Manageable Storage via Adaptation in WiND

Andrea Arpaci-Dusseau Remzi Arpaci-Dusseau John Bent Brian Forney
Sambavi Muthukrishnan Florentina Popovici Omer Zaki

Department of Computer Sciences
University of Wisconsin, Madison

Abstract

The key to storage manageability is adaptation. In traditional storage systems, adaptation is performed by a human administrator, who must assess problems, and then manually adjust various knobs and levers to bring the behavior of the system back to an acceptable level. Future storage systems must themselves adapt, and in doing so, reduce the need for manual intervention.

In this paper, we describe the Wisconsin Network Disks project (WiND), wherein we seek to understand and develop the key adaptive techniques required to build a truly manageable network-attached storage system. WiND gracefully and efficiently adapts to changes in the environment, reducing the burden of administration and increasing the flexibility and performance of storage for an eclectic range of clients. In particular, WiND will automatically adapt to the addition of new disks to the system, the failure or erratic performance of existing disks, and changes in client workload and access patterns.

1 Introduction

Data storage lies at the core of information technology. Whether in distributed file systems, Internet services, or database engines, nothing shapes the user's experience more than the reliability, availability, and performance of the I/O subsystem.

The basic elements of the storage device industry are in the midst of radical change. In particular, the advent of *network-attached storage devices* [13, 18] [1] will fundamentally alter the manner in which data is maintained, stored, and accessed. The combination of inexpensive yet powerful microprocessors inside every disk, and high-speed, scalable networks enables storage vendors to move disks off of slow, shared-medium busses and onto a storage-area network (SAN). Collectively, we refer to a group of network-attached disks as a *storage cluster*.

[1] The group at CMU refers to NASD as "network-attached secure disks"; since we wish to refer to something more general, we simply refer to disks on a network as network-attached storage devices.

Storage clusters provide many potential advantages over traditional storage architectures:

- **Scalability:** Not limited by a shared-medium interconnect such as SCSI, network-attached storage can deliver scalable bandwidth to multiple clients.
- **Fault Tolerance:** In a standard server environment, software failure on a single host can lead to data unavailability. In contrast, a storage cluster, accessible from more than one host, allows *multiple access paths* to data, thus increasing data availability.
- **Simplicity:** File systems built on network-attached storage can leave low-level layout decisions and performance optimizations to the drives, simplifying software and increasing maintainability [14].
- **Incremental Growth:** As compared to a typical RAID array, network-attached storage allows for essentially unlimited growth in the number of disks, removing the need for a "fork-lift" replacement of an entire RAID array [10].
- **Specialization:** Network-attached storage mixed within a cluster enables *specialization* of the system in direct response to need; if the system requires more disk bandwidth or capacity, one can buy more disks; if the system needs more CPU power, one can buy processing nodes.

1.1 The Problem: Management

However, storage clusters also introduce additional challenges, particularly regarding *manageability*. Whereas absolute performance was the goal of a great number of previous systems, manageability has become the new focus [16, 25]. Thus, a system that works consistently with little or no human intervention will be preferred over a system that sporadically delivers near-peak performance or requires a large amount of human attention to do so.

Manageability is more challenging in storage clusters due to their additional complexity. This complexity is a result of both the networking hardware and protocols between clients and disks, and the increasingly sophisticated nature of modern disks drives (*e.g.*, multiple zones [21], SCSI bad-block remapping [4], sporadic performance before absolute failure [29]). The result of complexity in both networks and disks is a system where unpredictable behavior (especially in terms of performance) is the norm, not the rare case [4]. The likelihood of unexpected behavior is compounded by an increasing demand for large-scale systems (*e.g.*, a storage-service provider with a 10,000-disk storage farm).

In spite of unpredictability, an ideal manageable storage system should behave as follows.

- **After Upgrade:** When a disk is added, the ideal system immediately begins utilizing it to its full capacity, both migrating data there to balance load (long-term data migration) and writing new data to the disk to increase throughput (short-term adaptation). The ideal system fully utilizes the disk regardless of its capacity or performance relative to other disks.

- **During Failure:** The complexity of modern disks has introduced a new range of failures: instead of a binary fail-stop model where components either work perfectly or not at all, there is a continuous range where a component may be working but not at full capacity [4, 29]. The ideal system utilizes such "performance faulty" components to the degree that each allows.

- **After Workload Delta:** With new applications or data-sets, the access patterns presented to a storage system may change and the previous layout of data across disks may no longer be satisfactory. The ideal system reacts by migrating data to better match current conditions; for example, frequently accessed data may be migrated to newer, faster disks or be spread across more disks for increased bandwidth.

- **Under Shared Access:** With network-attached storage, multiple clients and file systems may simultaneously share the underlying disks and therefore have incomplete knowledge of the activity at each disk. The ideal system adapts to contention at run-time and delivers whatever performance is available to the clients.

Traditional systems react to such scenarios with the assistance of a human administrator, who must assess the problem and then manually adjust various parameters to bring the performance of the system back to an acceptable level. *Adaptation* is key to manageable storage systems of the future, where the *system itself* reacts to changes in system behavior and automatically adjusts to problems, thereby reducing the need for manual intervention.

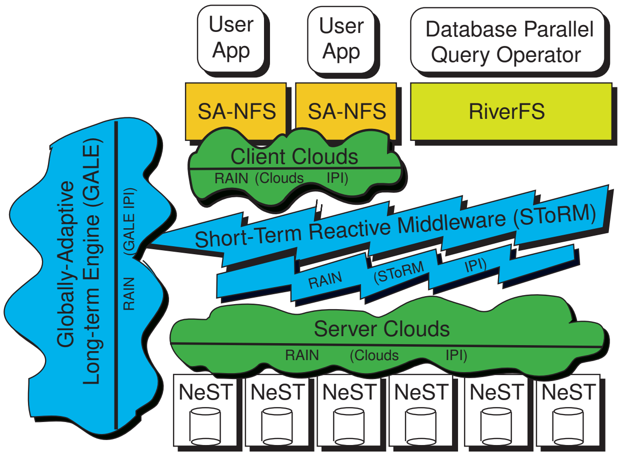

Figure 1: **WiND System Architecture**

1.2 Wisconsin Network Disks

In this paper, we describe the design and current status of Wisconsin Network Disks (WiND). Our main focus in WiND is to develop the key techniques necessary to build truly manageable network-attached storage. To achieve this goal and to fully exploit the potential of the underlying hardware, all of the software we develop will be distributed and scalable.

WiND is comprised of five major software components, broken down into two groups. The first three are the run-time and off-line adaptive elements of WiND: SToRM, GALE, and Clouds. The other two are key pieces of supporting infrastructure: RAIN and NeST. The overall system architecture is presented in Figure 1; we now briefly describe each of the components.

The first key piece of adaptation software is SToRM, which performs run-time adaptation for data access and layout. SToRM adapts to short-term changes in workload characteristics and disk performance by quickly adjusting how much each client reads from or writes to each disk.

However, short-term adaptation alone is not enough; it lacks global perspective. For this, we introduce GALE, a software layer that monitors system activity via on-line simulation and performs global, long-term optimizations to improve performance and reliability. For example, GALE may migrate or replicate data to improve read performance or to re-balance the workload across the drives. GALE tries to place data in accordance with the current "climate" of the system and the access patterns of applications. GALE also interacts in important ways with SToRM. For example, without data replication, SToRM has little or no flexibility as to where to read a given data block from; when needed, GALE replicates data to provide increased flexibility for SToRM, and thus improves the adaptivity of the system.

Both SToRM and GALE are designed to adapt data flows to and from disk; however, many requests to a network-attached disk may be satisfied from in-memory caches. Thus, Clouds provides flexible caching for network-attached storage. Clouds provides mechanisms and policies for both client-side and server-side caches, taking variable costs into account. Clouds also can employ cooperative techniques [2] to conglomerate server-side cache resources, and potentially hide disk variations from clients.

WiND also contains two key pieces of software infrastructure. The first is RAIN, an information architecture that encapsulates the acquisition and dispersal of information within SToRM, GALE, and Clouds. RAIN provides Information Programming Interfaces (IPIs) to each software subsystem, which hide details of information flows and greatly simplify system structure and maintainability. The second is NeST, which provides flexible and efficient single-site storage management.

Finally, we eventually plan to implement and evaluate two file systems on top of the WiND infrastructure. First, we will build a Striped, Adaptive version of NFS (SA-NFS), which adheres to the NFS file system interface, but is modified to stripe data blocks to disk in an adaptive fashion. We also plan to construct RiverFS, a parallel, record-oriented file system, designed to support high-performance parallel query operators found in database environments. RiverFS takes advantage of relaxed data semantics to more readily provide robust, consistent performance.

2 Adaptivity in WiND

In this section, we discuss the three major adaptive components of WiND: run-time adaptive layout and access in SToRM, off-line monitoring and adaptation in GALE, and adaptive caching with Clouds. These three pieces of software technology work in harmony to adapt to changes in the system and thus provide truly "hands off" network-attached storage.

2.1 SToRM

SToRM (Short-Term Reactive Middleware) is a distributed software layer that interposes between clients and servers to provide adaptive data access. SToRM is used by file systems to adapt to volatile disk behavior at run-time and deliver full bandwidth to clients without intervention.

The challenges of SToRM are two-fold. First, SToRM must adapt the data streams moving to or from disk. The general idea is that clients should interact with the higher performance disks more frequently; that is, a client should access proportionally more data from faster disks. Second, SToRM must achieve the first goal with low overhead, both in extra processing and book-keeping (or meta-data). However, adaptation and overhead are often at odds; thus, we seek the "knee of the curve" where adaptivity is high and overhead is still acceptable.

The challenges of adapting to allocating writes versus non-allocating writes and reads are substantially different: allocating writes can be placed anywhere (as long as one is willing to pay the cost of book-keeping), whereas non-allocating writes and reads have less freedom (*i.e.*, they must be performed on few locations where they are already stored on disk). We now discuss each operation.

Allocating Writes: When data is written to a file for the first time, space allocation occurs as blocks are committed to physical storage; we refer to such writes as *allocating writes*. For illustrative purposes, we assume data is striped across a set of remote disks with RAID level 0 (striping without redundancy). Algorithms for allocating data blocks across disks can be classified in terms of how frequently they evaluate the relative speeds of each disk and consequently adapt placement; we now discuss a range of these algorithms.

- **Level g_0:** Traditional striping does not gauge the performance of its storage components before allocating data across them and therefore assumes that all disks run at the same rate. The strength of this approach is that the only meta-data needed for block lookup is block size. The weakness is that all disks are treated identically and thus system performance tracks the rate of the slowest disk.

- **Level g_1:** The most primitive adaptive allocation algorithm adjusts to disks delivering data at different rates, but assumes that each disk behaves in a fixed manner over time. With a g_1 algorithm, the relative speeds of the disks are calculated exactly once, and then the amount of data striped to each device is made proportional to its relative speed. To lookup a logical block, a g_1 approach requires little additional meta-data: it must also know the *striping ratios* across disks.

- **Level g_n:** These algorithms periodically determine the relative performance of disks and adjust striping ratios accordingly. Each period with a new striping ratio is a *striping interval*. Additional meta-data is needed: the striping ratio and the size of each striping interval.

- **Level g_∞:** With the most adaptive algorithm, each client continuously gauges the performance of the system and writes each data block to the disk it believes will handle the request the fastest. The advantage of g_∞ approaches is that they can adapt most rapidly to performance changes and make small adjustments that cannot be reflected in simple integer striping ratios. The disadvantage is that a significant amount of meta-data must be recorded: the target disk and block offset for every logical block written.

To demonstrate the benefit of a g_1 versus an g_0 approach, we present the performance of a user-level library for file striping. These measurements were performed on an UltraSPARC I workstation with two internal 5400 RPM Seagate Hawk disks on a single fast-narrow SCSI and two external 7200 RPM Seagate Barracuda disks on a fast-wide SCSI. While these measurements were produced in the context of a single machine with multiple disks, we believe these results are general and apply to other environments with heterogeneous disks.

Table 1 compares the performance of g_0 and g_1 striping across all four disks. With g_0 striping, data is striped in blocks of 64 KB to each disk in the system. The table shows that g_0 striping is not effective with disks of different speeds, achieving only 77% of peak bandwidth. For g_1 striping, we gauge the relative performance of the disks via a simple off-line tool [3]; we measure that we can achieve 8.0 MB/s when writing simultaneously to the two Hawk disks, and 12.1 MB/s to the two Barracuda disks. This peak performance measured in isolation determines the proper ratio of stripe sizes: we write *two* blocks of data to each of the slower disks and *three* to each on the faster disks. Thus, with g_1 striping we achieve 95% of peak bandwidth.

One of the major research issues for SToRM is how to extend the adaptation algorithms (g_0 to g_∞) to other RAID levels. We will first concentrate on RAID level 1 (mirroring), because it has excellent performance properties and is conceptually simple since no parity is calculated. A straight-forward transformation of adaptive striping into mirrored, adaptive striping is to treat pairs of disks as a single logical disk and perform adaptive striping across the logical disks. The major disadvantage of this approach is that it introduces *performance coupling* across pairs of disks; the pair will run at the rate of the slow disk. Thus, SToRM should couple disks that have similar performance characteristics (calling on GALE for hints regarding the best pairing). Alternatively, mirroring may be performed lazily for files that can tolerate a window of potential loss; GALE can run later and fill in the mirrors for full reliability. A similar idea is proposed in AFRAID, where redundancy is sometimes relaxed to improve RAID-5 performance under small writes [26].

Non-Allocating Writes: With non-allocating writes, blocks are written over previously allocated blocks in a file. As a result, SToRM has no choice as to which disk receives a non-allocating write. However, this may still lead to acceptable performance in many cases.

Disks	SCSI Bus	Write (max)	Write (actual)
2 Seagate Hawks	Narrow	8.0	8.0
2 Seagate Barracudas	Wide	12.1	12.1
All disks (g_0 striping)	Both	20.1	15.5
All disks (g_1 striping)	Both	20.1	19.1

Table 1: **Benefits of g_1 Striping.** *The table shows the write bandwidth achieved with g_0 and g_1 level striping in MB/s. The first column lists the disks, and the second column the applicable SCSI buses. The Write(max) column shows the peak aggregate bandwidths of the disks, while the Write(actual) column shows the bandwidth achieved with the striping library.*

Given that SToRM initially allocates the amount of data to a disk depending upon the observed performance of that disk, it leaves a *performance footprint* on the storage system. A performance footprint has two contributing factors: that produced by the disks themselves and that produced by the workloads currently accessing the drives. The factor contributed by the devices is simply the speed at which file data can be sequentially written under no contention, given a particular layout of blocks on the disk. The factor contributed by the workload includes the access pattern of a single application (*e.g.*, sequential or random) and contention among multiple applications.

There is a very useful implication to formalizing the concept of a performance footprint. If *temporal performance locality* exists (*i.e.*, if a performance footprint changes little from that of the recent past), then clients that access the file later (with either non-allocating writes or with reads) will access data from the disks with the optimal performance allocation.

However, if temporal performance locality does not exist, then non-allocating writes are vulnerable to performance variations. The only completely general and flexible solution is to transform non-allocating writes into allocating writes, *e.g.*, build a multi-disk log-structured file system [24] that incorporates run-time adaptive techniques, akin to the g_∞ approach described above. In this case, adaptivity comes at a high cost.

We plan to evaluate the relative strengths and weaknesses of the range of allocation algorithms in terms of their adaptivity to changing performance footprints and the amount of meta-data required. Specifically, we will investigate algorithms for adaptively determining the length of each striping interval. When the performance footprint changes rapidly, the striping interval should be small to obtain the best bandwidths from the disks; when the performance footprint changes more slowly, the striping interval should be longer to amortize the cost of gauging and recording meta-data.

Reads: The freedom of reads depends on the level of replication of the file. For example, with simple striping, each block of a file is written to only one location; the block must be read back from that disk regardless of its later performance. The lack of freedom may be acceptable if temporal performance locality exists. When the performance footprint is no longer valid, we assume that GALE re-organizes or replicates the data, taking the current climate into account. SToRM must be able to adaptively take advantage of replicated sources of data under reads. Our earlier work on *Graduated Declustering* focused on the distributed, adaptive use of mirrors for parallel clients [5]; we plan to generalize it to handle more general-purpose workloads and a variety of replicated layout schemes.

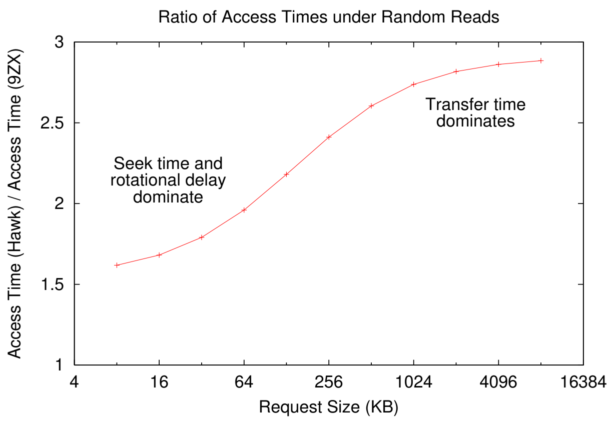

Figure 2: **Access Size Impact.** *The figure depicts the impact of access size on the performance difference between a Seagate Hawk and a more modern IBM 9ZX drive. The graph plots the ratio of performance between the two disks under random reads, varying the size of requests along the x-axis.*

2.2 GALE

Short-term adaptation does not solve all of the problems encountered in dynamic, heterogeneous environments. Short-term adaptations are analogous to greedy algorithms, which often do not arrive upon the best possible solution; both lack *global perspective*. To provide a long-term view of system and workload activity and to optimize system performance in ways not possible at runtime, we are building an additional software structure, the Globally Adaptive Long-term Engine (GALE).

GALE provides three basic services in WiND. First, GALE performs *system monitoring*, using both active and passive techniques to gather workload access patterns and device performance characteristics, and detecting anomalies in component behavior. Second, GALE decides when to perform a global optimization itself via *action instantiation*; for example, GALE may replicate an oft-read file for performance reasons. Finally, GALE provides information to SToRM and Clouds via *hint generation*.

System Monitoring: GALE inserts lightweight monitors into clients and servers to trace workload access patterns and measure response times. Beyond simple tracing, one particularly novel aspect of GALE is the use of *on-line simulation* to generate *performance expectations*. Periodically, GALE will take a set of actual disk requests and submit them to a disk simulator. The system will then compare the performance results from the simulation to measured performance, and take note if there are stark differences. Accurate disk simulators are readily available [11], and our initial experiments reveal that they can simulate a modern drive in real-time. Thus, by comparing real performance to simulated performance, GALE can detect when something in the system has gone awry.

Action Instantiation: After monitoring the system, GALE may choose to migrate or replicate data to better match the current "climate" of the system, which is done in two steps. The first step is a cost/benefit analysis: GALE must compare the costs of migration or replication versus their benefit for current workloads, taking care to ignore any short-term changes in the system. The second-step is the actual migration or replication.

Migration is useful for moving data off of a disk that is behaving in unexpected ways (as determined via simulation-comparison) and may soon fail and for reorganizing oft-read data to better match access patterns [20] or to utilize more disks for higher bandwidth.

Replication is important for giving SToRM additional flexibility when accessing data. By replicating a given block, SToRM can adaptively choose the best site from which to read, based on the current climate. For writes, we plan on exploring both active and lazy updating of replicas; the former has the disadvantage that it leaves little opportunity for adaptation while that latter has the disadvantage that replicas contain stale data.

We are also investigating the use of *multi-access* replication schemes, which are useful when the performance differential across two drives varies by workload. Figure 2 illustrates this performance differential for two drives, a Seagate Hawk and an IBM 9ZX. For small requests, the IBM drive is only 1.5x faster, due to the smaller edge it has in seek time and rotational delay. The larger the request, the more transfer rate dominates, and thus the larger the performance difference between the two drives. Thus, if GALE detects that both small/random and large/sequential accesses are important to overall performance, GALE creates two replicas, one whose data layout is optimized for small requests and one for large transfers. The challenge here is to simultaneously utilize the multi-access replicas for data availability.

Migration and replication both consume serious resources in the system and must be scheduled carefully. Idle time has been used successfully in other systems [15, 30], and we hope to use such times for GALE activities.

Hint Generation: GALE also provides hints that SToRM and Clouds can access to improve their decisions. Some examples of different hints that GALE can pass are: *data placement hints* (*i.e.*, the disks that a new file should be allocated upon), *ratio adjustment hints* (*i.e.*, the initial performance levels to expect from a set of disks), *fault-masking hints* (*i.e.*, data blocks that would benefit from fault masking via caching, as described in the next section), and *mirror-matching hints* (*i.e.*, the set of disks that should be used for mirrors). GALE may also return confidence indications per hint, allowing SToRM and Clouds to decide whether to use it or not.

2.3 Clouds

Clouds is a flexible and adaptive caching layer in WiND. It is flexible because clients of storage are not forced to use it – it can be used for all accesses, for a limited subset of files, or not at all. It is adaptive because all Clouds algorithms fundamentally take the varying cost of block access into account – not only can different blocks have different replacement costs, but those costs can change over time. Clouds is divided into two distinct components: client-side Clouds and server-side Clouds. We now explore both client Clouds and server Clouds.

Client Clouds: Clients of the network-attached storage system may cache blocks from a number of drives in the system. Clouds enhances existing operating-system buffer managers by enabling caching algorithms to take the variable cost of block access into account. Algorithms such as LRU do not take cost into account – they assume that all blocks are equally costly to fetch – and thus do not work well in a variable-cost environment.

Early theoretical work by Belady [6] established an off-line optimal cache-replacement algorithm – simply replace the block that will be referenced furthest in the future. However, no off-line generalization for caches with variable replacement costs is known, though there are good heuristics [31]. Thus, most recent work on Web Cache caching strategies focuses on extending or modifying LRU to work with different costs of replacement [9].

With Clouds, we will extend such approaches to function in our local-area environment, where access costs are much different than the wide-area and change more rapidly. An interesting challenge is to track replacement cost with low overhead. For example, it may be too expensive to track the cost of accessing each block; therefore, the caching algorithm may have to inform RAIN of a set of candidate blocks, and RAIN will return the costs for those. We will explore trade-offs in managing this information versus the performance of the algorithms. We will also apply variable-cost caching to non-LRU algorithms; recent work has shown that more sophisticated algorithms can behave like LRU when data fits in cache, while avoiding thrashing with larger data sets [28].

Our initial simulation results of client-side cost-aware caching algorithms are promising. When a single disk is performing poorly – perhaps stuttering before absolute failure, or perhaps just an older, slower disk – and three other disks perform at the expected level, traditional caching algorithms are adversely affected, because LRU does not account for the cost of accessing the slower disk. However, with a cost-aware cooperative approach, some of the performance differences can be masked.

Server Clouds: On the server-side, Clouds uses cooperative-caching algorithms [2] to manage the caches of the disks.[2] This enables a new ability: the disk caches can cooperate to cache blocks from "slow" disks, and thus hide the variable cost of disk access from clients, which we call *performance-fault masking*.

For example, assume that a single disk is running much slower than the rest, and that an application is sequentially accessing a particular data set across that disk and others. In this case, the server-side caches can cooperate and cache blocks from the slow disk, whose slowness can thus be "masked" from higher levels of the system, creating the illusion of a uniform set of disks.

Masking can occur for both reads and writes, and, as in SToRM, writes are somewhat easier to handle. The server caches can cooperate to buffer data that is destined for a slow disk, and thus hide its slowness from clients. Of course, if a large amount of data is written to that disk as compared to the total amount of buffer space available, the technique will not be successful. Masking reads behave analogously – server-side caches favor blocks cooperatively from slower disks, and thus potentially hide their latency. However, the blocks must first be in the caches! Thus, this technique is more effective for repeated reads, or when prefetching is employed to fill the caches with blocks from the slow disk(s).

Finally, we note the interaction between Clouds and SToRM. The server-side caches may cooperate to "hide" some of the variable behavior of disks from clients. Thus, there needs to be an interface between Clouds and SToRM such that SToRM is informed of the intention of Clouds and does not take action itself.

[2] Because we believe clients should be autonomous (*i.e.*, they should not have to trust all other clients), we do not plan to explore client-side cooperative algorithms.

3 Core Infrastructure

We have identified two pieces of infrastructure that are necessary to build an effective WiND system. The first is RAIN, a thin software layer responsible for efficiently gathering and dispersing information throughout the system. The second is NeST, a single-site flexible storage manager.

3.1 RAIN

The goal of RAIN (Rapid Access to InformatioN) is to distribute the current 'climate' of the system – how remote components are behaving – and thus enable effective and simplified implementations of SToRM, GALE, and Clouds. The information layer presents itself to higher-level layers via specialized Information Programming Interfaces, or IPIs. These interfaces insulate the algorithms used by the adaptive components from the details of how information is gathered, stored, and propagated.

The challenge of RAIN is to deliver accurate and low-cost information about current performance. One of the axes that we will investigate is whether information should be gathered explicitly or implicitly. We recognize that *explicitly* querying remote disks may not always be the most appropriate approach. First, there may not be an interface for obtaining the desired information. Second, accessing an explicit interface may be too costly – sending an explicit request consumes shared network resources that may be needed for data transfers and induces additional work on the remote disk. Finally, the explicit query could fail, forcing the requester to handle many different failure cases.

Therefore, we will investigate *implicit* sources of information, in which RAIN infers the desired characteristic by observing operations that already exist in the system. For example, by observing the time required for the most recent read from a particular disk or the number of outstanding requests to a disk, remote-disk performance can be inferred with little overhead. The central advantage of implicit methods is that they provide information for free – no additional communication is required, only the ability to deduce remote behavior from a local observation. However, there are disadvantages to these methods: the inferences that must be made are often subtle, and the information flow is restricted to the path of the data flow.

Because IPIs hide the method of gathering information, RAIN is free to switch methods at run-time to find that which are most effective (*e.g.*, explicit or implicit). A switch of methods depends on three variables: the *frequency* of climate changes (how chaotic and dynamic is the system?), the *accuracy* of the method (how good is the information obtained?), and the *overhead* it induces (how much of total system resources is spent?). Thus, we plan to evaluate the impact of the information-gathering style on higher layers of the system.

3.2 NeST

Another key piece of infrastructure is NeST(Network Storage Technology), our single-site storage manager, so named as its original intent was to provide storage for Condor [19]. NeST is a highly flexible and configurable I/O appliance, and therefore certainly will see application outside of the WiND environment. The main axes of flexibility that NeST provides are: the client protocol for communication (a WiND-specific protocol, HTTP, NFS, and a simple NeST native tongue); the concurrency architecture (threads, processes, or events); a range of locking and consistency semantics; and a flexible infrastructure for caching. By configuring NeST, one can deploy a specialized, highly-tuned I/O server that is well-suited for the current environment.

We are also investigating the interface NeST should provide to support adaptation. Our starting point is the object-level interface put forth by the National Storage Industry Consortium (NSIC), which is derived directly from the CMU NASD project. However, we have observed that this interface limits client-side algorithms; thus, we will develop alternatives that are more "adaptation friendly" and subsequently implement them within NeST.

4 Status

The WiND system is currently being developed on a cluster of 36 Intel-based machines, each running the Linux operating system, and connected together via both 100 Mbit/s and Gigabit Ethernet. Each PC contains five 9.1GB IBM Ultrastar 9LZX disks, and can be used as a network-attached storage device or client.

A basic prototype of SToRM is currently being developed, along with its RAIN IPI and many information-gathering alternatives. The client-side software plugs into Linux as a loadable kernel module, which talks to NeST via a homegrown WiND protocol. NeST is also up and running, and recent experiments show that it delivers full bandwidth to clients with both process and thread models – event-based NeST does not work well on Linux machines due to the limitations of the `select()` interface.

For GALE, we have studied Ganger's simulator [11], and found that its performance is suitable; events are simulated at a rate much faster than real-time. However, our initial results indicate that the simulator is not accurate on a per-request granularity, and therefore more sophisticated approaches may be required.

Finally, to better understand the caching algorithms of Clouds, we have developed a detailed simulation of our environment, which allows us to easily explore many algorithmic alternatives. After we understand the algorithmic trade-offs, we will proceed to an implementation in our prototype environment.

5 Related Work

The basic architecture of our storage system has been strongly influenced by the network-attached secure disk (NASD) project at Carnegie-Mellon [13]. For example, the NASD project introduced the drive object model that we use as a starting point. However, the focus of CMU NASD is support for traditional file systems and strong security, whereas we are concentrating on adaptivity, and hoping to leverage the security infrastructure that the CMU group develops.

The Petal storage cluster is also closely related [18]. Petal is assembled from a cluster of commodity PCs, each with a number of disks attached. Petal exports a large "virtual disk" to clients over a high-speed network. Petal's elegance and simplicity arises from careful separation of storage system functionality from the file system. Petal does contain a limited form of run-time adaptation, in that a client reading from a mirror picks the mirror with the shortest queue length. In WiND, we do not enforce a strong separation between the storage system and file systems; by exposing each disk to client-side file system software, adaptation across disks is made possible.

The IP Station from USC-ISI also explores network-attached peripheral devices [22]. In particular, they advocate the use of stock IP protocols for communication, rather than custom-tailored fast networking layers. This approach has two advantages, in providing easy compatibility among communicating devices and allowing the use of off-the-shelf, well-tested software. In WiND, we are using TCP/IP for those very reasons.

Several projects have taken network-attached storage a step further, running user code of some form on the drives [1, 23]. These "Active Disks" approaches lose the advantages of separation and specialization, but in network-limited environments can provide performance advantages. We believe that our adaptive techniques are applicable in this environment.

The Berkeley I-Store project discusses the concept of an "introspective" system built from intelligent disks [7]. In WiND, GALE is the component that provides a novel form of system introspection via on-line simulation. More recently, in [8], the authors propose a new set of benchmarks to evaluate system availability, which we hope to apply in our evaluation of WiND.

Robust performance has long been the goal of storage systems. For example, chained declustering balances load in a mirrored storage system under failure [17]. By carefully allocating data across the disks, read traffic avoids hot-spots typical in mirrored systems under failure. We seek to generalize this concept by adapting to all unexpected behavior of disks, not just absolute failure.

An excellent example of an adaptive system is the AutoRAID storage array [30]. AutoRAID presents a standard RAID interface to clients, but adaptively migrates data between two RAID levels: "hot" data is placed in mirrored storage for improved performance, and "cold" data is moved into RAID-5 storage to increase effective capacity. Such a "performance versus capacity" optimization could be placed into the GALE framework.

Adaptivity has also been explored within the context of parallel file systems [27]. In that study, the authors explore the use of fuzzy logic to adaptively select the proper stripe size for a storage system. Such approaches may also be applicable within WiND.

Long-term adaptation has shown promise in single-disk log-structured file systems [20]. With this approach, file layout is reorganized off-line to improve the read performance of LFS. Similar techniques could be employed by GALE, although GALE must generalize the task of reorganization to operate across multiple disks.

Finally, the issue of heterogeneous RAID strategies has been studied in the multimedia literature with regards to video servers [12]. These studies all assume static performance differences among the components, not the dynamic environment that we expect to develop.

6 Summary

Complexity is growing beyond the point of manageability in storage systems. Comprised of largely autonomous, complicated, individual components, and connected by complex networking hardware and protocols, storage systems of the future will exhibit many of the same properties – and hence, the same problems – of larger scale, wide-area systems. Thus, software programming environments for these platforms must provide mechanisms that facilitate robust global behavior in spite of chaotic and dynamic component behavior. Without adaptive mechanisms, storage will become increasingly difficult to manage, and require a high amount of human involvement.

Towards this end, in WiND, we are developing three pieces of adaptive software: SToRM, which provides reactive, run-time data access and layout, GALE, which performs system monitoring and off-line adaptation, and Clouds, which provides flexible caching. The underpinnings of the three layers of adaptation is the development of a rigorous information architecture called RAIN, and a flexible, general storage manager known as NeST. We believe that the successful development of these components will reduce the burden of storage administration.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active Disks. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, San Jose, CA, October 1998.

[2] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, and R. Wang. Serverless Network File Systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 109–26, December 1995.

[3] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. P. Patterson. High-Performance Sorting on Networks of Workstations. In *Proceedings of the 1997 ACM SIGMOD Conference*, pages 243–254, 1997.

[4] R. H. Arpaci-Dusseau. *Performance Availability for Networks of Workstations*. PhD thesis, University of California, Berkeley, 1999.

[5] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. A. Patterson, and K. Yelick. Cluster I/O with River: Making the Fast Case Common. In *IOPADS '99*, May 1999.

[6] L. A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5:78–101, 1966.

[7] A. Brown, D. Oppenheimer, K. Keeton, R. Thomas, J. Kubiatowicz, and D. A. Patterson. ISTORE: Introspective Storage for Data-Intensive Network Services. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Rio Rico, Arizona, March 1999.

[8] A. Brown and D. A. Patterson. Towards Maintainability, Availability, and Growth Benchmarks: A Case Study of Software RAID Systems. In *USENIX 2000*, San Diego, CA, June 2000.

[9] P. Cao and S. Irani. Cost-Aware WWW Proxy Caching Algorithms. In *USITS '97*, pages 193–206, December 1997.

[10] A. Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier. Cluster-based scalable network services. In *SOSP '97*, pages 78–91.

[11] G. R. Ganger, B. L. Worthington, and Y. N. Patt. The DiskSim Simulation Environment. Technical report, University of Michigan, CSE-TR-358-98, 1998.

[12] S. Ghandeharizadeh and R. Muntz. Design and implementation of scalable continuous media servers. *Parallel Computing*, 24(1):91–122, January 1998.

[13] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A Cost-Effective, High-Bandwidth Storage Architecture. In *ASPLOS VIII*, October 1998.

[14] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, H. Gobioff, E. Riedel, D. Rochberg, and J. Zelenka. Filesystems for Network-Attached Secure Disks. Technical Report CMU-CS-97-118, Carnegie-Mellon University, 1997.

[15] R. Golding, P. Bosch, C. Staelin, T. Sullivan, and J. Wilkes. Idleness is Not Sloth. In USENIX Association, editor, *Proceedings of the 1995 USENIX Technical Conference: January 16–20, 1995, New Orleans, Louisiana, USA*, pages 201–212, Berkeley, CA, USA, Jan. 1995. USENIX.

[16] J. L. Hennessy. The Future of Systems Research. *IEEE Computer*, 32(8):27–33, August 1999.

[17] H.-I. Hsiao and D. DeWitt. Chained Declustering: A new availability strategy for multiprocessor database machines. In *Proceedings of 6th International Data Engineering Conference*, pages 456–465, 1990.

[18] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *ASPLOS VII*, pages 84–92, Cambridge, MA, October 1996.

[19] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor—A hunter of idle workstations. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, San Jose, California, June 1988.

[20] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the Performance of Log-Structured File Systems with Adaptive Methods. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 238–251, Saint-Malo, France, October5–8 1997. ACM SIGOPS, ACM Press.

[21] R. V. Meter. Observing the Effects of Multi-Zone Disks. In *Proceedings of the 1997 USENIX Conference*, Jan. 1997.

[22] R. V. Meter, G. Finn, and S. Hotz. VISA: Netstation's Virtual Internet SCSI Adapter. In *ASPLOS XIII*, pages 71–80, Oct 1998.

[23] E. Riedel, G. A. Gibson, and C. Faloutsos. Active Storage For Large-Scale Data Mining and Multimedia. In *Proc. of the 24th International Conference on Very large Databases (VLDB '98)*, August 1998.

[24] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[25] M. Satyanarayanan. Digest of the Seventh IEEE Workshop on Hot Topics in Operating Systems. www.cs.rice.edu/Conferences/HotOS/digest/digest-html.html, March 1999.

[26] S. Savage and J. Wilkes. AFRAID— a frequently redundant array of independent disks. In *Proceedings of the 1996 USENIX Technical Conference*, pages 27–39, January 1996.

[27] H. Simitci and D. A. Reed. Adaptive disk striping for parallel input/output. In *Seventh NASA Goddard Conference on Mass Storage Systems*, San Diego, CA, March 1999. IEEE Computer Society Press.

[28] Y. Smaragdakis, S. F. Kaplan, , and P. R. Wilson. EELRU: Simple and Effective Adaptive Page Replacement. In *SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, Atlanta, GA, May 1999.

[29] N. Talagala and D. Patterson. An Analysis of Error Behaviour in a Large Storage System. In *IPPS Workshop on Fault Tolerance in Parallel and Distributed Systems*, 1999.

[30] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[31] N. E. Young. The K-server Dual and Lose Competitiveness for Paging. *Algorithmica*, 11(6):525–541, June 1994.