacity, storage reliability, I/O speed, and random w/r speed, the hybrid mix of mass storage and buffer disk is a promising choice. Fig. 4.6 shows the comparison of trends of

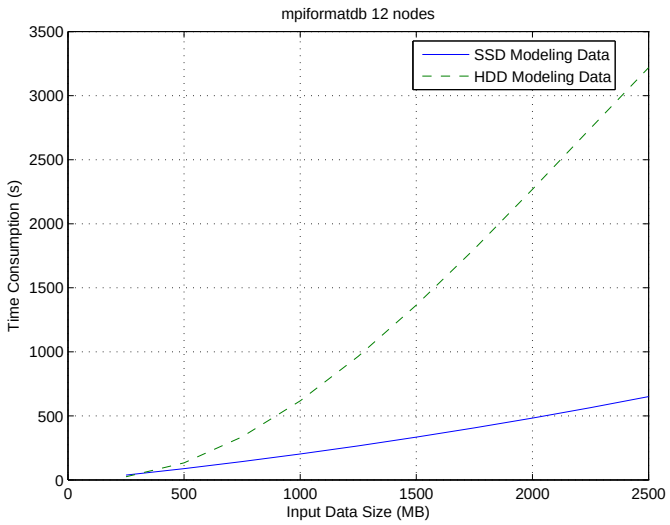


Figure 4.6: Time Consumption Trends Comparison: SSD vs. HDD.

using SSD and HDD. We can observe that the scalability of using HDD for mpiformatdb function is not good, compared with the SSD one. Thus in the following experiments, we use the SSD as the buffer disk.

Table 4.3: Time cost (in seconds) of performing mpiformatdb program under different input data size (in MB) on an ASN.

Running time (seconds)						
Devices	500 MB	750 MB	1 GB	1.25 GB	1.5 GB	1.75 GB
HDD	108.4	369.5	639.1	945.6	1385.0	1845.1
SSD	101.9	164.4	225.9	291.1	369.5	441.1

4.5.3 System Performance Evaluation

Figure 4.7 shows the system performance evaluation. The pp-mpiBlast testbed, which is configured by 12 computing nodes and 1 ASN (follows the pipelined processing pattern), is compared with two control experiments (native systems with different number of nodes) : the native mpiBlast running on 1) 12 nodes cluster (equals to the number of computing nodes in pp-mpiBlast testbed), and 2) 13 nodes cluster (equals to the total number of nodes in pp-mpiBlast testbed). The reason to choose two competitors is to present the performance improvement comprehensively.

In Fig. 4.7 and Fig. 4.8, the testbed with native mpiBlast contains 12 and 13 nodes, respectively. In each figure, results generated by pp-mpiBlast are using 3 different partition sizes: 250 MB, 500 MB, and 750 MB, which are presented by three sub-figures, respectively from left to right. Observed from the figures, the time consumption comparison results show that the performance of pp-mpiBlast beat both control testbeds: averagely reducing the execution time by 50% (i.e. 2X speedup). And Fig. 4.9 shows that: the improvement is greater when the input data size increase within a certain range of the input size, which is relative to the main memory size.

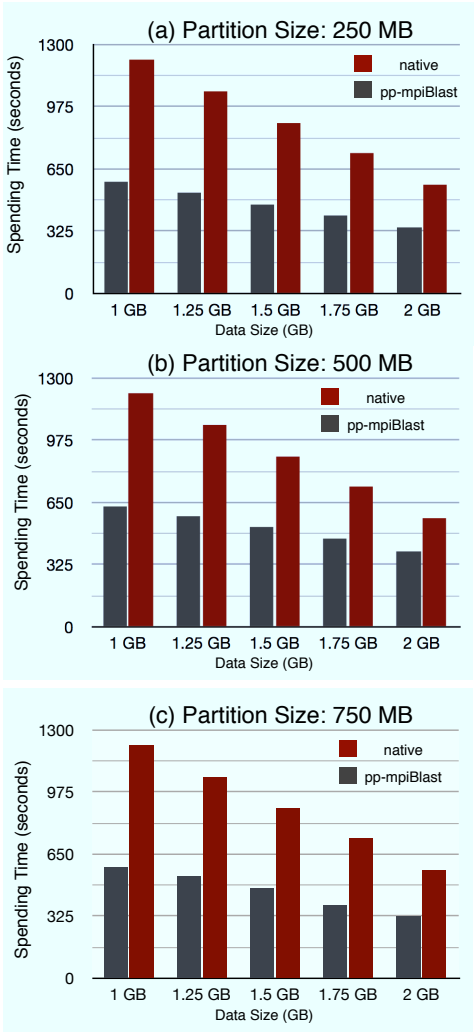


Figure 4.7: System Evaluation Results I: Execution time comparison between pp-mpiBlast system and native system (running on a computer cluster of 12 nodes). The pp-mpiBlast system contains a twelve nodes computing cluster and one ASN. Results are generated under 3 different partition sizes: 250 MB, 500 MB, and 1.25 GB, which are presented by three sub-figures, respectively from left to right.

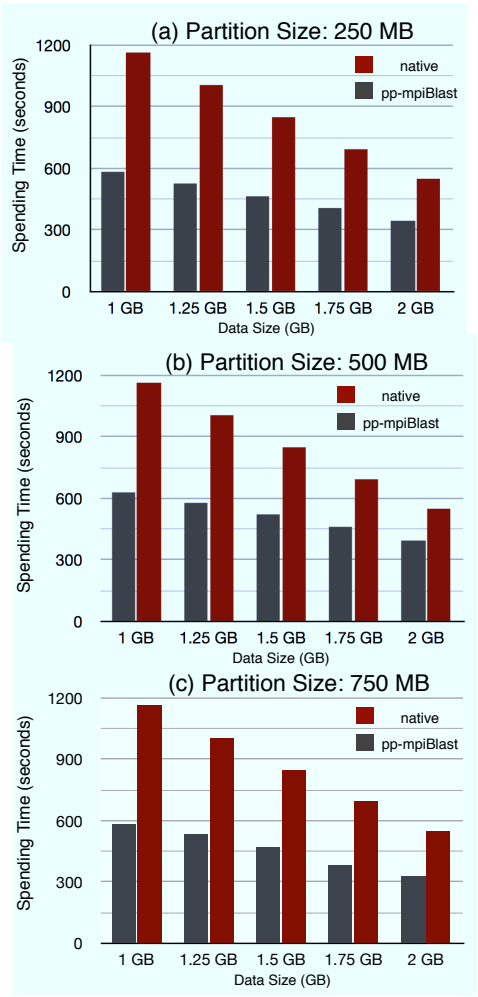


Figure 4.8: System Evaluation Results II: Execution time comparison between pp-mpiBlast system and native system (running on a computer cluster of 13 nodes). The pp-mpiBlast system contains a twelve nodes computing cluster and one ASN. Results are generated under 3 different partition sizes: 250 MB, 500 MB, and 1.25 GB, which are presented by three sub-figures, respectively from left to right.

We also compare the performance in terms of different partition size. Fig. 4.10 presents that 500 MB partition provides a better performance in general. Based on the data, the reason can be summarized as followings. 1) The smaller the better is not true because small partitions always generate more overheads. And 2) the larger the better is also not convinced since the . It means that a partition-size-threshold exists for optimal performance. The issue of measuring the threshold in quantity will be dug in our future work.

Based on the test results, we can see that using intra-application pipeline parallel processing model to extend mpiBlast improves the performance and scalability. However, as we mentioned in the previous sections, the approach is not general; it requires that the target application can be decomposed into stages, such as streaming and RMS applications. Also, applications parallelized using pipeline model are very sensitive to load balancing. In order to avoid bubbles or reduce their side effect, how to balance the heterogeneity issue between the ASN and computing nodes will be our next topic.

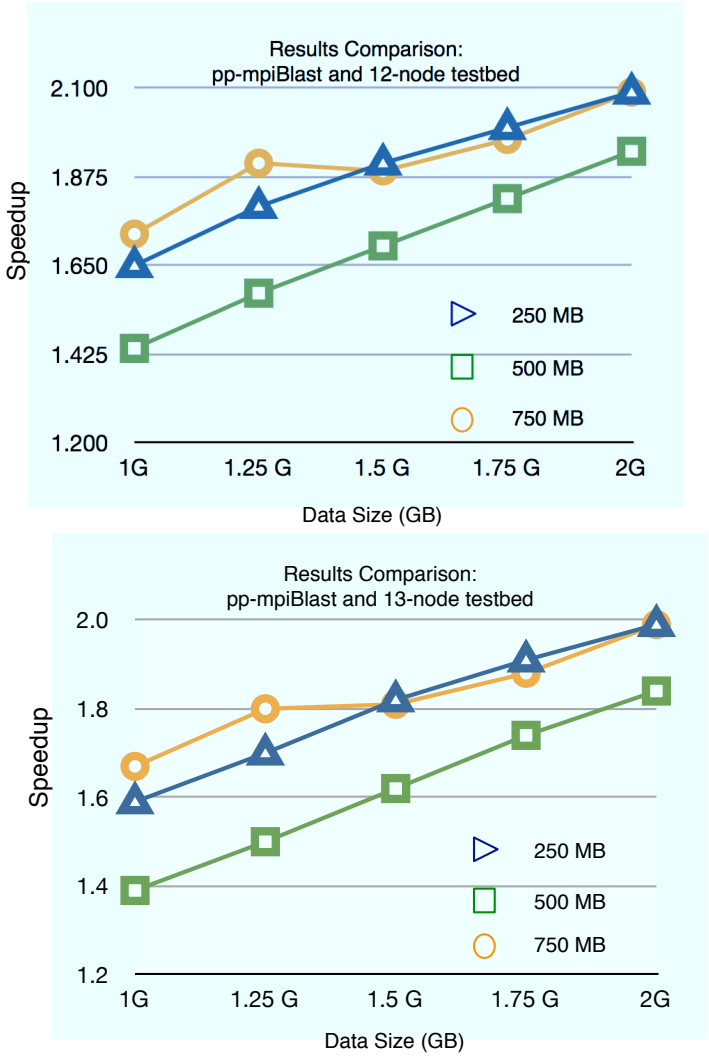


Figure 4.9: Speedup Trends: As input data size grows larger, the performance speedups of using pp-mpiBlast increase. Sub-figure on left is the comparison result between pp-mpiBlast and the 12-node testbed. And the right one is the result compared with the 13-node testbed.

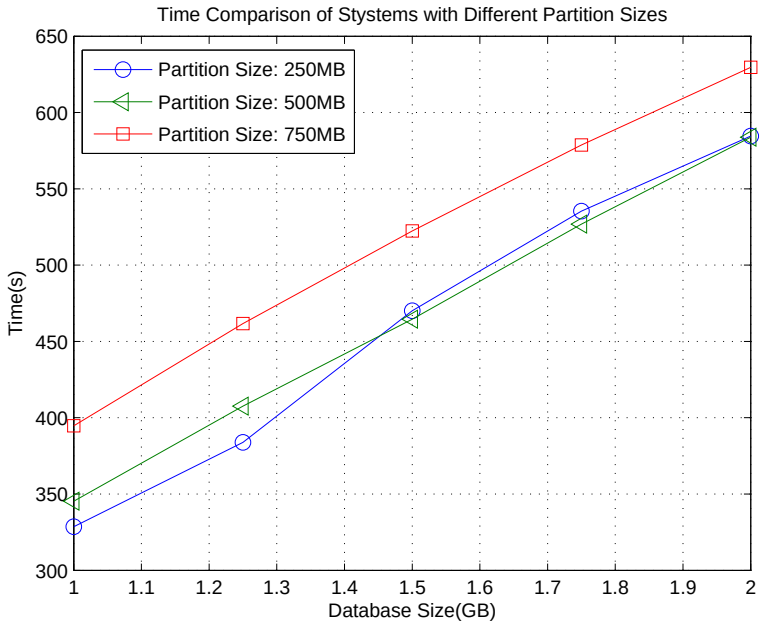


Figure 4.10: Time Consumption Curves Comparison: Different Partition Size.

4.6 Summary

We have presented a pipelining technique that allows computing nodes and active storage nodes in a cluster to perform CPU-intensive and data-intensive computing in parallel. The technique exploits parallelism among data processing transactions in a sequential transaction stream, where each transaction is sequentially handled by computing and storage nodes in the cluster.

One central trait of the pipelining technique is that it makes use of active storage to maximize throughput of data-intensive applications (e.g., bioinformatic applications) on high-performance clusters. We have showed that the synchronizations among computing and storage nodes can hinder active storage nodes from accelerating data processing operations. Our technique solves the blocking problem that prevents active storage and computing nodes from processing data in parallel. The implementation of pp-mpiBLAST demonstrates that active storage is an effective approach to improving data-intensive applications' overall performance by offloading data processing to storage nodes in clusters.

We implemented the pipelining technique in a bioinformatic application called pp-mpiBLAST, which incorporates mpiBLAST - an open-source parallel BLAST tool. The pipeline in pp-mpiBLAST processes a sequential transaction stream, in which each transaction contains a filtering/formatting task and a mpiBLAST task. The pipelining technique overlaps data filtering/formatting in active storage with parallel BLAST computations in computing nodes. Measurements made from a working implementation and analytic model have been encouraging. The pipeline and active storage not only reduce mpiBLAST's overall execution time by up to 50%, but they also achieved high scalability on large-scale clusters.

There are three directions to pursue future work. First, we intend to explore the extensibility of data-processing modules and operations that are preloaded into active storage. The modules and operations are likely to be application domain specific. Second, we will study the parallelisms among multiple active storage nodes. Third, we intend to investigate whether

similar partitioning and synchronization schemes can be applied to both read-intensive and write-intensive applications.

Chapter 5

HcDD: The Hybrid Combination of Disk Drives in Active Storage Systems

5.1 Motivation

Previous two chapters propose the architecture of a new active storage system and the pipeline parallelism processing pattern for applying the active storage system in a computing cluster. The new architecture helps in achieving the performance improvement by offloading the computation to the active storage, especially when read requests are performed on storage nodes. In this chapter, hybrid storage devices in an active storage system are introduced to boost write performance and to extend disk reliability.

Recently, the use of NAND-flash-based Solid State Devices (hereinafter referred to as SSDs) has evolved from specialized applications in mobile devices [18] and laptops to primary system storages in general-purpose computers and even data centers [72]. Evidence shows that *Tape is dead; disk is tape; flash is disk* [39]. Different from HDDs (magnetic hard disk drives) which rely on mechanical moving parts, SSDs are completely built on semiconductor chips resulting in high random access performance and lower power consumption. And the cost of commodity NAND flash – often cited as the primary barrier to SSD deployment [73] – has dropped significantly, increasing the possibility for dynastic changes in the storage arena. Although, today, flash disks have not dominated the whole storage market, SSDs begin to be deployed in enterprise servers and data centers. But, in data centers, engineerings still hesitate somehow to perform a large-scale deployment of SSDs due to their poor reliability and the limited lifespan issues [8].

As the cost of NAND flash has declined with increased density, the number of erase cycles that a flash cell can tolerate, on which the number of write operations depends (because of the erase-before-write characteristic), has suffered. Meanwhile, server applications, such

as OLTP (Online Transaction Processing) [44], normally demand a high-performance and highly reliable storage system. Some researchers believe that the stressful workload and fewer available erase cycles may further reduce lifetimes of SSDs, in some cases, even to less than one year [93]. From another perspective, when building terascale or petascale servers and data centers using only SSDs rather than HDDs, the cost – even though with the decreasing price of SSDs – is often beyond the acceptable budgets in most cases. Thus, HDDs are still regarded as indispensable components in the storage hierarchy because of their merits of low cost, huge capacity, above-average reliability, and fast sequential access speed. Instead of simply replacing HDDs with SSDs, researchers [49][68][51][80][33] believe that finding a complementary approach balancing the performance, reliability and cost is more attractive and challenging.

In this chapter, we propose a hybrid combination of disk arrays for active storage systems. The hybrid-disk design involves an enhanced duo-buffer design, which consists (1) a HDD-write buffer to server and de-duplicate write requests and an on-SSD buffer to provide parallel write processing. Read requests are all served by SSDs directly. But writes to an active storage are first transferred to its HDD buffer and perform de-duplication before migrating to the SSD. The de-duplication computation and I/O operations from the buffer disk to the major storage disk are offloaded to the dedicated processor in the active storage. This chapter also introduces an enhanced version of SSD's buffer, which supports internal parallelism processing. Together, the goal of this part of study is to minimize the number of writes sent to SSDs without significantly affecting the performance; by doing so, it reduces the number of erase cycles and thus extends SSDs lifetime.

We have made the following contributions in this chapter:

- We design and simulate a hybrid combination of storage devices for active storage systems.
- A HDD in our design is first assigned as a write buffer to SSDs, thereby supporting deduplication service.

- We design and simulate the internal-parallelism supporting cache on SSDs.
- A system simulator is built to evaluate the hybrid drives combination in active storage systems.

The rest of this chapter is organized as follows. Section 5.2 briefly introduces the background and related work. Then, the design and the implementation details are described in Section 5.3. It is followed by experimental results and the performance evaluation in Section 5.4. Last but no the least, Section 5.5 is the conclusion of this chapter.

5.2 Background

5.2.1 SSD and Hybrid Storage

NAND-flash-memory based SSDs, which used to be evolved in mobile devices and laptops, play a critical role in revolutionizing storage systems [21][52][54][56][67][81]. SSDs are completely built on semiconductor chips without any moving parts, giving rise to high random access performance and lower power consumption. And the cost of commodity NAND flash – often cited as the primary barrier to SSD deployment [73] – has dropped significantly, increasing the possibility for dynastic changes in the storage arena. In order to improve the storage performance, the San Diego Supercomputer Center (SDSC) has built a large flash-based cluster called Gordon, which adopts 256TB of flash memory [15]. This research project is backed by a $20 million grant from the U.S. National Science Foundation.

Different from the research case, however, enterprise data centers still are unable to adopt a large-scale deployment of SSDs due to high cost, poor reliability and limited lifespan of SSDs. In the perspective of the limited lifespan, the challenge is each block on flash-based storage media has limited erase times and each block has to be erased before being written. Thus erase-before-write operations not only degrade SSD performance, but also shorten SSD lifespan. Table 5.1 shows the lifespan of a single-level cell and a multiple-level cell NAND memory in terms of P/E (Program/Erase) cycles. Based on the SSD lifetime calculator

provided by the Virident website [40], the lifetime of a 200GB MLC-based SSD could be only 160 days if the write rate performing on the SSD is 50MB/s.

Thus, traditional magnetic hard disks are still regarded as indispensable in the storage hierarchy because of their merits of low cost, huge capacity, above-average reliability, and fast sequential access speed. Instead of simply replacing HDDs with SSDs, researchers [49][68][51][80][33][95][102]believe that it is attractive and chanllenging to investigate a cogent and complementary approach balancing the performance, reliability and cost.

Table 5.1: NAND Flash Memory Lifetime (P/E Cycles)

Generation	**SLC**	**MLC**	**eMLC**
5X	100,000	10,000	N/A
3X	100,000	5,000	35,000
2X	100,000	2,500	N/A

5.2.2 Internal Parallelism Processing on SSD

The inter-disk parallelism technique has been well explored since decades ago. Data striping is the basic idea of inter-disk parallelism. However, even for Hard Drives, inter-disk parallelism just start to be considered recently [88]. Since (1) there is no mechanical movements in SSDs and (2) an SSD has one or multiple identical elements which can work in parallel [4], we have better chance to improve internal parallelisms in SSDs than Hard drives. Park points out that intra-SSD parallelism is possible on die-level, package-level, and plane-level. Furthermore, parallelism-aware request processing is an effective solution to enhance intra-SSD parallelisms [100]. Chen and Zhang analyzed the essential roles of exploiting internal parallelism SSDs in high-speed data processing [19].

Unlike Hard Drives, SSDs have a Flash Translation Layer (or FTL) implemented to emulate a hard disk drive by exposing an array of logical block addresses (LBAs) to the host. FTL averagely spreads erase workloads on flash-based storage. A ill-designed FTL algorithm not only reduces the SSDs performance, but also wears out SSDs storage units rapidly. Chen and Zhang proposed CAFTL – a content-aware flash translation layer enhancing the lifespan

of flash memory based SSDs [20]. CAFTL removes unnecessary duplicate writes to reduce write traffic to flash memory. Gupta and Urgaonkar proposed the Demand-based Flash Translation Layer (DFTL), which selectively caches page-level address mappings [41] . A journal Remapping Algorithm JFTL was presented by Choi and Park [23]. JFTL writes all the data to a new region in an out-of-place update process by using an address mapping method [23] .

Reliability and performance are two major research challenges of Solid State Drives. Most current research attempts to design new Flash Translation Layer algorithms to improve reliability and enhance performance by utilizing the built-on-board cache. Hence, many research projects have been focused on using cache as write buffers. Kang applied Non-Volatile RAM (RAM) as write buffer for SSDs to improve overall performance [53]. Kim and Ahn proposed a buffer management scheme called BPLRU for improving random writes in flash storage. BPLRU buffers writes improve performance of random writes [57]. Soundararajan and Prabhakaran designed Griffin, a hybrid storage device, to buffer large sequential writes in Hard Drives [94]. Park and Jung also studied write buffer-aware address mapping for flash memory devices [77].

5.2.3 Data Duplication

Data deduplication is not a new problem; it is a specialized data compression technique for eliminating coarse-grained redundant data, thereby improving improve storage utilization in backup/archival systems [12][31][76][10][74][29][99] . Deduplication techniques are able to reduce the required storage capacity; duplicate data are deleted, leaving only one copy of the data to be stored, along with references to the unique copy of data. Different applications and data types naturally have different levels of data redundancies. Data-backup applications generally benefit the most from de-duplication due to the nature of repeated full backups of an existing file system.

In order to show data duplication is common, researchers from Ohio State University studied 15 disks installed on 5 machines, in which three kinds of file systems, (*i.e.,* Ext2, Ext3, and NTFS) can be found [20]. In terms of four different environments, the results show that the duplication rate ranges from 7.9% to 85.9% across the 15 disks. The results show that a well-designed approach to removing duplicate data is able to substantially extend the available storage space. When we consider the characteristics of NAND flash memory, such as no in-place overwrite and limited erase cycles, deduplication can further extend the lifespan of the flash memory.

Traditionally, the deduplication can be performed either "in-line" (*i.e.,* as data is flowing) or "post-process" (*i.e.,* after data have been written) [27]. Each of them cuts both ways. The "post-process" scheme stores new data on the storage media first. Then, a process initiated when disks are idle will analyze and delete duplicated data. There is no need to wait for deduplication computation before storing the data. When it comes to SSDs, the potential drawbacks include unnecessary storage space and write operations for duplicate data. As we described earlier, duplicated data waste the expensive memory space and also may reduce the lifespan of SSDs. On the other hand, the "in-line' scheme is the process where the deduplication calculations are handled in a real-time manner on the target device as the data enters the device. If a deduplicate engine spots a block that it has been already stored on the media, the engine just maps to the existing block without storing the new block. The benefit is that it requires less storage as data is not duplicated. On the negative side, however, "in-line" scheme is frequently suffered from the computation overhead, which reduces the data throughput of the device.

In this chapter, we propose the use of hybrid disk systems to provide a deduplication service in active storages. The deduplication service in this study is also hybrid – a combination of both "in-line" and "post-process" styles. The design details can be found in Section 5.3. The benefits of our design are: (1) in the perspective of a host machine, data

stores to an HDD-buffer in an active storage system directly without waiting for the deduplication service to complete, (2) the deduplication computation is offloaded to a dedicated processor in the active storage system, and (3) after the deduplication procedure, the unique data (i.e., deduplicated data) are written to the SSD in the active storage system.

5.3 The Design of HcDD – a Hybrid Combination of Disk Devices

5.3.1 System Architecture

In order to (1) extend the lifespan and improve the reliability of SSDs, (2) save the storage space and (3) boost the write performance of SSDs, we design two techniques (*i.e.,* two buffers) – hybrid combination of disk drives and enhanced on-SSD buffer with internal parallelism. As mentioned in previous sections, the benefits of the proposed schemes are:

- The elimination of duplicate writes and redundant data to SSDs via a hybrid deduplication mechanism combing both "in-line" and "post-process" approaches.
- The offloaded computation power for the deduplication calculation from host machines.
- The fast internal data transfer speed by parallel processing via an enhanced on-board buffer of SSDs.

In this section, we describe the design of a simulated hybrid disk system (hereinafter referred as HcDD), which stands for the hybrid combination of disk drives in an active storage system. The reasons that we conducted this hybrid disks research using the simulation are: (1) the modification of both the disk controller and the FTL module of SSDs are impossible without the support from hardware vendors and (2) it is important to evaluate the performance of an emerging architecture before its implementation.

Below are descriptions of modules that an active storage system consists.

- **Disk Drive.** The disk drive is a major component in any storage system, where data are permanently stored.

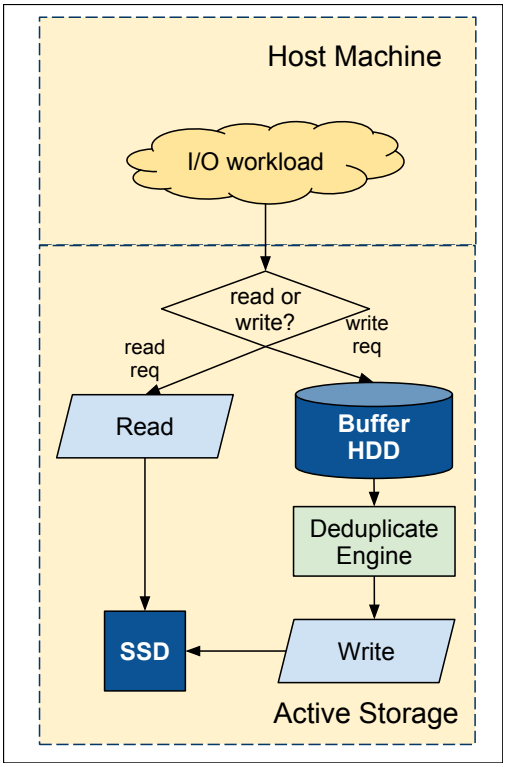


Figure 5.1: The System Configuration of HcDD – a hybrid active storage system.

- **Controller.** The controller is a processing unit (i.e. data management unit) for disks in a storage system. The controller communicates between disks and host machines, manages disk drives, and distributes data among disks.

- **Deduplication Engine.** It is in charge of computing fingerprints for incoming requests, looking the requests up in a fingerprint table, and deciding whether requests should be written to SSDs.

Figure 5.1 depicts the architecture of a HcDD in an active storage system. As we mentioned in previous chapters (see Chapter 3.3.1), the active node can be connected with a host machine or a cluster by either network connections or I/O connections. The active storage system has its own computation facilities (*i.e.,* a dedicated processor), the memory and storage facilities, including a disk controller, a deduplication engine as well as some disk drives. And there is a hybrid combination of two kinds of storage devices – SSDs serving

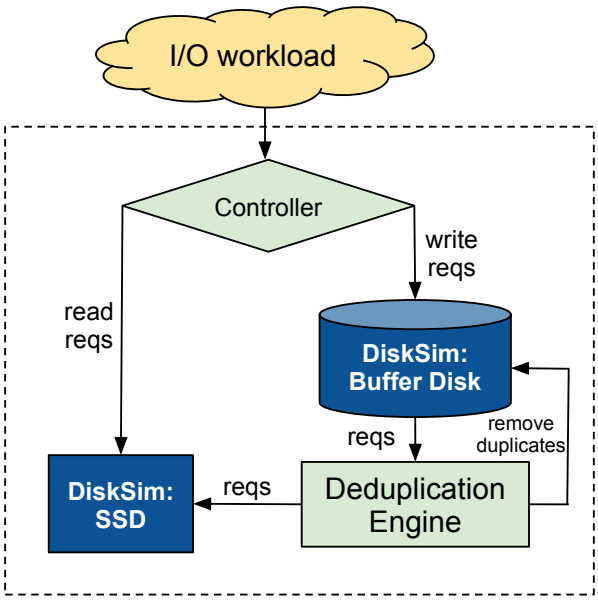


Figure 5.2: The System Architecture of HcDD. There are three modules: a controller, a deduplication engine, and some storage devices.

as main data disks, and HDDs serving as the write buffers – in each HcDD. Further, we enhance on-SSD buffer to support of internal parallelism processing.

5.3.2 Hybrid Combination of Storage Drives

Figure 5.2 is the configuration of the simulated hybrid combination of storage drives. The HcDD system mainly contains three types of modules: a controller, a deduplication engine, and storage devices. The controller module is in charge of managing and distributing input I/O requests. The duplication engine compares the fingerprint of each write request with existing ones in a fingerprint table. To simulate the storage system, we integrated DiskSim in our system. DiskSim, developed by Ganger *et al* [37][36], is a well-known disk simulator. DiskSim has been validated against several disk drives using the the published disk specifications and I/O workload traces. Since DiskSim only support the storage-level simulation, the developers also provide programming interfaces to integrate DiskSim with any system-level simulator.

The proposed hybrid storage model integrates both cost-effective HDDs and high-speed SSDs as a hybrid combination storage component in the active storage system. The controller handles data distribution, which avoid undesirable significant changes to existing file systems and applications. The controller directs write requests to a buffer disk (*i.e.,* HDD) and sends the read requests to the SSDs. Thus, in terms of read operations, the overhead caused by the controller can be ignored; there is no performance interference since the controller simply redirects requests without any computation overhead. Once a write request is issued by a host machine, the data will be written on the buffer disk of the active storage node. As soon as the data is buffed on the HDD, the data will be processed by the dedicated deduplication engine, which calculates the hash value and looks the data up in the hash table – before sending the data to the next step. Then, the deduplicated data will be written to SSDs. The duplicates are mapped to existing ones and removed from the buffer disk. We consider the deduplication process hybrid in nature because of the following reasons. From the perspective of the host machine, all the request are handled "in-line"; meaning that there is no calculation workload and thus no waiting time required. Meanwhile, within the active node, the deduplication is handled more similar to the "post-process" pattern, which stores new data on the storage media and then the deduplication engine will analyze the data looking for redundancy as soon as possible.

The workflow of deduplication engine is introduced in Figure 5.3. When a write request of the input workload trace is received at the buffer disk, the processing of deduplication can be described in four steps:

1. Before data are written to the buffer disk, the incoming request triggers the deduplication engine in the active storage node.

2. Each updated page in the buffer is computed a hash value (*i.e.* hash fingerprint) by the dedicated processor of the active storage node.

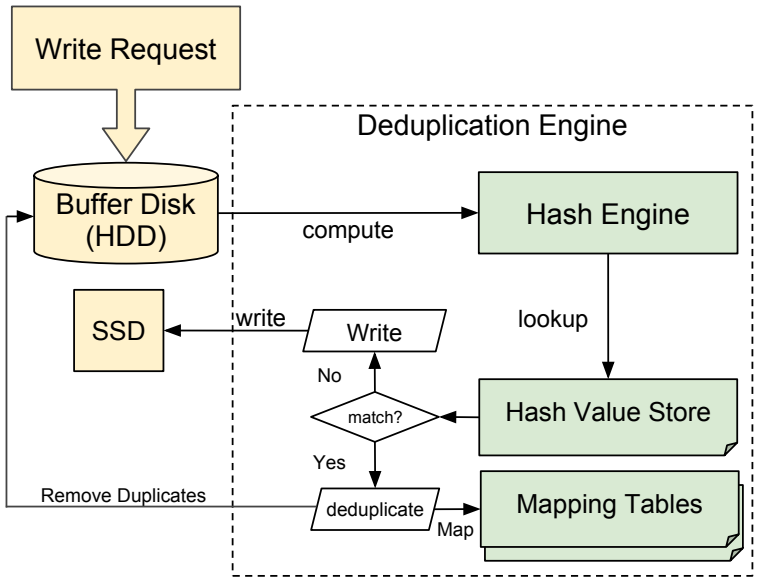


Figure 5.3: The Workflow of Hybrid Active Storage System. There are four steps: write to the buffer disk, calculate a hash value, compare the value against a value store, and write to the SSD or map to the table.

3. Then each hash value is looked up against a hash table, which maintains the fingerprints of data already stored in the SSD.

4. (a) If the hash value is fresh (*i.e.,* it is not found in the table) the write is performed as a regular operation in the SSD .

   (b) Otherwise, if the fingerprint is found, the mapping tables are updated by mapping the duplicate request to the physical location of the residing data. Then the write operation to flash is canceled. The goal of this step is to minimize the number of writes sent to the SSD without significantly impacting the performance; in doing so, HcDD reduces the number of erase cycles, which is the performance bottleneck to the NAND flash. In addition to improved perfrmance, extending SSDs' lifetime is a second benefit gained from the deduplication service.

5.3.3 Intra-parallelism buffer on SSD

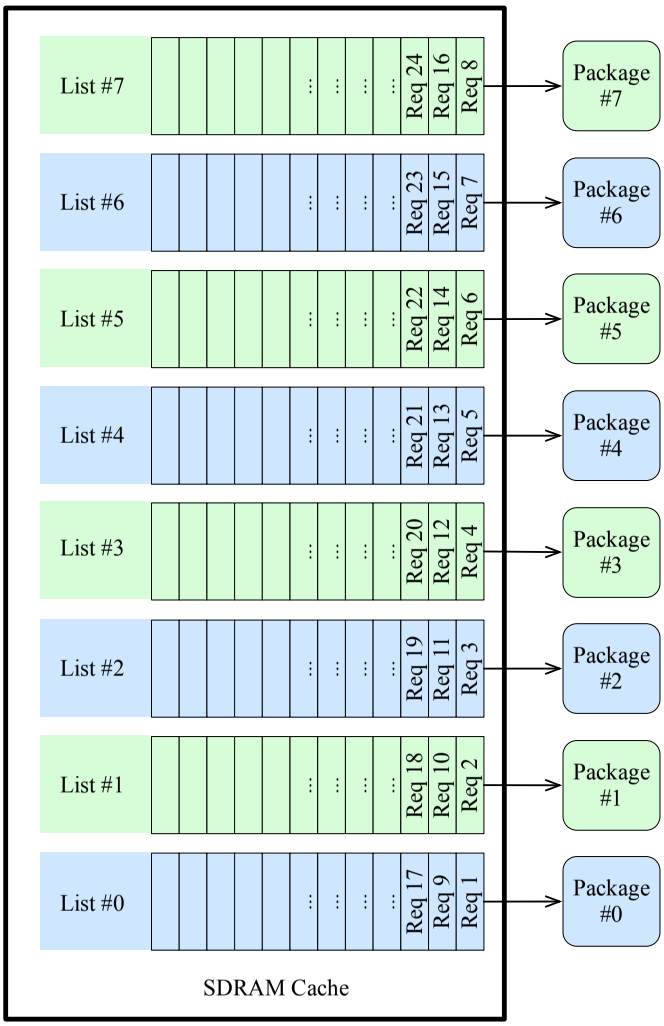


Figure 5.4: The Workflow of Hybrid Active Storage System.

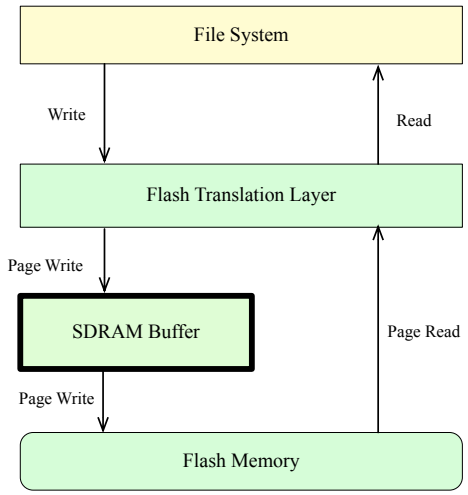


Figure 5.5: The Workflow of On-board Buffer.

An SSD is built by arrays of NAND flash memory, which is a semiconductor storage module [4][13]. Agrawal's [4] describes that an SSD has one or multiple identical elements (*i.e.* packages) and all of them can work in parallel. The challenge is which part of the multiple elements should be in charge of handling data distribution and parallel processing.

Unlike traditional hard drives, the flash-based storage has a Flash Translation Layer (FTL) which maps Logic Block Number (LBN) to Physical Block Number (PBN). Also, a remapping algorithm is designed in FTL for wear leveling. Thus, the block number in file system level is not the block number in flash memory level. Based on different remapping algorithms, the same LBN may lead to different PBN at different time periods. Hence, a buffer must be designed in the lower level of FTL to keep data consistency. Figure 5.4 presents our new architecture in a flash memory. We chose to use Synchronous Dynamic Random Access Memory (or SDRAM) as an on-board buffer for its high performance. The on-board buffer is the lower-level buffer in our "duo-buffer" design. Since the performance of random writes, especially re-writes, is the Achilles' heel in flash memory, the buffer is designed for buffering only write requests.

Figure 5.5 presents the software structure of an enhanced SSD with SDRAM buffer. In the buffer, the number of lists is the same as that of packages. Each list contains data for its

corresponding package. Since the buffer is built under FTL, the granularity in the buffer is 8 KB page. The size of pages can be tuned. Recall that all requests buffering in SDRAM are writes. Once the buffer is full, our algorithm will assign the same number of pages to each package in parallel since all packages are able to work independently. Below we presents an algorithm to enhance parallelisms by buffering writes. Even though there are nested loops, the number of parallelism pages and the number of packages are both small and fixed values, the time complexity is still approximately $O(n)$, where n is the number of packages.

Algorithm 1 Enhancing Package-Level Parallelism Algorithm. r represents one request. L_i represents the List stores the request in the buffer for the ith package. P_n represents the nth package.

```
if r_current is a write request then
  find its corresponding package P_i
  for r_j ← all requests in L_i do
    if r_j = r_current then
      remove r_j
    end if
    add r_current to L_i^tail
  end for
  if buffer is full then
    for m ← number of parallelism pages do
      for n ← number of packages do
        issue L_n^head to P_n
      end for
    end for
  end if
end if
```

5.4 Evaluations

The HcDD storage system is implemented and evaluated based on a comprehensive trace-driven simulator. In this section, I present the experiment environment and results from the HcDD simulation studies under a variety of configurations.

5.4.1 The Evaluation Environment

Our HcDD is a simulator emulates the behaviors of a hybrid active storage systems. There are two types of disk drive modules in the simulated system. The HDD module, a 15,000 RPM Seagate Cheetah 15.5K SAS hard disk drive, is provided by DiskSim 4.0 [36], which emulates a hierarchy of storage components including buses, controllers, and disks. The enhanced SSD module supporting internal-parallelism processing is implemented in a sophisticated SSD simulator, from Microsoft Research SSD extension [85] for the DiskSim simulation environment [36]. Although the Microsoft extension implements the major components of FTL (*i.e.,* indirect mapping, garbage collection and wear-leveling policies), it does not have a on-board buffer, which is an essential facility in newly released commodities. Thus, we designed and implemented an enhanced on-board buffer.

In this section, we mainly compare four different storage configurations: (1) HcDD, which consists a deduplication engine and the hybrid combination of both a RAID 0 disk array of two enhanced SSDs and a RAID0 disk array of two HDDs, (2) Hybrid Storage, which stands for traditional hybrid storage with one RAID0 array of two SSDs and one write-buffer RAID0 array of two HDDs (in the following figures, it is referred as Trad.Hybrid), (3) a RAID0 array of two SSDs (hereinafter referred as SSDs), and (4) a RAID0 array of two HDDs (hereinafter referred as HDDs).

We evaluate the HcDD deisgn by running simulations upon three real-world application traces (see Table 5.2): traceKernel, tracePhoenix, and Finanical2 [61]. "traceKernel' was collected while a target machine compiling the Linux kernel. "tracePhoenix" was collected while the target machine running a MapReduce application, WordCount, provided by the Phoenix MapReduce System [83]. "Financial" is from OLTP (Online Transaction Processing) applications running at a large financial institution provided by the laboratory for Advanced System Software of University of Massachusetts Amherst. "traceKernel" and "tracePhoenix" are collected on a HP ProLiant ML110 G6 workstation with an Intel Xeon X3430 processor, a 2GB main memory, and a 500GB 7,200 RPM Seagate Barracuda hard

disk drive. The operating system is Ubuntu 10.10 with the Ext3 file system. The logical address of all traces were evenly shrunk so that each request's address can be mapped to a physical address within the scope of the SSD configuration (32GB in this study). Table 5.2 presents the statistics of three traces.

Table 5.2: Num of Read/Write Requests

Trace	**Read Requests**	**Write Requests**
traceKernel	84,847	70,243
tracePhoenix	822,909	155,413
Financial	2,252,549	480,127

5.4.2 Internal Parallelism Supported Buffer for SSD

An SSD has one or multiple identical elements (*i.e.* packages) and all of them can work in parallel. In this section, we compare our internal-parallel SSD algorithm against the classic LRU cache management algorithm. The performance impact of parallelism levels and buffer size are presented and discussed as follows.

In the first test, two general purpose traces provided by the Microsoft Research SSD extension [85], iozone and postmark, are evaluated. The size of write buffer scales from 1 MB to 64 MB. From Figure 5.6, we observe that when the buffer size is small (1 MB or 2MB), internal-parallelism scheme surpasses the LRU one. However, when as the buffer size increases, the average response time of the internal-parallelism scheme is either similar to the competitor or worse. Thus, in Section 5.4.4, we choose a 1 MB buffer in the HcDD simulator.

In the second test, we use the random write trace. We do not provide the comparison of reads, because the internal-parallelism does not affect read operations in SSDs. There are two observations from Figure 5.7: (1) when the size of random write request is small (*e.g.,* 250 KB), the performance of both the internal-parallelism and LRU are similar, and (2) when the request size is 5 MB, the response time is reduced by the virtue of the internal-parallelism scheme compared with the LRU one.

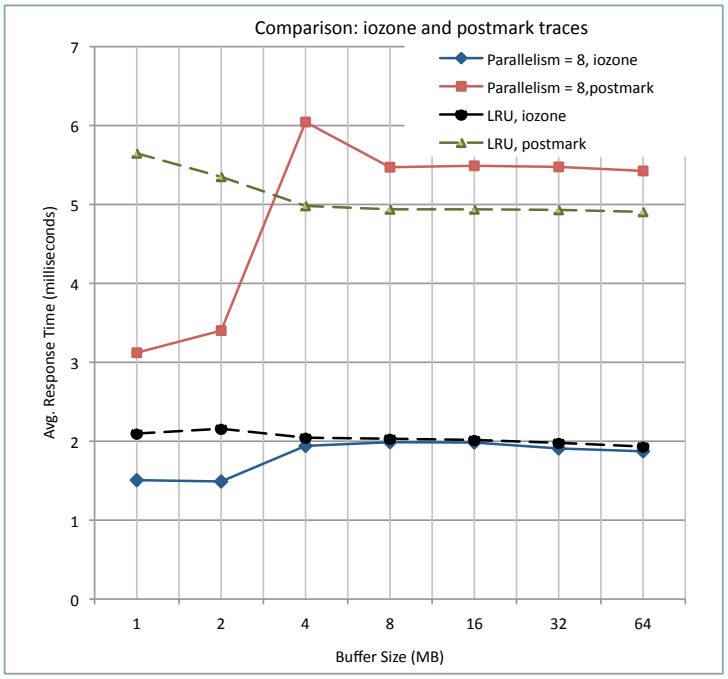


Figure 5.6: Performance Comparison of Iozone and Postmark

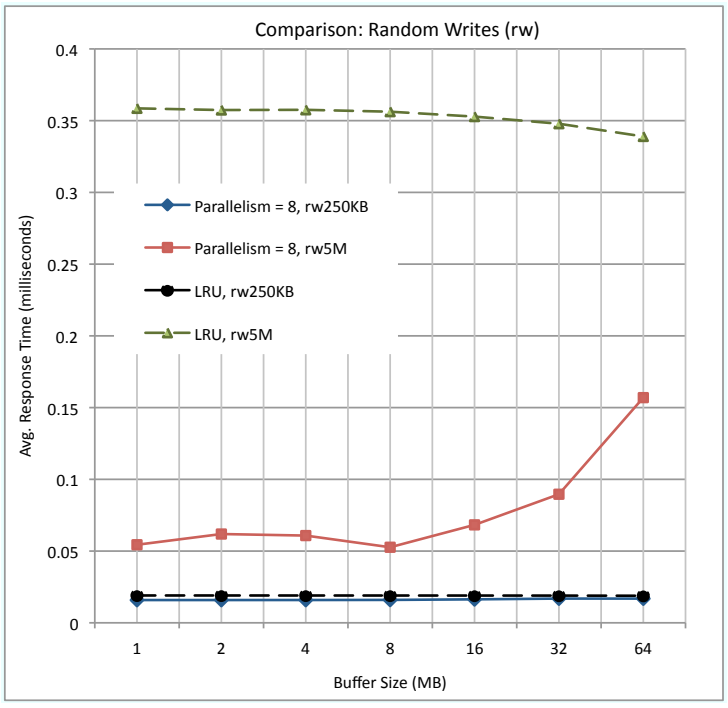


Figure 5.7: Performance Comparison of Random Write.

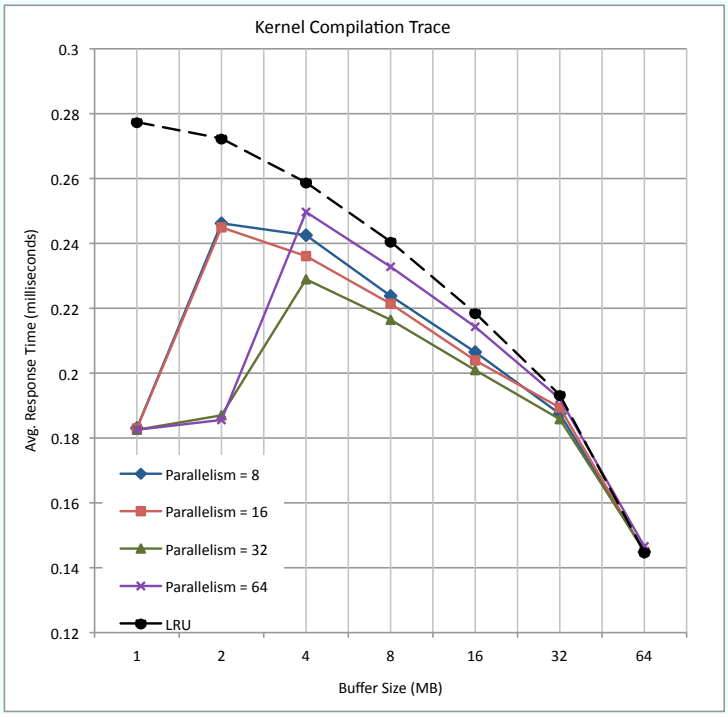


Figure 5.8: Performance Comparison of traceKernel.

Besides the comparison between the internal-parallelism algorithm and the LRU scheme, we evaluate the performance impact of interleaving levels (*a.k.a,* parallelism levels) in addition. Running two traces collected from real-world applications – traceKernel and tracePhoenix, we scale the parallelism level in the buffer from 8 to 64 and the size of write buffer from 1 MB to 64 MB. The result shows that (1) the internal-parallelism algorithm outperforms LRU in most cases, (2) parallelism level 8 is slightly better than other three levels. The optimal parallelism level depends on how many packages (elements) on a board, bus bandwidth, data access patterns, and the like.

Overall, the internal-parallelism algorithm improves performance in most cases, especially in the random writes case (with no impact on read-only workload), compared with the traditional LRU. As the on-board buffer becomes an essential component for the SSD, our solution provides an approach to fully utilizing the buffer and boosting the performance of the SSD.

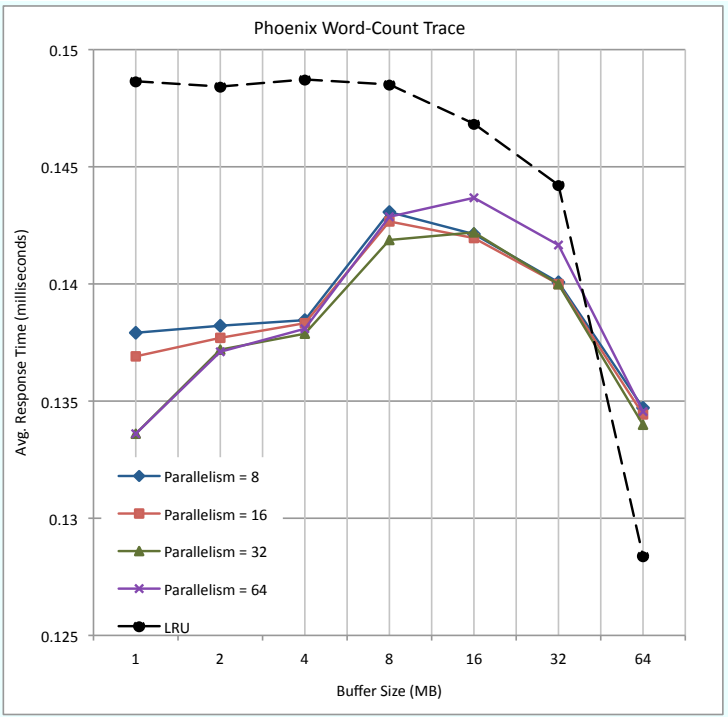


Figure 5.9: Performance Comparison of tracePhoenix..

5.4.3 Deduplication

We simulate the deduplication process within the HcDD storage system. The overhead of deduplication are derived from a real-implement statistic from a previous research [20]. Chen *et al.* run the deduplication function on the SimpleScalar-ARM simulator [65] to extract the total number of cycles for executing the function. The result for running SHA-1 hashing function to process a 4KB page is 47,548 cycles. Thus, we generated the latency by dividing the number of cycles by the processor frequency. In the HcDD simulator, we assign a low-level 1GHz dedicated processor for an active storage node. The overhead can be further minimize since even higher frequency processors are normal in the active storage node.

HcDD provides the service of removing duplicated writes to SSDs. Let us denote num_{total} and $num_{deduplicated}$ as the total number of requested writes and the number of

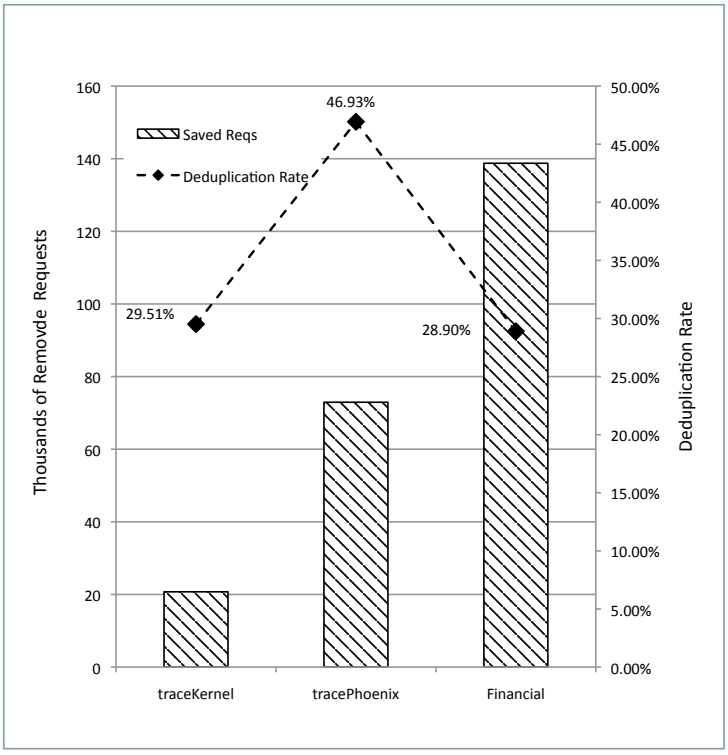


Figure 5.10: The number of removed requests and the deduplication ratio.

deduplicated writes actually written to a SSD, respectively. We have

$$DeduplicationRate = \frac{num_{total} - num_{deduplicated}}{num_{total}}. \tag{5.1}$$

Figure 5.10 shows the results of deduplication ratio and the number of "removed" requests. We observe that the deduplication rates of traceKernel, tracePhoenix and Financial are 29.15%, 46.93% and 28.9%, respectively. The number of removed write requests to the flash memory are 20,731, 72,933, and 138,748, respectively. The decrease of Program/Erase cycles can extend flash-memory's lifespan.

5.4.4 System Performance Evaluation

In this section, we evaluate the overall system performance of HcDD. The goal is to compare HcDD with three traditional storage architectures, *Trad.Hybrid*, *SSDs*, and *HDDs*.

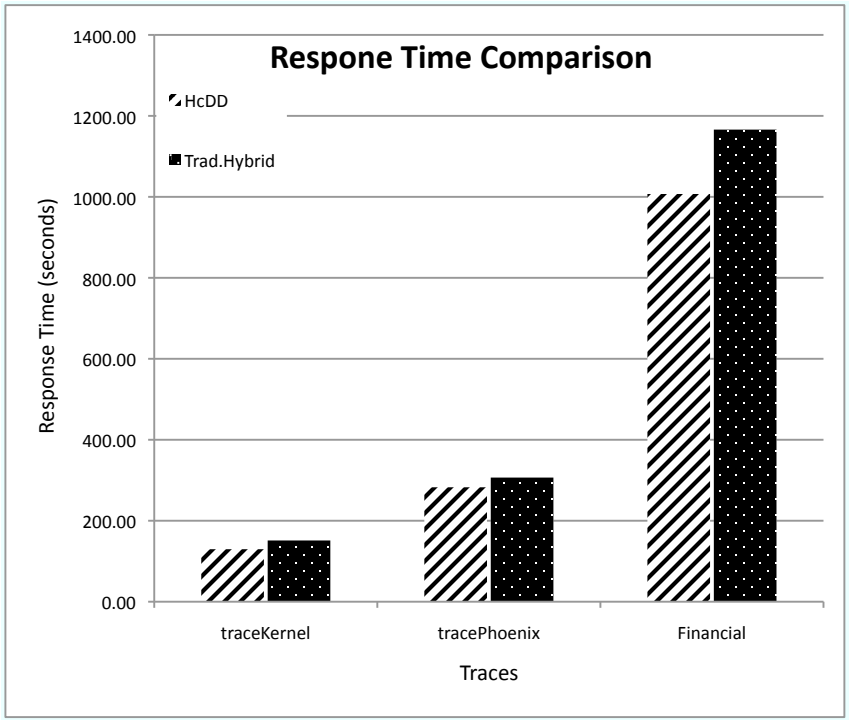


Figure 5.11: Processing Time Comparison between HcDD and traditional hybrid storage.

Response time (includes average response time) and number of write requests written to the SSD are two major metrics in this set of experiments to measure the speed and reliability, respectively. Before testing the system, we have two expectations: (1) In terms of response time, the performance of HcDD should be better than those of *Trad.Hybrid* and *HDDs* schemes but close to that of *SSDs*, and (2) in terms of reliability (we measure it using numbers of requests written on the flash memory in this study), HcDD should be better than *SSDs* and *Trad.Hybrid*. HcDD is not a "speed-oriented" design, because it targets at enhancing the reliability of SSDs and while providing short response times.

When compared with *Trad.Hybrid* (see Figure 5.11 and Figure 5.12), the HcDD reduces in both response time (averagely saved 12% of response time) and average response time (averagely saved 13% of average response time). And as mentioned in the previous section, there are 29.15%, 46.93% and 28.9% request respectively removed from being issued to the SSD. Even though the deduplication process in HcDD brings in computation overhead, the

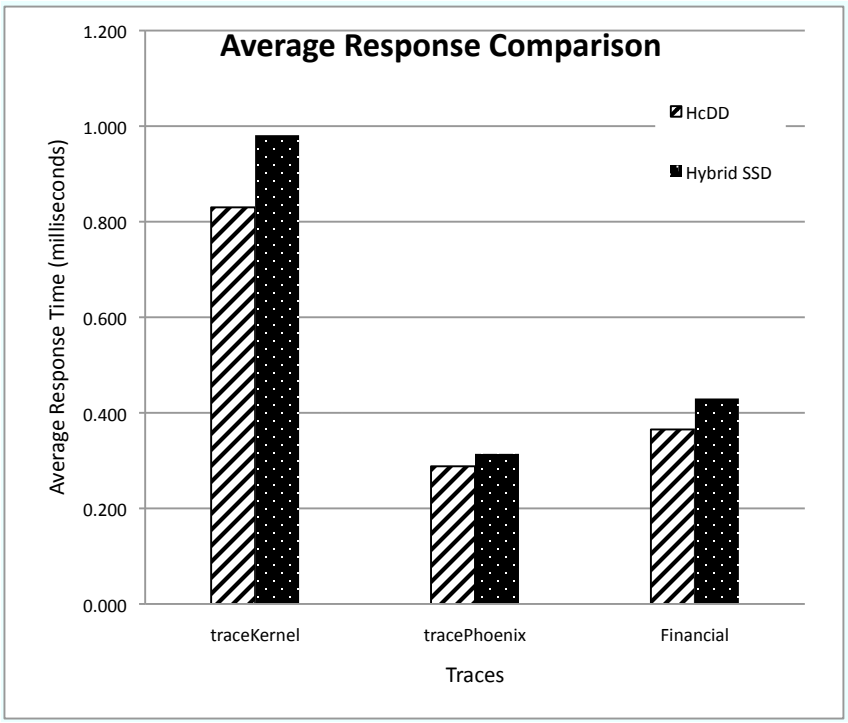


Figure 5.12: Average Response Time Comparison between HcDD and traditional hybrid storage.

performance of HcDD is still better because of fewer write requests forwarded to the SSD and the internal parallel data transferring on the enhanced SSDs.

When comparing HcDD with the *SSDs* scheme, we observe intriguing results. When it comes to traceKernel and tracePhoenix (see Figure 5.13), the response time of HcDD takes 54 seconds and 123 seconds longer than the *SSDs* approach. But when it comes to the Financial trace, HcDD outgoes the *SSDs* scheme by 50 seconds. This improvement depends on the deduplicated number of writes, because the poor random write performance of the flash memory can largely degrade the overall performance of data-intensive applications. We do not show the total response time of the *HDDs* scheme, because HDDs' performance is much worse than that of HcDD (*i.e.,* almost 3 times faster). Figure 5.14 presents the same trend in terms of average response time of HcDD and the two competitors.

In summary, based on the evaluation results, we can see that HcDD fulfills the previous expectations. It not only responds I/O requests with a very competitive speed (better

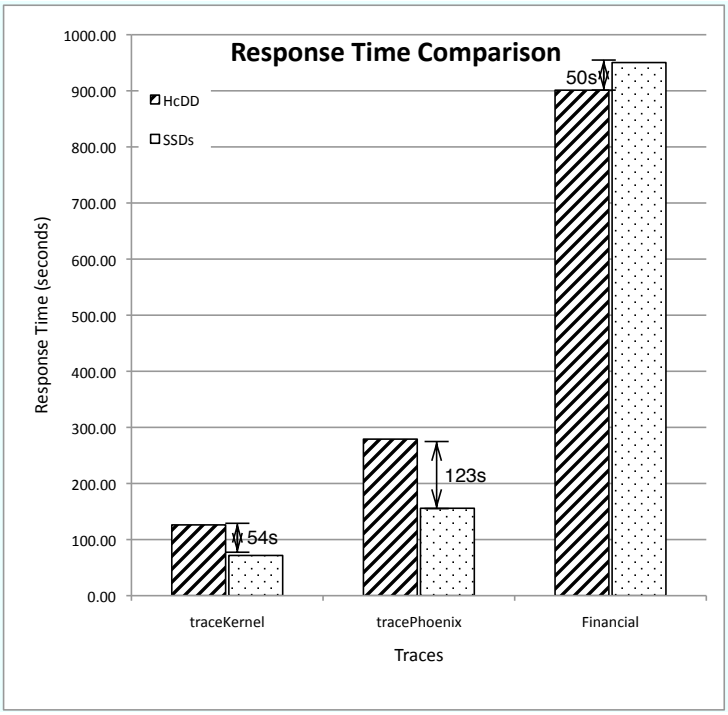


Figure 5.13: Response Time Comparison between HcDD and SSD.

than the traditional hybrid disk), but it also removes duplicate writes, which normally occupies 30% to 46% of the total requests, to the SSD. It saves the storage space and extend the lifespan. Thus, HcDD is a promising hybrid storage model, which balances the performance, saves the storage space and extends the lifespan of the SSD, for data-intensive server applications.

5.5 Summary

The use of NAND-flash-based Solid State Devices (hereinafter referred as SSDs) has evolved in primary storage systems in enterprise servers and data centers. SSDs are completely built on semiconductor chips without any moving parts resulting in high random access performance and lower power consumption. However, many engineers still hesitate to perform a large-scale deployment of SSDs due to their poor reliability and limited lifespans.

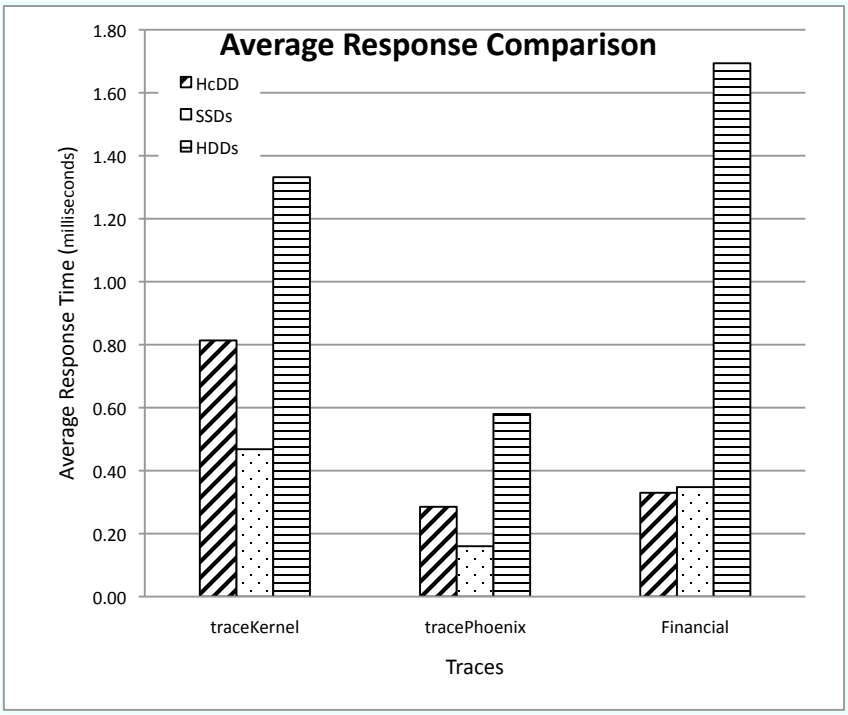


Figure 5.14: Average Response Time Comparison between HcDD and SSD.

In this chapter, we propose a hybrid combination of disk arrays designed for active storage system, useing hard disk drives as a write buffer to cache write requests and de-duplicate the redundant write requests to the SSD. Read requests are all served from SSDs directly. Unlike reads, writes to the active storage are first processed by HDDs and performed de-duplications before migrating to the SSDs. The de-duplication computation and I/O operations are offloaded to a dedicated processor in the active storage system. This chapter also introduces an enhanced FTL algorithm for SSD's buffer, which supports internal parallelism processing. The proposed "duo-buffer" architecture minimizes the number of writes issued to SSDs without significantly impacting the I/O performance, thereby extending the lifespan of flash memory. The new architecture also utilizes the parallel on-board packages to boost the write performance of SSDs.

In order to evaluate the proposed architecture, we design and implement a trace-driven storage system simulator for the hybrid active storage system. We compare our internal-parallelism algorithm with the classic LRU scheme in terms of single-SSD drive performance.

The results shows that our proposed algorithm outgoes LRU in most cases, when the interleaving level is 8 and buffer size is 1 MB or 2 MB. Then, we compare the overall performance of HcDD with the other three storage configurations, including the traditional hybrid storage scheme, the SSDs disk array and the HDDs disk array. The evaluation results are generated using several traces collected from daily usages and some applications in academic and financial areas. In terms of the system response time, HcDD beats the traditional hybrid disk array in all three tests and even performs slightly better than the SSD disk array in the data-intensive trace test. Based on the evaluation, we can observe that HcDD is a promising configuration for data-intensive applications because it (1) utilizes the parallel processing capability of on-disk buffer, and (2) balances the system I/O performance and the SSD lifespan.

Chapter 6

Conclusion and Future Work

In this dissertation, we have designed an active storage architecture, studied a pipeline-parallel processing pattern to apply the active storage node in the cluster system, and explored a new disk configuration, the hybrid combination of disk devices, of the active storage system. This chapter concludes the dissertation by summarizing the contributions and describing future directions.

6.1 McSD: Multicore-Enabled Smart Storage for Clusters

Processor-enabled smart storage can improve I/O performance of data-intensive applications by processing data directly using storage nodes, because smart nodes avoid moving data back and forth between storage and host computing nodes. Thanks to the escalating manufacturing technology, it is possible to integrate multi-core processors into smart storage nodes. In this study, we implemented a prototype called McSD for multicore-enabled smart storage that can improve performance of data-intensive applications by offloading data processing to multicore processors employed in storage nodes of computing clusters.

Our McSD system differs from conventional smart/active storage in two ways. First, McSD is a smart storage nodes rather than a smart disk. Second, McSD can leverage multi-core processors in storage nodes to improve performance of data-intensive applications running on clusters. The four major contributions of this study are:

- A prototype of next-generation multicore-enabled smart data storage.
- A programming framework, which include MapReduce-like programming APIs and a runtime environment for multicore-based smart storage in the context of clusters.

- Development of three benchmark applications to test McSD for clusters.
- Single-application experimental results and multiple-application performance evaluation.

McSD along with its programming framework enables programmers to write MapReduce-like code that can be automatically offload data-intensive computation to multicore processors residing in smart storage nodes. The McSD programming framework allows smart storage nodes to take full advantages from embedded multi-core processors. The APIs and a runtime environment in this programming framework automatically handles computation offload, data partitioning, and load balancing. The McSD prototype was implemented in a testbed–a 5-node cluster containing both host computing nodes and McSD smart-storage nodes. Our experimental results were taken by running three real-world applications on the testbed. The tested data-intensive applications include Word Count, String Matching, and Matrix Multiplication. Our multicore-enabled smart storage system - McSD - significantly reduces the execution time of the three applications. Overall, we conclude that McSD is a promising approach to improving I/O performance of data-intensive applications.

6.2 Using Active Storage to Improve the Bioinformatics Application Performance: A Case Study

We have presented a pipelining technique that allows computing nodes and active storage nodes in a cluster to perform CPU-intensive and data-intensive computing in parallel. The technique exploits parallelism among data processing transactions in a sequential transaction stream, where each transaction is sequentially handled by computing and storage nodes in the cluster.

One central trait of the pipelining technique is that it makes use of active storage to maximize throughput of data-intensive applications (e.g., bioinformatic applications) on high-performance clusters. We have showed that the synchronizations among computing

and storage nodes can hinder active storage nodes from accelerating data processing operations. Our technique solves the blocking problem that prevents active storage and computing nodes from processing data in parallel. The implementation of pp-mpiBLAST demonstrates that active storage is an effective approach to improving data-intensive applications' overall performance by offloading data processing to storage nodes in clusters.

We implemented the pipelining technique in a bioinformatic application called pp-mpiBLAST, which incorporates mpiBLAST - an open-source parallel BLAST tool. The pipeline in pp-mpiBLAST processes a sequential transaction stream, in which each transaction contains a filtering/formatting task and a mpiBLAST task. The pipelining technique overlaps data filtering/formatting in active storage with parallel BLAST computations in computing nodes. Measurements made from a working implementation and analytic model have been encouraging. The pipeline and active storage not only reduce mpiBLAST's overall execution time by up to 50%, but they also achieved high scalability on large-scale clusters.

6.3 HcDD: Hybrid Combination of Disk Drives in Active Storage

The use of NAND-flash-based Solid State Devices (hereinafter referred as SSDs) has evolved in primary system storage in enterprise servers and data centers. SSDs are completely built on semiconductor chips without any moving parts, which results in high random access performance and lower power consumption. But, meanwhile, users still hesitate a little bit to perform a large-scale deployment of SSDs because of their poor reliability and the limited lifespan.

In this chapter, we propose a hybrid combination of disk arrays designed for active storage system, which uses hard disk drives as a write buffer to cache write requests and de-duplicate the redundant write requests to the SSD. Read requests are all served from the SSDs directly. But writes to the active storage are first processed to the HDDs and perform de-duplication before migrating to the SSDs. The de-duplication computation and I/O operations are offloaded to the dedicated processor in active storage. This chapter also

introduces an enhanced FTL algorithm of SSD's buffer, which supports internal parallelism processing. Together, the proposed architecture minimizes the number of writes sent to the SSD without significantly impacting the performance, which can extend the lifespan of flash memory, and utilizes the parallel on-board packages to boost the write performance of the SSD.

In order to evaluate the proposed architecture, we design and implement a trace-driven storage system simulator, the hybrid combination of disk drives (HcDD), for the active storage node. First, we compare our internal-parallelism algorithm with the classic LRU one in terms of single SSD drive performance. The results shows that our proposed algorithm outgoes the classic LRU algorithm in most cases, when the interleaving level is 8 and buffer size is 1 MB or 2 MB. Then, we compare the overall performance of HcDD with the other storage configurations, including traditional hybrid storage scheme, the SSDs disk array and the HDDs disk array. The evaluation results are generated using several traces collected from daily usages and some applications in academic and financial areas. In terms of the system response time, HcDD beats the traditional hybrid disk array in all three tests and even performs slightly better than the SSD disk array in the data-intensive trace test. Based on the evaluation, we can observe that HcDD is a promising configuration for data-intensive applications because it (1) utilizes the parallel processing capability of on-disk buffer, and (2) balances the system I/O performance and the SSD lifespan.

6.4 Future Work

In the course of designing and implementing active and hybrid storage systems, we have found two interesting techniques that may further improve the system in terms of performance and energy-efficiency. This section overviews those two techniques: the memory compression and the parallel processing using wimpy nodes.

6.4.1 Memory Compression

In many areas, the compression technology has been used in many settings to increase the effective size of a storage device or to increase the effective bandwidth. Other researchers have proposed to integrate compression into the memory hierarchy. It has been proved to be a feasible approach to improve the I/O performance of data-intensive applications [96], [16]; the basic idea is to reserve some memory and use this memory region in compressed form instead of holding pages. In general, the in-memory-compression approach creates RAM based block device, which acts as swap region. Pages swapped to a disk are compressed and stored in memory itself. Compressing pages and keeping them in RAM virtually increases its capacity, which result in allowing more data to fit in given limited amount of the main memory. Because of the increase of available memory capacity, accesses to local disk, *i.e.,* the swap partition, are fewer. Thus, in a certain measure, the memory compression balances the performance loss suffered from the data compression cost; the in-memory-compression offers more available memory space at a affordable price.

Existing studies of memory compression only focus on general purpose usage, such as the compcache [1]. Tuduce and Gross [96] mentions that the potential benefits of memory compression depend on three issues: (a) the size of the compressed area, (b) an application's compression ratio, and (c) an application's access pattern. In order to benefit the wimpy machines, we are going to tailor the existing project in terms of (1) the hardware's characteristics and (2) data access pattern.

6.4.2 Wimpy System Board

For decades, the notion of "performance" has been synonymous with "speed. High-performance supercomputers and clusters consume egregious amounts of electrical power and produce so much heat that extravagant cooling facilities must be constructed to ensure proper operation [2]. Recently, along with the reconsideration of the power-consumption, "green" or energy-efficient computing becomes widespread not only in the personal devices

and appliances but also in supercomputers and cluster computers arena. For example, the Green500 list [2], a complement to the Top500 supercomputers [3], was inaugurated on 2008. Its unveiling ushered in a new era where supercomputers can be compared by performance-per-watt.

FAWN [7] is an exploration of building well-matched cluster systems by fully utilizing the low-power efficient embedded CPUs with flash storage. It is claimed to have the potential of achieving high performance and be fundamentally more energy-efficient than conventional architectures for serving massive-scale I/O and data-intensive workloads. Anderson *et al.* proposed the principles of the FAWN architecture and the design and implementation of FAWN-KV (key-value storage system).

FAWN has a few limitations, since the published results only focused on key-value workloads, where near-perfect performance scaleup complements the poor performance of individual wimpy nodes [62]. In our work, inspired by the FAWN, we want to apply the wimpy system boards to general purpose unstructured data processing for some real-world data-intensive applications.

Inspired by researches of using wimpy nodes, the energy-efficient parallel active storage system (PASS) empowers the traditional storage node in the cluster with the parallel wimpy data processing peripherals. Thus, the storage system is capable to process some data-intensive tasks close to where the data stores in an energy-efficiency way.

The proposed architecture of PASS is depicted in Fig 6.1. The hardware testbed contains four types of machines: a client PC, a front node of the cluster, several computation machines, and a storage system. Note that the storage node is made up of a end-node connected to the disk array and the computation module built by some wimpy system boards. We consider the I/O path of the storage device as a programmable computational substrate, which supports general purpose computing at certain points on the path. In the other words, the computation module is considered within the I/O path. For example, in Fig 6.1, the red line indicates a programmable I/O path. There are two levels of parallelism computation:

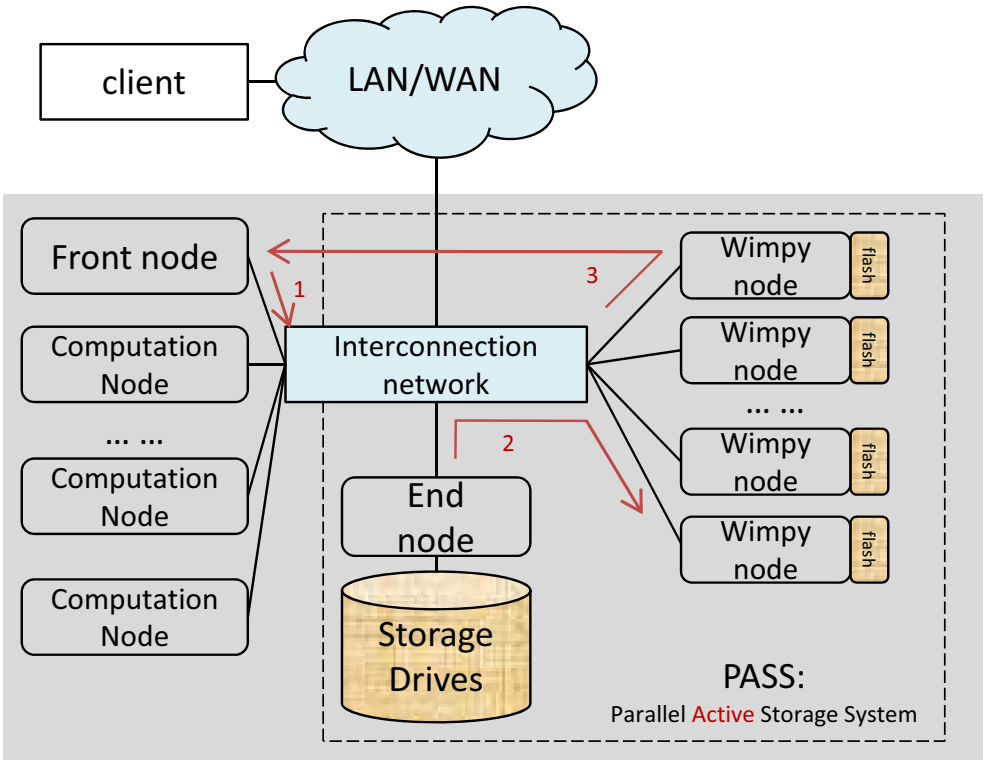


Figure 6.1: The Work Flow of PASS

(1) from the cluster perspective, the tasks are processed in a pipeline approach, and (2) the data is processed in parallel under MPI standards within the computation module of the storage system.

We will develop a benchmark suite in order to evaluate the PASS system. It will include the cluster-level benchmarks, (*e.g.,* the computation and the I/O benchmarks), and real data-intensive applications, (*e.g.,* BioParallel and mpiBLAST). Besides the"speed", the system will be evaluated in terms of new metrics, such as "performance per watt", "energy efficiency for improved reliability", and *etc*.

6.5 Conclusion

This dissertation has presented an active and hybrid storage system designed for offloading data-intensive applications to where data stores. The experimental results reported

in the dissertation have shown that the proposed techniques can deliver promising performance improvements by cooperating active storage system with host machines. We applied our scheme to a real world bioinformatics application as a case study. In addition, the proposed system utilizes the hybrid combination of storage devices in order to achieve a balance between reliability and performance.

Bibliography

[1] Compcache project, 2010. `http://code.google.com/p/compcache/`.

[2] Green500 list, 2010. `http://www.green500.org/`.

[3] Top500 supercomputer sites, 2010. `http://www.top500.org`.

[4] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for ssd performance. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 57–70, Berkeley, CA, USA, 2008. USENIX Association.

[5] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, October 1990.

[6] Khalil Amiri, David Petrou, Gregory R. Ganger, and Garth A. Gibson. Dynamic function placement for data-intensive cluster computing. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '00, pages 25–25, Berkeley, CA, USA, 2000. USENIX Association.

[7] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: a fast array of wimpy nodes. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 1–14, New York, NY, USA, 2009. ACM.

[8] D.G. Andersen and S. Swanson. Rethinking flash in the data center. *Micro, IEEE*, 30(4):52 –54, july-aug. 2010.

[9] Apache. Apache hadoop, 2006. `http://lucene.apache.org/hadoop/`.

[10] A. Arasu, C. Re, and D. Suciu. Large-scale deduplication with constraints using dedupalog. In *Data Engineering, 2009. ICDE '09. IEEE 25th International Conference on*, pages 952 –963, 2009.

[11] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, 2009.

[12] D. Bhagwat, K. Eshghi, D.D.E. Long, and M. Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems, 2009. MASCOTS '09. IEEE International Symposium on*, pages 1 –9, sept. 2009.

[13] Simona Boboila and Peter Desnoyers. Write endurance in flash drives: measurements and analysis. In *Proceedings of the 8th USENIX conference on File and storage technologies*, FAST'10, pages 9–9, Berkeley, CA, USA, 2010. USENIX Association.

[14] Enrique V. Carrera, Eduardo Pinheiro, and Ricardo Bianchini. Conserving disk energy in network servers. In *Proceedings of the 17th annual international conference on Supercomputing*, ICS '03, pages 86–97, New York, NY, USA, 2003. ACM.

[15] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. *SIGPLAN Not.*, 44:217–228, March 2009.

[16] R. Cervera, T. Cortes, and Y. Becerra. Improving application performance through swap compression. In *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 46–46, Berkeley, CA, USA, 1999. USENIX Association.

[17] Li-Pin Chang. Hybrid solid-state disks: combining heterogeneous nand flash in large ssds. In *Proceedings of the 2008 Asia and South Pacific Design Automation Conference*, ASP-DAC '08, pages 428–433, Los Alamitos, CA, USA, 2008. IEEE Computer Society Press.

[18] Feng Chen, Song Jiang, and Xiaodong Zhang. Smartsaver: Turning flash drive into a disk energy saver for mobile computers. In *Low Power Electronics and Design, 2006. ISLPED'06. Proceedings of the 2006 International Symposium on*, pages 412 –417, oct. 2006.

[19] Feng Chen, Rubao Lee, and Xiaodong Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of 17th International Symposium on High Performance Computer Architecture*, ISCA '11. IEEE Computer Society, 2011.

[20] Feng Chen, Tian Luo, and Xiaodong Zhang. Caftl: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, FAST'11, pages 77–90. USENIX Association, 2011.

[21] Shimin Chen. Flashlogging: Exploiting flash devices for synchronous logging performance. In *In Proceedings of SIGMOD'09*, June 2009.

[22] Steve C. Chiu, Wei-keng Liao, Alok N. Choudhary, and Mahmut T. Kandemir. Processor-embedded distributed smart disks for i/o-intensive workloads: architectures, performance models and evaluation. *J. Parallel Distrib. Comput.*, 65(4):532–551, 2005.

[23] Hyun Jin Choi, Seung-Ho Lim, and Kyu Ho Park. Jftl: A flash translation layer based on a journal remapping for flash memory. *Trans. Storage*, 4:14:1–14:22, February 2009.

[24] Ritendra Datta, Jia Li, and James Z. Wang. Content-based image retrieval: approaches and trends of the new age. In *Proceedings of the 7th ACM SIGMM international workshop on Multimedia information retrieval*, MIR '05, pages 253–262, New York, NY, USA, 2005. ACM.

[25] Garth Gibson David, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, Erik Riedel, David Rochberg, and Jim Zelenka. File server scaling with network-attached secure disks. In *In Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 272–284. ACM Press, 1997.

[26] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[27] S. DEBNATH, B. SENGUPTA and J. LI. Chunkstash: speeding up inline storage deduplication using flash memory. In *In Proceedings of USENIX'10*, June 2010.

[28] David J. DeWitt and Paula B. Hawthorn. A performance evaluation of data base machine architectures (invited paper). In *Proceedings of the seventh international conference on Very Large Data Bases - Volume 7*, VLDB '1981, pages 199–214. VLDB Endowment, 1981.

[29] J. Dinerstein, S. Dinerstein, P.K. Egbert, and S.W. Clyde. Learning-based fusion for data deduplication. In *Machine Learning and Applications, 2008. ICMLA '08. Seventh International Conference on*, pages 66 –71, dec. 2008.

[30] David H. C. Du. Intelligent storage for information retrieval. *Next Generation Web Services Practices, International Conference on*, 0:214–220, 2005.

[31] A.K. Elmagarmid, P.G. Ipeirotis, and V.S. Verykios. Duplicate record detection: A survey. *Knowledge and Data Engineering, IEEE Transactions on*, 19(1):1 –16, jan. 2007.

[32] E.J. FELIX, K. FOX, K. REGIMBAL, and J. NIEPLOCHA. Active storage processing in a parallel file system. In *In Proc. of the 6th LCI International Conference on Linux Clusters: The HPC Revolution*, 2006.

[33] Chen Feng, David Koufaty, and Xiaodong Zhang. Hystor: Making the best use of solid state drives in high performance storage systems. In *In Proceedings of International Conference on Supercomputing, ICS 2011*, ICS '11, Tuscon, Aizona, 2011. ACM.

[34] Wu Feng and etc. mpiblast: Open-source parallel blast, 2010. `http://www.mpiblast.org/`.

[35] Blake G. Fitch, Aleksandr Rayshubskiy, Michael C. Pitman, T. J. Christopher Ward, and Robert S. Germain. Using the active storage fabrics model to address petascale storage challenges. In *PDSW '09: Proceedings of the 4th Annual Workshop on Petascale Data Storage*, pages 47–54, New York, NY, USA, 2009. ACM.

[36] Greg Ganger. The disksim simulation environment, 2011.

[37] Gregory Robert Ganger. System-oriented evaluation of i/o subsystem performance. Technical report, 1995.

[38] Garth A. Gibson and Rodney Van Meter. Network attached storage architecture. *Commun. ACM*, 43(11):37–45, 2000.

[39] Jim Gray. Tape is dead, disk is tape, flash is disk. ram locality is king, 2007.

[40] Jim Gray. Flash memory pe cycles, 2011.

[41] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 229–240, New York, NY, USA, 2009. ACM.

[42] S. Gurumurthi, A. Sivasubramaniam, M. Kandemir, and H. Franke. Drpm: dynamic speed control for power management in server class disks. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 169 – 179, june 2003.

[43] Sudhanva Gurumurthi. Should disks be speed demons or brainiacs? *SIGOPS Oper. Syst. Rev.*, 41:33–36, January 2007.

[44] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. Oltp through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 981–992, New York, NY, USA, 2008. ACM.

[45] James Hays and Alexei A. Efros. Scene completion using millions of photographs. *Commun. ACM*, 51:87–94, October 2008.

[46] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: a mapreduce framework on graphics processors. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269, New York, NY, USA, 2008. ACM.

[47] Jiahua He, Jeffrey Bennett, and Allan Snavely. Dash-io: an empirical study of flash-based io for hpc. In *TG '10: Proceedings of the 2010 TeraGrid Conference*, pages 1–8, New York, NY, USA, 2010. ACM.

[48] Windsor W. Hsu, Honesty C. Young, and Alan Jay Smith. Projecting the performance of decision support workloads on systems with smart storage (smartstor). In *ICPADS '00: Proceedings of the Seventh International Conference on Parallel and Distributed Systems*, page 417, Washington, DC, USA, 2000. IEEE Computer Society.

[49] An i A. Wang, Peter Reiher, and Gerald J. Popek. Conquest: better performance through a disk/persistent-ram hybrid file system. In *In Proceedings of the 2002 USENIX Annual Technical Conference*, pages 15–28, 2002.

[50] Sami Iren and Steve Schlosser. Database storage management with object-based storage devices. In *Proceedings of the 1st international workshop on Data management on new hardware*, DaMoN '05, New York, NY, USA, 2005. ACM.

[51] Yongsoo Joo, Youngjin Cho, Kyungsoo Lee, and Naehyuck Chang. Improving application launch times with hybrid disks. In *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '09, pages 373–382, New York, NY, USA, 2009. ACM.

[52] William K. Josephson, Lars A. Bongo, Kai Li, and David Flynn. Dfs: A file system for virtualized flash storage. In *Proceedings of the 8th USENIX conference on File and storage technologies*, FAST'10, Berkeley, CA, USA, 2010. USENIX Association.

[53] Sooyong Kang, Sungmin Park, Hoyoung Jung, Hyoki Shim, and Jaehyuk Cha. Performance trade-offs in using nvram write buffer for flash memory-based storage devices. *IEEE Trans. Comput.*, 58:744–758, June 2009.

[54] S. KAWAGUCHI, A. NISHIOKA and H. MOTODA. A flash-memory based file system. In *In Proceedings of USENIX Winter*, pages 155–164, Jan 1995.

[55] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (idisks). *SIGMOD Rec.*, 27(3):42–52, 1998.

[56] H. KIM and S. AHN. Bplru: A buffer management scheme for improving random writes in flash storage. In *In Proceedings of FAST'08*, Feb 2008.

[57] Hyojun Kim and Seongjun Ahn. Bplru: a buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 16:1–16:14, Berkeley, CA, USA, 2008. USENIX Association.

[58] Youngjae Kim, S. Gurumurthi, and A. Sivasubramaniam. Understanding the performance-temperature interactions in disk i/o of server workloads. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 176 –186, feb. 2006.

[59] Peter M. Kogge, Jay B. Brockman, Thomas Sterling, and Guang Gao. Processing in memory: Chips to petaflops. In *In Workshop on Mixing Logic and DRAM: Chips that Compute and Remember at ISCA '97*, page pages, 1997.

[60] R. Kumar, D.M. Tullsen, P. Ranganathan, N.P. Jouppi, and K.I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 64 – 75, june 2004.

[61] University of Massachusetts Amherst Lab for Advanced System Software. Umass trace repository, 2009.

[62] Willis Lang, Jignesh Patel, and Srinath Shankar. Wimpy node clusters: What about non-wimpy workloads. In *DAMON '10:Proceedings of the Sixth International Workshop on Data Management on New Hardware*. ACM, 2010.

[63] H Lin, X Ma, W Feng, and N Samatova. Coordinating computation and i/o in massively parallel sequence search. *Parallel and Distributed Systems, IEEE Transactions on*, PP(99):1 –1, 2010.

[64] Heshan Lin, Xiaosong Ma, Praveen Chandramohan, Al Geist, and Nagiza Samatova. Efficient data access for parallel blast. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, page 72.2, Washington, DC, USA, 2005. IEEE Computer Society.

[65] SimpleScalar LLC. Simplescalar 4.0 simulator, 2010.

[66] Xiaonan Ma and A.L.N. Reddy. Mvss: an active storage architecture. *Parallel and Distributed Systems, IEEE Transactions on*, 14(10):993 – 1005, oct. 2003.

[67] T. MAKATOS, Y. KIONATOS, M. MARAZAKIS, M. D. FLOURIS, and A. BILAS. Using transparent compression to improve ssd-based i/o caches. In *In Proceedings of EuroSys'10*, April 2010.

[68] Bo Mao, Hong Jiang, Dan Feng, Suzhen Wu, Jianxi Chen, Lingfang Zeng, and Lei Tian. Hpda: A hybrid parity-based disk array for enhanced performance and reliability. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, 2010.

[69] Gokhan Memik, Alok Choudhary, and Mahmut T. Kandemir. Design and evaluation of smart disk architecture for dss commercial workloads. In *ICPP '00: Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing*, page 335, Washington, DC, USA, 2000. IEEE Computer Society.

[70] Anurag Acharya Mustafa, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. In *In Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 81–91, 1998.

[71] David Nagle, Denis Serenyi, and Abbie Matthews. The panasas activescale storage cluster: Delivering scalable high bandwidth storage. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, pages 53–, Washington, DC, USA, 2004. IEEE Computer Society.

[72] D. NARAYANAN, E. THERESKA, A. DONNELLY, S. ELNIKETY, and A. ROWSTRON. Migrating enterprise storage to ssds:analysis of tradeffs. In *In Proceedings of EuroSys'09*, March 2009.

[73] Dushyanth Narayanan, Eno Thereska, Austin Donnelly, Sameh Elnikety, and Antony Rowstron. Migrating server storage to ssds: Analysis of tradeoffs. In *In EuroSys*, 2009.

[74] Howard B. Newcombe and James M. Kennedy. Record linkage: making maximum use of the discriminating power of identifying information. *Commun. ACM*, 5:563–566, November 1962.

[75] Jon Oberheide, Evan Cooke, and Farnam Jahanian. Cloudav: N-version antivirus in the network cloud. In *Proceedings of the 17th conference on Security symposium*, pages 91–106, Berkeley, CA, USA, 2008. USENIX Association.

[76] Nohhyun Park and D.J. Lilja. Characterizing datasets for data deduplication in backup applications. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1 –10, 2010.

[77] Sungmin Park, Hoyoung Jung, Hyoki Shim, Sooyong Kang, and Jaehyuk Cha. Write buffer-aware address mapping for nand flash memory devices. In *Modeling, Analysis and Simulation of Computers and Telecommunication Systems, 2008. MASCOTS 2008. IEEE International Symposium on*, pages 1 –2, 2008.

[78] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fourth Edition, Fourth Edition: The Hardware/Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[79] Juan Piernas, Jarek Nieplocha, and Evan J. Felix. Evaluation of active storage strategies for the lustre parallel file system. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 28:1–28:10, New York, NY, USA, 2007. ACM.

[80] L. Prada, J.D. Garcia, J. Carretero, and F. Garcia. Saving power in flash and disk hybrid storage system. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems, 2009. MASCOTS '09. IEEE International Symposium on*, pages 1 –3, 2009.

[81] T. PRITCHETT and M. THOTTETHODI. Sievestore: a highly-selective, ensemble-level disk cache for cost-performance. In *In Proceedings of ISCA'10*, June 2010.

[82] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[83] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[84] IBM Research. Ibm unstructured information management architecture.

[85] Microsoft Reserach. Ssd extension for disksim simulation environment, 2009.

[86] E. RIEDEL. Object based storage (osd) architecture and systems, 2006.

[87] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. *Computer*, 34(6):68–74, 2001.

[88] Sriram Sankar, Sudhanva Gurumurthi, and Mircea R. Stan. Intra-disk parallelism: An idea whose time has come. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 303–314, Washington, DC, USA, 2008. IEEE Computer Society.

[89] Seagate. The advantages of object-based storage secure, scalable, dynamic storage devices., 2005.

[90] Anand Lal Shimpi. Anandtech: Intel's 90nm pentium m 755: Dothan investigated, Dec 2007. `http://www.anandtech.com/cpuchipsets/`.

[91] Muthian Sivathanu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Evolving rpc for active storage. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ASPLOS-X, pages 264–276, New York, NY, USA, 2002. ACM.

[92] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-smart disk systems. In *FAST '03: Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 73–88, Berkeley, CA, USA, 2003. USENIX Association.

[93] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending ssd lifetimes with disk-based write caches. In *Proceedings of the 8th USENIX conference on File and storage technologies*, FAST'10, pages 8–8, Berkeley, CA, USA, 2010. USENIX Association.

[94] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. Extending ssd lifetimes with disk-based write caches. In *Proceedings of the 8th USENIX conference on File and storage technologies*, FAST'10, pages 8–8, Berkeley, CA, USA, 2010. USENIX Association.

[95] Guangyu Sun, Yongsoo Joo, Yibo Chen, Dimin Niu, Yuan Xie, Yiran Chen, and Hai Li. A hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1 –12, Jan 2010.

[96] Irina Chihaia Tuduce and Thomas Gross. Adaptive main memory compression. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 29–29, Berkeley, CA, USA, 2005. USENIX Association.

[97] Mustafa Uysal, Anurag Acharya, and Joel Saltz. Evaluation of active disks for large decision support databases. Technical report, Santa Barbara, CA, USA, 1999.

[98] C. WHITE. Consolidating, accessing, and analyzing unstructured data. 2005.

[99] William E. Winkler. The state of record linkage and current research problems. Technical report, Statistical Research Division, U.S. Census Bureau, 1999.

[100] Seon yeong Park, Euiseong Seo, Ji-Yong Shin, Seungryoul Maeng, and Joonwon Lee. Exploiting internal parallelism of flash-based ssds. *Computer Architecture Letters*, 9(1):9 –12, 2010.

[101] Jin Hyuk Yoon, Eyec Hyun Nam, Yoon Jae Scong, H. Kim, B.S. Kim, Sang Lyul Min, and Yookun Cho. Chameleon: A high performance flash/fram hybrid solid state disk architecture. *Computer Architecture Letters*, 7(1):17 –20, jan. 2008.

[102] Jin Hyuk Yoon, Eyee Hyun Nam, Yoon Jae Seong, Hongseok Kim, Bryan Kim, Sang Lyul Min, and Yookun Cho. Chameleon: A high performance flash/fram hybrid solid state disk architecture. *IEEE Computer Architecture Letters*, 7:17–20, 2008.