# Data Reliability Techniques for Specialized Storage Environments

Technical Report UCSC-SSRC-09-02
March 17, 2009

Rosie Wacha
rwacha@cs.ucsc.edu

Storage Systems Research Center
Baskin School of Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064
http://www.ssrc.ucsc.edu/

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

# DATA RELIABILITY TECHNIQUES FOR SPECIALIZED STORAGE ENVIRONMENTS

A project submitted in partial satisfaction of the
requirements for the degree of

MASTERS OF SCIENCE

in

COMPUTER SCIENCE

by

**Rosie Wacha**

December 2008

The project of Rosie Wacha
is approved:

_____
Professor Darrell D. E. Long, Chair

_____
Professor Ethan L. Miller

## Acknowledgments

I would like to thank the following people for their help and support: Darrell Long, Ethan Miller, Thomas Schwarz, Scott Brandt, Gary Grider, James Nunez, John Bent, Ralph Becker-Szendy, Neerja Bhatnagar, Kevin Greenan, Bo Hong, Bo Adler, Alisa Neeman, Esteban Molina-Estolano, Valerie Aurora, Julya Wacha, Diane Wacha, and Noah Wacha.

I also want to thank the following organizations for funding my research: UC Regents, Graduate Assistance in Areas of National Need (GAANN), Los Alamos National Laboratory (LANL), and the Institute for Scalable Scientific Data Management (ISSDM).

# Contents

# List of Figures

# List of Tables

## Abstract

Data Reliability Techniques for Specialized Storage Environments

by

Rosie Wacha

Data reliability has been extensively studied and techniques such as RAID and erasure coding are commonly used in storage systems. Real workload data is also important for storage systems research. We developed a tool to streamline the process of releasing workload data by automatically removing all non-I/O activity from software. The tool creates a Synthetic Parallel Application (SPA) that has the same I/O behavior as the original program when it is run. Next, we address reliability in the context of two specific storage environments, namely sensor networks and tape archives.

Sensor networks are made up of individual nodes that are highly constrained in power. Due to reduced storage costs, nodes are increasingly storage-based and transmitting data to a base station is reduced in order to conserve power and camouflage the network in hostile environments. We investigated the tradeoff between power and reliability for storage-based sensor networks using Reed-Solomon, XOR-based codes, and mirroring. Results show that our Reed-Solomon implementation provides higher reliability with more flexibility but with a higher energy cost. Also, the $\mathtt{XOR}_2$ reliability scheme we designed provides reliability close to that of 4-way mirroring at half the storage space overhead.

Commercial tape drives have high reliability ratings. However, many individual drives make up an entire archive. In order to achieve good write performance, data is often written in a striped pattern so that several tape drives are used to store a single file. Thus reliability is a significant concern and additional reliability techniques are often used. We investigated the performance overhead of row-diagonal parity (RDP) in the context of a large tape archive. Results show that our parallel implementation scales well for small numbers of nodes, with twice the initial write bandwidth of data when the stripe size (and number of nodes) is doubled. Future work will compare the performance of RDP with Reed-Solomon and evaluate scalability with higher numbers of nodes.

Reliability can be achieved in many ways. The SPA project can help improve storage reliability by allowing software that normally could only be tested in a single environment to be run on different hardware setups. Sensor nodes often have very limited power available due to the locations where they are often deployed. The reliability of data measured from one node is not always essential, particularly if another nearby node measured the same data. The choice of reliability technique for a sensor network must be made in the contex of these constraints. The data stored in tape archives is often never read, but if it is needed it must be there. We can sacrafice some extra hardware as long as performance is not significantly lowered. This project investigates these three areas of reliability.

# Chapter 1

# Introduction

One of the central requirements for most file systems research is good workload data. Most of the time this data is contained in a log of I/O requests, known as a trace. Collecting and releasing traces is not glamorous – file systems researchers typically only do this out of necessity. No one really wants to collect traces because it is a time consuming process and there are privacy concerns that must be addressed before the data can be released.

The first part of this project is a tool that simplifies the process of collecting traces of real parallel applications and releasing them to the public. The basic input to the tool is a parallel application that can be run on a cluster. The tool runs the application and collects traces at each node. Then these traces are automatically analyzed to detect all I/O behavior and a new program, called a Synthetic Parallel Application (SPA), is written that will perform the same I/O activities at the same times. All non-I/O behaviors in the trace are ignored and not present in the SPA. Our results show that I/O traces collected from running the SPA closely match the original traces.

The second part of this project is an investigation of reliability for two storage environments: sensor networks and tape archives. Good data reliability can be achieved by simply mirroring data on several disks. More copies of data provide more reliability. However, the hardware cost quickly grows unmanageable. Particularly in environments where traditional disks are not used or are only part of the storage system, more sophisticated reliability strategies are helpful.

Sensor nodes that store their data locally are increasingly being deployed in hostile and remote environments such as active volcanos and battlefields. Observations gathered in these environments are often irreplaceable, and must be protected from loss due to node failures. Nodes may fail individually due to power depletion or hardware/software problems, or they may suffer correlated failures from localized destructive events such as fire or rock fall. While many file systems can guard against these events, they do not consider energy usage in their approach to redundancy. We examine tradeoffs between energy and reliability in three contexts: choice of redundancy technique, choice of redundancy nodes, and frequency of verifying correctness of remotely-stored data. By matching the choice of reliability techniques to the failure characteristics of sensor networks in hostile and inaccessible environments, we can build systems that use less energy while providing higher system reliability.

Tape drives were invented by IBM in the 1950s [11]. Tape archives are still used for data that is written once and then rarely read or updated. Fast write performance can be achieved by writing data in a striped pattern. A very large file is broken up into several chunks and each chunk is written to a separate tape device. For example, a 128GB file might be broken up into 128 chunks where each is written to a tape. The time to write that file would be the time to write 1GB. Striping like this is actually done on a much larger scale. The problem is that reliability for that large file is degraded significantly when only striping. If any one of those 128 tape drives is damaged, that file cannot be reconstructed. Reliability in the context of such high performance requirements is quite challenging. For example, suppose a 1 GB tape cartridge is expected to last 30 years [2], which is a mean time between failures a little above $10^5$ hours. If the entire archive contains 4000 cartridges, we expect to see a failure every day. In high performance computing the stripe width can be very large, meaning that a single file may be broken into thousands of pieces and each piece is stored on one devices. A single parity provides some protection, but with thousands of devices it is not sufficient. We implemented a software RAID that performs mirroring, RAID 4, and Row-Diagonal Parity (RDP). We measured the performance of RDP to determine how much processing time is required to compute two parities.

In summary, we investigated reliability from several points of view in some very specific contexts. The first is that of the I/O workload and how it can affect the choice of reliability method for a storage system. The SPA provides a method for running the I/O subset of proprietary or private code on untrusted hardware. This allows more applications to be used as benchmarks for new algorithms and can help improve data reliability. Sensor networks typically have very specific constraints. Some of these are limited power, cheap hardware that is more likely to fail, and deployment in hostile environments, each of which further increase the likelihood of node failure. The choice of reliability technique must address these constraints and provide reasonable reliability in creative ways. For example, storing a mirror copy of data from one node to another far away in the network can protect the data better than storing that copy at a nearby neighbor. Lastly, tape archives have unusual access patterns and requirements. Individual hardware components are relatively reliable. In larger systems, components are often utilized in parallel to improve performance but resulting in a much lower overall system reliability. A large file can be quickly written to tape, but then that file requires all those tape drives to be functional in order to reconstruct that one file. The performance impact of adding erasure coding techniques is important and must be addressed to ensure that performance isn't degraded to near what it was without striping.

# Chapter 2

# Synthetic Parallel Applications

## 2.1 Introduction

Workload data is useful for file systems researchers, particularly for simulations of new algorithms and designs. This data is available in a variety of forms, such as traces and benchmarks. Traces can be logs of the behavior of the entire file system or as small as a single application. Traces can be quite large, especially if the application is very long-running or performs many actions. For this reason, it is difficult to create traces regularly for changing workloads and applications. Also, both because of their large sizes and the private information contained, they are difficult to share with researchers outside a particular organization.

Benchmarks are used to evaluate a system under a specific load. For convenience, benchmarks are often used many times to evaluate many different systems since there is a large cost in designing new appropriate benchmarks. Benchmarks are often designed as synthetic programs which don't perform a necessarily useful programmatic function but stress the system in a specific way to determine certain characteristics of the system, such as peak I/O bandwidth. While this type of benchmark is useful for comparing systems under specific requirements, they don't capture system metrics under "typical" conditions of user applications on a system. The ideal situation would be if we could release real user applications as benchmarks that then could be used to compare systems, which is conceptually the goal of the first part of this project. We created a tool that creates an I/O skeleton program, called a Synthetic Parallel Application, from a real scientific program. This work was completed while working at Los Alamos National Laboratory.

The second long-term future goal of this project is to create accurate synthetic I/O workload models of real workloads. We plan to extract the essential properties of the real I/O workload into a hidden Markov model. The model will contain access pattern information and time durations representing the time required to access file segments. These two types of data will be modified to create new synthetic workloads that represent how the program would behave on different systems. The models could be modified and extended to larger clusters, allowing even greater flexibility and accuracy than is possible using the basic synthetic applications. This part of the project is proposed work that hasn't been completed.

## 2.2 Related Work

Workload studies are published regularly as previous workloads become outdated and I/O patterns change [6, 43, 30]. Typically, a sanitized version of the trace data is released with these publications so that researchers, as well as storage system developers, can simulate new I/O system algorithms with realistic access patterns. These types of trace analysis papers are published infrequently because the amount of work involved in gathering accurate traces, finding useful information, and removing private data is so high. Perhaps traces would be published more often if the process of obtaining and sanitizing them was more seamless.

There are several projects that focus on more accurate accurate trace collection and replay. Buttress [4] deals with timing accuracy for I/O benchmarking and replay. Specifically, the tracing tool preserves extreme burstiness of asynchronous I/O. This work allows us to create models from I/O traces that we can expect to be accurate. Aranya et al. [5] implemented Tracefs, an actual file system that captures traces at the VFS level. Auto-pilot [49] is a framework for producing accurate, reliable, and informative benchmarking results.

Other work uses I/O tracing and analysis to detect, isolate, and correct problem. Pinpoint [12] uses statistics to correlate components with failed requests to pinpoint the cause of faults. Magpie [7] is a tool for analyzing workload behavior that detects events and correlates them by application. Magpie is implemented as a tool that runs online using a low-overhead event tracker, a request parser which extracts requests from actual event logs, and an event schema which groups related event types. Triage [27] uses a controller and adapts to system and workload changes. Proteus [26] does automatic detection of workloads that contend for certain resources and uses control theory to schedule requests. Façade [33] improves I/O efficiency by throttling I/Os to avoid storage device saturation. Black box [3] models system components as black boxes. The algorithm can identify an individual node which caused a performance delay without modifying applications.

Several research projects have looked at the process of getting performance information from parallel I/O subsystems. Madhyastha [34] developed an I/O classification system using Hidden Markov Models (HMMs). The motivation for the project was to classify I/O access patterns online in order to make system decisions to improve performance. An individual HMM is created and trained for each program. There is significant overhead in introducing a new program but low overhead to run the program multiple times. However, the authors do not present overhead results in this paper. Event duration times are ignored in the paper in an effort to remain platform independent, but this area is listed as future work and is part of the motivation for our research. Mesnier et al. [36] attacked the problem of synchronization of nodes across a large cluster. This project automatically determines data dependencies between nodes in I/O traces. Their solution uses a technique called I/O throttling where a single node is slowed down during replay in order to see which other nodes stop making progress and wait on that one node. This work does not address other parallel I/O issues such as accurate node clocks or barrier detection. Hong [24] studied two characteristics of I/O workloads and two techniques for creating synthetic traces. He verified that disk traffic is self-similar and that ignoring some long-range dependence does not significantly affect disk behavior. Self-similarity in disk traces motivates the use of Markov models for I/O modeling. See Section 2.5 for more details.

Our work is on the extreme end of simplifying the process of gathering and releasing traces on clusters. We analyze source code with the goal of removing all sensitive information that is unrelated to the I/O behavior of the application. The tool is tied to a single programming language for output code, but could be extended to a more general system. The input can be any language, since the true input to the tool is a system call trace.

## 2.3 How to Create the SPA

The purpose of a Synthetic Parallel Application (SPA) is to mimic the I/O of a *single run* of a real application. Note that this is a limitation particularly for nondeterministic programs, but was chosen for simplicity and to reduce the performance overhead of creating the SPA. It may possible to extend this work to perform several runs and create the SPA by averaging the timing and I/O data into a single SPA, but this still provides only a single run for performance analysis.

The SPA can be run as a way to measure the I/O requirements of user applications and our goal is to improve accuracy with respect to timing and data access needs. The Synthetic Parallel Application is created from system call logs of the original application. The Unix application *strace* is used to capture the I/O system calls made at each node that the original application runs on. The process of creating a SPA involves a few steps that will be discussed in detail in the next sections. First, we capture logs of the I/O events the original application performed at each node. These logs are analyzed to create a global structure of the program that does not depend on individual node usage. Lastly, that structure is converted into a real, runnable, parallel program, which we call the SPA.

### 2.3.1 Capture Logs of I/O Events at Each Node

The logs are generated by the Unix tool *strace*. Strace captures and displays system calls that are made by an application. Each call log that strace creates is a list of requests made to the kernel. Strace creates a log at each node the parallel application is running on. The data in the logs needs to be generalized to get a full representation of the entire parallel program.

The first step in creating the SPA is to create a description of the events of the parallel application we are modeling. We run the application under strace, carefully ensuring that all I/O events generated by the application at each node are captured into a log file. We capture and reproduce the following I/O events:

- open
- close
- read
- write

### 2.3.2 Create the Event Queue

The event queue is an internal representation of all events read from the trace. Each I/O event in the event queue contains the following fields:

- Node ID
- Timestamp at start of each event, microsecond accuracy
- Timestamp at end of each event, microsecond accuracy
- Name of event
- Options of event (e.g. file name, file handle, size of I/O)

The system call logs that strace creates contain a low-level representation of I/O events containing local data that we need to generalize to create a full representation of the entire parallel program. For example, the `open` system call uses the file name of the file to be opened, but subsequent `read, write,` and `close` system calls are specified by an integral file handle. These file handles are unique at each node as long as the file remains open. Therefore, we must track file handles in order to know which I/O events correspond to which files. In order to track file handles within an strace log, we maintain the set of currently open file handles matched with their file names.

Before an I/O event is added to the event queue, we verify that the event is *sequentially valid*. Sequential validity means that events occur in a realistic sequence. For example, a file handle cannot be written to before the file is opened. We ignore any events that occur without the required setup. These types of events are rare and only show up at the beginning of our traces. We believe they are MPI-specific files that are opened before strace starts tracing.

### 2.3.3 Barriers

A barrier in a parallel program forces every node to wait to continue running until all nodes have reached the barrier. This effectively synchronizes the nodes so that any actions will begin at roughly the same time. We determined that placing a barrier at the start of the SPA improved the results of the SPA, but we will need to look into detecting additional barriers in the original application. This is particularly important for longer running applications. We could use a heuristic to improve barrier detection in combination with monitoring network traffic. When MPI has a barrier, the nodes need to synchronize and the notification for all nodes to start running again should be detectable. By strategically placing barriers in our SPA, we can better match the timing of the events in the original application.

### 2.3.4 Writing the Parallel Code

Creating code from the event queue is a straightforward process because of the careful way the event queue is constructed. Specifically, all error handling is done before creating the event queue, and each event is valid sequentially (e.g. no read occurs without a prior open to the

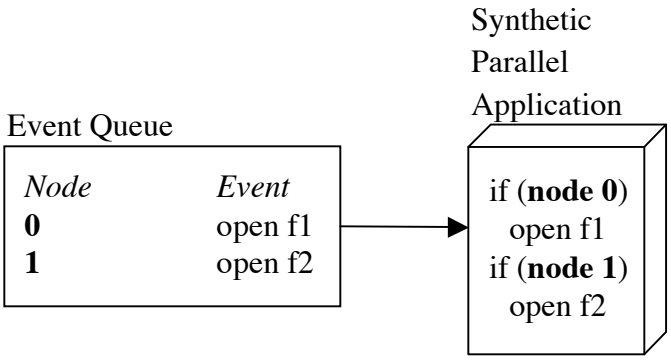


Figure 2.1: Event queue events are transferred into the SPA by placing conditionals around the block.

file). Each event is simply transcribed into the parallel code, with a conditional to specify the node that performs that event, as shown in Figure 2.1. The events are ordered the same way as they were in the original application. It would be possible to reorder events in the event queue before processing into code, but that is not the goal of this particular project. Our goal was to closely mimic the behavior of the scientific application.

### 2.3.5 Limitations

The SPA generator described has several limitations. Because we only trace a single run without analyzing source code in any way, we ignore all nondeterminism. An example of this is parallel make which parallelizes source code building by breaking up components that don't depend on each other and executing the builds in parallel. The code compiled on each node can vary dramatically from one run to the next, depending on how quickly each component completes and whether or not any nodes are overloaded. One way we could address this limitation is to perform multiple runs of the program. The first step would be to perform some basic statistical analysis to verify that certain parallel programs do not exhibit this type of nondeterminism. In the case of a truly nondeterministic program, the best solution may involve looking at source code.

Asynchronous I/O is another issue that we did not investigate. Asynchronous I/O means that the application does not wait for the I/O to complete so that it can immediately continue running other work. Processing is likely to interleave more fully with I/O because the process doesn't have to wait for I/Os to complete before executing more instructions.

Another type of I/O that is not included in the SPA is prefetching. This type of I/O is executed automatically by the file system based on previous access patterns and expected future access. The goal is to preload files from the disk that an application might need in the future in order to make I/Os more sequential and reduce waiting time in the future. Our SPA will include

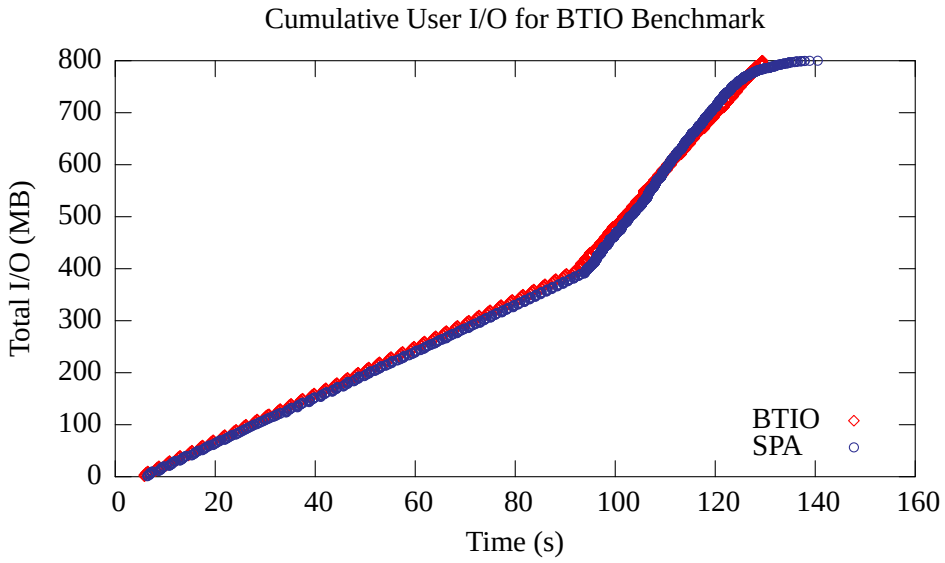


Figure 2.2: BTIO results; the BTIO application contains a compute and write phase followed by a read and verification phase.

all I/O that was requested by the application, and thus a future run may differ due to lower level I/O policies. Caching is another aspect of file system behavior that we have not investigated. Because we trace at the application level, we ignore any lower level disk activities such as prefetching and caching. We do not detect whether a read is serviced by the disk on-demand or earlier. The best way to create an accurate SPA in this case would be to turn prefetching off as much as possible when creating traces of the original application. This will help to make the timing of I/O events more system independent. The SPA can then be used on the same system to evaluate different prefetching methods.

## 2.4 Results

To evaluate the validity of the SPA, we compare the workloads of two original applications and SPAs generated based on those applications. All programs are run under strace and the resulting strace output files are analyzed to create SPAs as described in the previous section.

We ran the tests on a 164 node Linux cluster at Los Alamos National Laboratory. The first benchmark is the BTIO benchmark [48] from the NAS Parallel Benchmarks version 3.2.1. The results of this test are in Figure 2.2. We found that for this benchmark, the SPA is able to match the original application fairly closely. The overall behavior of the SPA matches, with a higher slope at 95 seconds where the benchmark switches from calculating and writing out results to reading them back for verification.

The second benchmark is the MPI-IO Test benchmark, developed at LANL. The re-

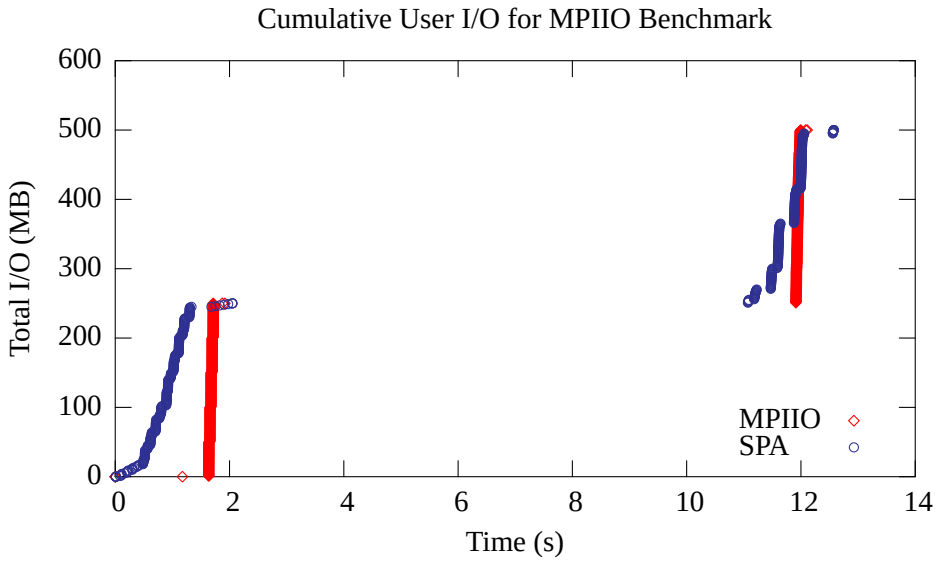


Figure 2.3: MPIIO results; the benchmark is configured to perform a series of writes followed by a 10*s* sleep and another write phase.

sults of this test are in Figure 2.3. Because this benchmark ran for a short period of time, the results show more clearly the differences between the original application and the SPA. It is not realistic to expect the I/O timing to match exactly for these two application runs. Even when running one application twice in a row, the timing will differ slightly due to resource usage and contention. The important difference that we need to address is that the SPA actually performs I/O at a different rate than the original application. The slope of the original application is steeper, meaning that I/Os are performed more quickly. This is also slightly visible in Figure 2.2 where the far right tail of the graph for the SPA extends past the original BTIO application. We may analyze this behavior as future work.

### 2.4.1 Overhead of Strace

Another area of future work is to determine the overhead of strace for some of the available benchmarks. There is an strace option where you can estimate overhead and ideally improve the traces by accounting for the overhead in the traces. It would be useful to try that technique and see if it results in a lower measured overhead for the application. The strace option is supposed to amortize the overhead across the entire run of the program, so that the timestamp of each event is slightly earlier, and the strace more accurately represents a run of the application without strace.

## 2.5 Future Work: Analyzing Traces with Hidden Markov Models

We have started looking at extending our work to model application I/O behavior using hidden Markov models (HMMs). The main motivation is that a better representation of I/O will allow more flexibility in replaying that I/O on future systems. Once we have a model that we believe to be accurate, we will be able to modify it in order to create new workloads based on existing data instead of only matching available workloads.

### 2.5.1 Background: Hidden Markov Models

A Markov chain is a finite state machine with probabilities associated with the transitions between states. This means that we have a graph of states or configurations and the probability of moving from one state to another is known. The *Markov assumption* is that the probability of each state and output at that state depends only on the previous state, meaning that there is no long term history of influence. Hong [24] showed this to be a reasonable assumption for I/O workload access patterns.

A Markov chain is a model of an event stream where the states visited in the model are uniquely determined by the output sequence of events. In other words, there is only one possible Markov chain that can be created from a specific event stream. Creating this type of model from data is straightforward. The state structure must be chosen ahead of time; this is the non-automated part of creating the model. Measuring the probabilities is automated in the following way. Consider the transition from state $i$ to state $j$. Calculate the total number of times this transition is taken and divide that by the total number of times transitions are taken from $i$ to any state (including $j$). This gives the probability that state $j$ follows state $i$. Repeat this process for all transitions in the model.

Hidden Markov models [41] are a generalization of Markov chains where multiple states representing the same output symbol are allowed. This allows the property that an HMM state sequence cannot be uniquely determined based on an output sequence, and thus the state sequence is *hidden*. The main challenge in using HMMs is building the set of states that best represent the output data sequence to be modeled. Several properties of HMMs make them particularly well suited to the problem of performance prediction.

- Multiple states representing the same visible output. This allows more than one expected value for the average access time for a particular file segment. For example, the first time a file segment is accessed, it is likely to take a longer time to fetch it from disk than the second time if it is being accessed from memory. Other possible reasons for varying times are RAID policy changes or file migration onto a different physical hard drive.
- The Markov assumption greatly simplifies the model by not remembering history. This may be a limitation on the prediction capability, but simplicity is preferred when possible.

### 2.5.2 Using Hidden Markov Models

This work addresses the area of benchmarking storage systems. We are interested in improving the ability to make performance predictions based on available I/O workload knowl-

edge. The basic idea is that by creating a representative model of a workload, we can extrapolate how that workload would look on other systems.

Consider a scientific application that was built to run on 32 nodes. Suppose we are interested in the I/O behavior that application would perform if it could instead run on 64 nodes. Our goal is to create an accurate model of the scientific application from running it on 32 nodes. With that model, we will extrapolate how the I/O would look if we could run on 64 nodes. These synthetic workloads should be accurate representations of how the original scientific application would behave if it could fully utilize the additional hardware available in the new setup. Then we can simulate the performance of the synthetic workload under various I/O systems algorithm changes, such as caching or prefetching. Note that this solution makes several strong assumptions. The first is that I/O is the bottleneck of the application and changing the number of computation nodes will directly impact the size and number of I/O requests. We also assume good scaling of the application, meaning that the problem being solved can be subdivided in arbitrary ways and has fine granularity. This may not be valid for many applications, but still provides a useful way to predict I/O behavior even when a similar program would not realistically solve the same problem.

We will create synthetic workloads based on hidden Markov models (HMM) of real workloads. The models specify read and write events and the duration of those events. The optimal model will classify an I/O workload in such a way that it can be used to accurately recreate the same load on the storage system. The model can also be modified to represent similar workloads of interest, such as increasing the amount of load each compute node places on the storage subsystem.

Hidden Markov models are Markov models with hidden states, meaning that some states do not correspond to actual events in the data. These hidden states allow the model to capture system behavior that we don't measure directly because we can't observe those events or they are difficult to observe. We leverage previous research [34] which used HMMs to characterize workloads using traces. This work created an HMM for each file accessed, where each state corresponded to a segment of that file. This is implemented as a two dimensional array where each file segment has a probability of accessing every other file segment. In practice, many of those probabilities are 0. If a file is only accessed sequentially, then the only nonzero probability is to the next block. In order to reduce memory usage, they consolidate any states where multiple segments are only accessed sequentially into a single state with a larger segment size.

Our work will add features to improve the workload accuracy of the model. The first addition is a measure of time, in order to capture event durations. Using these event durations, we also plan to determine a way to predict the behavior of a particular workload on a different system setup.

The HMM will have a few different types of states. The first layer will be states corresponding to file segments. These segments represent the granularity with which I/O requests are made. We also plan to add *wait* states to incorporate event durations into our HMM. Event durations can be incorporated into the HMM as separate states. Each file segment (state) will have a probability distribution for which segment will be next accessed as well as for the duration of the current state. Since I/O durations are not completely deterministic, we will implement the

duration as having an average and standard deviation measured from multiple runs of the same workload. We will use confidence intervals to get the percentages for each value in a range of possibilities around the average value. The additional runs will introduce significant overhead in creating the model, but this step is necessary to add sufficient certainty about the expected durations on a certain system.

We need multiple states for each observable output because there are multiple ways to access a file. If a file is accessed sequentially, the state sequence should be different than if it is accessed randomly. This will be possible by having additional hidden state sequences that represent different types of access patterns.

With the time of events measured, we hope to determine how to predict how the workload will change on a new system. For example, if our trace was taken on a 64 node cluster, we want to predict how different the workload will be when run on a 128 node cluster. This assumes that the original program will scale with the addition of more nodes. We could incorporate nonlinear scaling, so that tripling the number of computation nodes would only produce double the amount of I/O work, for example. In the simplest case, we will assume linear scaling of I/O requests in size and frequency.

Here is our basic algorithm for creating an HMM of a workload:

```
Repeat the following until delay values tend to converge:
    Run program to get an I/O trace
    For each I/O event in I/O trace:
        If (this access and previous access) is unique
            Create a new state
            Initialize average delay (to delay for this access)
            Initialize number of accesses (to 1)
        Else (i.e. this access and previous was seen before)
            Put delay into average, increment number of accesses
```

This algorithm was designed in order to improve memory efficiency by not storing transitions between states that are never taken in practice. Because every I/O event is added to the HMM along with its transition, all transitions that have probability greater than zero will necessarily be present in the model.

### 2.5.3 Predicting Performance

The goal of this section is to determine viability of a new system setup before acquiring and configuring the actual hardware based on workloads known to be important. The idea is to have a detailed description of the system in various dimensions, such as number of compute nodes and hard drives, and run the real workload on some of these dimensions. By keeping most dimensions fixed and varying a single dimension at a time, we can measure the impact of that dimension on workload performance. Maybe there is a machine learning technique that would apply here.

### 2.5.4 Determining Bottlenecks

One goal of workload analysis is to determine which computer subsystem is the bottleneck of a given application. The bottleneck of any system is clearly an area that should be focused on when adding new resources.

Consider the following experiment. Run an application with 1 node, 2 nodes, 4 nodes, and 8 nodes. Plot the I/O performance such as throughput in MB/s, network throughput, and CPU time. The data plotted should be the left edge of a logarithmic curve. The expected bottleneck is the curve for which the slope changes the fastest or where the slope is the smallest first.

### 2.5.5 Parallel Accesses

Do we want to have separate HMMs for each node and then combine them? Or does it make more sense to have it combined with the ability to reconstruct individual node behavior? This depends on whether we are most interested in individual requests or the requirements of the program as a whole. I tend to think that combining into a single model makes more sense, so that is what I have chosen for now.

There are some issues that we will address with respect to the parallel programming model. We will create a single model for the entire program, thus combining parallel thread behaviors into a single control model. Our model will keep track of whether or not multiple nodes access the same segment of a file. This will be done by keeping an individual I/O access count for each CPU node.

### 2.5.6 Trace Format

The obvious approach is to continue analyzing strace I/O traces. One clear benefit is the ease in which straces are produced on any Linux-based operating system. One drawback is that there is overhead associated with running an application under strace. Another drawback is that there is a limited amount of information available at the system call level. System calls provide the interface between user-space and kernel-space, thus we are tracing right between the application and hardware.

It may be desirable to capture traces at a lower level than strace allows in order to get more detailed information about what the hardware is actually doing when processing requests. In particular, knowledge about whether a particular file segment is stored in memory would be useful in predicting the effect of increasing total memory. This might be possible to determine by tracing the calls to mmap (an optional field for strace), but I have not investigated this approach in detail. It might be necessary to trace at other levels in order to predict performance changes with respect to certain hardware.

At a higher level, we also miss some information that would be very useful in mirroring behavior. Specifically, we do not have access to any application logic and algorithms. This is desirable in the context of producing code that doesn't contain proprietary information. On the other hand, it makes it much harder to deduce and replay certain application behaviors such as barriers.

## 2.6 Conclusion

We implemented a tool that creates synthetic parallel applications from real applications. The SPA is created from I/O traces of the original application, thus reducing the likelihood of inadvertently putting any code directly from the original application into the synthetic one. The tool was shown to work on several benchmarks by creating code that mimics the I/O from those benchmarks accurately to the granularity of less than one second.

We have considered many areas for future work. One topic is to look at modeling application I/O behavior using hidden Markov models. An application model could then be systematically modified in order to produce more general behavior. For example, the number of files accessed by the program could be doubled. Even more ambitious changes could be made, such as modifying the model to require twice as many nodes. This requires certain assumptions about scalability of the underlying algorithm, but can be quite useful in testing future cluster and storage requirements.

# Chapter 3

# Reliability in Sensor Networks

## 3.1 Introduction

The availability of inexpensive gigabyte-scale local storage on sensor nodes [37] and the high cost of radio operations relative to storage operations are enabling sensor nodes that store data locally in between data collection events [35]. Storage-based sensor networks are used to monitor volcanoes, battlefields, habitats, seismic events, traffic, and the stability and integrity of engineered structures such as buildings and bridges [47, 13]. However, the difficulty of gathering data from sensor nodes in hostile and inaccessible environments has also made it harder to deploy base stations that accumulate nodes' data. Base stations installed with sensor networks are easily detected in contested land areas such as borders, and are an obvious target for network disruption. Base stations in inaccessible and natural environments are single points of failure because they may suffer from power outages or malfunction, causing data loss; in a volcano-based sensor network, "[f]ailures of the base station infrastructure were a significant source of network downtime" [47]. Some networks try to avoid this problem by deploying multiple base stations or specialized storage nodes [46], increasing the both the likelihood of the detection of the network, and the system cost. Data loss in centrally-controlled sensor networks is likely to be more severe because nodes do not retain the observations they have already uploaded to the base station. Moreover, a base station cannot easily transmit data to a receiver when none is nearby, as is often the case in remote environments. Such environments are better suited to occasional data collection, requiring nodes to reliably maintain their data over long periods of time.

Individual sensor nodes typically suffer from relatively high failure rates, as compared to traditional storage devices. Moreover, sensor nodes are more likely to suffer *correlated* failures due to environmental dangers. Individual failures may be caused by battery depletion, hardware or software errors, or physical damage. In contrast, correlated node failures may be caused by larger-scale physical damage caused by a destructive event such as flood, rock fall, or fire. Unfortunately, the latest data from destroyed nodes is often the most valuable because it may record details of the event, making it even more important for the observations gathered by the nodes to survive their destruction. However, it is also imperative that sensor nodes create and maintain back-up copies of their data without overwhelming their energy budgets.

We discuss the tradeoffs between energy and reliability in sensor networks that store data for long periods of time: weeks to years. These tradeoffs can be made in three separate areas: redundancy techniques, choice of nodes which store the redundant data, and frequency of integrity checks on the remotely stored redundant data. We do not expect the energy expenditure of reliable storage in sensor networks to be less than the energy expended by nodes to upload their data to a base station; rather, our goal is to make sensor network storage much more reliable by increasing the likelihood that sensor data survive despite individual and correlated node failures. By providing energy-efficient storage operations, sensor networks can more easily provide raw data, instead of aggregated and representative values, to their intended audience, potentially facilitating more robust forecast and analysis models.

We assume that the network is comprised of sensor nodes severely constrained in power, storage, and processing. We also assume that nodes have limited radio range, so communication with distant nodes requires multi-hop routing. Since our research is primarily concerned with energy-reliability tradeoffs, we fold the costs for interference and retransmission into the cost for transmitting data between nodes. We assume that each node has a battery-backed RAM for buffering data and NAND flash memory for persistent storage [35], though new non-volatile memory technologies such as phase change memories may further simplify the architecture [29].

## 3.2 Issues in Reliability

Analyzing tradeoffs between energy usage and file system reliability depends on making good choices for redundancy techniques, nodes for remote storage, and frequency of checking integrity of redundant data, while considering the high failure rate of sensor nodes and the likelihood of occurrence of correlated failures [38].

### 3.2.1 Redundancy Techniques

As with traditional file systems, sensor nodes may use either mirroring or erasure coding to store data reliably. Transmission costs dominate the energy usage at a node compared to simply storing the data locally. Transmitting data costs two hundred times more energy [35] than storing data locally. As a result, due to the relative position of nodes and the base station, the transmission cost of mirroring data to another node may be lower than that of uploading data to a base station. This is specifically the case when the transmitting node is in the center of the network and the base station is installed at the edge of the network, or vice versa. The storage overhead of mirroring is also very high: tolerating $n$ failures requires the system to store $n+1$ copies of the data. In contrast, processing costs dominate energy usage for erasure codes.

We compared the performance (energy consumption expressed in mJ and throughput expressed in MB/s) of encoding using Reed-Solomon (RS) codes [40] based on GF($2^8$) [20] to XOR-based codes [50, 22] on an ARM9E 400 MHz processor that consumes 94 mJ/s [1]. The first column in Table 3.1 represents the RS code implementation for parameters $(n,m)$, where $n$ is the number of data nodes, and $m$ is the number of parity nodes. RS codes were implemented as table lookups, where each multiplication requires two lookups. Each lookup

Table 3.1: Energy expenditure of erasure codes in mJ/s and throughput in MB/s.

| Code Size | Energy Expenditure (mJ/s) | | Throughput (MB/s) | |
|---|---|---|---|---|
| | **RS** | **XOR** | **RS** | **XOR** |
| (5, 3) | 3.515 | 1.205 | 2.674 | 7.798 |
| (6, 2) | 3.133 | 0.6 | 3 | 15.654 |
| (9, 3) | 4.82 | 0.524 | 1.95 | 17.953 |
| (10, 2) | 3.92 | 0.653 | 2.4 | 14.4 |
| (17, 3) | 5.193 | 0.588 | 1.81 | 15.99 |
| (18, 2) | 4.36 | 0.589 | 2.156 | 15.972 |
| $\texttt{XOR}_1$ | — | 0.74 | — | 12.76 |
| $\texttt{XOR}_2$ | — | 0.75 | — | 12.72 |

table is 256 bytes in size, consuming 512 bytes of memory. The second column in Table 3.1 represents the most fault-tolerant XOR-based codes for the same parameters. These codes have the storage efficiency of $n/(n+m)$. The last two rows present the performance of highly fault-tolerate XOR-based codes that we developed. The $\texttt{XOR}_1$ code we designed is an instance of a WEAVER code [22] that tolerates two-node failures.

Reed-Solomon codes consume 3–10 times more energy than XOR-based codes due to more complex finite field calculations [22, 21] with our implementation, but provide higher reliability (*e.g.*, a $(5,3)$ RS code can tolerate *all* three-node failures but an XOR-based $(5,3)$ code may only be able to tolerate at *most* three-node failures). However, it may be possible to tolerate some node failures without losing data because very closely-located sensor nodes may be observing similar phenomena. In order to tolerate correlated failures, closely-located sensor nodes must spread their information over a large physical area. The energy expenditure of $\texttt{XOR}_1$ and $\texttt{XOR}_2$ schemes is comparable to most XOR-based codes but better than that for RS codes. We are currently exploring the suitability of several less processor intensive XOR-based codes, based on the research done by Wylie and Swaminathan [50], to sensor networks. Future work will also compare the amount of energy required to transmit data to neighboring nodes with the computation cost for generating code data. The transmission cost may significantly dominate the compuation cost such that the code choice is less relevant.

### 3.2.2 Node Choice

The impact of correlated failures caused by localized damage can be mitigated by spreading redundant data over a large physical area. However, there is a cost in energy to send the data further away. Even when the number of nodes in two redundancy groups is the same, the choice of the layout of nodes significantly impacts both reliability and energy requirements.

Mirroring alone is energy-consuming for making sensor network storage reliable. In order to reduce energy expenditure, it may be better to mirror data only to nearby nodes and to use erasure codes for nodes that are further away. This approach can quickly replicate data nearby, guarding against individual node failure, and can use widespread replication to protect

| A | B | C | D | E |
|---|---|---|---|---|
| B⊕C | C⊕D | D⊕E | E⊕A | A⊕B |

Figure 3.1: $\texttt{XOR}_1$ redundancy method for a 5-node sensor network.

| A | B | C | D | E |
|---|---|---|---|---|
| B⊕C | A⊕D | A⊕E | A⊕C | A⊕B |
| D⊕E | C⊕E | B⊕D | B⊕E | C⊕D |

Figure 3.2: $\texttt{XOR}_2$ redundancy method for a 5-node sensor network.

against correlated node failures. Systems such as OceanStore [42] use erasure codes to tolerate relatively large numbers of failed nodes; we plan to do the same for making sensor network storage reliable. Our file system has the advantage of using less-expensive XOR-based codes in place of RS codes by carefully placing redundant data on particular nodes. When using a $(5,3)$ XOR-based code, by arranging data so that the "fatal" three-node sets cover a large physical area, the sensor network can gain nearly all benefits of RS codes with the computational cost of XOR-based codes. By choosing several nodes located somewhat far from each other, the system is more safe because they are less likely to suffer correlated failures simultaneously. The system could provide additional reliability by choosing some very distant nodes as part of its redundancy group.

We designed two XOR-based backup layouts which are called $\texttt{XOR}_1$ and $\texttt{XOR}_2$. In the $\texttt{XOR}_1$ scheme, each node stores its own data and the XOR of data from two other nodes. The number corresponds to how many copies are stored as a parity on each other node. Figure 3.1 shows the 5 node case where nodes and their data are specified by the letters A-E. Each column represents all data stored at a given node, including redundant data as backup for other nodes. Node A stores its own data and $B \oplus C$; node B stores its own data and $C \oplus D$; node C stores its own data and $D \oplus E$; node D stores its own data and $A \oplus E$; and node E stores its own data

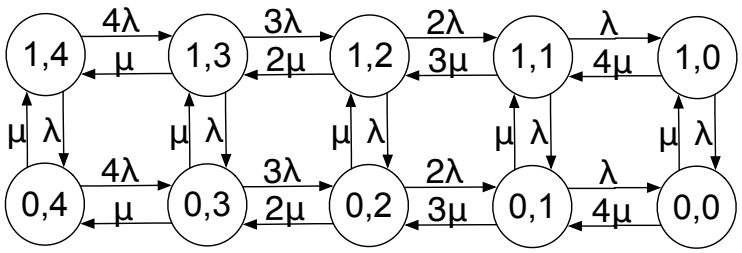


Figure 3.3: Markov model, where $\lambda$ and $\mu$ are the average failure and repair rates, respectively, of exponential distributions.

and $A \oplus B$. In the $\texttt{XOR}_2$ scheme, each node stores its own data and data from four other nodes as two-node XORs. Figure 3.2 shows the same 5 node case where each column represents all data stored at a given node, including redundant data as backup for other nodes. Node A stores its own data and $B \oplus C$ and $D \oplus E$; node B stores its own data and $D \oplus E$ and $A \oplus C$; node C stores its own data and $A \oplus E$ and $B \oplus D$; node D stores its own data and $A \oplus C$ and $B \oplus E$; and node E stores its own data and $A \oplus B$ and $C \oplus D$. The storage overhead of $\texttt{Mirror}_4$ is four times that of the original data set. The storage overhead of $\texttt{XOR}_1$ and $\texttt{XOR}_2$ schemes is, respectively, two and three times the original data set. Figure 3.4 shows that $\texttt{XOR}_2$ delivers availability similar to $\texttt{Mirror}_4$, but at a lower overhead. $\texttt{Mirror}_4$ can tolerate at most four node failures, while $\texttt{XOR}_1$ and $\texttt{XOR}_2$ schemes can, respectively, tolerate at most two- and three-node failures. Markov models provide good approximate analysis, but do not work well for "irregular" XOR codes or for systems that experience correlated failures; these are better suited to simulation. However, they provide an approximate initial analysis.

We use a simple Markov model to analyze the availability of the $\texttt{Mirror}_4$, $\texttt{XOR}_1$, and $\texttt{XOR}_2$ schemes. Figure 3.3, depicts 4-way mirroring, but can easily be generalized to an *n*-node redundancy group. The transitions are exponentially distributed with mean failure rate $\lambda$, and mean repair rate $\mu$. For simplicity, we let $\rho = \lambda/\mu$. When $\rho \approx 1$, this means that failures are replaced at the same rate that failures occur. On the other hand, if failures are not replaced as quickly as they occur, $\rho \approx 0$. In a real sensor network, we expect replacements to occur infrequently, so that $\rho \approx 0$. State $(0,0)$ represents the failed state.

The availability of a node's data when $\texttt{Mirror}_4$, $\texttt{XOR}_1$, and $\texttt{XOR}_2$ schemes are used to create redundancy in the sensor network are given by:

$$\texttt{Mirror}_4 = 1 - \frac{\rho^5}{(1+\rho)^5},$$

$$\texttt{XOR}_1 = \frac{10\rho^2 + 5\rho + 1}{(\rho+1)^5}, \text{ and}$$

$$\texttt{XOR}_2 = 1 - \frac{5\rho + 1}{(\rho+1)^5}.$$

Table 3.2: MTTDL, in hours, for $\texttt{Mirror}_4$, $\texttt{XOR}_1$, and $\texttt{XOR}_2$ schemes with and without repair.

| | $\texttt{Mirror}_4$ | $\texttt{XOR}_1$ | $\texttt{XOR}_2$ |
|---|---|---|---|
| MTTDL with repair (h) | $4.87 \times 10^{11}$ | $2.42 \times 10^{6}$ | $6.50 \times 10^{8}$ |
| MTTDL w/o repair (h) | 4932 | 1692 | 2772 |

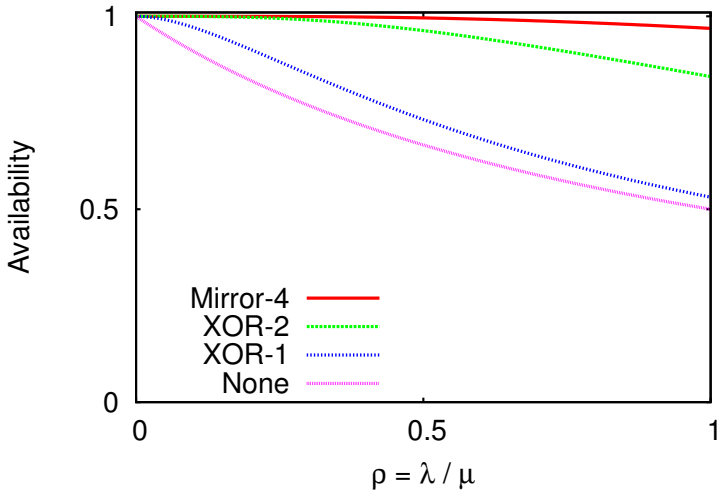


Figure 3.4: Data availability of $\texttt{Mirror}_4$, $\texttt{XOR}_2$, $\texttt{XOR}_1$, and no redundancy.

These availability models are simple and assume that the nodes may be repaired. In the case of no repair, steady-state does not exist and so the system must be modeled using differential equations. These equations quickly become unmanageable, and so a better solution would be to use simulation, which has the additional advantage of being able to model correlated failures.

Modeling mean-time-to-data-loss (MTTDL) is easier, and uses the same transition matrix that would be used for modeling with differential equations. We assume that both failures and repairs are exponentially distributed. We solve all these models by building a transition matrix $M$, as discussed by Schwarz [44], and computing

$$MTTDL = -[1,1,1,\ldots,1] \cdot M^{-1} \cdot [1,0,0,\ldots,0].$$

Table 3.2 presents the MTTDL for $\texttt{Mirror}_4$, $\texttt{XOR}_1$, and $\texttt{XOR}_2$ schemes, with and without repairs. For this example we assume that nodes are organized into five-node redundancy groups and choose $\rho = 5.56 \times 10^{-3}$, which assumes that failures occur on average every 3 months and nodes are repaired, on average, in 12 hours.

### 3.2.3 Frequency of Integrity Checks

Regardless of the technique used to generate redundancy, each sensor node must periodically check to ensure that its back-up data is still being stored correctly. If a node replicates

its data to distant nodes, then its integrity checks and their responses must also travel further, thereby expending more energy. Moreover, the more frequently a node checks the correctness of its back-ups, the more energy it expends. Furthermore, additional energy is expended at the responding node which must generate a signature and transmit it back over multiple hops. However, in a system where node failure is frequent, it is necessary to detect small problems before they grow bigger and cause data loss. It may be energy-wise to allow small problems to become a little bigger, but not fatal, because the energy cost to restore redundancy is sub-linear. We are currently exploring the energy tradeoffs between more frequent integrity checks with that of the overall reliability of the system.

We plan to use algebraic signatures [45] to verify the correctness of remotely-stored redundant data. Although algebraic signatures are not cryptographically secure, they change in response to small changes in the data from which they are generated. Moreover, they can be used in conjunction with XOR or RS codes to ensure that a set of returned signatures is consistent. An algebraic signature operation requires a node to calculate a function on its own piece of stored redundant data, thereby, generating a small (4–8 byte) signature. When combined, these signatures obey the same relationship as the data from which they were generated; if the signatures form a valid code word in the XOR or RS scheme, the underlying data is highly likely to be consistent as well—the chance of agreement with an underlying error is approximately $2^{-b}$ for a $b$-bit signature.

## 3.3 Optimizations

We are currently researching several optimizations that can help reduce energy requirements for making sensor network storage reliable. For example, it may be possible to piggy-back integrity check messages and responses on other network traffic such as “hello” or “ack” messages or on other traffic related to updating routing and neighborhood tables. Such piggy-backing has the potential of reducing transmission cost because integrity check messages are relatively small and the marginal cost of including additional information in another message is minimal. In order to reduce energy expenditure of reliability, some redundancy can be generated at remote nodes to reduce the total volume of data that must be transmitted over large distances. Sending all data to a remote node and letting it distribute it to its nearby neighbors may also be more energy efficient than the originating node distributing its data to all nodes. Energy expended in transmission can be further reduced by using some of the “routing” nodes or the intermediate nodes in the path between a source node and its destination back-up node.

## 3.4 Related Work

Koushanfar, *et al.* [28] identify computing, storage, communication, sensing, and actuating as resources and propose backing-up a resource running low with one that is abundantly available. However, the application software that computes resource availability may itself consume lots of energy. The solutions presented by Kamra, *et al.* [25] and Lin, *et al.* [31] are designed for sink-based network architectures. Although our solution is applicable to both dis-

tributed and centrally-controlled networks, we assume a distributed network architecture without a sink. Lin, *et al.* [32] use decentralized fountain codes to introduce redundancy into the network. Ghose, *et al.* [19] present a Resilient Data Centric Storage (R-DCS) scheme to reduce energy consumption while increasing resilience to node failures. Schemes presented by authors [32, 19] require a complete picture of the network. This may not always be possible with *ad hoc* networks [31]. In contrast, we assume nearly homogeneous nodes with no single point of failure. This assumption may not hold well in *ad hoc* networks deployed by dropping nodes from an airplane or artillery shell. Dimakis, *et al.* [15] use decentralized erasure codes to reduce latency and unreliability between query times and the time at which data reaches the data collector. The authors assume a fixed ratio between the number of storage nodes and the number of nodes that contain original data.

## 3.5 Conclusion

"Sense and store" sensor networks are gaining popularity due to the recent availability of gigabyte-scale local storage on sensor nodes, and because storage operations are more energy efficient than radio operations. It is important to make the data stored locally on sensor nodes reliable because sensor nodes suffer from unusually high failure rates (both individual and correlated). We discussed three factors that influence energy-reliability tradeoffs—redundancy techniques, node choice, and frequency of integrity checks. We presented a simple analytical model for modeling the availability of a node's data, and are currently exploring these issues in more detail using simulation-based models. Our research on energy-reliability tradeoffs will enable long-term reliable storage in sensor nodes and enable their deployment in environments where frequent data collection is infeasible. We showed that our implementation of Reed-Solomon provided higher and more flexible reliability at the expense of a higher energy cost. We also showed that $\texttt{XOR}_2$ provides reliability close to that of a 4-way mirroring scheme, but at a much lower storage space overhead. This project [8] was a collaborative effort completed at UCSC. My significant contributions are the design of the $\texttt{XOR}_1$ and $\texttt{XOR}_2$ reliability schemes, the Markov models, and the availability equations.

# Chapter 4

# Parallel Redundant Array of Independent Streams (PRAIS)

## 4.1 Introduction

Tape drives [11] are well suited for archival storage of data that is infrequently accessed. Random read access of data has very low performance because the individual tape drive that contains the data must be loaded and the magnetic tape must be rotated to the location of the data within the tape. On the other hand, writing data onto tape drives is fast because tape drives are efficient at sequential data streaming.

When archival data is eventually accessed, the probability that the tape containing the data has suffered an unrecoverable data loss (either a full or partial failure) is relatively high. Reliability of enterprise tape drives in the short term is much better than disk drives, but the large amount of time between accesses leaves a long period of time for the drives to degrade due to an improper storage room environment including mishandling during physical moves.

An erasure coding system can help protect data against these types of failures by storing the tape in an array with some redundant data. However, adding redundancy also adds complexity and can degrade performance. We implemented a parallel algorithm that computes two parities and simulates a tape archive, though the data is written to disk. We show that our algorithm scales linearly for small installations.

## 4.2 Related Work

The performance of writing to a tape archive is crucial, particularly when used as a backup for a large disk-based storage system. A typical usage case for tape is as a nightly backup. If it takes longer than all night to copy the data to tape, this method would not be very useful. Tape archives use striping to improve performance [16, 17]. Data is broken up into pieces which are written to all tapes sequentially, in a sort of striping pattern across the devices. This technique greatly improves read and write performance, but at the cost of reliability. If a file is striped across 128 tape drives, all 128 must accurately retrieve their share of the file

| Disk A | Disk B | Disk C | Disk D |
|---|---|---|---|
| $D_{A1}$ | $D_{A1}$ | $D_{C1}$ | $D_{C1}$ |
| $D_{A2}$ | $D_{A2}$ | $D_{C2}$ | $D_{C2}$ |
| $D_{A3}$ | $D_{A3}$ | $D_{C3}$ | $D_{C3}$ |
| $D_{A4}$ | $D_{A4}$ | $D_{C4}$ | $D_{C4}$ |

Figure 4.1: RAID 1 mirrored data layout requires high 2x storage space but good read performance. Any single failure is tolerated, as well as some multiple failures such as disks A and C in this example.

in order for it to be reconstructed. We look at improving the reliability that is decreased when using striping.

Storing data in a redundant format is an easy way to improve reliability. This is a very common technique in disk systems as well. The simplest version is mirroring or higher levels of replication. Tandem in the 1980s offered reliable computer systems that used a pair of mirrored disks [9]. These types of fault tolerant systems were popular with the banking industry. Currently, the Google File System (GFS) [18] needs to provide high availability without noticeable degradation when failures occur and therefore triplicates its chunks. Since the storage overhead of replication is high and since many data centers are not limited by performance, other types of redundancy generation are important. Mathematically, all types of redundancy can be described as an erasure correcting code that stores $m$ data blocks on $n$ disks such that all data is accessible despite up to $k$ disk failures. For example, a system of $m$ triplicated disks uses $n = 3m$ disks in total and can always survive up to $k = 2$ failures, and often many more.

One well known technique that is often implemented in hardware is Redundant Array of Independent Disks (RAID) [39], which is a set of $m$ data devices associated with $n$ parity devices. RAID level 0 has no parity and is now known as a Just a Bunch Of Disks (JBOD). Figure 4.1 shows the RAID 1 layout which stores identical data on two disks. Figure 4.2 shows two layouts which use parity to reduce the storage overhead of storing redundant data. Parity of several data blocks is calculated by taking the exclusive-or of all data bits. RAID 4 places parity blocks on a dedicated disk. This presents a bottleneck on the parity disk because the parity disk must be accessed for every write. In order to remove the bottleneck on that one disk, parity is distributed across all disks in the RAID 5 layout.

Early work extended RAID 5 to tolerate multiple disk failures by using additional parities for data stripes. One of the clear benefits to XOR-based codes is that they are computationally efficient and easy to understand. Hellerstein et al. [23] present binary (XOR-based) linear codes in a combinatorial framework. Their codes tolerate up to three failures in the general case and they suggest that codes tolerating more than three failures would not be useful in the future. EVENODD[10] is a maximum distance separable (MDS) code with optimal performance, no recursive computations, and tolerates up to two disk failures. MDS codes are

RAID 4

| stripe | disk 1 | disk 2 | disk 3 | disk 4 |
|---|---|---|---|---|
| A | $D_{A1}$ | $D_{A2}$ | $D_{A3}$ | $P_A$ |
| B | $D_{B1}$ | $D_{B2}$ | $D_{B3}$ | $P_B$ |
| C | $D_{C1}$ | $D_{C2}$ | $D_{C3}$ | $P_C$ |
| D | $D_{D1}$ | $D_{D2}$ | $D_{D3}$ | $P_D$ |

RAID 5

| stripe | disk 1 | disk 2 | disk 3 | disk 4 |
|---|---|---|---|---|
| A | $D_{A1}$ | $D_{A2}$ | $D_{A3}$ | $P_A$ |
| B | $D_{B1}$ | $D_{B2}$ | $P_B$ | $D_{B3}$ |
| C | $D_{C1}$ | $P_C$ | $D_{C2}$ | $D_{C3}$ |
| D | $P_D$ | $D_{D1}$ | $D_{D2}$ | $D_{D3}$ |

Figure 4.2: RAID 4 and RAID 5 data and parity layouts have a lower storage overhead than RAID 1, but only tolerate a single failure. The parity distribution of RAID 5 eliminates the parity disk bottleneck.

RDP

| stripe | disk 1 | disk 2 | disk 3 | disk 4 | row parity | diag. parity |
|---|---|---|---|---|---|---|
| A | $D_{A1}$ | $D_{A2}$ | $D_{A3}$ | $D_{A4}$ | $P_A$ | $P_{S1}$ |
| B | $D_{B1}$ | $D_{B2}$ | $D_{B3}$ | $D_{B4}$ | $P_B$ | $P_{S2}$ |
| C | $D_{C1}$ | $D_{C2}$ | $D_{C3}$ | $D_{C4}$ | $P_C$ | $P_{S3}$ |
| D | $D_{D1}$ | $D_{D2}$ | $D_{D3}$ | $D_{D4}$ | $P_D$ | $P_{S4}$ |

Figure 4.3: The Row-Diagonal Parity layout tolerates any two failures.

optimally space efficient.

### 4.2.1 Row-Diagonal Parity (RDP)

Row-Diagonal Parity [14] is an XOR-based code similar to EVENODD but proven computationally optimal. Figure 4.3 shows the layout of RDP with dedicated parity disks. This is simpler to understand, but parity can be distributed on all disks as in RAID 5. Each data segment belongs to one row parity segment and one diagonal parity segment. The diagonal parity includes data segments and row parity segments if they are included in the diagonal. RDP protects data against up to two disk failures. We investigated RDP [14] as an erasure coding method for streams of data. Systems using RAID 4 or other methods with a single parity for each redundancy group are protected against a single failure. In an archival system with strict data reliability requirements, it is better to protect against a second failure. We measure the implications and investigate the tradeoff between additional parities and the effect on reliability and performance.

Figure 4.4 illustrates an example RDP layout. The numbers on each disk represent

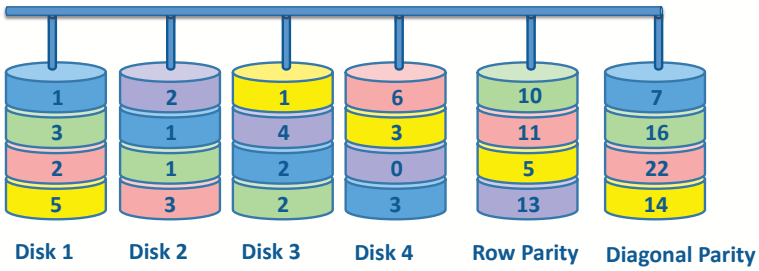


Figure 4.4: Example RDP layout with integers for data. For a single failure, reconstruct using either row or diagonal parity. After a double failure, first reconstruct a diagonal that only lost one element. Then, reconstruct that row, and repeat this process for all data and parity elements.

sample integer data and (for simplicity) the parities are computed with addition rather than exclusive-or. To show how data is reconstructed after two failures, suppose disks one and two fail. Notice that the yellow diagonal lost just one block. Thus the yellow block on disk 1 can be reconstructed using the remaining blocks in the yellow diagonal parity stripe. Now row parity can reconstruct the pink block on disk 2 in the bottom row. The rest of the reconstruction follows in the same manner.

## 4.3 PRAIS Implementation

RDP was implemented in a parallel software scheme using C with Message Passing Interface (MPI). Figure 4.5 shows the architecture of the software. Each node in the cluster represents a storage device along with one master node that coordinates error handling. Node 1 is the parity node and either reads parity and sends to another node or receives parity and writes it out to disk Each other node computes the XORs of data as it is read from disk. The file is broken up into RDP "sets" where each set is a self-contained box where all row parities and all diagonal parities can be computed using that set. In Figure 4.5, the data segment *A* represents an entire diagonal parity set with several rows and diagonal parity elements. Each set is allocated to a node and all parities are computed at that node and then segments are sent to the appropriate node that will write that segment of data. A larger number of nodes means that the stripe width is larger so that data can be written to disk more quickly and each node does less work.

## 4.4 Evaluation

Our experiments were run on a 164 node Linux cluster at Los Alamos National Laboratory. The first experiment consists of 500MB of data striped and written with RDP. Figure 4.6 shows the speed of several runs using 8-16 nodes. The work involved is that each node computes parity and writes data to a disk device and sends data to other nodes. The bandwidth of the initial write is about 100 MB/s with 8 nodes and 200 MB/s with 16 nodes. As expected,

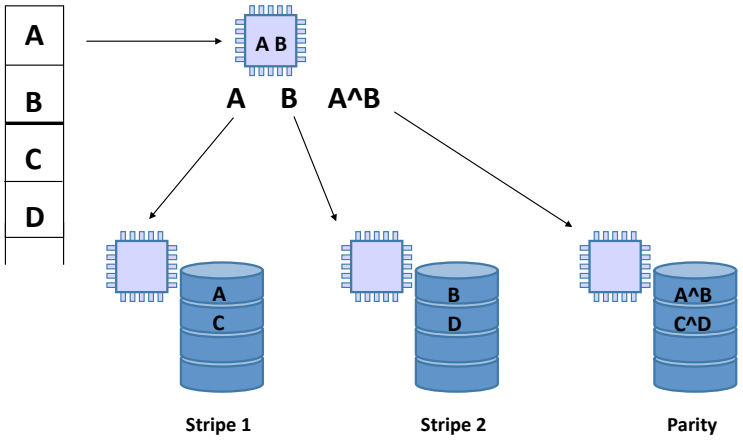


Figure 4.5: PRAIS architecture.

doubling the number of nodes doubles the bandwidth because the work is spread out evently.

The second experiment was to reconstruct missing data after two failures where the total size of data is 500MB and it is striped and stored with RDP. Figure 4.7 shows the results of reconstructing data. You can see that the bandwidth is lower than the initial write. This is because reconstructing two data elements involves first reading all remaining data, then computing both parities. Bandwidth is approximately 75 MB/s with 8 nodes and 150 MB/s with 16 nodes. This shows that our algorithm scales for small numbers of nodes.

## 4.5 Conclusion

We completed a parallelized software implementation of RAID 4 and row-diagonal parity. We showed that our parallel implementation scales well for small numbers of nodes: doubling the number of nodes doubles the initial write and reconstruction bandwidths. The motivation of this project is to investigate the tradeoffs of using higher levels of redundancy than RAID 4 in a tape archive. Future work is to do a more complete evaluation comparing the performance of our parallel RDP implementation with higher redundancy erasure codes to determine the cost of tolerating additional failures. We would also like to evaluate the reliability for a simulated tape archive and compare the actual improvement in reliability provided with the computational cost. This will help determine how useful this work can be for tape archives and other applications such as wide area network data movement and disk archives. This project was completed at the Los Alamos National Laboratory (LANL) during a summer internship in 2004.

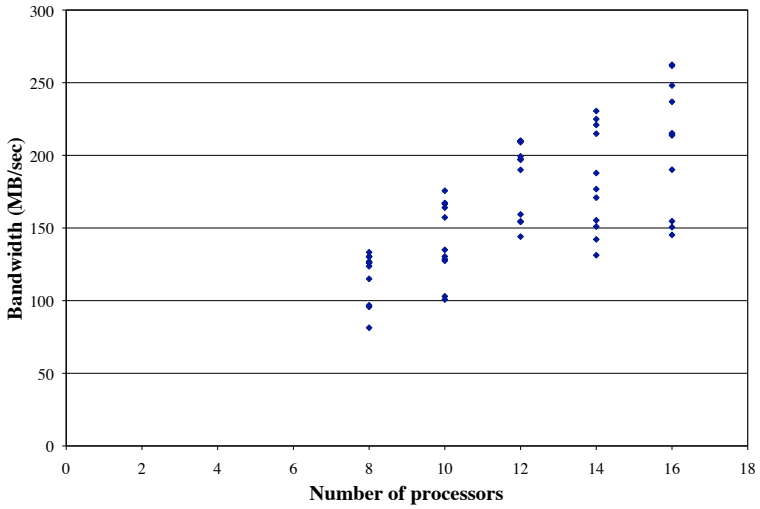


Figure 4.6: Performance of initial write of 500MB of data.

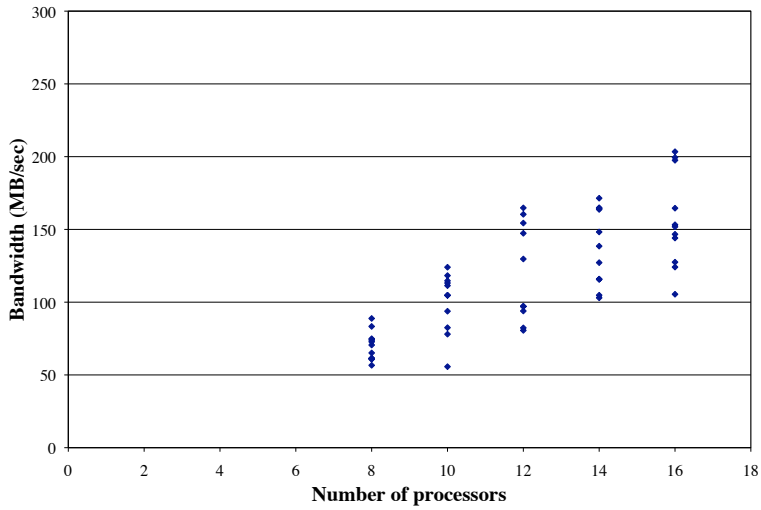


Figure 4.7: Performance of reconstruction of 500MB of data.

# Bibliography


[1] www.arm.com/products/CPUs/ARM926EJ-S.html.

[2] Sun StorageTek T10000. http://www.sun.com/storagetek/tape_storage/tape_media/t10000/.

[3] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, October 2003.

[4] Eric Anderson, Mahesh Kallahalla, Mustafa Uysal, and Ram Swaminathan. Buttress: A toolkit for flexible and high fidelity I/O benchmarking. In *Proceedings of the Third USENIX Conference on File and Storage Technologies*, San Francisco, CA, March 2004.

[5] Akshat Aranya, Charles P. Wright, and Erez Zadok. Tracefs: A file system to trace them all. In *Proceedings of the Third USENIX Conference on File and Storage Technologies*, pages 129–145, San Francisco, CA, April 2004. USENIX.

[6] Mary Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. In *Proceedings of the thirteenth ACM Symposium on Operating Systems Principles*, 1991.

[7] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 259–272, San Francisco, CA, December 2004.

[8] Neerja Bhatnagar, Kevin M. Greenan, Rosie Wacha, Ethan L. Miller, and Darrell D. E. Long. Energy-reliability tradeoffs in sensor network storage. In *Proceedings of the 5th Workshop on Embedded Networked Sensors*, 2008.

[9] Dina Bitton and Jim Gray. Disk shadowing. In *Proceedings of the 14th Conference on Very Large Databases (VLDB)*, pages 331–338, 1988.

[10] Mario Blaum, Jim Brady, Jehoshua Bruck, and Jai Menon. EVENODD: An optimal scheme for tolerating double disk failures in RAID architectures. In *Proceedings of the 21st annual international symposium on Computer architecture*, pages 245–254, 1994.

[11] R. Bradshaw and C. Schroeder. Fifty years of IBM innovation with information storage on magnetic tape. *IBM Journal of Research and Development*, 47(4):373–383, July 2003.

[12] Mike Y. Chen, Anthony Accardi, Emre Kıcıman, Dave Patterson, Armando Fox, and Eric Brewer. Path-based failure and evolution management. In *Networked Systems Design and Implementation (NSDI)*, pages 309–322. USENIX, 2004.

[13] Chee-Yee Chong and Srikanta P. Kumar. Sensor Networks: Evolution, Opportunities, and Challenges. In *Proc. of the IEEE*, volume 91, pages 1247– 1256, 2003.

[14] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of the Third USENIX Conference on File and Storage Technologies*, 2004.

[15] Alexandros G. Dimakis, Vinod Prabhakaran, and Kannan Ramchandran. Ubiquitous access to distributed data in large-scale sensor networks through decentralized erasure codes. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, page 15. IEEE Press, 2005.

[16] Ann L. Drapeau and Randy H. Katz. Striped tape arrays. In *Proceedings of the 12th IEEE Symposium on Mass Storage Systems*, 1993.

[17] Ann L. Drapeau and Randy H. Katz. Striping in large tape libraries. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 378–387, 1993.

[18] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, 2003.

[19] Abhishek Ghose, Jens Grossklags, and John Chuang. Resilient data-centric storage in wireless ad-hoc sensor networks. In *Proceedings of the 4th International Conference on Mobile Data Management*, pages 45–62. Springer-Verlag, 2003.

[20] Kevin Greenan, Ethan L. Miller, and Thomas Schwarz. Analysis and construction of Galois fields for efficient storage reliability. Technical Report Technical Report UCSC-SSRC-07-09, University of California, Santa Cruz, 2007.

[21] Kevin M. Greenan, Ethan L. Miller, and Thomas J. E. Schwarz, S.J. Optimizing galois field arithmetic for diverse processor architectures and applications. In *Proceedings of the 16th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2008)*, September 2008.

[22] James Lee Hafner. WEAVER codes: Highly fault tolerant erasure codes for storage systems. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST)*, 2005.

[23] Lisa Hellerstein, Garth A. Gibson, Richard M. Karp, Randy H. Katz, and David A. Patterson. Coding techniques for handling failures in large disk arrays. *Algorithmica*, 12:182–208, 1994.

[24] Bo Hong. Techniques for synthetic I/O workload generation. M.sc. thesis, University of California at Santa Cruz, September 2002.

[25] Abhinav Kamra, Jon Feldman, Vishal Misra, and Dan Rubenstein. Data persistence for zero-configuration sensor networks. In *ACM Special Interest Group on Data Communications (SIGCOMM)*, 2006.

[26] Magnus Karlsson and Christos Karamanolis. Non-intrusive performance management for computer services. In *Middleware 2006*, pages 22–41, Melbourne, Australia, November 2006.

[27] Magnus Karlsson, Christos Karamanolis, and Xiaoyun Zhu. Triage: Performance differentiation for storage systems using adaptive control. *ACM Transactions on Storage (TOS)*, 1(4):457–480, 2005.

[28] F. Koushanfar, M. Potkonjak, and A. Sangiovanni-Vincentelli. Fault tolerance techniques in wireless ad-hoc sensor networks. In *Proc. of IEEE Sensors*, volume 2, pages 1491–1496, 2002.

[29] S. Lai. Current status of the phase change memory and its future. *IEDM Technical Digest*, pages 10.1.1– 10.1.4, 2003.

[30] Andrew Leung, Shankar Pasupathy, Garth Goodson, and Ethan L. Miller. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the 2008 USENIX Technical Conference*, 2008.

[31] Song Lin, Benjamin Arai, and Dimitrios Gunopulos. Reliable hierarchical data storage in sensor networks. In *19th Int'l Conf. on Scientific and Statistical Database Mgmt, SSDBM*, pages 26–35, 2007.

[32] Y. Lin, B. Liang, and B. Li. Data persistence in large-scale sensor networks with decentralized fountain codes. In *INFOCOM 2007. 26th IEEE Int'l Conf. on Computer Communications*, pages 1658–1666, 2007.

[33] Christopher R. Lumb, Arif Merchant, and Guillermo A. Alvarez. Façade: Virtual storage devices with performance guarantees. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, CA, April 2003.

[34] Tara M. Madhyastha and Daniel A. Reed. Input/output access pattern classification using hidden Markov models. In *Fifth Workshop on I/O in Parallel and Distributed Systems (IOPADS)*, pages 57–67, November 1997.

[35] Gaurav Mathur, Peter Desnoyers, Deepak Ganesan, and Prashant Shenoy. Ultra-low power data storage for sensor networks. In *IPSN '06: Proceedings of the fifth international conference on Information processing in sensor networks*, pages 374–381. ACM, 2006.

[36] Michael P. Mesnier, Matthew Wachs, Raja R. Sambasivan, Julio Lopez, James Hendricks, Gregory R. Ganger, and David O'Hallaron. //TRACE: Parallel trace replay with approximate causal events. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, pages 153–167, February 2007.

[37] A. Mitra, A. Banerjee, W. Najjar, D. Zeinalipour-Yazti, V. Kalogeraki, and D. Gunopulos. High-Performance Low Power Sensor Platforms Featuring Gigabyte Scale Storage. In *IEEE/ACM 3rd Int'l Workshop on Measurement, Modelling, and Perf. Anal. of WSNs*, 2005.

[38] Suman Nath, Haifeng Yu, Phillip B. Gibbons, and Srinivasan Seshan. Subtleties in tolerating correlated failures in wide-area storage systems. In *Networked Systems Design and Implementation (NSDI)*, 2006.

[39] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, 1988.

[40] James S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software, Practice and Experience*, 27(9):995–1012, 1997.

[41] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[42] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: the OceanStore prototype. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, 2003.

[43] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Technical Conference*, 2000.

[44] Thomas Schwarz. *Reliability and Performance of RAID Systems*. PhD thesis, Univ. of California at San Diego, 1994.

[45] Thomas S. J. Schwarz and Ethan L. Miller. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, page 12. IEEE Computer Society, 2006.

[46] Bo Sheng, Qun Li, and Weizhen Mao. Data storage placement in sensor networks. In *ACM International Symposium on Mobile Ad Hoc Networking and Computing*, pages 344–355, 2006.

[47] Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh. Fidelity and yield in a volcano monitoring sensor network. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 381–396. USENIX Association, 2006.

[48] Parkson Wong and Rob F. Van der Wijngaart. NAS parallel benchmarks I/O version 2.4. Technical Report NAS-03-002, NASA Advanced Supercomputing (NAS) Division of NASA Ames Research Center, January 2003.

[49] Charles P. Wright, Nikolai Joukov, Devaki Kulkarni, Yevgeniy Miretskiy, and Erez Zadok. Auto-pilot: A platform for system software benchmarking. In *Proceedings of the 2005 USENIX Technical Conference*, pages 175–188, Anaheim, CA, April 2005.

[50] Jay J. Wylie and Ram Swaminathan. Determining fault tolerance of XOR-based erasure codes efficiently. In *Proceedings of the 2007 International Conference on Dependable Systems and Networks (DSN 2007)*, 2007.