Researching system administration

by

Eric Arnold Anderson

B.S. M.S. Carnegie Mellon University 1994

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Dave Patterson, Chair
Professor Doug Tygar
Professor Peter Menell

Spring 2002

The dissertation of Eric Arnold Anderson is approved:

Chair Date

Date

Date

University of California at Berkeley

Spring 2002

Researching system administration

Abstract

Researching system administration

by

Eric Arnold Anderson
Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Dave Patterson, Chair

System administration is a phenomenally important, yet surprisingly ignored sub-field of Computer Science. We hypothesize that this avoidance is because approaches for performing academic research on system administration problems are not well known. To reduce the difficulty of performing research, we present a small set of principles that can be used to evaluate solutions, a classification of existing research on system administration, and three approaches to research on system administration that we illustrate with the research that we have done.

First, we demonstrate the approach of "Let the human handle it" with the CARD cluster monitoring system. We show that CARD is more flexible and scalable than earlier approaches. We also show that monitoring is necessary for system administration, but that this research approach is not a complete solution to system administration problems.

Second, we demonstrate the approach of "Rewrite everything" with the River I/O programming infrastructure. We show that River adapts around performance anomalies improving the performance consistency of I/O kernels. By rewriting the entire application, we could explore a substantially different approach to program structuring, but this research approach limits the completeness of the resulting system.

Third, we demonstrate the approach of "Sneak in-between" with the Hippodrome iterative storage system designer. We show that Hippodrome can find an appropriate storage system to support an I/O workload without requiring human intervention. We show that by using hooks in existing operating systems we can quickly get to a more complete system, but that this research approach can be restricted by the existing interfaces.

Finally, we describe a substantial number of open research directions based both on the classification that we developed of existing research, and on the systems that we built. We conclude that the field of system administration is ripe for exploration, and that we have helped provide a foundation for that exploration.

Professor Dave Patterson
Dissertation Committee Chair

Contents

List of Figures

Chapter 1

Introduction

System administration has great economic importance. Studies indicate the cost per year of administering systems as one to ten times the cost of the actual hardware [Gro97, And95, Coub]. Moreover, system administrators are in remarkable demand, with average salaries growing by over 10% per year [SAN]. As a consequence, many companies have made reducing total cost of ownership one of their primary goals [Mica, Micb, Pacb].

Despite this substantial commercial interest, there is little academic work on system administration. Only a few schools have classes on system administration [Nem, Coua, Ext], a only a small number of research projects have specifically targeted system administration [BR98, Asa00].

We choose to focus on system administration of large sites because we believe that the problems faced by large sites are more complex than those faced by end users, and because we believe that if we can make the large sites manageable, they will be able to support the end users. Indeed, some researchers and business people have proposed having only a web browser on the machines used by end users, and hosting all of the applications of centralized, large sites. This centralization reduces the administration problem for the end users, but at best leaves the problems the same for the new centralized sites.

This dissertation serves three related purposes. First, it identifies approaches for academic research on system administration. Second, it demonstrates the approaches by examining three systems, each built using a different research approach. Third, it enables future research by both identifying principles for evaluating system administration research, and by identifying directions of future research.

We start by describing principles for evaluating system administration research in section 1.2. We identify and explain the principles to help researchers avoid some work of deploying their systems. The principles help identify areas where a particular solution to a system administration problem both assists and complicates the job of system administrators. We use these principles to evaluate the systems we built, but as we did not identify the principles until after we had developed all of the systems, they did not influence

our selection of problems.

Chapter 2 describes the field of system administration. We examine the history of the USENIX systems administration conference and categorize the work both by task and by problem source. The categorization helps us to understand the prior research and identify important problems to attack.

Chapter 3 considers the challenge of monitoring and diagnosing problems in a cluster of computers, and examines our first approach to system administration research: "Let the human handle it." We describe CARD [AP97], which shows that we can scale up to a large cluster and achieve substantial flexibility. CARD uses relational tables for flexibility, aggregation to scale in performance and visualization and a novel communication method to reduce network load. CARD does not prevent problems, but merely brings them to light, still leaving a human to repair the actual problem.

Chapter 4 considers the challenge of automatically adapting around performance anomalies, and examines our second approach to system administration research: "Rewrite everything." We show that performance anomalies can substantially impact cluster applications, and we describe a new programming infrastructure for I/O-centric applications. Rivers [ADAT+99] demonstrates that for database kernels, the system can minimize the impact of performance anomalies improving system predictability and avoiding the need for administrators to spend time tuning the system.

Chapter 5 considers the challenge of resource provisioning and long term variability in I/O workloads, and examines our third approach to system administration research: "Sneak in-between." We describe Hippodrome, a system that iteratively adapts to I/O workloads by analyzing a workload, designing a new system that supports the workload, and implementing the new design. By iterating until the workload stabilizes, Hippodrome can identify the appropriate amount of resources needed for a workload. By periodically executing the loop, persistent changes can be factored into the design. In both cases, the administrator is freed from the difficulty of configuring, sizing, and updating the configuration of complex storage systems

Chapter 6 identifies future directions for research based on the task categorization in Chapter 2, and the work done in Chapters 3-5. This chapter helps future researchers focus and structure their efforts, reducing the effort necessary to start researching system administration problems.

Chapter 7 presents our conclusion, reviews the three approaches to research that we have described, and summarizes the results from each of the systems we built.

1.1 Overview of system administration

System administration is a very general term, so we start with a definition and an overview. The environment for system administration is the hardware and software that forms the computer system used by some set of users. Administrators have three main responsibilities. First, they are responsible for configuring the system so that the users can get their jobs done. Second, they are responsible for maintaining the system

against both internal failures and internal or external attacks. Third, they are responsible for training users to use the system effectively.

This description is painfully general as it excludes few tasks which are even peripherally related to the computer system. However, this generality is accurate and is similar to a description of the medical profession: "If it has to do with the human body, doctors need to understand it." As with the medical profession, both general system administrations and specialists exist. As the system administration profession is relatively new, most administrators are still generalists, having learned the profession by apprenticeship to other administrators [SAG] and by personal experience.

Specialists are starting to develop along two different axis. Some administrators are specializing in certain types of hardware, for example Cisco certified engineers [Cis], who are trained to manage Cisco networking hardware. Others are specializing in certain types of problems, for example computer security experts.

The system administration field is strongly influenced by the importance of the users. The users at a site are the largest determinant of the responsibilities of an administrator. Consider, for example, the difference between an administrator for a research group at a university, an administrator at a Wall Street financial firm, and an administrator for a large web site. The research administrator may favor flexibility and ability to determine what the users have done to the system, the financial administrator may focus on uptime during trading, and the web administrator may be primarily concerned with scalability.

Users further differ in the type of interaction they prefer, their sophistication, the types and variety of tasks they want to perform, and their expectation of responsiveness from the administrators. System administrators are often called on to do anything having to do with the computer system, especially if they do a good job. Indeed, Zwicky, Simmons, and Dalton claimed that the most important thing for limiting the responsibilities of the administrators was to explicitly identify tasks they would *not* perform [ZSD90].

System administration research looks at problems with users and administrators interaction with the system, which makes system administration research somewhat different from most computer science research. Research on human-computer interaction (HCI) is similar, and many of the difficulties faced in evaluating solutions in HCI are faced in evaluating system administration solutions. Deploying a system widely and using it in production provides deep, valuable understanding. It takes a long time, however, and requires a lot of work beyond that needed for the research. Although we believe that researchers in system administration benefit from this experience and should attempt to deploy, we also realize that other evaluation criteria allow for more efficient research. We therefore describe a collection of principles by which a proposed solution to a system administration problem can be evaluated.

1.2 Principles of system administration

The acid test for a system administration solution would be to widely deploy the solution in the field and measure the improved productivity or reduced costs derived from the solution. Because this test is extremely hard to perform in practice, we looked for easier methods for evaluating system administration.

We have identified eight principles of system administration from discussions with administrators, reading of papers, and personal experience. Although the principles are desirable in isolation, several conflict with each other. This conflict is one of the central tensions in system administration: people want their systems to be able to do many different things, and yet still be simple and cheap. The conflict is magnified because different organizations, and different people within a single organization, have different goals and hence want different behaviors from a system.

The eight principles in approximate order of importance (as determined below in Figure 1.1) are:

1. **Dependability** — The system should behave as expected. If something worked the day before, and the user has not initiated a change, then it should work today. If the user has tried to change the system, the new behavior should be predictable given the change. Dependability is related to transparency (below), but is more aimed at answering the question "what will the system do" than the question "how did the system do it." Dependability captures most of the security issues; an insecure system, or one that leaks information, is not dependable as it may not behave as expected.

2. **Automation** — It is almost always better to remove the human from the loop by having the system automatically perform a task. We expect that applying the principle of flexibility (below) will allow for customization of automated tasks. Occasionally, the work required to automate a special case is larger than the gain saved by the automation, in which case performing the work by hand is better [LRNL97]. Automation also tends to hide details of what is happening, and hence conflicts with the goal of transparency.

3. **Scalability** — System growth is unavoidable. People want bigger, better, faster. Many articles have been written on the exponential growth of users and systems on the Internet and how quickly companies expand. Therefore, a system with a specific fixed maximum will cause problems when the needs of the users outgrows the capacity of the system.

4. **Flexibility** — A system administration solution should be able to work in many different ways. People want to modify and fiddle with their systems to get the system to conform to the desires of the user. Inflexible systems force users to conform to the system. Similarly, inflexible systems may not be able to work well with other tools. A common task for administrators is getting two separate tools to work with each other.

5. **Notification** — Once a problem occurs, the system should notify the human, rather than requiring the human to continually check the system for problems. Users vastly prefer an environment where when problems occur, the administrator is already fixing them, or better yet, the administrator can fix the problems before they ever have an impact on the users. Unfortunately, the goal of notification can work against the goal of schedulability (next) because the notification can interrupt the administrator needlessly, or worse, incorrectly.

6. **Schedulability** — It is almost always better to be able to schedule tasks that have to be performed. One large Internet service provider estimated it cost over $1000 to get an engineer out of bed in the case of an unexpected network problem [Mal]. By adding redundancy to the system, system availability can be maintained without required human intervention to fix problems. Delaying the intervention of humans can both increase the speed of recovery, and reduce the chance of human error made under pressure. In addition, much as with computers, interrupts for people are expensive. System administration is already an interrupt driven job. Scheduling tasks improves the productivity of the administrators.

7. **Transparency** — It should be clear what the system is doing. Administrators often have to determine why a particular problem occurred so that they can guarantee it will not happen again. A non-transparent system makes this much more difficult because the administrator has to guess at the internals of the system and speculate about what could be causing the problem. Similarly, this shows the utility of logging past events. It is easier to see into the system if the system tracks what has been done. Transparency may conflict with the goal of automation.

8. **Simplicity** — A simpler system is both easier to use and administer because fewer things can go wrong. Keeping a system simple, for example by partitioning different applications, reduces the number of interactions present in the system. The more interactions present, the greater the chance of conflicts between pieces of the system, and resulting problems for administrators. Furthermore, by restricting the choices users can make, a system can be tested more completely, and has a better chance of catching user errors. Unfortunately, a simple system may not be able to do the tasks that users want, and hence the principle of simplicity conflicts with the principle of flexibility.

The principles of administration give us a method for evaluating a solution without having to perform widespread deployment of the system. We can examine a system and rate the system as positive, neutral, or negative depending on its effect on each of the principles. Having done the rating, then the actual evaluation of the tradeoff depends on the particulars of a site. Some sites may value flexibility above all, others may consider dependability or scalability the key principle. As experience is gained in measuring

	Sites			
Principle	Financial	Research	Web Site	Total
Dependability	High	Medium	High	8
Automation	Medium	Medium	High	7
Scalability	Low	High	High	7
Flexibility	Low	High	Medium	6
Notification	High	Low	Medium	6
Schedulability	High	Low	Low	5
Transparency	Low	High	Low	5
Simplicity	Medium	Low	Low	4

Figure 1.1: Estimated importance of the various principles to each of the different sites. Estimation was done by author based on discussions with administrators at each different type of site, and the issues they had focused on in papers they had written. Each site was arbitrarily allowed three highs, two mediums, and three lows to force some differentiation, as otherwise all sites would want each principle with high importance. Importances are relative not absolute. The total was calculated by rating a high as three, medium as two and low as one. This table is only intended to give a rough idea of the importance of each principle.

systems relative to the principles, we may even find we can assign numeric values to the different axes, so the magnitude of the benefit or loss becomes apparent.

As mentioned above, the conflict between principles is inherent. Hence, we believe that it is best to identify how a particular solution affects each of the principles. For example, a cluster solution which uses multiple computers instead of a large symmetric multi-processor (SMP) may be more scalable, but is is clearly less simple, and it requires more automation to maintain consistency between the hosts in the cluster.

Figure 1.1 re-evaluates the three different sites we described earlier, using the principles as a matrix. We estimated the relative importance based on discussions with administrators at each of the different types of sites. Some of the principles appear more important than the others, and we have sorted the table accordingly using a simple rating system. We then used the sorted table to order the principles shown above.

We evaluate each of the systems in Chapters three to five based on the principles described above. We also examine the successful and unsuccessful aspects of each system. That examination shows that academic research can be used to approach system administration problems, but that care must be taken to avoid potential pitfalls. In particular, failure to deploy a system at least partially can result in a researcher missing limitations in their system. We will show issues that we have learned from deploying the systems we built when we describe each system.

Chapter 2

The field of system administration

Examining the tasks performed by system administrators helps us understand the field as a whole, and improves the chance of research being relevant. There are many different approaches to learning about tasks: experience, discussions, surveys, and examination of publications. We have drawn from all of these sources: I have worked as an administrator at both a university and at a Internet service provider. We have had discussions with many other administrators primarily at the USENIX Systems Administration conferences (LISA). We ran a survey [And95] to determine where time is spent, and draw from the other surveys that have been done. For this chapter, we describe two categorizations of the first twelve years of LISA proceedings to help us understand the subjects that administrators consider sufficiently important that they publish their results.

Other people have also created different descriptions of the field of system administration. The system administrator's body of knowledge project [Hal99] has been working to categorize all of the tasks performed by administrators. The certification project by the System Administrators Guild(SAGE) has produced a number of study guides [Gui02] that identify important problems administrators have to solve. Mark Burgess's book [Bur00] describes the tasks performed by system administrators and generalizes the problems faced into many aphorisms. The Unix system administration handbook [NSSH01] provides in-depth details about the problems faced by administrators. SAGE keeps a list [Gui] of many additional books about system administration.

2.1 A model of tasks

The traditional approach for categorization is to group related papers by the problem described. Therefore, we started with this approach for all 342 of the papers from the first 12 years of the LISA conference. When we completed this initial step, we discovered that we had 64 separate categories. Hence, to provide additional structure, we continued the process and built a second level of the hierarchy starting

with the categories rather than the papers. Figure 2.1 sorts the categories by popularity at each level, with ties broken alphabetically. We show the paper count for each of the categories in brackets after each category name.

The breadth of tasks that administrators perform is clearly shown in the length of the list in Figure 2.1. This is one of the reasons that it is difficult to gain a complete understanding of the field. In addition to the 45 categories shown, there were 19 papers which were sufficiently different that they required a unique category.

Luckily, some of the categories are more popular. Backup, Mail, Application Installation, Site Configuration and Accounts comprise over a third of all the papers. This means that there are some areas where we can focus our research in order to improve the impact of the research. In addition, research on software installation or monitoring may cover multiple of the bottom level categories, providing another avenue for relevant research.

There are some potential concerns about this categorization that should be addressed. The simplest of which is that there were errors in the classification. The papers were all read by a single person, so the categories are mostly consistent, but with 342 papers, a few errors probably occurred in classification. Furthermore, while I have worked as a system administrator both at SURAnet and at Carnegie Mellon University, I have clearly not personally performed all of the tasks described. Another concern is that the program committee may also have biased the accepted papers based on their views of what should be in the conference, or because of a limited selection of available papers. Finally, some papers may be missing because companies consider the information to be proprietary. Despite these concerns, we believe that surveying the existing research still provides one of the best ways of getting at hard data about important problems in system administration.

2.2 A model of problem sources

Because of the concern about completeness, Figure 2.2 shows a second model based on the source of a problem. The source of the problem is labeled on the edges leading out from the center (the happy state) of the state transition diagram. The edges leading back in to the center represent tasks performed to return the system to a happy state. This model was derived in part from the time surveys, which indicated that administrators spent about a third of their time on each of these tasks.

The generality of this model allows it to cover all system administration tasks. Either administrators are trying to improve people (training) or trying to improve machines. If they're trying to improve the machines, it's either because the machines need to do something different (configuration management) or because they need to get back to doing what they used to do (maintenance).

Services [75]

- Backup [28]
- Mail [20]
- Printing [11]
- News [5]
- NFS [4]
- Web [3]
- DNS [2]
- Database [2]

Software Installation [57]

- Application Installation [32]
- OS Installation [14]
- User Customization [8]
- Software Packaging [3]

Monitoring [44]

- System Monitoring [14]
- Resource Accounting [6]
- Data Display [5]
- Network Monitoring [5]
- Benchmarking [4]
- Configuration Discovery [4]
- Host Monitoring [4]
- Performance Tuning [2]

Configuration Management [40]

- Site Configuration [27]
- Host Configuration [7]
- Site Move [4]
- Fault Tolerance [2]

Tools [40]

- Trouble Tickets [9]
- Secure Root Access [8]
- General Tool [6]
- Security [6]
- File Synchronization [4]
- Remote Access [3]
- File Migration [2]
- Resource Cleanup [2]

User Management [35]

- Accounts [23]
- Documentation [4]
- Policy [3]
- User Interaction [3]
- White Pages [2]

Network [19]

- Network Configuration [9]
- LAN [4]
- WAN [4]
- Host Tables [2]

Administrator Improvement [18]

- Self Improvement [7]
- Models [5]
- Software Design [4]
- Training Administrators [2]

Only one paper on topic [19]

Figure 2.1: 9 categories Categories derived from categorizing the 342 papers in the first 12 years of proceedings of the LISA conference

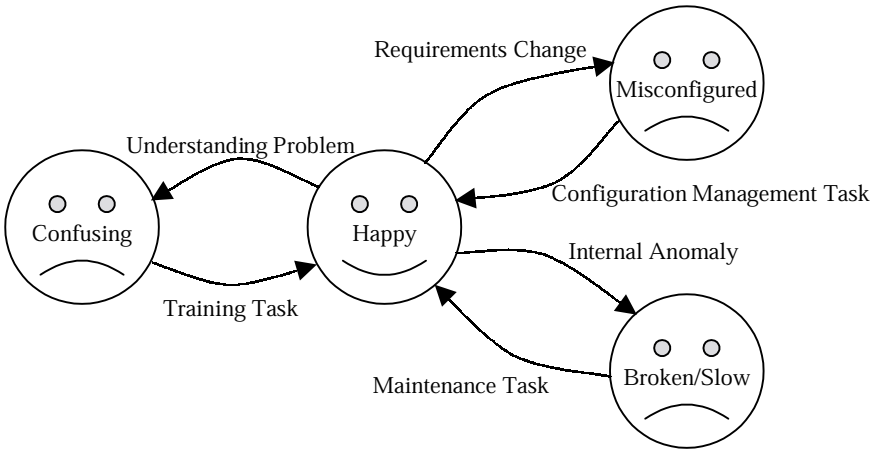


Figure 2.2: System state transition diagram. Edges out indicate problems that occur making the system less usable. Edges in indicate tasks performed by system administrators to restore the functionality of the system.

2.2.1 Examination of the different categories

Until people stop changing how they want to use a system, configuration management tasks will remain. Only by freezing how the system is used can we eliminate configuration management tasks. Even a simple appliance like a toaster has a few configuration tasks (plugging it in, adjusting the amount of toasting). The tasks have been simplified by limiting choices; adding choices inherently increases complexity. Configuration management tasks exemplify the conflict between the principles of flexibility and simplicity.

Maintenance tasks may be eliminated by building systems that recover from internal faults. Recovering from internal faults is additionally challenging because some of the faults are caused by malicious intruders. Furthermore, automating some maintenance tasks is extremely risky, for example, purchasing and installing replacement hardware. Hence, the goal should be to make the task schedulable, rather than forcing an administrator to deal with the problem immediately. Maintenance tasks exemplify the principles of automation and schedulability, and their conflict with the goal of notification.

Training tasks may be partially transferable out of the organization and into the schools. Users could be trained in the tools they will be using, and administrators could be trained in system administration. Earlier education would mean people would only have to learn the specifics of a site rather than the general knowledge. Alternately, the various tools that are being used could be improved to reduce the need for training. Researchers in Human Computer Interaction have been looking at this for some time, and have made a number of strides, but more work remains. Training tasks exemplify the principles of simplicity and dependability, and their conflict with the principle of flexibility.

2.3 Historical trends of the LISA conference

Given the two models, we can classify the papers using the models, and examine the trends that result over time. This examination will help us identify areas which are missing, and areas which have been studied thoroughly.

2.3.1 Task model trends

Figures 2.3 and 2.4 show the papers over the last twelve years categorized by the Task Model. For completeness, we show all of the papers that were shown in Figure 2.1.

We can see that some tasks, such as backup, application installation and accounts alternated between very heavy and light years. This alternation probably indicates some amount of duplicated effort in the very heavy years. Detailed examination of the papers shows two possible causes for this pattern. In some cases (application installation, OS installation), widely applicable solutions have not been found, and people are still making new, slightly different attempts. In other cases (backup, accounts), there was some change in the external world that caused previous solutions to stop working. For example, backup was a task that was successfully solved in the past, but with disk capacity and bandwidth growing faster than tape capacity and bandwidth, it has returned as a problem of dealing with larger scale.

We can see that some tasks, such as printing and trouble tickets, have received a consistent amount of work per year. This pattern is probably a good sign, as it means that slow and steady progress is being made without too much duplication of effort. Further examination of the papers indicates that in most cases the papers do build upon each other, but a few are not sufficiently related.

Mail alternated between the steady work and the heavy work models. Detailed examination of the papers indicates that this is because of effects from both of the previous descriptions. Initial work was fairly steady until the explosion of the Internet increased the size of mailing lists, and commercialization resulted in problems with SPAM.

Similarly, some tasks, such as system monitoring and network configuration, see punctuated bursts of activity. This pattern probably indicates that the problem occurred simultaneously due to some external change such as sites scaling up, or new applications. This intuition is confirmed by reading the papers; changes in the outside world often necessitated improved solutions. It would be nice if there were some way for different people to coordinate their work as they simultaneously discover new problem areas. This would reduce the amount of duplicated work, and probably also improve the resulting solution as it will deal with the idiosyncrasies of multiple sites.

It is not clear what we can learn from the tasks with fewer papers. In a few cases, we can infer that certain areas did not become problems until fairly recently. The WWW is an obvious example; configuration discovery, LAN, WAN, and NFS problems also appear to have only become problems recently. If we read

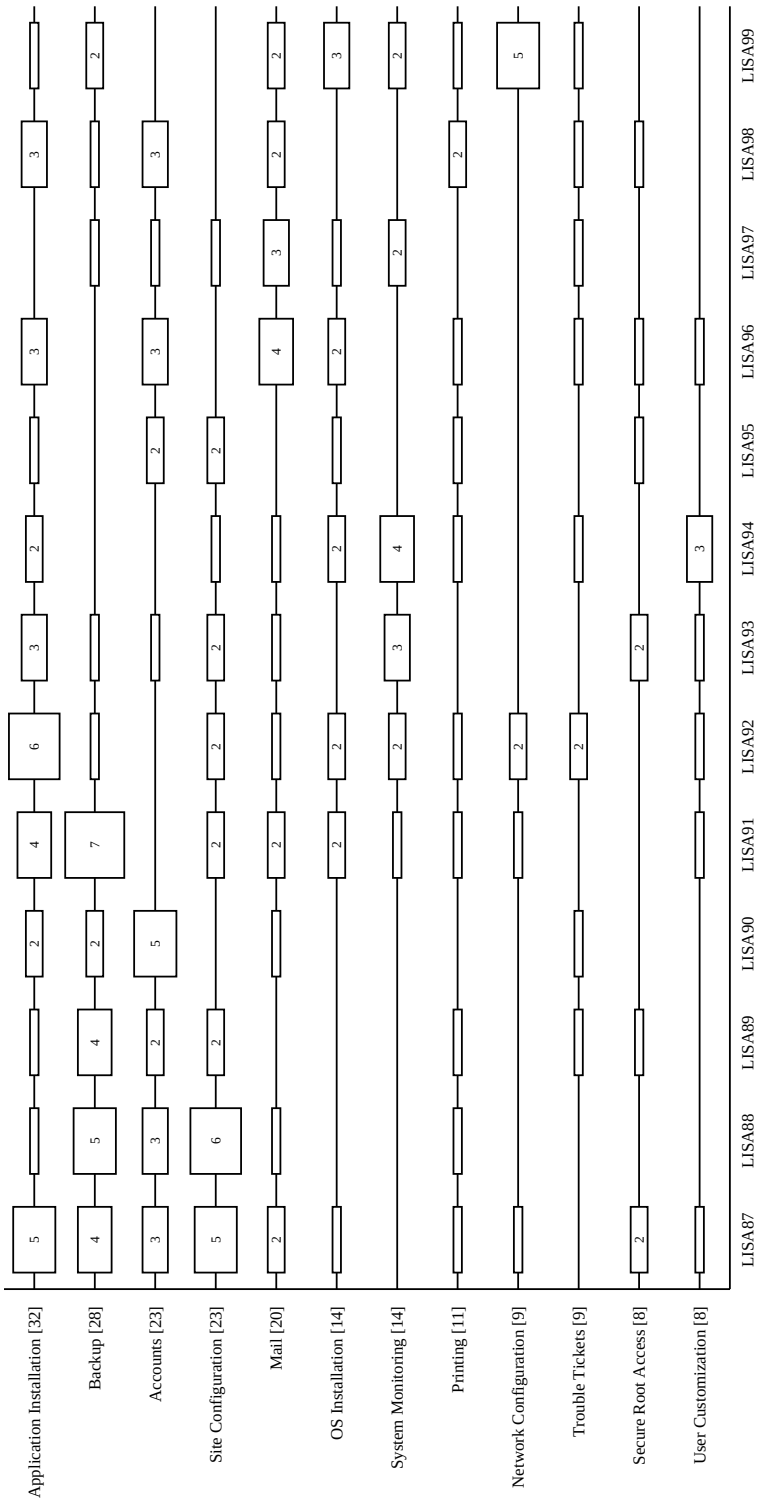


Figure 2.3: Breakdown of number of papers/conference/category for categories with at least 8 total papers. Sorted by popularity of a category, ties broken alphabetically. Height of a box, and the number inside, indicates number of papers. Total number of papers in a category is shown in brackets after the category name. The remainder of categories are shown in Figure 2.4.

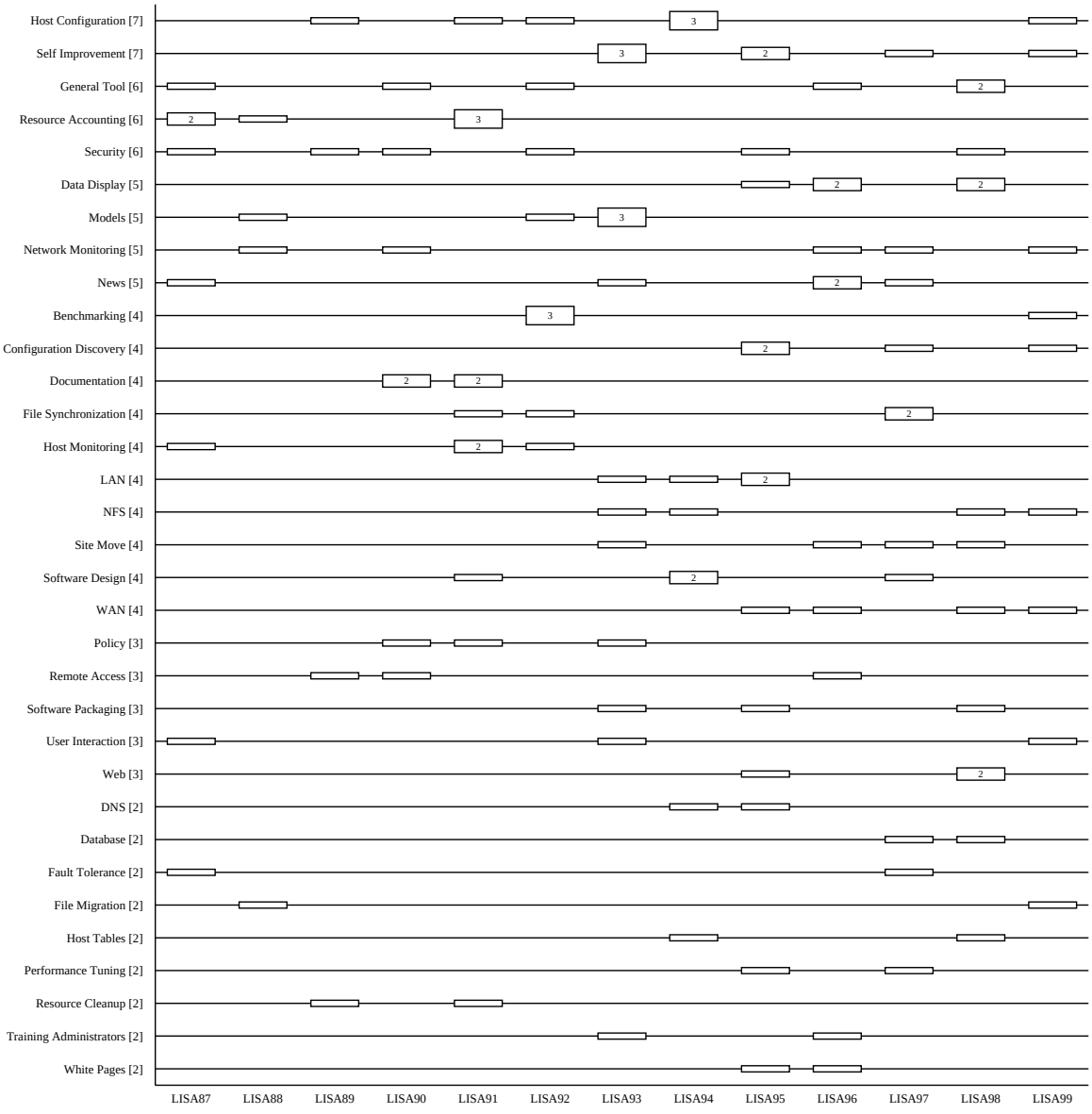


Figure 2.4: Continuation of Figure 2.3 for categories with 2-7 papers overall. Sorted by popularity of a category, ties broken alphabetically. This figure is included for completeness, but care should be taken in drawing conclusions given the small number of papers.

the papers, and examine this history of the field, we can find that this is mostly true, but with a small sample size, it is risky to draw any firm conclusions.

2.3.2 Source model trends

Figure 2.5 shows the papers over the last twelve years categorized by the three-state problem source model shown in Figure 2.2.

We can see that the number of training task papers has been remarkably small. In fact, further examination of the papers in those categories indicates that they are mostly papers on improving the skills of administrators. The one oddity is LISA93, in which a third of the papers were on many different training issues. Some of the training papers cover software design issues for administrators, others suggest how to improve interactions with other administrators, users or managers. A few of the training papers cover how to train new administrators, but surprisingly none of the papers cover training users to take better advantage of software or provide better problem summaries. Training is an area where some work should be done, although it is more difficult to analyze because it involves people. We suspect that some crossover with the field of sociology would shed light on these problems, but have insufficient experience to be sure.

We can also see that maintenance tasks comprise the second largest fraction of papers. Unfortunately, interrupt-style maintenance tasks contribute greatly to administrator stress. Beyond simply eliminating maintenance tasks by having systems automatically repair themselves, we should strive to convert maintenance tasks to schedulable tasks. We could designed to operate in degraded mode by minimizing the impact of failures impact on end users. If degraded mode resulted in only a slight slowdown, administrators would not have to respond immediately every time a problem occurred, but instead could delay responding until a related problem occurs.

Finally, we can see that configuration management tasks are the most prevalent of the papers, which is unsurprising given that many tasks eventually require some change in configuration. The authors focus on the automation of those changes. Moreover, configuration tasks generally lead to a tool, and tools are easier to write a paper about than are solutions from the other two categories.

2.4 Examination of important tasks

We now examine the important tasks performed by system administrators in more detail. We summarize the area and examine the research history. In the research history, we reference some of the better papers on each topic, so that readers intrigued about a particular topic will be able to find additional information on that area.

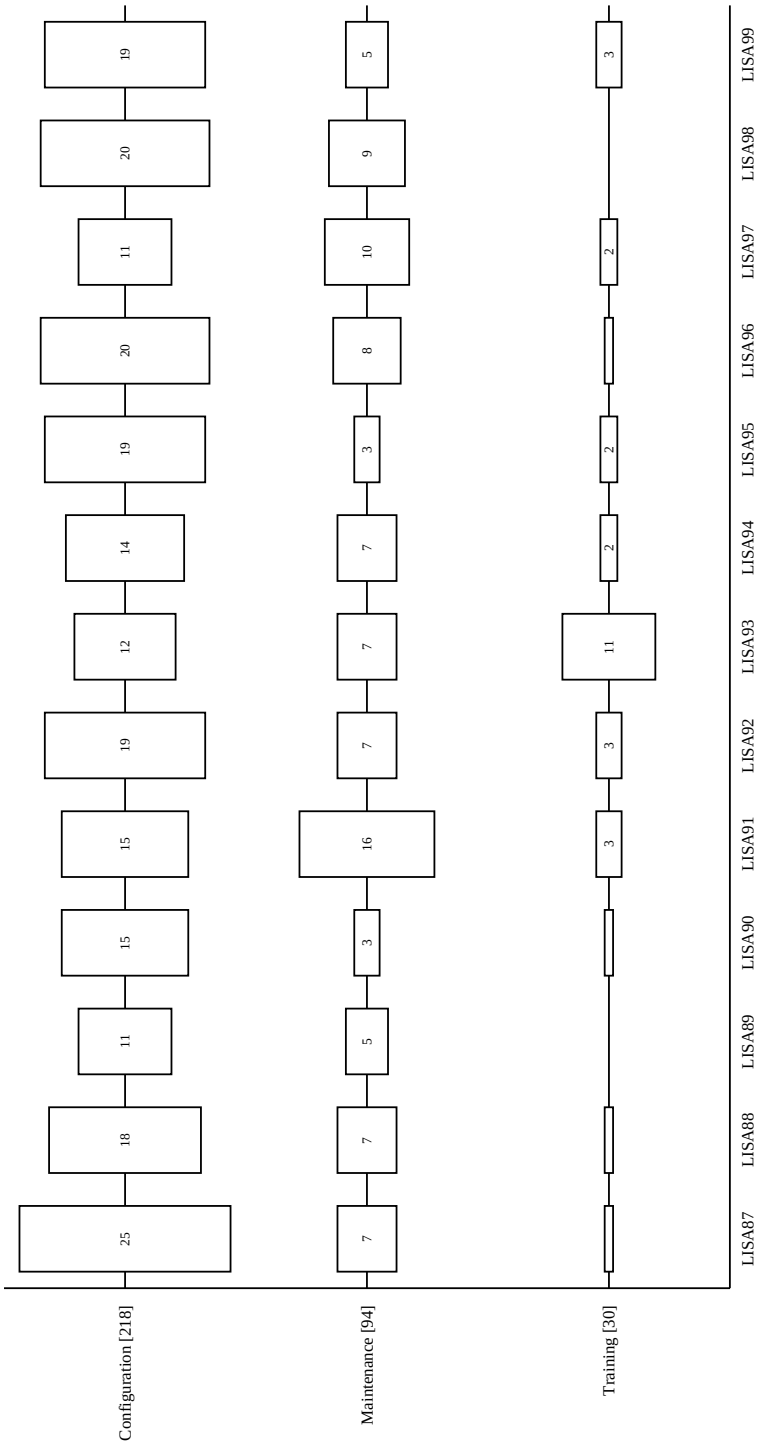


Figure 2.5: Breakdown of number of papers by year and category. They are sorted by popularity. Height of a box, and the number inside, indicates the number of papers. The total number of papers in a category is shown in brackets after the category name.

2.4.1 SW installation: OS, application, packaging and customization

There are four categories of software installation: Operating System (OS) Installation, Application Installation, Software Packaging, and User Customization. Operating system installation deals with the problem of taking the raw machine and putting the operating system on it so it can boot. Application installation is the addition of optional (non-OS) packages to a machine. Software packaging is the step of creating an installable package. User customization happens when users need to change the way the software operates.

OS installation usually puts files in specific places and has limited support for multiple versions on a single machine. Research into operating system installation has taken a cyclic path. In the very beginning, the OS was installed by either cloning a disk and then putting it in the new machine, or by booting the new machine off some other media (e.g. floppy disk, network) and then copying an image to the local hard drive. Those solutions were then modified to support customization of the resulting installation and easier upgrades [Zwi92, Hid94]. The tools were then scaled to allow fast installation across the entire enterprise [SMH95]. By then large-scale PC OS installation needed to be supported, and the cloning solution [Tro96] reappeared.

Application installation usually puts packages into separate directories, and uses symbolic links to build composite directories, so multiple versions are easily supported, and programs can be beta tested easily before being made generally available. Application installation has had many more papers written on it than OS installation, probably because vendors did not supply tools to install additional applications. The initial solution was to build packages in separate directories and link them into a common directory [MWCR90, CW92]. These tools were then extended to support customization per host [Won93]. Recently, the caching and linking pieces were untangled and refined into separate tools [Cou96, Bel96].

Relatively few papers have been written on software packaging, probably because most of the application installation tools use source code trees rather than binary packages. These papers cover the patching of software for different host types, and the subsequent generation of installation packages [Sta98].

The papers on user customization cover two separate areas of customization: Selecting which packages are accessed by a user [FO96, WCM93], and customizing application behavior [EL92]. The package selection tools started as simple shell scripts that adjusted environment variables to enable packages, and later were refined to work faster and more flexibly. The customization tools have dealt with different aspects of making it easer to control the behavior of programs and have been targeted at beginning users.

Software installation has commonly focused on the principles of automation, flexibility and scalability. All of the tools have been designed to automate some sort of task. OS installation has most clearly taken the path of scalability, starting at approaches which scaled to tens of machines in a day and growing to approaches which handled thousands in a day. Application installation has mostly focused on flexibility,

with some of the symbolic linking approaches also improving transparency as they allow the administrator to determine the package responsible for a particular file. The user-customization work is, of course, driven by the principle of flexibility, but also is an application of simplicity.

2.4.2 Backup

Backup addresses four separate, but related problems: User Error, Independent Media Failure, Correlated Media Failure (e.g. Site Failure, Software Error), and Long Term Storage. All the solutions are based on some type of redundant copy, but the particulars of each are different. Damage due to user error can be reduced by online filesystem snapshots. Independent media failure can be remedied by techniques like RAID. Correlated media failure requires use of additional uncorrelated media (e.g. Off-site tape, remote duplicates with different software). Finally long term storage requires very stable media, and an easily read format. Consider how few people can still read data written on punchcards, or even 9-track tape. Most of the focus in backup has been on independent media failure, usually by creating copies on tape, although people have looked at the other issues.

Research on backup has passed through many stages. The first was correctness: Does the right data get written? [Zwi91] Are backups happening regularly and on schedule? [MK92] Do restores work? Having achieved correctness, research turned to scaling backup solutions to the enterprise. The solution was staging disks so that backups could stream to tape [dSlG93]. Having solved the correctness and scalability problems, research on backup paused. But then the onward march of technology reintroduced scalability as a problem. Disk bandwidth and capacity are starting to outstrip tape bandwidth and capacity, leading to solutions requiring multiplexing of disks and tapes [Pre98].

Research on backup has commonly focused on the principles of dependability and scalability. The basic purpose for backup is to recover data after some sort of loss. Not arbitrarily losing data is, of course, a property of a dependable system. The initial research on backup was all about dependability. The followup research has been about scaling backup solutions as the underlying technology changes.

2.4.3 Configuration: site, host, network, site move

Configuration tasks are modification to the setup of hardware and software so that the environment matches the requirements of a particular organization. Simple configuration tasks include installing the appropriate exports and resolv.conf files. Complicated ones include migrating from an MVS platform to a UNIX one and purchasing the new system. Wise administrators will plan for a configuration change before it becomes an emergency.

The first few LISA conferences included many papers which summarized their site's configuration. Research then forked in two directions. Some papers looked at how to store and extract configuration

information from a central repository, either using available tools such as SQL [FS89], or by designing their own language [RM94]. Other papers looked at using a level of indirection to make configuration changes transparent to users [Det91].

The great growth spurt in the computer industry lead to complete site moves, either as part of a merger, separation, or just to handle growth [Sch93]. Similarly, the great amount of research in this area led some people to examine the question, "What properties of site design make it easier to administer?" [TH98]. Recently, a mobile user base caused dynamic network re-configuration to become a problem [VW99].

Configuration tasks have commonly focused on the principles of automation and flexibility. The research of extracting configuration information was primarily to automate this previously human-intensive task. The automatic extraction of configuration information also helped with the transparency of the system. The research on setting configurations was also about automation, and as the complexity of the desired configurations in creased, flexibility became important.

Configuration is probably the weakest categorization. The original intent was that host configuration would cover host issues, network configuration would cover network issues, and site configuration would cover global site issues. However, the line between host and site is at best blurry. We therefore believe that someone should re-examine the papers in these areas, and see if they can find a better categorization.

2.4.4 Accounts

Managing user accounts at first seems very simple. But upon further examination, we find there are additional subtleties because an account identifies users. This identification leads to associated real world meaning such as security and privacy. Therefore, authentication, rapid account creation, and managing the associated user information become important.

Accounts research started with the goal of simplifying the account creation process. Scripts were designed that automated the steps of accumulating the appropriate information about users, adding entries to password files, creating user directories, and copying user files [CKCS90]. Because the scripts were site-specific, they were able to do better error checking. Once creating accounts became easy, accounts research paused until enough people needed accounts that scalability became a concern. Sites with thousands of accounts, usually schools, needed to create many accounts quickly because of high turnover in the user population. Their solutions tended to have some sort of central repository storing account information (often an admissions' database), with complementary daemons on client nodes to extract the needed parts of the database [Spe96]. Some of the recent papers considered auxiliary details such as limiting accounts to certain hosts, account expiration, and delegating authority to create accounts [Arn98].

Accounts research has been driven by the principles of automation and simplicity. Initial research focused on simplifying the process of creating accounts to reduce errors. Sites with thousands of users

required a great deal of automation to handle account setup. Finally, some of the recent research simplified account creation enough to remove the administrator from having to do any of the work.

2.4.5 Mail

Electronic mail has been one of the driving applications on the Internet since its inception. This role makes it unsurprising that it ranks extremely high on the list of applications. It is the highest of the applications that are used by end-users on a regular basis. There is a vast amount of email, traveling around the world-wide network, leading to a lot of effort in interoperability and scalability.

Early research in mail targeted interoperability between the wide variety of independently developed mail systems. This research and the reduction in variety over time, combined with SMTP as a standard mail interchange protocol, solved the interoperability problem. Research then turned to flexible delivery and automating mailing lists [Cha92]. There was then a brief pause in the research. However, as the Internet continued to grow, research on scaling delivery of mail both locally and in mailing lists [Kol97] was needed. At the same time, commercialization caused SPAM to become a problem [Har97].

Mail has commonly focused on the principles of flexibility, automation, and scalability. The initial work in interoperability was about making the work flexible enough to deliver between systems. The mailing list work was done to automate the initially human-intensive task of administering mailing lists. Scalability arrived as sizes continued to grow requiring new techniques to manage ever larger lists.

2.4.6 Monitoring: system, network, host, data display

Monitoring solutions help administrators understand what is happening in the environment. There are problems of system, network and host monitoring, and the associated problem of data display. Monitoring solutions tend to have two variants: instantaneous and long term.

Research in monitoring has progressed along a number of axes. First, there has been work in monitoring specific sources from file and directory state [RL91] to OC3 links [AkcTW96]. Simultaneously, generic monitoring infrastructure [HM92, AP97] has been developed. Finally, as the amount of data available has increased, some work on data display has been done [Oet98].

The categorization here was by the type of thing being monitored (host, network system). Perhaps a better classification would be by the axes described in the research history.

Monitoring research has commonly focused on the principles of transparency, scalability, and notification. The focus of monitoring is determining what is going on inside of a system, and hence is in support of transparency. As the size of systems increased, work was necessary on scalability to handle larger clusters. Some of the systems incorporated support for notification via paging an administrator when a problem occurred.

2.4.7 Printing

Printing covers the problems of getting print jobs from users to printers, allowing users to select printers, and getting errors and acknowledgements from printers to users.

Early research in printing merged together the various printing systems that had evolved [Fle92]. Once the printing systems were interoperable, printing research turned to improving the resulting systems, making them easier to debug, configure, and extend [PM95]. As sites continued to grow, scaling the printing system became a concern, and recent papers have looked into what happens when there are thousands of printers [Woo98].

Printing research has commonly focused on the principles of flexibility, scalability, and simplicity. The initial research worked on making the different systems interoperate in support of flexibility, and then made them easier to work with, again for flexibility. Site growth drove the need for scalability, and the resulting complexity of the system required some simplicity for the access to the system for users.

2.4.8 Trouble tickets

Trouble ticket tools simplify the job of accepting a problem report, assigning the problem report to an administrator, fixing the problem, and closing the problem's ticket. Trouble ticket systems usually have a few methods for getting requests into the system (e-mail, phone, GUI), and provide tools for querying and adjusting the requests once they are in the system.

Trouble ticket systems began as email-only submission tools with a centralized queue for requests [GHN90]. Later, the systems were extended so that users could query the status, and tickets could be assigned to particular administrators [Kob92]. The systems were improved to support multiple submission methods such as phone [Sco97] and GUI, and to support multiple request queues [Rue96].

Trouble ticket research has commonly focused on the principles of automation, flexibility, and notification. They have been designed to provide a tool for managing the status of problems rather than human editing of files. Many of the tools automatically notify users or administrators when the trouble ticket status changes.

2.4.9 Secure root access

Security in general is the problem of protecting systems and data against non-authorized individuals. For the purpose of Unix system administration, one of the key problems has been providing access to the special privileged user root. Secure root access is the general problem of providing temporary privileges to a partially trusted user. Many actions need to be taken as root, and giving out the root password is clearly a poor decision. The questions then are how to give out privileges, how to track their use, and how to retain some amount of security.

	Principles							
Categories	Dependability	Automation	Scalability	Flexibility	Notification	Schedulability	Transparency	Simplicity
SW installation		X	X	X				
Backup	X		X					
Configuration		X		X				
Accounts		X						X
Mail		X	X	X				
Monitoring			X		X		X	
Printing			X	X				X
Trouble tickets		X		X	X			
Secure root access			X				X	
TOTAL	1	5	6	5	2	0	2	2

Figure 2.6: A comparison of the categories described in details and the principles commonly found in the papers in that category. X's are put where principles showed up most commonly, for any given paper, it may or may not address the principles described.

Research in secure root access has gone down two separate paths. One path has been to examine how to provide secure access to commands within a host. This has gone through many iterations, slowly adding in more complex checking of programs and arguments [MHN+, Hil96]. The other has been to provide secure access remotely [RG95].

Secure root access has commonly focused on the principles of flexibility and transparency. The tools have been improved to add more and more complex argument checking so that they are more flexible. Furthermore, all of the tools have been designed to keep logs so that it is transparent what modifications have been made.

2.5 Conclusions and analysis

We have categorized all of the papers in the first 12 years of the LISA conference according to two separate models. We have made the categorization available so that others can examine our choices, correct mistakes, or provide better categorizations. Hopefully, this chapter will encourage people to think differently about the field and problems that it presents, and as a result build better tools and processes. Figure 2.6 lists

the categories and shows their relationship to the principles from the first chapter. Some of the principles, such as dependability have not gotten the direct focus that they should have, although dependability is an implicit goal in many of the papers.

We have examined the historical trends of the LISA conference according to the two models. Trends help us see that some areas are under served, and some are probably over-served. We can also see the bursty nature of research in system administration, probably because the same problem occurs to everyone at the same time. As a result, we recommend that a central clearinghouse of problems be created to facilitate collaboration and improve the resulting tools.

Finally, we examined some of the important task areas. We have looked at the history of the research in each area, as well as the principles that are most related to each task. In Chapter 6, we propose based on the same task areas, a number of directions for future research. We believe that this sort of analysis should be performed every few years. The Database community gets together and decides which areas of research were successful, and which require more work [SSU91, SSU96]. Their reports have helped their community show their results and focus their efforts. Hopefully, this analysis of system administration will be a starting point toward doing the same for system administration.

Chapter 3

CARD: extensible, scalable monitoring for clusters of computers

When we started looking for an initial project in system administration, monitoring stood out as an important problem. We had found it was roughly as important as configuration from a time survey [And95], and the Network of Workstations (NOW) [ACPtNt95] project was building the NOW-2 cluster consisting of over 100 nodes. Previous work in monitoring had not attempted to scale up to that many nodes at high update frequency, so we decided that would be an excellent challenge. The work was then published in the LISA '97 systems administration conference [AP97]. This approach to system administration research we refer to as "Let the administrator handle it."

Recall from Chapter 1 that we examined eight principles of system administration, sorted in order of estimated importance: Dependability, Automation, Scalability, Flexibility, Notification, Schedulability, Transparency, and Simplicity. This monitoring work was focused on scalability (#3), flexibility (#4), and dependability (#1). There was existing work on notification (#5) for a monitoring system, so we did not examine that problem. As we describe below, the system achieved scalability and some flexibility, but turned out to be hard to automate (#2) because of the complexity in the implementation, reducing the system's dependability (#1). It is likely that a re-implementation of similar ideas could fix the complexity and resulting dependability problems while still keeping the scalability and flexibility.

We address four monitoring problems in CARD (Cluster Administration using Relational Databases). First, we handle the evolution of software and hardware in our cluster by using relational tables to make CARD flexible. Second, we use timestamps to detect and recover from node and network failures, making it more dependable. Third, we improve data scalability by using a hierarchy of databases and a hybrid push/pull protocol for efficiently delivering data from sources to sinks. Fourth, we improve visualization scalability by statistical aggregation and using color to reduce information loss. In our prototype implemen-

tation, CARD gathers node statistics such as CPU and disk usage, and node information such as executing processes. We synthesized and adapted research from other fields to help solve these monitoring problems.

The remainder of this chapter is structured as follows. Section 2 describes our four solutions, section 3 describes our experience with our implementation, and section 4 describes the related work. Section 5 summarizes our conclusions from building the system. This chapter is based on [AP97].

3.1 Four problems and our solutions

We now describe our solutions to four problems of monitoring large clusters. First, we explain how we handle the evolution of software and hardware in a cluster. Second, we explain how we deal with failures in the cluster and our software. Third, we explain how we increase data scalability. Fourth, we explain how we display the statistics and information from hundreds of machines.

3.1.1 Overview

We make CARD flexible and extensible by gathering and storing the data in relational tables [Cod71]. Because the tables use named columns, old programs do not have to change as new types of data are added. Our prototype uses SQL [CAE+76] to access the data, so in addition to providing data for the visualization applet, administrators can execute ad-hoc queries. The column names help administrators understand the structure of the data when browsing, and the database includes tables that describe the columns in more detail.

We use timestamps to detect and recover from failures in CARD and the cluster. Since data is updated periodically, failures are detected when the updates stop. The timestamps also help for getting a consistent view of changing data. Finally, stale data is expired when the timestamps are too old.

We scale our data capacity as machines are added by building a hierarchy of databases. The hierarchy allows us to batch updates and infrequently update upper-level databases, and to specialize nodes to interesting subsets of the data. Specialization and infrequent updates reduce the scope and the freshness of the data, however, the full, fresh data is still available from the leaf-level databases.

We gracefully scale the amount of data displayed in a fixed amount of space through statistical aggregation of data. We then reduce the information loss by using different shades of the same color to display dispersion. These two techniques have allowed us to meaningfully display multiple statistics from hundreds of machines.

We reduce the amount of requests and data transferred over the network by using a hybrid push-pull protocol. Our protocol sends an initial SQL request and a repeat rate. The query is executed repeatedly,

and the results are forwarded to the requester. The hybrid protocol achieves the best of both a request-response (pull) protocol and an update (push) protocol.

3.1.2 Handling rapid evolution using relational tables

Cluster software is evolving at a rapid pace, so a monitoring system needs to be extensible to keep up with the changes. This evolution means that new data will be placed in the system, and usage of the data will change. A system with only one way of storing or querying data will have trouble adapting to new uses.

We believe that flexibility and extensibility can be achieved by using a relational table to store all of the data. The table format increases flexibility by decoupling the data users from the data providers, which means that arbitrary processes can easily put information into the database, and arbitrary consumers can extract the data from the system. The database increases extensibility because new tables can be easily added, and new columns can be added to existing tables without breaking old applications. Queries only address columns in tables by name, and hence the new columns do not affect the old queries. Finally, if a full database is used, then SQL queries can combine arbitrary tables and columns in many ways, and the database will automatically use indices to execute the queries efficiently. As the use of the database changes, new indices can be added to maintain the efficiency of the queries.

Using data structured into tables and made available remotely over the network is a significant departure from previous systems. They generally use a custom module for data storage and only a few provide any remote access to the data[AkcTW96, Dol96, Fin97, HA93, HM92, SB93, SL93, SA95, SW91, Sim91, Wal95]. Although building an integrated module can increase efficiency for a single consumer of the data, some of that improvement is lost with multiple consumers. Furthermore, the flexibility of the system is reduced because adding new data producers and consumers is more difficult. Indeed, by using a relational structuring throughout the system, we can specialize the database implementation based on the usage pattern, using a fast in-memory database for local nodes, and a full database for long term storage and analysis.

3.1.3 Recovering from failures using timestamps

The second problem we address is detecting and recovering from failures. We use timestamps to detect when parts of the system are not working, identify when data has changed, and determine when data has become old and should be rechecked or removed.

Timestamps help detect failures when data sources are generating periodic updates. If the timestamp associated with the data is not changing, then the check has failed, which indicates that the remote node is either slow or broken. This solution works even if the updates are propagating through multiple

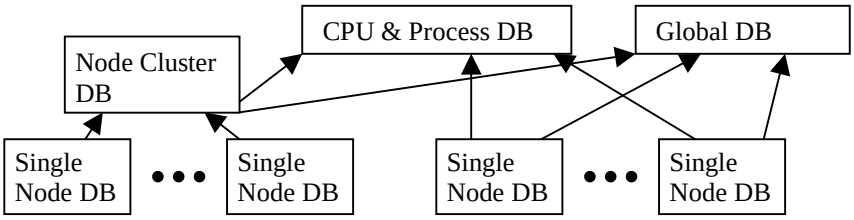


Figure 3.1: A hierarchy of databases. At the lowest level are single node databases. These hold information gathered from a single node. The top level shows a few forms of specialization. The node cluster database gathers information about all the single nodes in its cluster. The CPU and process database stores a subset of the data at the full frequency, and takes advantage of the batching possible because of the node cluster database. The global database stores all of the information about the cluster, but at a reduced frequency.

databases in the hierarchy because the timestamps are associated with the data and do not change as the data moves.

We also use timestamps for consistency control [CL85]. Timestamps allow quick comparisons of data to determine if it has been updated. We have a timestamp associated with both the data and the time for when the data was placed in the database. Remote processes maintain a last timestamp (t0). To synchronize with the database, they get a new timestamp from the database (t1), get all the data that was added since t0, and set t0 to t1. By repeating this process, the remote program can be kept weakly synchronized with the database. Moreover, if the machines' time is synchronized [Mil95], then the remote program also knows the freshness of their data. Timestamp consistency control is very simple to implement in comparison to other consistency protocols [GLP75], and if the database is accessible, then the data is always available regardless of other failures in the system, whereas other protocols may deny access to ensure stricter consistency.

Finally, we use timestamps to eliminate stale data. Stale data can occur because of failures or removals. The timestamps allow the data to be automatically removed after a table specific period, which means that the system will automatically recover to a stable state. Multiple timers allow slowly changing data like physical memory to be updated infrequently yet not be declared stale.

3.1.4 Data scalability using hierarchy

Systems that can take advantage of multiple machines are usually more scalable. Figure 3.1 shows a hierarchy of databases possible in our system. Using a hierarchy provides several benefits.

First, a hierarchy allows updates to a database to be batched. Batching updates reduces the number of packets that need to be transmitted over the network. Batching is possible in part because the individual updates are not serialized and hence the latency of the network is less important. Finally, batch updates can be processed more efficiently by a database.

Second, a hierarchy allows specialization of nodes. Although a single database may not be able

to handle the full update rate for all of the information that is being collected, a single database may be able to handle a useful subset of the data at the full rate. For example, statistics about CPU usage and processes could be stored in a single database, allowing it to handle more nodes. Furthermore, the nodes at different positions in the hierarchy can be specialized to the usage. The node-level databases could only support simple select queries and keep the data in-memory with only intermittent flushes to disk.

Third, a hierarchy allows reduction in the data rate. For example, an upper level database could keep all of the information gathered at an interval of a few minutes. As the amount of data gathered grows, the interval can be reduced. Infrequent updates allow a single database to keep a complete, but more slowly changing, copy of the database.

Fourth, a hierarchy over multiple machines allows for fault tolerance. Multiple databases can be storing the same information, and hence if one of the databases crashes, other nodes will still have access to the data.

3.1.5 Data transfer efficiency using a hybrid push/pull protocol

Our system needs to efficiently transfer data from sources to sinks because we have to transfer data both within the hierarchy and to programs that are using the data. Most previous work used pull-based transmission (polling); a few used push-based transmission (updates). The choice of a particular method depends on the use of the data. Infrequent updates work well with pull, but as the frequency increases, push become more efficient. To improve the flexibility of our system, we have developed a hybrid push-pull protocol to minimize wasted data delivery, maximize freshness, and reduce network traffic.

The canonical pull protocol is RPC [Sun86]. SNMP [CFSD90, CMRW96], a protocol used for monitoring, is also a mostly pull protocol. A pull-based system requires the sink to request every piece of data it wants from the source. If the sink wants regular updates, it polls the source. Since the source knows it wants regular updates, all of the request packets are wasted network bandwidth. Furthermore, if the data is changing slowly or irregularly, some of the polls will return duplicates or no data. However, polling has the advantage that since the data was requested, the sink almost always wants the data when it arrives. Between polls, the data goes steadily out of date, leading to a tradeoff between the guaranteed freshness of the data and the wasted traffic.

Pointcast [Poi97] and Marimba Castanet [Mar97] use a push protocol. A push protocol delivers data all the time, forcing sinks to discard data if they did not want it. Multicast [DC90] prunes the distribution tree for the packets it pushes to receivers as they indicate a lack of interest. Broadcast Disks [AFZ97] distribute data to all receivers using the underlying broadcast nature of some physical networks.

A push system is ideal when the sink's needs match the source's schedule since the data is current, and the network traffic is reduced because the sink is not generating requests. However, if the sink does not

want the data, the network traffic to transmit the data was wasted. Furthermore, sinks have to wait until the source decides to retransmit in order to get data. These conflicting forces lead to a tradeoff between wasting bandwidth with un-needed updates and delaying updates to clients.

We use a hybrid push-pull model. Sinks send a request to a source along with a *count* and an *interval*. The source will process the request until it has sent *count* updates to the requester. The source will suppress updates that occur more frequently than *interval*, and updates where nothing has changed. If a sink wants the pull model, it sets the *count* to one. If a sink wants the push-model, it sets the *count* to infinity. If the sink wants updates for a certain period of time, it sets the *count* and *interval* to intermediate values. The *interval* allows the sink to reduce the rate of updates so that it is not over-run. When the updates are no longer needed, the sink can cancel the request.

In our implementation we use SQL to describe the request. This allows sinks to precisely describe the data they want, instead of indiscriminately getting information as happens in other push systems. Conveniently, this protocol is only slightly more complicated than a push protocol; the only addition is the addition of the count, as any periodic push already required the specification of an interval.

3.1.6 Visualization scalability using aggregation

We have found that we need to use aggregation to scale the visualization to our whole cluster. We tried having a strip chart for every statistic we wanted, but ran out of screen space trying to display all of our machines. We therefore aggregate statistics in two ways: First, we combine across the same statistics for different nodes. For example, we calculate the average and the standard deviation across the CPU usage for a set of nodes. We then display the average as the height in the strip chart, and the standard deviation as the shade. Second, we aggregate across different statistics. For example, we combine together CPU, disk and network utilization to get a single statistic we call machine utilization.

Aggregating data together risks losing important information. We minimize this effect by taking advantage of shade and color when displaying the data. We use shade to indicate the dispersion of the data that has been aggregated. Highly variable data is therefore darker, taking advantage of the eye's ability to perceive a large number of shades [Mur84, HSV]. We use color to help draw distinctions and identify important information. For example, we use different colors for the I/O and user CPU usage, and we identify groups of down machines in red. Figure 3.2 shows both the use of shade and color from a snapshot of our system while running.

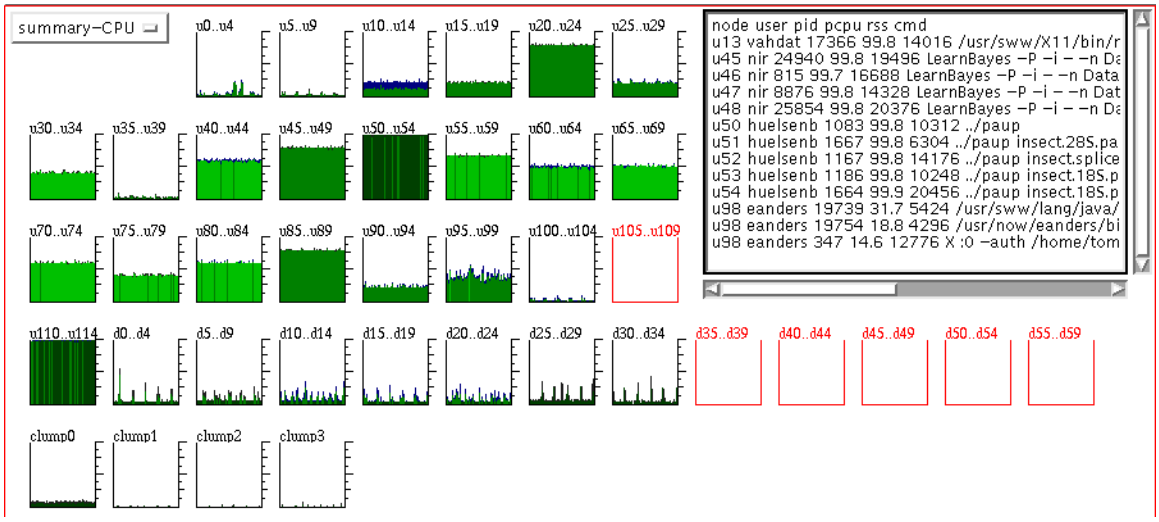


Figure 3.2: Snapshot of the Java interface monitoring our entire cluster which consists of 115 Ultra 1's (u0-u114), 60 Sparc 10's or 20's (d0-d59), and 4 Enterprise 5000 SMP's (clump0-clump3). Aggregation has been done with averages (height) and standard deviation (shade) across groups of five machines except for the SMP's. The darker charts are more balanced across the group (u50..u54 all are at 100%), and the lighter charts are less balanced (u40..u44 have three nodes at 100% since the average is 60% and the usage is not balanced). All charts show the system CPU time in blue over the green user CPU time; u13 has a runaway Netscape process on it. u105-u109, and d35-d59 are down or removed from the cluster, and so shown in red. Processes running on selected nodes are shown in the text box in the upper right hand corner. The figure was post-processed to remove the gray background normally present in the Java applet to make it print better.

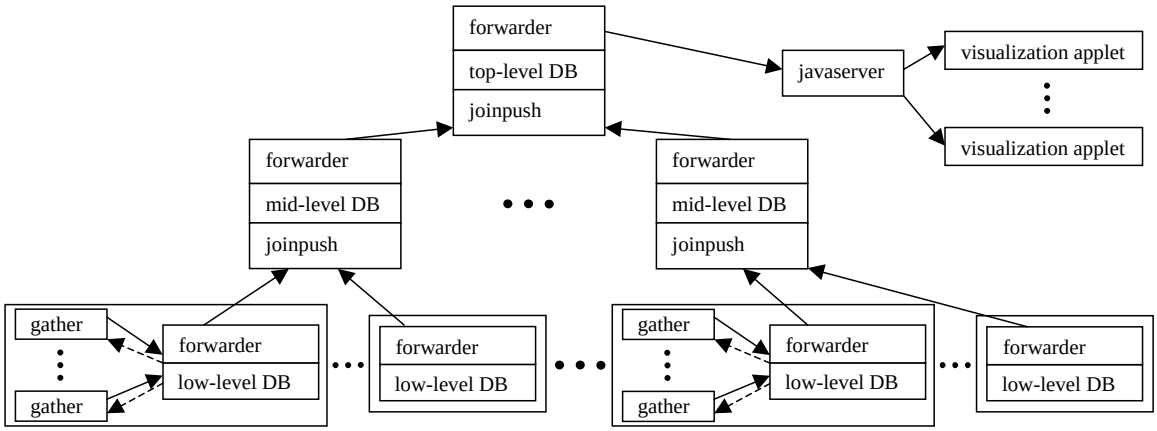


Figure 3.3: Architecture of our system. The gather processes are replicated for each forwarder/node-level DB group. The top level databases can also be replicated for fault tolerance or scalability. The javaserver acts as a network proxy for the visualization applets because they can not make arbitrary network connections. The forwarder and joinpush processes are associated with a database and serve as the plumbing that moves data through the system.

3.2 Implementation

We had a number of choices to make to implement our solution. We first chose to implement the relational tables using the MiniSQL [Min97] database. This in turn caused us to implement timestamp recovery and the hybrid push/pull protocol as programs outside of the database. We wrote a Java applet for the visualization, which forced us to write a Perl script as an intermediary between the applet and the databases. Finally, we implemented startup and shutdown of our system as an outside program that used `rsh` to get to each node.

Figure 3.3 shows the data flow among the major components of our system, explained in more detail below. The `forwarder` process, associated with each database, accepts SQL requests from sinks, executes them periodically, and forwards the results to the sinks. The `joinpush` process merges the updates pushed from the `forwarder` processes into the mid-level databases, and then the process is repeated for the upper-level databases. The databases are shown sandwiched between the forwarder and the joinpush because that is logically how they operate. For a small network, there might only be a two level hierarchy. The `javaserver` process acts as a network proxy for the Java visualization applet because applets cannot make arbitrary network connections. The `visualization` applet accepts updates from the `javaserver`, and displays the data in strip charts or a text window for the user. The applet also provides a simple way to select a pane of information to view.

Figure 3.4 summarizes the implementation properties of each part of our prototype.

Component	Language	Lines of code
gather	Perl	500 (core) + 100/data-source
forwarder	C	500 + 500 shared with joinpush
joinpush	C	800 + 500 shared with forwarder
javaserver	Perl	300
visualization	Java	600
table definitions	SQL	300
node-startup	Perl	400
remote-restart	Perl	500
total	N/A	4200

Figure 3.4: Implementation properties of the components in our prototype. Lines of code have been rounded to a multiple of 100. The table definitions, node-startup and remote-restart components are only shown in the table; they are used to initialize the database, start a single node, and check and restart failed nodes.

3.2.1 Storing relational tables

We chose to use MiniSQL [Min97] to store our relational tables because it is freely available to universities. We did not expect the SQL limitations in MiniSQL to be a problem as we were using simple SQL queries.

In addition, because MiniSQL comes with source code, we were able to extend it when necessary. For example, we added micro-second timestamps to the database so that we could mark data changing on a short time-scale. We also modified the client software to support batched updates by only waiting for a response from the server at the end of a batch.

3.2.2 Building the hierarchy with the hybrid push/pull protocol

We have implemented the hierarchical structure as shown in Figure 3.3 with the `forwarder` program and the `joinpush` program. All `forwarders` and `joinpushes` are associated with a database running on the same node. The `forwarder` program implements the sending side of our hybrid protocol by repeatedly polling the database for updates, and if there are updates for a particular client, forwarding them along. The `joinpush` program implements the receiving side of our hybrid protocol. It is responsible for contacting the appropriate forwarders to build the hierarchy, merging together the updates from the various forwarders, and pushing the updates into its associated database.

The forwarder and joinpush programs are both implemented in C taking advantage of Solaris threads in order to achieve better efficiency. We initially tried implementing those programs in Perl, but the code was too inefficient to support 150 nodes, and using threads reduced concerns about blocking while reconnecting.

Figure 3.5 shows the architecture of the forwarder. The accept thread gets outside connections from clients and immediately forks a client thread to deal with each client. The client threads take requests

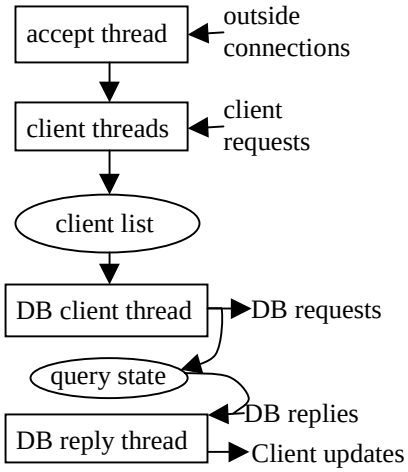


Figure 3.5: Architecture of the forwarder. The left column shows either threads or important data structures in the forwarder. The right column shows the interactions with other processes.

for updates from the clients, and put those requests in the structure associated with each client. The use of threads allows us to easily handle slow clients without concerns of blocking. The database client thread walks the list of clients, and issues the requests which are pending to the database. When the response comes back from the database, it is matched with the information stored at request time, and the reply thread sends any new updates to the appropriate client.

Figure 3.6 shows the architecture of the joinpush process. The list of forwarders and the data to request is configured externally to joinpush to simplify the implementation. The reconnect thread forks separate threads to connect to each of the forwarders and issue a request. When a connection is made, the connection is added to the connections list, and the update thread waits around for updates from any of the forwarders. It generates an appropriate database update. The reply thread will generate an insert request if the reply indicates that the data was not yet in the table.

3.2.3 Visualization applet

We chose to display the information using a Java applet. The advantage of this choice is that data can be viewed from any Java enabled browser. There are also a few disadvantages. First, a network proxy has to be written to connect the applet to the databases because of the security restrictions in java. Second, the monitoring system is dependent on a running web server. In our implementation, all of the work is done in the network proxy, so that the applet remains extremely simple.

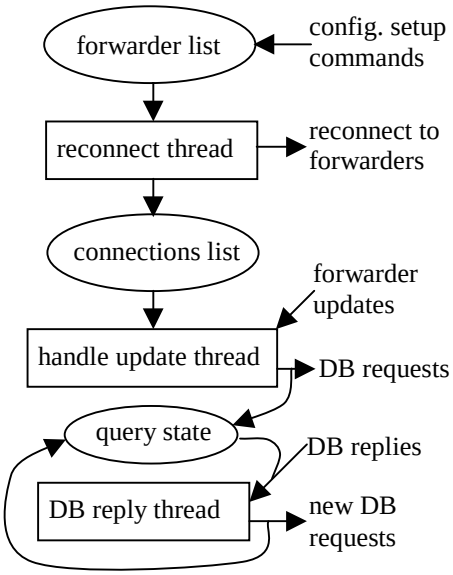


Figure 3.6: Architecture of joinpush. The left column shows either threads or important data structures in joinpush. The right column shows the interactions with other processes. The configuration commands are handled analogously to how clients are handled in the forwarder.

3.2.4 Gathering data for the leaf databases

Data is added to the system by the gather process, which is also implemented in Perl. We originally examined a threaded implementation of the gather process, but we were unable to get a sufficiently stable multi-threaded implementation. We therefore use a multi-process implementation. We have a directory of files which all get spawned by the first process, which then waits around and flushes old data from the database. We currently have processes that extract CPU, I/O, network, process and machine statistics.

3.3 Experience

Our experience with CARD has been mixed. It has allowed us to discover problems with the cluster, an example of that was shown in figure 3.2. However, it has had sufficient difficulties to keep us from using it on an ongoing basis.

Our experience using relational tables has been very positive. We initially stored all of our configuration information in separate, per-program files. We found that the database provided a convenient, centralized location for configuration information, and as a result, moved all of if into relational tables. We implemented a simple caching mechanism for local nodes so that they could continue to function even if the master configuration database was unavailable. This choice improved the simplicity of our system because it meant we used a single method for transporting information rather than having to use a separate method to move configuration information.

Resource usage of MiniSQL did not initially appear to be a problem. The database uses 1-2% of the CPU on an Ultra 1/170 to update 20-30 statistics a second. The upper level databases seem to peak somewhere between 1500 and 2000 updates/second. We found this utilization acceptable when we were testing our system. However, as we started using it while running cluster-wide parallel problems, the monitoring system interfered with the parallel program. We found that the 1-2% CPU usage introduced by the monitoring system dramatically slowed down the SPMD parallel programs that people ran on the cluster. We did not find that the extra network traffic affected the programs, both because the network traffic was extremely low ($< .1\%$), and because many of the parallel programs used a separate network for communication. We estimated that we would need to get the CPU utilization down by an order of magnitude in order to make the impact un-noticeable.

We found that some optimizations in our system increased its complexity and made it more difficult to manage. In particular, mapping semi-constant strings onto integer indices required keeping the mapping tables identical among all the instances. It was particularly confusing when two different nodes communicated using slightly different mapping tables. The data values that were reported were effectively meaningless.

One of the problems we encountered while developing the system was keeping it running as we made additional revisions of the software which would sometimes cause crashes due to bugs. We chose to implement a centralized process that checked on the existing nodes and restarted them if necessary. Unfortunately, we discovered that there were failure modes where the process stopped handling requests, either temporarily or permanently, but did not exit. Keeping those processes from accumulating resources required careful design.

We use two methods to ensure that old CARD processes terminate quickly. First, each process creates a pid file and locks the file. The reset-node operation attempts to lock each file and if the lock fails, the process is sent a signal. Using a lock guarantees that we will not accidentally try to kill off a second process which just happens to have the same process id. Second, we write an `epoch` file each time the node is reset. The epoch file stores the time that this node's monitoring was started. Existing processes can check the epoch file, and if they started at a different time than the epoch file, then they know they should exit. We added the second approach because we occasionally saw processes not exit despite having been sent a signal that should cause them to exit.

Having a centralized system to restart nodes worked tolerably while debugging, but required a list of the nodes participating in the system. The central node also had to be able to contact and log into the remote note to restart the monitoring system. The merging nodes also had a list of leaf nodes that they were supposed to contact, which led to polling from the merging nodes to determine when a leaf node has restarted.

We have found the timestamps associated with the data to be extremely useful. For example, an early version of CARD failed to send the timestamp from the javaserver to the visualization applet. When the javaserver generated repeated updates of old data, the client was fooled into thinking the system was performing correctly. As we now forward timestamps along with the data, we would detect this error.

The fact that we display information through a Java applet raises a few privacy concerns. In particular, outside users can see all of the statistics of the cluster. Given that we are an academic institution, we have not been very concerned about maintaining secrecy of our usage information. However, all of the standard techniques for improving privacy could be added to our system. For example, access could be limited by IP address, or secure, authenticated connections could be established via the secure socket layer [FKK96]. To ensure privacy, it would be necessary to apply this protection to the javaserver, the forwarder, and the MiniSQL server. To prevent bogus information from being added into the database, it might also be necessary to protect the joinpush process.

3.4 Re-implementing CARD

It is clear from our experience that CARD needs to be re-implemented to achieve our goals of flexibility and scalability without sacrificing CARD's dependability. This section describes the choices we would make in re-implementing CARD to address some of the problems described in the previous section.

We do not believe that we can sufficiently reduce the overhead of the system without tuning the database implementation to CARD. We have already used indices and replaced strings with integers to make the database run quickly, but the overhead of parsing SQL, and the generality of databases makes it unlikely that we will be able to achieve very low overhead.

There is evidence that writing a task-specific database can lead to performance improvements. The work of continuous profiling [ABD+97] showed that for their application, overhead of monitoring and storing the data could be reduced by many orders of magnitude from what we measured. It is therefore plausible to believe that a database system tuned for the in-memory, simple column matching uses found in CARD could reduce the CPU overhead by at least an order of magnitude.

We believe that the system would then become a hybrid system. On the individual nodes, a custom, tuned implementation would support the simple in-memory matching queries needed to extract information from a node. Each node would probably only need 128k of memory for the data, as there is only a small amount of information recorded on each node. The merged nodes could either still be the simple efficient implementation, or if desired a full database. The merged databases would be kept off of the cluster, so the resources used by database would not impact the applications running on the cluster. Furthermore, the updates to the full database could be batched, which further increases its efficiency.

A second benefit of the CARD-specific database would be that we could merge the gathering and

push-pull processes into the database. This change would further reduce the overhead on the leaf nodes, and in addition would reduce the complexity of the system running on those nodes by reducing the number of processes. To support the hybrid system, however, the separate `joinpush` and `forwarder` programs would remain for use with a more complex database.

A third benefit of the CARD-specific database would be to simplify the string to integer mapping that we used to speed up the database. Given that most of the strings in a monitoring system are constant, it is more efficient to compare them as integers rather than as strings. Our initial implementation kept a big table which handled the mapping, but we had difficulties with it getting out of date on some nodes. In the re-implementation, we believe that as pairs of processes communicate, they should start by exchanging their tables. The two processes can adjust the mappings so that they are the same on both nodes by re-numbering where necessary. By keeping a priority order on nodes, we can guarantee that this process will terminate with all the nodes sharing the same mapping.

Since the in-memory database would still represent the data as relational tables, and would implement the query operations using a subset of SQL, the only flexibility we have sacrificed on the individual nodes is the ability to make complex queries. However, those queries are usually not useful until you have more data collected together on one of the merged nodes, so we expect that we have sacrificed only non-useful flexibility.

After simplifying the programs running on the leaf nodes, we still have the problem of keeping those programs running with current versions of the system. Our prototype did this by checking and re-starting nodes from a central location, leading to a lot of complexity and failure cases. A better implementation would be to have a stub on each node which contacts one of many central databases, downloads the required code to the local disk, and runs it from there. Furthermore, it would have been better if each program could restart the programs it required. The implementation as done depended on the central server to re-start nodes. It also would have been better to use multicast to locate the leaf level servers. This choice would have eliminated the need for the list of leaf nodes. The merging and aggregation nodes would still have to be configured, but this would be relatively easy given the wide hierarchy we expect to see.

3.5 Related work

The most closely related work is TkIned [SL93, Sch97]. TkIned is a centralized system for managing networks. It has an extensive collection of methods for gathering data. Because it is distributed with complete source code, it can be extended by modifying the program. Since the data is not accessible outside of the TkIned program, new modules either have to be added to TkIned, or have to repeat the data gathering. TkIned provides simple support for visualization and does not aggregate data before displaying it. TkIned's centralized pull model limits its scalability.

Pulsar [Fin97] uses short scripts (pulse monitors) which measure a statistic, and send an update to a central display server if the value is out of some hardcoded bounds. Pulse monitors run infrequently out of a cron-like tool. Pulsar can be extended by writing additional pulse monitors, and adding them to a configuration file. Pulsar's centralized design is not fault tolerant, and only simple support for external access to updates. Pulsar does not support monitoring of rapidly changing statistics.

SunNet Manager [SNM] and HP Openview [Paca] are commercially supported, SNMP-based network monitoring programs. Other companies can extend them by writing drop-in modules to manage their equipment. SunNet Manager can use SNMP version 2 [CMRW96], or Sun proprietary protocols to support communication between multiple monitoring stations. As with other monolithic systems, SunNet Manager and HP Openview have poor scalability and weak extensibility.

The DEVise [LRM96, Liv97] system is a generic trace file visualization tool. DEVise supports converting a sequence of records (rows in a table) into a sequence of graphical object, displaying the graphical objects, and performing graphical queries on the objects. DEVise uses SQL queries to implement the graphical queries, and supports visualizing trace files larger than the physical memory on a machine. Unfortunately, it does not support online updates to visualized data, and so does not directly match our needs, but we are using a similar idea of translating database updates into graphical objects.

Multi Router Traffic Grapher (MRTG) [Oet98] supports fetching data from a variety of sources, and aggregating together the time series data by averaging. This part of the approach is similar to our aggregation, except that MRTG does not retain any of the dispersion statistics.

SGI's performance co-pilot [SGI] gathers data similar to how we do, and puts it into a centralized proprietary (but accessible) format. It then provides various tools to perform 3-d visualization of performance data, and to replay historical information. It does not use a standard format like relational tables, nor does it perform generalized aggregation.

A variety of programs follow the same structure as buzzerd [HM92]. They have a centralized monitoring station with some list of thresholds for values. When a metric exceeds a threshold, the system will page a system administrator. As with all centralized systems, this has scalability problems, and does not help the administrator with seeing the state of the system.

3.6 Conclusion

Decoupling data visualization and data gathering through the relational database has greatly improved the flexibility and structure of our system. It led to our success in using relational tables for flexible data storage and access. It also led to the idea of using a hierarchy and a hybrid protocol for efficient data transfer. Timestamps have been very useful in detecting internal system failures and automatically recovering from them. Since the machines are used on a research project [ACPtNt95] exploring how to use

hundreds of machines in cooperation to solve complex problems, aggregation in visualization was required by the scale and class of the system we wanted to monitor. We expect in the future to monitor more of the software and hardware in our cluster, including more research systems.

We can see how the scalability (principle #3 from Chapter 1) of the system derived from the use of the hierarchy, hybrid push-pull protocol, and the visualization approaches. The system also achieved partial flexibility (principle #4). However, the complexity of the monitoring system made it less dependable (principle #1). The use of relational tables made it easy to add in additional data, but some of the premature optimizations to try to achieve sufficient efficiency made the flexibility more difficult to use. The numerous programs involved in the system (database, 2-4 gather processes, joinpush, forwarder, javaserver, java applet) dramatically increased the complexity of the system (conflicting with principle #8), and the automation (principle #2) applied to getting that combination working was insufficient. For these reasons, we suggest that future work take the ideas presented in this paper, and ideas from the proposed re-implementation, but not directly follow the implementation as it was done.

CARD illustrates the research approach of "Let the administrator handle it." This approach has the advantages that some sort of monitoring is necessary for debugging the really hard problems, and there will always be hard problems. This advantage tells us that all systems need to have monitoring ability, and also tools to make changes. However, this approach to research does not reduce administrator's workload. It may make their lives better because they achieve faster problem resolution, but it may make their lives worse because more problems will be apparent. As a result, some other approaches are necessary. The bottom line to this approach is that it is required, but it is not sufficient.

Chapter 4

River: infrastructure for adaptable computation

The work that was done on monitoring was valuable because it looked at the case where we had legacy applications that were not designed for easy administration. Monitoring allows us to track what is going on in the system. If a human is watching the monitoring output, or is notified of a problem, then the administrator can go and fix the problem. However, the disadvantage of this approach is that it is very human intensive, and is hence not a complete solution to the problem.

We then took a step back and asked the question "If we could re-engineer systems to be easier to maintain, how would we do this?" A partial answer to that question is the River system. We chose an important class of applications, namely cluster-based, data-intensive applications, and a particular type of problem that affects them, namely performance perturbation. We then set out to build a new version of the primitives in database systems which were robust to perturbations of various components that were being used. The net effect was that for transient performance problems on hosts or disks, River will automatically adapt to minimize the degradation as opposed to traditional parallel systems which degrade substantially if any node is slightly perturbed. This approach to system administration research we refer to as "Rewrite everything."

The work was jointly done with Remzi Arpaci-Dusseau and Noah Treuhaft, and has been published both in [ADAT+99] and then in Remzi's dissertation [AD99]. This chapter is based on [ADAT+99] which was written with Remzi Arpaci-Dusseau, Noah Treuhaft, David Culler, Joseph Hellerstein, Dave Patterson, and Katherine Yelick. Unlike the earlier work, we focus on how the ideas in River could be applied to system administration, rather than the earlier focus on parallel computing. We repeat enough of the explanation so that readers can see how they worked for the database primitives we examined, and leave the full explanation for the other papers.

This work was primarily focused on the principle of dependability (#1 from Ch. 1), as it intended to get consistent, reasonable performance despite changes in the underlying system. It was secondarily focused on the principle of scalability (#3), as it was done in the context of a large cluster of computers.

We will show River achieved the dependability goal while it was running, but had some problems during startup, and did not handle the problem of partial failures. We will also show that River achieved the goal of scalability. We propose some approaches to fixing the startup issues and partial failures.

4.1 Introduction

Cluster I/O systems exhibit *performance heterogeneity*, which often causes the common-case performance to be much worse than the peak performance. Performance heterogeneity comes from both hardware and software differences. Machines in a cluster do not have to be identical, and even if they are, the inner cylinders of a disk have much less bandwidth than the outer[Met97], and two apparently identical disks can have different bandwidths depending on the locations of unused "bad" disk blocks. Software problems also cause performance heterogeneity because of unexpected operating system activity, uneven load placement, or a heterogeneous mixture of operations across machines. In practice, peak numbers from cluster measurements are done after rebooting the entire cluster to "clean" it, and guaranteeing that no other programs are sharing the cluster. Performance heterogeneity often requires administrators to go and re-tune the system. Moreover, as administrators add resources, they are likely to increase the heterogeneity of the cluster.

Since eliminating performance heterogeneity is nearly impossible, we instead chose to design the *River* I/O system to adapt around performance heterogeneity. River uses data-flow programming and achieves common-case near-maximal performance to I/O-intensive applications. River uses two basic system mechanisms: a *distributed queue* (DQ) balances work across consumers of the system, and *graduated declustering* (GD) adjusts the rate of producers so that all parts complete at the same time. DQ's work similarly to load balancing routers, with the addition of back-pressure to keep from overrunning consumers. GD is a generalization of Chained Declustering [HD90]; the data is mirrored across multiple disks, and the production rate is varied so that all of the sources approach completion at the same time. These two techniques limit the cases when the administrator has to re-tune the system.

To provide enough flexibility in the system to adapt around performance bottlenecks, River uses a data-flow style of programming. Data flow programming is a natural match for many applications such as implementing database query plans [Gra90] and scientific data-flow systems [KRM98, SCN+93].

The River system has been designed for data warehousing applications where a large amount of data flows through the system. Although techniques like partitioning can be used to handle small update workloads (such as TPC-C), we have not experimented with these approaches.

We demonstrate River with a number of data-intensive applications, and use them to validate the performance of the system. In all cases, River provides near-ideal performance in the face of severe performance perturbations. We then describe how to apply these principles to problems in system administration.

When a "traditionally" designed cluster application is perturbed, it dramatically reduces it's performance. We measured the effect of perturbation on the NOW-Sort [ADADC+97] application. If a single file on a single machine has poor layout (inner tracks versus outer), overall performance drops by a factor of 1.5. When a single disk is a "hot spot", and has a competing data stream, performance drops by a factor of 3. CPU loads on any of the machines decrease performance proportional to the amount of CPU they steal. Finally, when the memory load pushes a machine to page to disk, a factor of 5 in performance is lost.

The rest of this chapter is structured as follows: Section 4.2 describes the design of the system and its current implementation, *Euphrates*. Section 4.3 validates the performance properties of our dynamic I/O infrastructure, with measurements of both distributed queues and graduated declustering. Related work is found in Section 4.4. Section 4.5 analyzes how River applies to system administration. Finally Section 4.7 summarizes the Chapter.

4.2 The River system

This section describes the design of the River environment, as well as the current implementation, *Euphrates*. We present the River data model: how data is stored and accessed on disk. We explain the components of the River programming model. Finally, we examine how a typical River program is constructed.

4.2.1 The data model

Data in River is a typed collection of records. Record types are stored as named fields of a given base type. Data is therefore analogous to a single table in a database. We store data as records because records are meaningful to applications whereas an application will have to merge together bytes in a byte stream to provide application level meaning.

Single disk collections

Data can be accessed on disk as an *unordered* or *ordered* collection. Unordered collections allow the system to optimize the data accesses. Ordered collections arrive as a stream, and the read order for a single stream from disk will be the same as the write order at that disk.

The Euphrates implementation uses the underlying Solaris 2.6 UNIX file system (UFS) to implement record collections. To read from disk, we use either `read()` with `directio()` enabled (an

unbuffered read from disk), or the `mmap()` interface, both of which deliver data at the raw disk rate for sequential read access. Using `directio()` eliminates double-buffering in the file system. Writes to disk use the `write()` system call, with or without `directio()` enabled. Because we use UFS, we do not have location information to schedule un-ordered I/Os more efficiently. A disk manager running on a raw disk would enable optimizations when collections are accessed in the unordered mode.

Parallel collections

We build ordered or unordered parallel collections by merging together streams from several single-disk collections. We store the meta-data on which disks are used in NFS and serialize access to the meta-data through the process which starts the parallel application. Naturally, improved performance is only available if multiple disks can be accessed at the same time.

Redundancy

We use mirroring to improve the consistency of application performance. Earlier work on chained declustering [HD90] showed that in a system where mirrors are interlaced, during a partial failure, a read-only load can be balanced evenly across the remaining, working disks. This balance is achieved through a carefully-calculated distribution of read requests to the mirror segments on the working disks.

We generalize this technique for better performance consistency by creating *graduated declustering*. In the common case, all disks storing a mirrored collection are functional, but each may offer a different bandwidth (for reasons enumerated earlier) to any individual reader. Under traditional approaches to mirroring, these variations are unavoidable because a reader will choose one mirrored segment copy from which to read the entire segment. Such variations can lead to a global slowdown in parallel programs, as slow clients complete later than fast ones.

To remedy this weakness, we approach the problem somewhat differently. Instead of picking a single disk to read a partition from, a client will fetch data from all available data mirrors, as illustrated in Figure 4.1. Thus, in the case where data is replicated on two disks, disk 0 and disk 1, the client will alternatively send a request for block 0 to disk 0, then block 1 to disk 1; as each disk responds, another request will be sent to it, for the next desired block.

However, this alone does not solve the problem, as we want all of the reads in a parallel system to complete at the same time. Graduated declustering must adjust the bandwidth so that each reader finishes at close to the same time. Clients that receive less than the expected bandwidth from one of the two disk mirrors must receive more bandwidth from the other mirror as compensation. Thus, the implementation of graduated declustering must somehow observe these bandwidth differences across clients and adjust its bandwidth allocation appropriately.

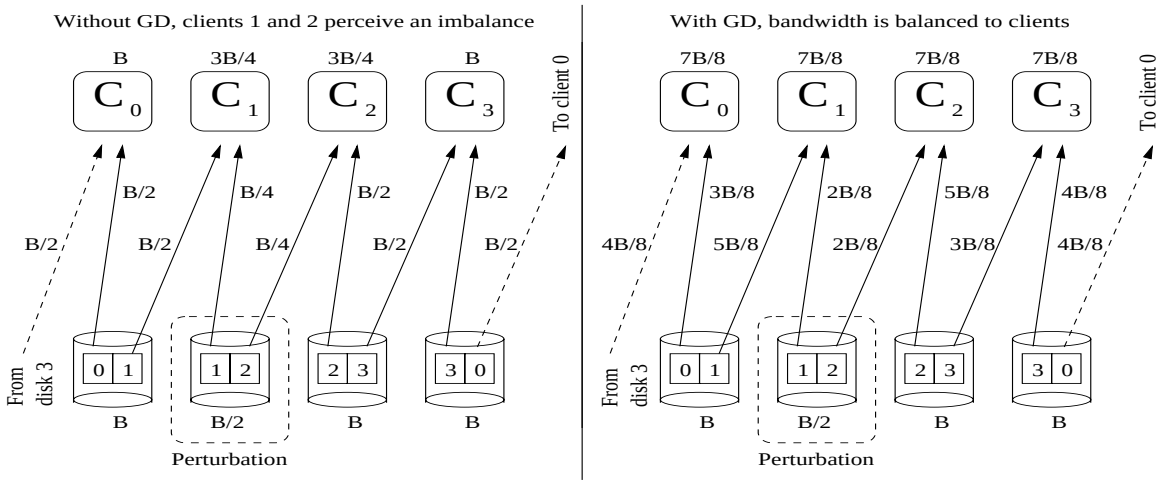


Figure 4.1: **Graduated declustering.** *These two diagrams depict two scenarios, without and with graduated declustering under a perturbation. Unperturbed disks normally deliver B MB/s of bandwidth, and the one perturbed disk delivers half of that, B/2. On the left, the disk serving partitions 1 and 2 to clients is perturbed, and thus only half of its bandwidth is available to the application. Left unchecked, the result is that clients 1 and 2 do not receive as much bandwidth as clients 0 and 3. On the right, the bandwidths from each disk have been adjusted to compensate for the perturbation, as is the case with graduated declustering. With the adjustments, each client receives an equal share of the available bandwidth.*

The Euphrates implementation of GD uses a simple algorithm to balance load amongst data sources. Each disk manages two different segments of a parallel collection, and continually receives feedback from two consumers as to the total bandwidth that the consumers are receiving. When a performance inequity between two clients is detected, the disk manager biases requests towards the lagging client, and thus attempts to balance the rates at which the readers progress. An example of the result of such a balancing is shown in the right-side of Figure 4.1. There, both disks 0 and 2 compensate for a perturbation to disk 1 by allocating 5/8 of their bandwidth to clients 1 and 2. The resulting bandwidths to each client are properly balanced.

4.2.2 The programming model

River provides a generic data flow environment for applications, similar to parallel database environments such as Volcano [Gra90]. Applications are constructed in a component-like fashion into a set of one or more *modules*. Each module has a logical thread of control associated with it, and must have at least one input or output channel, often having one or more of each. A simple example is a filter module, which gets a record from a single input channel, applies a function to the record, and if the function returns *true*, puts the data on a single output channel.

Modules are connected both within a machine and across machine boundaries with queues. A queue connects one or more producers to one or more consumers and provides rate-matching between

modules. By dynamically sending more data to faster consumers, queues are essential for adjusting the work distribution of the system.

To begin execution of an application, a master program constructs a *flow*. A flow connects the desired set of modules, from source(s) to sink(s). Any time a single module is connected to another, a queue must be placed between them. When the flow is instantiated by the master program, the computation begins, and continues until the data has been processed. Upon termination, control is returned to the master program.

River modules

A module is the basic unit of programming in River. Modules operate on records, calling `Get()` to obtain records from one or more input channels, and then calling `Put()` to place them onto one or more output channels. For convenience, we refer to a set of records that is moving through the system as a *message*. Logically, each module is provided a thread of control. More details and examples of the programming model can be found in [ADAT+99].

In Euphrates, modules are written as C++ classes. In the current implementation, each module is given its own thread of control, which has both its benefits and drawbacks. The main advantage of this approach is that applications naturally overlap computation with data movement; thus, the user is freed from the burden of carefully managing I/O. However, thread switches can be costly. To amortize this cost, modules should pass data (a set of records) amongst themselves in relatively large chunks. In our experience, this has not complicated modules in any noticeable fashion; thus, we felt that the inclusion of complex buffer management was not worth the implementation effort.

Queues

Queues connect multiple producers to multiple consumers, both in the local (same machine) and distributed (different machines) cases. During flow construction, queues are placed between modules so that messages can be transmitted from producers to consumers. Modules that are placed on either side of local or distributed queues are oblivious to the type of queue with which they interact.

Messages in River may move arbitrarily through the system, depending on run-time performance characteristics and the constraints of the flow. Dynamic load balancing is achieved by routing messages to faster consumers through queues that have more than one consumer.

To improve performance, ordering may be relaxed across queues. In a multi-producer queue, a consumer may receive an arbitrary interleaving of messages from the producers. The only ordering guarantee provided in a queue is point-to-point; if a producer places message *A* into queue *Q* before message *B*, and if the same consumer receives both messages, it receives *A* before it receives *B*. This ordering is necessary,

for example, to retain the ordering of a disk-resident stream. By attaching a single consumer to the single producer of a stream, the ordered property of the stream can be properly maintained.

In Euphrates, the local queues are simply a in-memory queue protected by a shared mutex. The remote queues operate either using a randomized back-pressure algorithm, or if large, ordered messages are required, by using a consumer based pull method. More details and examples of the queue implementation can be found in [ADAT+99].

Flow construction

To execute a program in the River environment, one or more modules must be connected together to form a *flow*. A flow is a graph from data source(s) to sink(s), with as many intermediate stages as dictated by the given program.

There are three phases involved in instantiating a flow: construction, operation, and tear-down. During construction, a *master program* specifies the global graph, describing where and how data will flow, including which modules to use and their specific interconnection. When the construction phase is complete, the master program instantiates the flow. In the operation phase, threads are created across machines as necessary, and control is passed to each of the modules. The flow of data begins at the data sources, and flows through the system as specified by the graph, until completion.

Flow construction can be performed programmatically or graphically. Flows are constructed as graphs using *create node* and *attach* operations. The node create operation takes arguments to specify how many, or precisely which physical nodes should be used for the logical flow node. The attach operation connects logical nodes together with the appropriate physical queues, and has options to control how the multiple physical nodes will be connected. After the flow is constructed, a go routine is called which instantiates all of the nodes and executes the parallel program. This routine remotely starts all of the modules on the specified nodes, provides them with appropriate initialization arguments, and connects them together locally or remotely to the other modules.

In the Euphrates implementation, numerous languages can be used to program flows. A C++ interface is available, but we have found it overly cumbersome to re-compile codes for each simple change to a flow. Therefore, we provide both Tcl and Perl interfaces, allowing for the rapid assembly of flows in a scripting language.

4.3 Experimental validation

In this section, we perform experiments to validate the expected performance properties of the system. First, we explore the absolute performance and adaptability of the distributed queue. We will

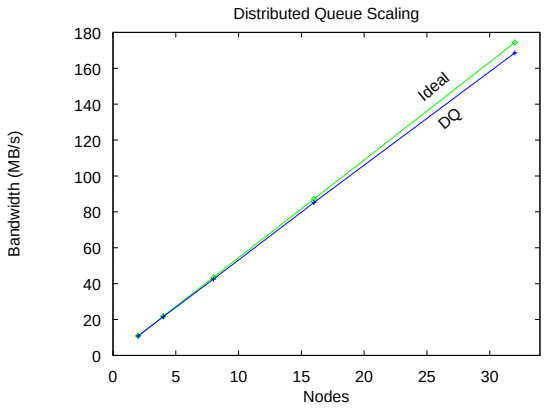


Figure 4.2: **Distributed queue scaling.** *In this experiment, the scalability of the DQ is under scrutiny. During the run, from 1 to 32 producers read data blocks from disk and put them into the distributed queue, and 1 to 32 sources pull data from the DQ. The ideal line shows the aggregate bandwidth that is available from disk. Cluster parameters are described in section 4.3.1*

see that the distributed queue is effective in balancing load across consumers, and moving more data to faster consumers. We then perform experiments on graduated declustering, examining the performance adaptability of disk sources. We will see that graduated declustering transparently provides performance robustness to disk slowdowns.

4.3.1 Hardware and software environment

The River prototype, Euphrates, ran on a cluster of Ultra1 workstations running Solaris 2.6 [KVE+92] and connected together by the Myrinet local-area network[BCF+95]. Each workstation had a 167 MHz UltraSPARC I processor, two Seagate Hawk 5400 RPM disks (one used for the OS and swap space in the common case), and 128 MB of memory.

All communication is performed with Active Messages (AM)[MC96]. AM exposes most of the raw performance of Myrinet while providing support for threads, blocking on communication events, and multiple independent endpoints. Other fast message layers [PLC95, vEBBV95, vECGS92] require polling the network interface, which mostly defeats the single-node sharing desirable for an I/O infrastructure such as River.

4.3.2 Distributed queue performance

Absolute performance

First, we explore the scaling behavior of the distributed queue. In the first experiment, data is read from *n* disks, put into a distributed queue, and consumed by *n* CPU sinks. We scale *n* from 1 to 32. Figure 4.2 shows that the scaling properties are near ideal. Reading from 32 disks, Euphrates achieves 97

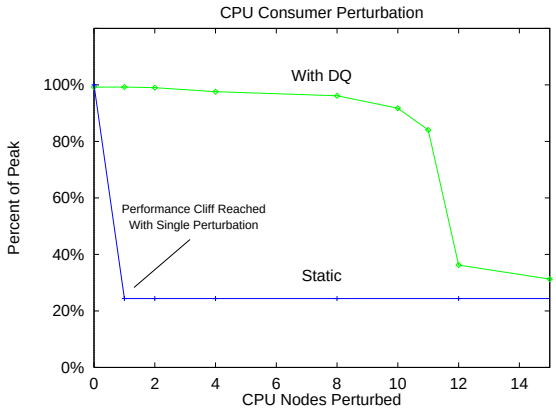


Figure 4.3: **DQ read performance under perturbation.** *This figure shows the percent of peak performance achieved as consumer perturbations are added into the system. Without a DQ to balance load across unperturbed consumers, performance drops as soon as a single consumer is slowed. With a DQ, performance is unaffected until a large number of nodes are perturbed. A CPU perturbation steals 75% of the processor; the test consists of 15 producers and 15 separate consumers.*

percent of the peak performance. The performance writing to disks through a DQ (not shown) scales equally well.

Performance under perturbation

We next examine the results when one or more consumers is arbitrarily slower than the rest. This type of perturbation could arise from dynamic load imbalance, hot spots in the system, or CPUs or disks with different performance capabilities.

Figure 4.3 shows the effect of slowing down 1 to 15 CPU consumers both with and without a DQ, when reading from 1 to 15 disks. In the static case, without a DQ, work is pre-allocated across consumers; thus, if a single consumer slows down, the performance is as bad as if all consumers had slowed down.

When a DQ is inserted between the producers (disks) and consumers, more data flows to unperturbed consumers, thus flowing around the hot spots in the system. Euphrates avoids a substantial slowdown until 10 nodes (60%) are perturbed, because the CPU's were under-utilized in this test.

Figure 4.4 shows the effects of slowing down 1 to 15 disks when performing writes. In this experiment, we place a DQ between CPU sources (which generate records) and the disks in the system. Once again, the static allocation behaves quite poorly under slight perturbation. With the DQ, performance degrades immediately because the disks are fully utilized, but it degrades gracefully as Euphrates adapts to the slower disks. Unfortunately, this adaptation comes at a slight cost. When all disks are perturbed, the DQ is slightly slower because of the randomized distribution of data.

We have now demonstrated that the distributed queue has the desired properties of balancing load among data *consumers*; however, without mirroring, each *producer* of data has a unique collection of

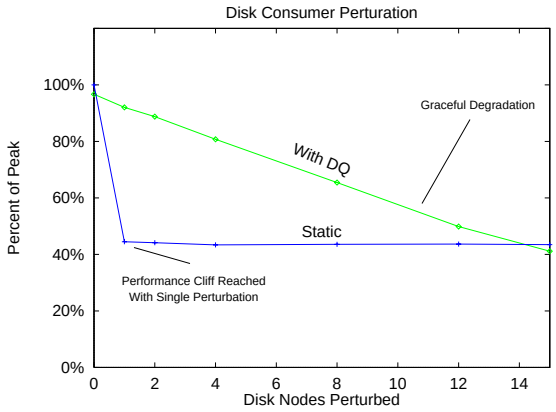


Figure 4.4: **DQ write performance under perturbation.** *This figure shows the effect of disk perturbation during writes, and how the DQ dynamically adapts. The system under test consists of 15 disks. Instead of falling off the performance cliff, the DQ routes data to where bandwidth is available, and thus gracefully degrades. In this case, each perturber continually performs sequential, large-block, writes to the local disk, stealing roughly half of the available bandwidth.*

records, and to complete a flow, must deliver that data to the consumers. Thus, when the producers are the bottleneck in the system (as is often the case when streaming through large data sets), slowing a single producer will lead to a large global slowdown, as the program will not complete until the slow producer has finished. This “producer” problem is the exact problem that graduated declustering attempts to solve.

4.3.3 Graduated declustering

We now describe our experimental validation of the graduated declustering implementation. We find that both the absolute performance and behavior under perturbations is as expected.

Absolute performance

The performance of graduated declustering under reads, with no disk perturbation, is slightly worse than the non-mirrored case. This effect is a direct result of our design, which always fetches data from both mirrors instead of selecting a single one, in order to be ready to adapt when performance characteristics change. Multiplexing two streams onto a single disk has a slight cost, because a seek must occur between streams. Increasing the disk request size to 512KB or 1MB amortizes most of the cost of the seek, and thus Figure 4.5 shows that we achieve 93 percent of the peak non-mirrored bandwidth. Writes, each of which must go to two disks, halve performance as is expected for mirrored systems, so temporary streams are best put onto un-mirrored disks.

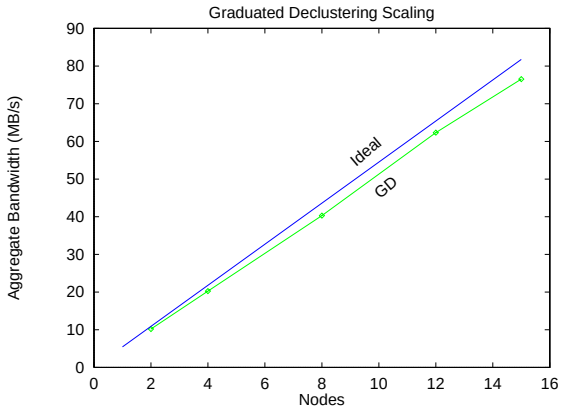


Figure 4.5: **Graduated declustering scaling.** *The graphs shows the performance of GD under scaling. The only performance loss is due to the fact that GD reads actively from both mirrors for a given segment; thus, a seek cost is incurred, and roughly 93% of peak performance is delivered.*

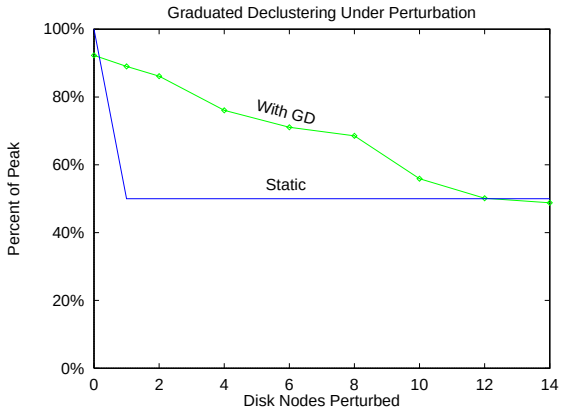


Figure 4.6: **GD performance under read perturbation.** *The graphs shows the performance of GD under read perturbation. Performance degrades slowly for the GD case, whereas a typical non-adaptive mirrored system suffers immediate slowdown. Each perturber is a competing read-stream to disk.*

Performance under perturbation

The real strengths of GD come forth for read-intensive workloads, such as decision support or data mining. In these cases, applications reading from a non-adaptive mirroring system would slow to the rate of the slowest disk of the system. With GD, the system shifts the bandwidth allocation per disk, and thus each consumer of the data receives data at the same rate.

Figure 4.6 shows the results of a 28-machine experiment using half the machines as disk nodes and the other half as data consumers. As explained above, the performance of GD as compared to no mirroring is slightly worse in the unperturbed case. However, a single perturbation slows the application on the non-GD system to the bandwidth of the slow disk, which in this case delivers data at roughly half of peak rate due to a single competing stream. With GD, performance degrades slowly, spreading available bandwidth evenly

across consumers. When all disks are equally perturbed, however, the performance of GD once again dips below the non-GD system, again due to the overhead of seeking between multiple streams.

4.3.4 Supporting a trace-driven simulator

A trace driven simulator can require fast access to a long sequence of data. We allowed for data to be copied into the River system and then copied out directly into the simulator. This improved the performance of the simulator over accessing the files via NFS. This usage is identical to the disk read/write cases that we showed above, and so we show no graphs for it. Most of the performance improvement came because River runs over the fast AM2 implementation on Myrinet, whereas NFS runs over traditional TCP over Ethernet.

4.3.5 One-pass hash join

The hash join stressed the partitioning aspects of our system, and demonstrated that application flexibility is required to gain performance robustness. In particular, the preliminary results showed that we could not adapt to the static partitioning present in the algorithm. More details on our implementation are present in [ADAT+99]. Follow on work [AH00] demonstrated that performance robustness for hash join could be achieved by modifications to the join algorithm.

4.3.6 One-pass external sort

External sorting stressed the ordering properties of our system. We added the client-based pull to the distributed queue so that the large, sorted blocks are distributed evenly. This database kernel showed that some of the benefits of River can be trivially achieved, and others require modifying the statically parallel structure of previous algorithms.

We provide a few of the graphs to demonstrate the behavior of the external sort, and refer the reader to [ADAT+99] to see the full collection of graphs and detailed discussion of the behaviors.

Figure 4.7 shows that the Euphrates sorting algorithm is reasonable, we compare it to an idealized static sorting algorithm with perfect parallelism and no cost for the in-memory sort. Euphrates achieves 90% of the ideal performance primarily because the in-memory sort has not been tuned.

Figure 4.8 shows that the Euphrates sorting algorithm performs much better than the idealized sort would under perturbation of the partitioning part of the sort. In particular, we see the same graceful degradation that we saw when we perturbed readers in the distributed queue experiments.

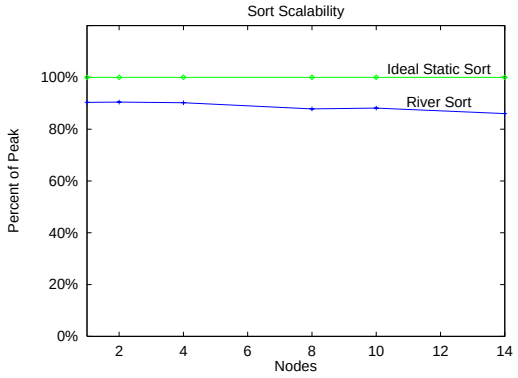


Figure 4.7: **Parallel external sort scaling.** *This figure shows the scaling behavior of the sort built in the River framework, as compared to an idealized statically-partitioned sort. The River sort scales well; its only deficiency is an under-tuned in-memory sort, resulting in the slightly less than perfect scaling.*

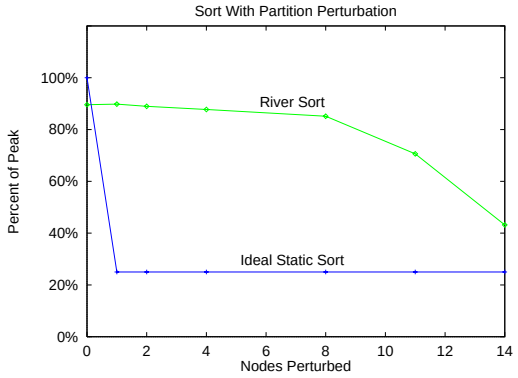


Figure 4.8: **Perturbing the sort partitioner.** *This figure shows the sort when the partition modules are perturbed. The disk and sort modules run on one set of 14 machines, and the partition modules run on another set. The River sort is compared to a "perfect" sort that is statically partitioned. Each perturbation steals 75 percent of the CPU.*

4.4 Related work

River relates to work from a number of often distinct areas: file systems, programming environments, and database research. In this section, we discuss work from the three areas.

4.4.1 Parallel file systems

High-performance parallel file systems are abundant in the literature: PPFS [HER+95], Galley [NK96], Vesta [CF96], Swift [CL91], CFS [Nit92], SFS [LIN+93], and the SIO specification [BBD+94]. However, most assume performance-homogeneous devices; thus, performance is dictated by the slowest component in the system, and few of them provide a programming environment, leading to simple static partitioning approaches in the programs.

More advanced parallel file systems have specified higher-level interfaces to data via collective I/O (a similar concept is expressed with two-phase I/O) [Kot94, CBH+94]. In the original paper, Kotz found that many scientific codes show tremendous improvement by aggregating I/O requests and then shipping them to the underlying I/O system; the I/O nodes can then schedule the requests, and often noticeably increase delivered bandwidth. Because requests are made by and returned to specific consumers, however, load is not balanced across those consumers dynamically. Thus, though these types of systems provide more flexibility in the interface, they do not solve the problems we believe are common in today's clustered systems.

Finally, there has been recent file-system work extolling the virtue of "adaptive" systems [MRC+97, SS97]. As hardware systems increase in complexity, it can be argued that more intelligent software systems are necessary to extract performance from the underlying machine architecture. Whereas some of these systems employ off-line reorganization to improve global performance [MRC+97], the goal of River is balance load on-line (at run-time). However, long-term adaptation could also be useful in River.

4.4.2 Programming environments

There are a number of popular parallel programming environments that support the single program, multiple data (SPMD) programming style, including messaging passing environments such as Message Passing Interface [The93] and Parallel Virtual Machine [GS93], as well as explicit parallel languages, such as Split-C [CDG+93]. These packages all provide a simple model of parallelism to the user, thus allowing the ready construction of parallel applications. However, none provide any facility to avoid run-time perturbations or adapt to hardware devices of differing rates. Our own experience in writing a parallel, external sort in Split-C led us to realize some of the problems with the SPMD approach; while it was possible to run the sort well *once* – NOW-Sort broke the world record on two database-industry standard sorting benchmarks – it was difficult to attain a high-level of performance consistently [ADADC+97, ADADC+98].

There have been many parallel programming environments that are aligned with our River design philosophy of run-time adaptivity. Some examples include Cilk [BJK+95], Lazy Threads [GSC96], and Multipol [CDI+95]. All of these systems balance load across consumers in order to allow for highly-irregular, fine-grained parallel applications. River focuses on I/O systems and thus avoids the problems of slow remote memory performance found for parallel programming applications. Indeed, as we demonstrated, remote I/O performance is equal to local I/O performance.

Perhaps more similar to the River environment is Linda, which provides a shared, globally-addressable, tuple-space to parallel programs [Car87, GCCC85]. Applications can perform atomic actions on tuple-space, inserting tuples, and then querying the space to find records with certain attributes. Because of the generality of this model, high performance in distributed environments is difficult to achieve [BKT92]. Thus, while the distributed aspects of River could be built on top of Linda, they would likely suffer from performance and scaling problems.

4.4.3 Databases

Perhaps most relevant to River is the large body of work on parallel databases. Data flow techniques are well-known in the database literature [DG92], as it stems quite naturally from the relational model [Cod70]. Indeed, two of the applications we implemented were database kernels.

One example of a system that takes advantage of unordered processing of records is the IBM DB2 for SMPs [Lin98]. In this system, shared data pools are accessed by multiple threads, with faster threads acquiring more work. This structure is referred to as "the straw model", because each thread "slurps" on its data straw at a (potentially) different rate. Implementing such a system is quite natural on an SMP; a simple lock-protected queue will suffice, modulo performance concerns. With River, we argue that this same type of data distribution can be performed on a cluster, due to the bandwidth of the interconnect. The straw model also only handles the consumer side of the problem, and ignores the problems addressed by graduated declustering.

There are a number of parallel databases found in the literature, including Gamma [CABK88], Volcano [Gra90], and Bubba [DGS88]. These systems all use similar techniques to distribute data among processes. Both the Gamma *split table*, Volcano *exchange operators*, and a generalized split table known as a "river" in [BBGS94] are used to move data between producers and consumers in a distributed memory machine. However, all use static data partitioning techniques, such as hash partitioning, range partitioning, or round robin. These functions all do not adapt at run-time to load variations among consumers.

Current commercial systems, such as the NCR TeraData machine, exclusively use hashing to partition work and achieve parallelism. A good hash function has the effect of dividing the work equally among processors, providing consistent performance and achieving good scaling properties. However, as

Jim Gray recently said of the TeraData system, “The performance is bad, but it never gets worse” [Gra97]. Consistency and scalability were the goals of the system, perhaps at the cost of getting the best use of the underlying hardware.

4.5 Applying river to system administration problems

The obvious, immediate application of River to system administration is using it for database applications. If databases used ideas as demonstrated in River, then the system administration problems of performance anomalies would be greatly reduced. Administrators would still have to perform long-term analysis to see whether the load of the system was changing substantially, and they would still have the other tasks associated with databases, but one class of problems would have been eliminated.

Netnews, which uses a flood-fill algorithm for distributing news articles [KL86], already adapts around a particular server being slow in the same way that our distributed queue algorithm adapts around a slow node. Adapting netnews to using the pipelining approach used in River rather than the single article at a time approach used in netnews could further improve the netnews performance.

Web-server load balancing also attempts to adapt around slow nodes in a number of ways. Prior to the existence of load-balancing routers, Netscape used a cluster of heterogeneous machines to serve web pages and performed asymmetric load balancing in the clients [MFM95]. In fact load-balancing routers [Cis00] and the follow-on improvements [PAB+98] distribute web accesses across a set of servers, potentially incorporating server load information. Web server load balancing could be extended to include the DQ idea of getting all the related parts of a query to complete at the same time for the more complex web pages that are now being designed. Similarly to how completing part of a scan is not very useful, delivering part of a web page is not very useful.

In general, the idea of adapting around performance anomalies in the ways demonstrated by the distributed queue and graduated declustering ideas can be applied to most of the problems in system administration. The distributed queue ideas apply when the data is logically read-only, and the cost of transferring data over the network is lower than the cost of locally processing it. The graduated declustering ideas apply when there are multiple sources for data, and the data can be divided into a number of small chunks.

4.6 System administration problems in Euphrates

The Euphrates implementation introduces a few additional system administration problems. In particular, it has problems with startup, and it does not deal well with failures. The problems with startup stem from having a single node which sets up the entire flow. This node can be a bottleneck for setting up a flow across many nodes. A better implementation would have been to have the remote startup happen in a

duplicated tree structure to reduce the sequential, variable delay of starting a single node at a time. For our experiments, the startup delay of a few seconds was not important as the flow would then execute for many minutes, and we ignored the runs where startup failed. For flows which are short lived, however, this startup delay will dramatically reduce the overall performance. Moreover, if the same startup node is used for all flows, then that node could become even more of a bottleneck.

River also did not attempt to handle complete failures, just slowdowns. There are a number of places where failures can be a problem. The first place is during startup the central node constructs the flow and reads all of the summary information for the layout of data across the disks. If the central node fails during construction, it is probably best to just re-build the flow from another node. To make sure that the layout summary is available, the summary should be replicated across multiple nodes and disks.

Handling disk failures for graduated declustering is relatively simple. The node receiving from both of the disk nodes can time-out after a node has failed, and then re-send the requests pending at the failed node to the working one. Once the requests are re-sent, River will automatically adapt around the failed node.

Handling processing node failures is more difficult. A number of approaches are possible. First, all of the operations could be duplicated across pairs of nodes. Then if a node fails, there is another one performing exactly the same calculation which will be able to take over. This solution requires the nodes in the graph to be deterministic, and for the messages to be delivered to both of the nodes in the same order. The former problem is application specific, and the latter problem has been addressed in a number of systems [BC91, vRHB94]. Unfortunately, this approach leads to a factor of two slowdown over the non-replicated implementation.

A second approach to handling processing node failures is to duplicate the inputs to nodes. If the data is saved on a separate node, then it can be re-played at a new node started to replace the first node. This approach again assumes that the nodes are deterministic, and it has the problem of wasting space. That space can be reclaimed if nodes can detect when they have passed the data on to another node. For example, for a filter node, once the data has passed through the filter and on to the next node, the pre-filtered data does not need to be retained. The space wastage can be further reduced by selectively re-computing data as necessary. For example, if data is going from disk, through a filter and then into a sort module, there is no need to store any of the data as it can be simply re-constructed from disk.

4.7 Conclusions

As hardware and software systems spiral in size and complexity, systems that are designed for controlled environments will experience serious performance defects in real-world settings. This problem has long been realized in the area of wide-area networking, where the end-to-end argument [SRC84] per-

vades the design methodology of protocol stacks such as TCP/IP. In such systems, it is clear that a globally-controlled, well-behaved environment is not attainable. Therefore, applications in the system treat it as a *black box*, adjusting their behavior dynamically based on feedback from the system to achieve the best possible performance under the current circumstances.

Complexity has slowly grown beyond the point of manageability in smaller distributed systems as well. Comprised of largely autonomous, complicated, individual components, clusters exhibit many of the same properties (and hence, the same problems) of larger scale, wide-area systems. This problem is further exacerbated as clusters move towards serving as a general-purpose computational infrastructure for large organizations. As resources are pooled into a shared computing machine, with hundreds if not thousands of jobs and users present in the system, it is clearly difficult, if not impossible, to believe that the system will behave in an orderly fashion.

To address this increase in complexity and the corresponding decrease in predictability, we introduce River, a substrate for building I/O-intensive cluster applications. River is a confluence of a programming environment and an I/O system; by extending the notion of adaptivity and flexibility from the lowest levels of the system up into the application, River programs can reliably deliver high performance. Even when system resources are over-committed, performance of applications written in this style will degrade gracefully, avoiding sudden (and often frustrating) extensions of the expected run time.

From our initial study of applications, we found that avoiding perturbations among consumers is relatively straight-forward via distributed queues. One important issue in balancing load is the *granularity* of ordering required by the applications. The most fine-grained applications (those that can balance load on the level of the individual records) are the simplest to construct in a performance-robust manner. While distributed queues have proven excellent as load balancers, they do require the programmer to insert them where appropriate in the flow.

Avoiding perturbations at the producers is the other problem solved by River, with graduated declustering. By dynamically shifting load away from perturbed producers, the system delivers the proper proportion of available bandwidth to each client of the application.

River achieved its goals, and so systems built on it would have more robust performance, and hence would be more dependable, the #1 principle of system administration. Furthermore, River also demonstrated scalability, the #3 principle of system administration. River lost some of the simplicity (principle #8) of static partitioning, but that tradeoff is fine. River had no effect on, or was irrelevant to the other principles.

The dependability that River achieved was not propagated throughout the entire system. There was variability, and occasional failures introduced by the single node starting the entire system, and River did not attempt to handle multiple simultaneous applications; they would have treated each other as perturbers rather than cooperating to each do better (for example by sharing a read stream). Furthermore, the use of

NFS to store meta-data about the location of data in the system introduced another single point of failure into the system reducing it's dependability. It would have been better to have the NFS information be a cache which could be safely reconstructed from the individual nodes and cross checked during a run.

Again, as with the work on monitoring, we see that actually deploying and using the system is critical to determining how well it achieves each of the principles. We deployed the system and had external people building on top of it as part of a class, which helped us to learn how well River matched with the principles described in Chapter 1. If River had been simply used a few times to take the measurements, we would not have learned as much as we did.

River illustrates the research approach of "Rewrite everything." This approach has the advantage of minimizing dependencies on earlier systems. Therefore, researchers can try out completely different ideas, and experiment with substantial restructuring of systems. Unfortunately, the burden of rewriting everything makes it hard to get a complete system. As a result, this approach often forces simplified examples, such as the database primitives we evaluated. In addition, deployment is much more difficult because the system is unlikely to be complete. Therefore the principles are much more valuable for evaluating systems produced with this type of research. The bottom line to this approach is that it is flexible, but hard to validate.

Chapter 5

Hippodrome: running circles around storage administration

Rivers showed that by re-writing applications in a new form allowed us to design applications that adapt to performance anomalies. However, the Rivers approach only handles short-term adaptation, requires modifications to applications, and is unable to determine the amount of resources that should be used for a given application.

We designed Hippodrome to address these problems. Hippodrome is an iterative loop for automatically determining the amount of storage-system resources necessary to support a given application. It does this without modification to applications. By running Hippodrome on a regular basis, we believe that it could handle long-term adaptations to workload changes. This approach to system administration research we refer to as "Sneak in-between."

This work was primarily focused on the principle of automation (#2 from Ch. 1). Hippodrome successfully automates problems that were previously done by hand. Hippodrome has achieved limited scalability so far, as it has only been tested on a single disk array. The design component has been tested on multiple arrays, so there is good reason to believe it would work with multiple arrays.

This chapter is based on work and an early draft written jointly with Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, and Alistair Veitch at Hewlett Packard Laboratories. A substantially revised version of the paper was published as [AHK+02]

5.1 Introduction

Enterprise-scale storage systems containing hundreds of disk arrays are extremely difficult to manage. The scale of these systems, the thousands of design choices, and the lack of information about work-

load behaviors raise numerous management challenges. Users' demand for larger data capacities, more predictable performance, and faster deployment of new applications exacerbate the management problems. Worse, administrators skilled in designing, implementing, and managing these storage systems are expensive and rare.

In this paper, we concentrate on the particularly important problem of *initial system configuration*: designing and implementing the storage system that is needed to efficiently support the application(s) of a particular workload. Initial system configuration refers to the process that must occur before the storage system can be put into production use. It possesses several key challenges:

- **System design**: Generating a good system design is difficult, due to the thousands of device settings and a lack of workload information. Administrators face an overwhelming number of design decisions: which storage devices to use, how to choose the appropriate RAID level and the accompanying device settings, and how to map the data onto the configured devices. The design choices often interact with one another in poorly understood ways, resulting in a very complex design process.

  Initial system configuration is further complicated because administrators often know little about the workloads that will execute on the system being designed. Even in cases where workload information exists – such as when migrating or merging existing applications onto a new system – the workloads may behave unexpectedly when combined or when run on a different system.

- **Design implementation**: Implementing the chosen design is time-consuming, tedious, and error-prone. During this step, administrators must interact with numerous graphical and command-line user interfaces to run hundreds of very specific commands to create logical units[1] (LUs) on the disk arrays, create physical and logical volumes[2] at the hosts, and set multiple inter-related parameters correctly. Unfortunately, a mistake in any of these operations or in the order they are performed is difficult to identify, and can result in a failure of the applications using the storage system.

Traditionally, these storage management tasks have been undertaken by human experts, utilizing "rules of thumb" gained through years of experience. For example, one common approach involves estimating the requirements for bandwidth and the number of I/O operations per second (IOPS) based on intuitive knowledge of the application(s) and measurements taken on a similar, existing system. Budgetary constraints and growth expectations also contribute to the initial system configuration. Administrators select RAID 1 if the workload is I/O intensive, and RAID 5 otherwise. They then map the application data onto

[1] A logical unit is the element of storage exported from a disk array, usually constructed from a subset of the array's disks, configured using a particular RAID layout (e.g., a RAID 5 redundancy group). An LU appears to be a single virtual "disk" to the server accessing it.

[2] A physical volume is the device file that is used to access an LU. Logical volumes provide a level of virtualization that enables the server to split the physical volume into multiple pieces or to stripe data across multiple physical volumes.

these LUs in an ad-hoc manner, for example, by partitioning the storage for different applications and then striping across the individual LUs. After generating this initial system configuration, they may tune the storage system by measuring it and rearranging the data to match, or they may choose to put the system into production, and wait until there are complaints before improving the system.

This ad-hoc process is expensive because it usually involves the administrator trying a variety of designs. Determining a suitable design is hard for a human to handle well, because of the many inter-related parameters. Moreover, implementing the design can be extremely tedious and error-prone because it requires the administrator to execute a large number of intricate steps, in the right order, without making any mistakes. As a result, it takes a long time to set up the storage system. Furthermore, the results are often over-provisioned and hence expensive, or under-provisioned and hence perform poorly.

In this chapter, we describe Hippodrome, a system that automatically solves the problems of the manual, ad-hoc approaches described above. Hippodrome automatically designs and implements a storage system without human intervention. Hippodrome is an iterative loop that analyzes a running workload to determine its requirements, calculates a new storage system design, and migrates the existing system to the new design. By systematically exploring the large space of possible designs, Hippodrome can make better design decisions, resulting in more appropriately provisioned systems. By analyzing a workload's requirements explicitly, Hippodrome's loop converges to a design that supports the workload. Finally, by automating these tasks, Hippodrome decreases the chance of human error and frees administrators to focus on the applications that use the storage system.

The remainder of this paper is organized as follows. Section 5.2 presents the component requirements and loop progression that results in Hippodrome. Section 5.3 describes our experimental setup, methodology and workloads. Section 5.4 presents the results of applying Hippodrome to initial system sizing of synthetic workloads and the PostMark file system benchmark. Section 5.5 discusses related work and Section 5.6 summarizes the results.

5.2 System overview

We first introduce the process of initial system configuration by explaining the current practices used by system administrators. We then show how the administrators' practice can be viewed as an iterative loop. We next describe how their manual ad-hoc approach can be automated. Finally, we present a progression of increasingly sophisticated automatic loops, starting with a simple, automatic version of the ad-hoc loop, and continuing with increasingly sophisticated components to culminate in the Hippodrome loop. We show that each of the intermediate systems have substantial problems that prevent them from solving the problem of initial system sizing.

5.2.1 Today's manual loop

The process that administrators use to determine an initial system configuration can be viewed as an ad-hoc iterative loop. Each stage is performed manually, with some support from commercially available storage products.

First, administrators use the workload's capacity requirements and a guess about its performance requirements to build a trial system. Such performance information may come from previous experience with the application on a different system, or from knowledge of similar applications. They select a RAID level for the data based on these requirements, as well as budgetary constraints. For instance, they may select RAID 1 if the workload is I/O-intensive, and RAID 5 otherwise, to minimize the overall storage capacity required for the data. They use the command-line or graphical user interface of the disk array management tools and a logical volume manager (LVM) to create an initial storage system. The disk array manager is used to create LUs of the appropriate RAID level on the disk array, and the LVM is used to create the corresponding physical volumes and to assign the application *stores*[3] to the physical volumes. Then, databases may use stores to hold their tables and indexes, or filesystems may use them to hold users' data.

The administrators then measure and observe the system using various system- and array-specific monitoring tools to see how it performs using simple metrics such as the number of IOPS and/or total I/O system bandwidth (MB/s). The Veritas Volume Manager [Ver00], for example, supports a command, *vxstat*, to measure I/O activity for the LUs of a server. This Volume Manager's Visual Administrator will display an illustration of the storage, using color to draw the administrators' attention to the high-activity LUs.

Administrators compare the observed performance to their expectations and to the maximum attainable performance documented by device manufacturers. These comparisons often reveal that various parts of the system may be over- or under-utilized. In these cases, they propose a new system design that they hope will provide better balance by shuffling the load between LUs, purchasing additional storage resources, or both.

Administrators then implement the proposed design by configuring newly purchased resources as described above, and by using array tools and the LVM to migrate the stores to the appropriate target LUs. For instance, the HP XP512 [Hew00a] and EMC Symmetrix [EMC00] disk arrays provide assistance for moving data within a single disk array.

The administrators then start the cycle again at the measurement step, and continue until a satisfactory performance level is achieved. The loop is completed when all LUs are below some threshold utilization, and the administrator (and the users) are satisfied with the system's performance. Even for rela-

[3] Each store in our system is implemented as a logical volume. A store is a logically contiguous block of storage. We use stores rather than logical volumes because some storage systems provide other abstractions to virtualize, and our system could use those instead of the logical volume abstraction.

tively well-understood applications, this process can take many weeks of time and effort. It typically takes well over a month for a team of experts to design and build a system for a TPC-H benchmark submission, part of which is spent designing and implementing the storage system. Our own experience in setting up these storage systems indicates that it takes one person about a day to set up a trivial configuration with completely unknown performance properties, and at least a week to generate a configuration which appears to work adequately.

This iterative configuration process can occur only if a pool of storage resources is available to the administrators. Today, this pool of resources is made available in one of two ways. First, administrators purchase storage resources based on their prediction of how many storage devices are necessary for the workload. These predictions often over-provision to compensate for inaccurate predictions, or to build in headroom for future growth. Once the purchase has been made, they will iteratively refine the usage of these resources. Second, administrators for larger systems may take their applications to a system vendor's capacity planning center (CPC), to use the CPC's large pool of resources to determine the appropriate storage and compute resources necessary to support their target workload.

The increasing demands of storage management are resulting in several new models for storage system provisioning, as well. Service providers, such as Exodus [Gro01], allow enterprises to lease storage from a pool of storage made available by third party providers. Companies including HP, IBM and Compaq support instant capacity on demand (ICOD) for storage, enabling customers to expand storage systems nearly instantaneously. These models imply that there is a pool of storage resources available to be allocated during initial system sizing.

Finally, the iterative configuration process can only occur if there is a method for generating a representative workload on the system before it is deployed into production use. In any of the storage pool scenarios described above, the administrator may set up the application(s) to be run on the new system. A representative input workload may come from a log of application requests on an existing production system, which is then replayed against the system being configured.

5.2.2 The iterative loop

Analyzing the ad hoc process described in the previous section, we observe three stages that are followed in sequence:

- **Analyze workload**: Analyze the running system to determine its performance characteristics. This information can then be used to produce a better system design. If the system is not yet running, the analysis output is based on the capacity requirements of the workload, and any guesses the administrator may have about the workload's performance requirements.

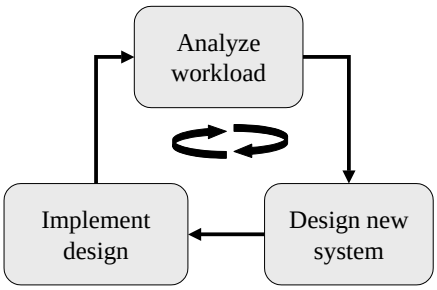


Figure 5.1: Three stages of an iterative loop for managing storage systems

- **Design new system**: Based on available inputs (typically previous observations of the system behavior), design a system that should better match the workload requirements.
- **Implement design**: Implement the system – create the LUs on the storage devices, build the logical volume data, and migrate the existing application setup (if any) to the new design.

As described in the previous section and shown in Figure 5.1, these stages can form an iterative loop. The loop can be bootstrapped at the design stage using only the capacity requirements of the application(s), which provide an absolute lower limit on the number of storage devices required. An initial guess at performance, perhaps obtained through experience with similar applications, can also be used as a starting point.

Once this initial system design has been created, the loop iterates to generate a design that better meets the actual requirements of the workload. On each iteration, it analyzes the workload on the current system, summarizing information about I/O and capacity usage. The design stage uses the summary to generate an improved system design. Finally, it implements the new system design and migrates the existing system to the new design.

During the course of several iterations, the storage system performance is improved through the addition of more devices over which the load can be distributed until the performance of the application as a whole, that is, both server and storage, is not limited by the storage system. The loop converges on a suitable system design when the workload's performance requirements are satisfied and the number of storage devices in the system stops changing.

The time to converge is determined by how long each iteration takes and how many loop iterations must be performed. The time for each iteration is dominated by running the application and implementing the design. Application run times can range from minutes to hours. Implementing the design can also take minutes to hours, because it involves moving some fraction of the (potentially sizeable) data in the system.

The number of loop iterations depends on the size of the final system and the degree of mismatch between the initial design and the final design. The number of iterations may be reduced if the initial design is made using an initial performance guess.

Sometimes the user of the system may not be willing to buy the amount of storage required to support the performance requirements of the workload. In this case, the loop can be configured to produce a system design that is limited to a maximum price with the storage workload balanced across the available devices. Although this design will not meet the workload's performance requirements, it will meet the user's cost constraints. Conversely, the user may wish to purchase more resources to accommodate future growth or to leave headroom for unexpected peak loads.

A simple example may help to illustrate how the different components of the loop work together. Consider a workload that uses 10 filesystems. Each filesystem needs a logically contiguous part of the storage system. We call each of these parts a *store*, as described above. Assume that each store is 1 GB in size and that the LUs in the storage system are 18 GB in size, each capable of performing 100 I/Os per second (IOPS). The initial capacity-only design will place all ten stores on a single LU. Now, assume that when the application runs, it performs 50 IOPS to each filesystem. During the subsequent iteration of the loop, the analysis stage summarizes the capacity and I/O requirements of the workload. The design stage uses this information to choose a new design that has at most two stores on each LU, as only two stores will fit onto an LU without exceeding the 100 IOPS throughput limit. Finally, the loop implements the new design by migrating eight of the ten stores from the single LU onto four new LUs allocated in the design stage.

As demonstrated in our example, there are four key components used to implement the iterative loop shown in Figure 5.1. The first component, which implements the *analyze workload* stage, monitors a workload's performance and summarizes its capacity and performance requirements for an input to the design system stage. The *design system* stage is implemented by two components: a performance model and a design engine, or *solver*. The performance model component encapsulates the maximum performance capabilities of the storage device. The solver provides the ability to design a new, valid storage system (e.g., one that does not exceed the available capacity or I/O performance of any device in the system, as determined by the model). The final component performs the *implement design* stage, including migrating the existing design to the proposed one. The implementation of each of the analysis, model, design, and migration components can range from simple to complex.

In the following subsections, we describe a progression of successively more sophisticated versions of the iterative loop, by describing the improvements made to each of the components. We begin with a simple automated loop, which implements the manual loop executed by human administrators today, and progress to the automated Hippodrome loop, which employs advanced components to handle most of

the complexities of I/O workloads. Figure 5.2 illustrates this progression. Each step of the progression is a coherent implementation of the loop that is more accurate, more flexible, or faster than the previous approach. For example, the solution described in Section 5.2.4 achieves a balanced load in the final system, whereas the one in Section 5.2.3 does not. We will describe how making improvements to some of the components requires improvements to other components. For example, a solver that can create a design that moves multiple stores requires a migration component that can migrate multiple stores as a logically single operation.

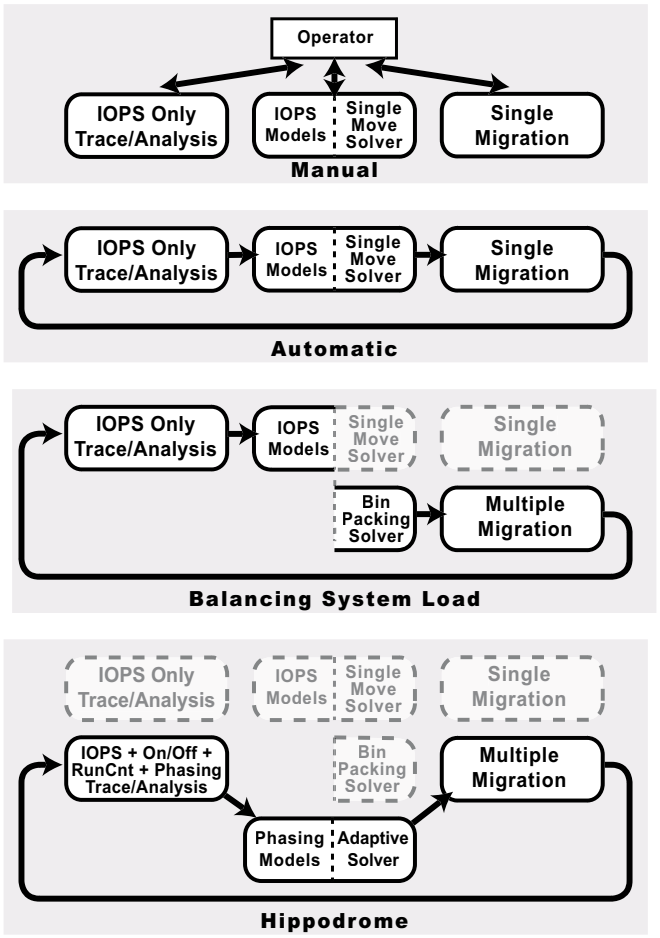


Figure 5.2: Loop progression. As the analysis, solver and migration components improve, so does the resulting loop approach. The automatic approaches are described in more detail in Section 5.2.3 through Section 5.2.5.

5.2.3 Automating the loop

The primary disadvantage of today's manual loop is that it relies on administrators to make all of the decisions and to do all of the work. Administrators must gain enough experience to determine when

an LU is overloaded and to decide which stores to move. They need to test many possible actions and determine which ones work and which ones do not. Some tools provide a degree of automation for moving stores within a single disk array. However, if they use multiple arrays, which is common in enterprise-scale systems, they must manually move stores between arrays. Because of these problems, this approach is extremely human-intensive, and hence slow, expensive, and error-prone.

We can remove the human from the loop by automating each of the manual stages described above. This simple automated loop is shown as the second approach in the loop progression of Figure 5.2.

Analysis component

The workload analysis component of the simple automated loop takes a trace[4] of I/Os from the running workload and calculates a summary of the trace. The summary consists of two parts: *stores* and *streams*. The stores represent the capacity requirements of the logical volumes in the system. The streams represent the I/O accesses to a store, in this approach the number of I/Os to the store divided by the elapsed time of the trace (IOPS). Each stream refers to accesses to a single store, and each store has at most one corresponding stream. In the Hippodrome approach below, the streams will capture many more properties.

Model component

The performance model for the automated loop adds together the IOPS for each stream on a particular LU and compares the sum to a pre-specified maximum, obtained from manufacturers' specifications or from direct measurement. For example, if a disk can re-position in about 10ms, then an LU consisted of a single-disk can perform about 100 IOPS.

Design component

The design component automates the simple "move one store from an overloaded device" algorithm sometimes used by administrators. It picks one store from an overloaded LU and checks to see whether it fits (according to the models) on another LU. If it does, then the store is moved to that LU. If it does not fit on any of the remaining LUs, more storage is added to the system by, for example, enabling additional ICOD storage, and the store is moved to the new LU.

Migration component

The migration component of the automated loop copies the data for the store to be moved to the new location, and deletes the old copy. Because we are addressing initial system configuration, we can stop

[4]The CPU overhead of taking the trace in our experience is 1-2%; the traces can take up a few GB for a day-long trace, which is negligible as the trace only has to be kept until analysis is run.

Figure 5.3: Problems with the simple loop. Each disk can handle 100 IOPS. The first example shows a failure of load balancing. The second example shows a failure to purchase a minimal number of disks. Stores are moved to available disks one at a time in this example.

the application during the migration phase, so we do not have to worry about application accesses to the store during the migration execution. The migration stage also does not need to worry about space problems on the target device because the solver would not suggest the new location for the store unless sufficient free space exists on the target LU.

Problems with the simple automated loop

Because the above approach is a simple automated version of what the administrator does manually, it has a number of problems: it may not balance the load in the system, it may allocate more resources than required, and the simplistic models that it uses may lead to poorly provisioned systems.

First, Figure 5.3 shows that the simple automated loop may not balance the load in the system, because it makes all of the design and migration decisions locally. Consider a scenario where each LU is capable of handling 100 IOPS, and the starting point is generated using only capacity information. Imagine we start with nine stores requiring 25 IOPS, all packed onto a single LU. After four iterations, the first LU would still contain five stores (at a total of 125 IOPS), and the second LU would contain the remaining four stores (at a total of 100 IOPS). One additional iteration would move the fifth store from the first LU onto a third LU. The final system would then have two LUs, each with four stores and 100 IOPS total load, and a

third LU with a single store and 25 IOPS total load. A more balanced design would put three stores and an aggregate load of 75 IOPS on each of the three LUs.

Second, Figure 5.3 also shows that this approach may use more resources than necessary to satisfy the workload in the final system, also because of localized decision making. Consider a system with three LUs each capable of 100 IOPS. Two LUs each have four stores at 20 IOPS each (for a total of 80 IOPS), and the third has two stores at 60 IOPS each (for a total of 120 IOPS). The solver will choose one of the 60 IOPS stores from the overloaded LU and move it to a new LU. A better choice would be to swap two of the 20 IOPS stores on one of the first two LUs with the 60 IOPS store on the third LU, creating a system design of 2 20 IOPS stores and 1 60 IOPS store on two of the LUs and 4 20 IOPS stores on the remaining LU, which fits within the available IO capacity of the LUs.

Finally, since this approach uses a simplistic measure of performance, it ignores many aspects of device utilization, such as request size, request type and sequentiality. For example, a workload that performs 100 random 64k reads/second is much more disk-intensive than a workload that performs 100 sequential 64k reads/second, but the IOPS metric considers those two access patterns to result in the same device utilization.

5.2.4 Balancing system load

Figure 5.2 shows we can build upon the simple automated loop approach described in the previous section by incorporating new design and migration components. These components tackle the problems of unbalanced final systems and purchasing too many devices. This approach continues to use the analysis and model components of the previous approach, so we do not discuss those components here.

Improved design

To make the loop produce a balanced final system and not over-provision, we must improve the design stage. The problem of efficiently packing a number of stores with capacity and IOPS requirements is very similar to the problem of multi-dimensional bin packing. Although bin-packing is an NP-complete problem, there are several algorithms that produce good solutions in practice [dlVL81, JDU+74, Ken96]. We extend the bin-packing algorithms to balance the load after generating a successful solution. The final load-balancing can be done by removing individual stores and attempting to re-assign them to a location that results in a more balanced solution. The final load-balancing step is restricted to produce a solution no more expensive than the input to that step.

Improved migration

The bin-packing algorithms may propose a new system design that requires moving multiple stores. Unfortunately, there may not be sufficient space on the target LU(s) to move all of the stores. For example, if all of the devices are nearly full, and we have to swap some of the stores, then we may need to temporarily move a store to scratch space to perform the swap. The previous approach did not have this problem because the solver guaranteed that the single store to be moved would fit onto the target LU. This guarantee does not hold for multiple store migration. As a result, we need a migration component which can move multiple stores in a single iteration.

For this approach, multiple-store migration consists of a planning phase and an execution phase. The planning phase calculates a plan which tries to minimize the amount of scratch space which is used and minimize the amount of data which needs to be moved. The migration problem is also NP-complete, as it is reduceable to subset sum [GJ79], so we use a simple greedy heuristic that will move stores to the final location if possible, and will otherwise choose a candidate store and move all of the stores blocking it into scratch space. This heuristic creates a sequential plan for the migration. If we can move parts of a store at a time instead of having to move the entire store,[5] we can use the more advanced algorithms found in [AHH+01], which generate efficient parallel plans.

In the execution phase we can apply the same approach used in the previous automated loop, that is copying the stores to the appropriate destination (either scratch space or the final destination). Another possible approach is to copy the data from a "master copy" of the stores to the final destination. This second approach, commonly used in capacity planning centers, has the disadvantage of requiring double the storage capacity to hold a copy of both the master and working data stores.

Problems with the load-balancing loop

The primary limitation of the load-balancing approach is that the simplistic IOPS models used so far do not sufficiently capture the performance differences between sequential and random accesses, reads vs. writes, and the on/off behavior of streams. Thus, the challenge remaining is to more accurately model the performance of storage systems.

More complex models will also highlight a problem with the bin-packing algorithms. They assume that each of the requirements (e.g., performance and capacity) are additive. For example, if the utilization of store *s1* is *u1*, and the utilization of store *s2* is *u2*, they assume that the utilization of *s1* and *s2* on the same device is *u1+u2*. These assumptions are fine for the models used in the current approach, since both the IOPS and capacity requirements are additive. However, more complex performance models are not additive.

[5]The HP-UX logical volume manager, which provides the underlying mechanisms for migration execution, does not currently support moving part of a store.

Attribute	Description	Units
request_rate	mean rate at which requests arrive at the device	requests/sec
request_size	mean length of a request	bytes
run_count	mean number of requests made to contiguous addresses	requests
queue_length	mean size of the device queue	requests
on_time	mean period when a stream is actively generating I/Os	sec
off_time	mean period when a stream is not active	sec
overlap_fraction	mean fraction of the "on" period when two streams are active simultaneously	fraction

Figure 5.4: Workload characteristics generated by Hippodrome's analysis stage.

5.2.5 Hippodrome

Hippodrome, shown at the bottom of Figure 5.2, builds upon the previous approach by greatly improving the performance models and improving the design component to take advantage of them.

Improved analysis

The simplistic models used in previous approaches required only very simple analyses. In Hippodrome, we improve the analysis component to capture properties necessary to improve the device models. In particular, Figure 5.4 shows all of the attributes we add.

We model an I/O stream as a series of alternating *on/off* periods. During an on period, we measure four parameters separately for reads and writes. The first parameter is the *request rate*, which is the mean of the I/O request rates during on periods. The second parameter is the mean *request size*. The third parameter is the *run count*, which is the mean number of sequential requests. A request is sequential if its start offset is at the location immediately after the end offset of the previous request. The fourth parameter is the *queue length*, which is the mean number of requests outstanding from the application(s). Because streams can be on or off at different times, we also model the inter-stream phasing. The *overlap fraction* is approximately the fraction of time that two streams' on periods overlap. The actual definition used by the models is slightly more involved because of the queuing theory used in the models and is described in [BGJ+98].

Improved performance models

Hippodrome uses the table-based models described in [And01], which improve on the simplistic performance models of previous approaches by differentiating between sequential and random behavior, read and write behavior, and on-off phasing of disk I/Os.

The performance models have three complimentary parts. The first part reduces the sequentiality of interfering streams and increases the overall queue length of overlapping streams. The second part uses

tables to estimate the utilization of each individual stream based on the new, updated metrics. The third part combines together the utilizations for multiple streams based on the phasing information to calculate the overall utilization of each LU.

The models take as input for both reads and writes the mean request rate, request size, queue length and sequentiality, as described in the analysis section.

The input queue length and sequentiality are adjusted to take into account the effect of interactions between streams on the same LU using the techniques described in [UAM01]. The sequentiality is decreased for two streams that are on simultaneously, because the overlap will cause extra seeks. The queue length is increased because there will be more outstanding I/Os, giving the disk array more opportunity for re-ordering.

The utilization of each stream is calculated using a table of measurements. The model looks up the nearest table entries to the specified input values for the stream, and then performs a linear interpolation to determine the maximum request rate at those values. Given the maximum request rate, the utilization is the mean request rate of the stream divided by the maximum possible request rate.

The third part of the model then calculates the final utilization of each LU by combining the estimated stream performance values using the inter-phasing algorithms found in [BGJ+98]. The algorithms use queuing theory techniques so that the utilization of two streams is proportional to the fraction of time that they overlap.

Improved design

Introducing the more complex models violates the bin packing algorithms' assumption that individual stream utilizations are additive, as described in Section 5.2.4. Because two sequential streams cause inter-stream seeks, the utilization of two simultaneous sequential streams is higher than the sum of the utilization of either stream individually. Conversely, because two streams may not both be on at the same time, inter-stream phasing implies that the utilization of two streams may be less than the sum of the utilization of the individual streams. We therefore need an improved design component that can cope with the more accurate, but more complex model predictions.

The adaptive solver [AKS+01] in the Hippodrome design stage builds on the best-fit approaches found in [dlVL81, JDU+74, Ken96] and augments them with backtracking to help the solver avoid local minima in the search space of possible designs.

The adaptive solver operates in three phases. The first phase of the solver algorithm attempts to find an initial, valid solution. It does this by first randomizing the list of input stores, and then individually assigning them onto a growable set of LUs. The solver will assign stores onto the best available LU, and if the store does not fit onto any available LU because the resulting utilization or capacity would be over

100%, then the solver will allocate an additional LU. The best LU is the one closest to being full after the addition of the store, since the aim is to minimize the number of LUs.

The second phase of the solver algorithm attempts to improve on the solution found in the first phase. Randomized backtracking extensions are used, which enable the solver to avoid the bad solutions that would have been found by the simpler algorithms. The solver randomly selects an LU from the existing set, removes all the stores from it, and re-assigns those stores in a similar manner to the assignments of the first phase. It repeats this process until all of the LUs have been reassigned, and then goes back and repeats the entire reassignment process two more times[6]. At the end of this phase, we have a near-optimal but non-balanced assignment of stores to LUs, using the minimum necessary storage configuration.

The third phase of the solver algorithm load-balances the best solution found in phase two in the same way as for the bin-packing algorithm. The solver removes a single store from the assignment and then re-assigns it with the goal of producing a balanced packing, rather than the goal of a tight packing that was used in the first two phases. The solver has already packed the stores tightly in the first two phases, and guarantees that the balanced solution does not increase in cost. The third phase repeats the process of randomly selecting a store and re-assigning it, with the aim of producing a more balanced solution.

Experiments with this solver have found that it produces good solutions. For the cases where we can prove optimality (e.g. synthetic workloads), the solver generates optimal solutions. For more complex cases, we cannot prove optimality because the problem we are addressing is NP-complete; in practice, the solver seems to do well on realistic inputs.

5.2.6 Hippodrome vs. control loops

The Hippodrome (and the load balancing) loop does not behave like a simple control (or, feedback) loop, because it contains models of the system it is designing. As a result, if the workload remains constant, the design that is generated also remains constant. This is different from a control loop which will increase and decrease the available resources and use some metric to perform a “binary search” for the correct amount of resources. Even if the workload does remain constant, a control loop may have to continually adjust the resources to see if the metrics of interest are changing.

Both Hippodrome and control loops take a period of time to converge, but for different reasons. A control loop takes the time to converge because it tries to adjust the set of control parameters of the system based on the inputs. In the Hippodrome case, the workload is actually changing while the system is trying to converge. In the beginning, the workload can not actually run at its target rate, and as a result when the workload is given an expanded system, it uses the expanded resources and may still request more until the storage system is no longer the bottleneck. Once the system has converged, the workload’s requirements are

[6] A configurable parameter, two is more than sufficient for these workloads.

met and the system no longer changes.

The Hippodrome loop can exhibit the *appearance* of oscillation if the workload is running very close to the border between a resource increment. For example, if an LU can support 100 IOPS, and a workload requires 100 IOPS with a standard deviation of 2 IOPS, the system will oscillate between one and two LUs as the standard deviation causes the requirements to go above and below 100 IOPS.

5.2.7 Breaking the loop

With each of the loops presented in this section, a few basic assumptions have been made. It is possible that these assumptions are not true which in turn forces the loops not to converge to a valid configuration.

The first assumption is that the host operating system is capable of providing information on the workload, such as the request rate of a given workload. In the case of Hippodrome the additional set of workload characteristics shown in Figure 5.4 are also required. Fortunately, the measurement interfaces on most modern operating systems make it possible to record this information. A related issue is the fidelity of the information – ideally, the system traces all I/Os, and does not drop or otherwise summarize the I/O records. Doing so would result in inaccurate information being supplied to the design stage, which would in turn result in a design that did not match the actual workload requirements. Although we cannot control this, our experiments on HP-UX systems have shown that this is not a problem, except under extremely loaded conditions. We have observed this only in the laboratory, using specialized tools, and never using real-world applications.

The second assumption is that applications do not modify their behavior based on knowledge of their data layout on the storage system. Such applications would, in our belief, interact poorly with any of the loop approaches presented, as they would not maintain a constant workload behavior as iterations of the loop modify the storage system. In this case, it is possible that the loops would be unable to converge to a stable design. Since the role a logical volume manager is to virtualize the storage system and most applications rely on logical volumes (either raw or through a file system), the physical data layout is not visible by the applications. This makes it difficult for applications to modify their behavior based on the data layout.

Finally, overly optimistic performance models could potentially cause Hippodrome to settle on a design that does not support the given workload. This is due to the fact that the design stage depends on the models to allocate resources. Overly pessimistic models, on the other hand, cause Hippodrome to generate over-provisioned designs that cost more than the necessary amount to support the workload. Although the current models incorporated into Hippodrome generally do a good job, we have encountered a few situations where our models were overly pessimistic, and some other situations where they were overly optimistic.

In summary, there are a few scenarios that may “break the loop”. Two of these are external to Hippodrome, and there is little we can do about them, except identify them when they occur, so that remedial action can be taken. The third, that of inaccurate models, is of more concern, since the models are fundamental to the correct operation of the system; this is currently an active area of investigation.

5.3 Experimental overview

In this section, we give an overview of the set of experiments we run to determine how Hippodrome performs. Our experiments focus on the following questions:

- **Convergence**: How fast does the Hippodrome converge to a valid system design that supports a given workload?
- **Stability**: Does Hippodrome produce stable system designs that do not oscillate between successive loop iterations after convergence?
- **Resource allocation**: Does Hippodrome allocate a reasonable set of resources for a given workload?

5.3.1 Workloads

Our evaluation is based on a variety of synthetic workloads and a expanded version of the PostMark [Kat97] benchmark described below. The synthetic workloads are useful for validating whether the Hippodrome loop performs correctly, because we can determine the expected behavior of the system. The PostMark benchmark is useful because it lets us investigate how Hippodrome performs under a slightly more realistic workload that simulates an email system.

In our experiments, we used synthetic workloads shown in Table 5.5 with fixed-size, random requests. It generates a load that ranges between 12.5 IOPS to 50 IOPS for each individual stream. We also used workloads that exhibit complex phasing behavior where groups of streams had correlated on/off periods. We generated these workloads using a synthetic load generator capable of controlling the access patterns of individual streams. For each stream, it generates the access pattern based on the request rate, request size, sequentiality, maximum number of outstanding requests and the duration of on/off periods. We used the Poisson arrival process for each stream in the synthetic workloads and limited the number of requests outstanding from a stream at a given time to a maximum of four requests.

We also used the PostMark benchmark [Kat97], which simulates an email system. The benchmark consists of a series of transactions, each of which performs a file deletion or creation, together with a read or write. Operations and files are randomly chosen. Using the default parameters, the benchmark fits entirely in the array cache, and exhibits very simple workload behaviors, so we scaled the benchmark to use 40 sets

Parameter	Always on	Phased
Store size (MB)	1024	1024
Number of stores	100	100
Request size (KB)	32	32
Request rate (IOPS/stream)	12.5, 25	50
Request type	read	read
Request offset	1KB aligned	1KB aligned
Run count	1 (random)	1
On/Off periods (sec)	always on	4.5 / 5.5
Correlated Groups	n/a	2 stream groups
Arrival process	open Poisson	open Poisson

Figure 5.5: Common parameters for synthetic workloads.

of 10,000 files, ranging in size from 0.5 KB to 200 KB. This scaling provides both a large range of I/O sizes and sequentiality behavior. In order to vary the intensity of the workload, we ran multiple identical copies of the benchmark simultaneously on the same filesystem. The data for the entire PostMark benchmark was sized to fit within a single 50 GB filesystem.

5.3.2 Experimental infrastructure

Figure 5.6 shows our experimental infrastructure, which consists of an HP FC-60 disk array [Hew00b] and an HP 9000-N4000 server. The FC-60 array has 60 disks, each of which is a 36 GB Seagate ST136403LC, spread evenly across six disk enclosures. The FC-60 has two controllers in the same controller enclosure with one 40 MB/s Ultra SCSI connection between the controller enclosure and each of the six disk enclosures. Each controller has 512 MB of battery-backed cache (NVRAM). Dirty blocks are mirrored in both controller caches, to prevent data loss if a controller fails. The FC-60 is connected to a Brocade Silkworm 2800 switch via two FibreChannel links, one for each controller. The switch is present because our SAN includes disk arrays and hosts not used in the experiments.

Our HP 9000-N4000 server had seven 440 MHz PA-RISC 8500 processors and 16 GB of main memory, running HP-UX 11.0. The host uses a separate FibreChannel interface to access the controllers in the disk array.

We have configured each of the LUs in the system as 6 disk RAID-5 with a 16 KB stripe unit size. The common configuration allows us to avoid the multi-hour reconfiguration time.

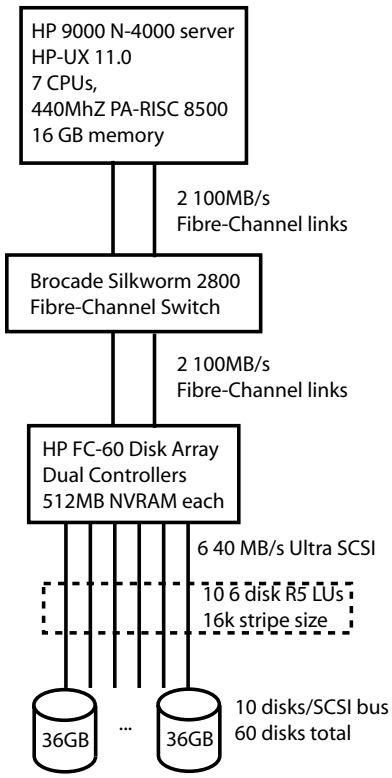
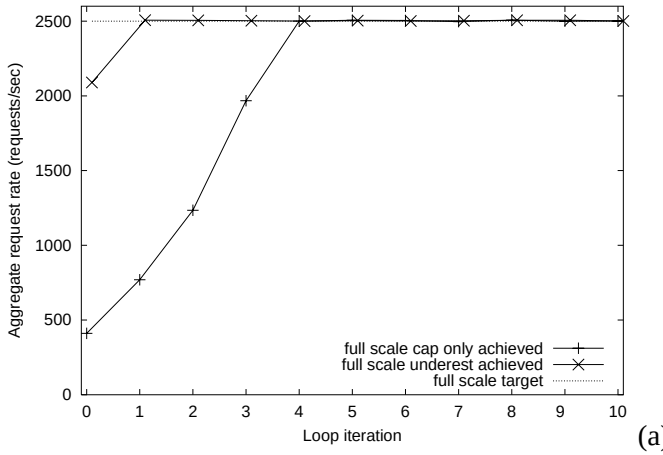
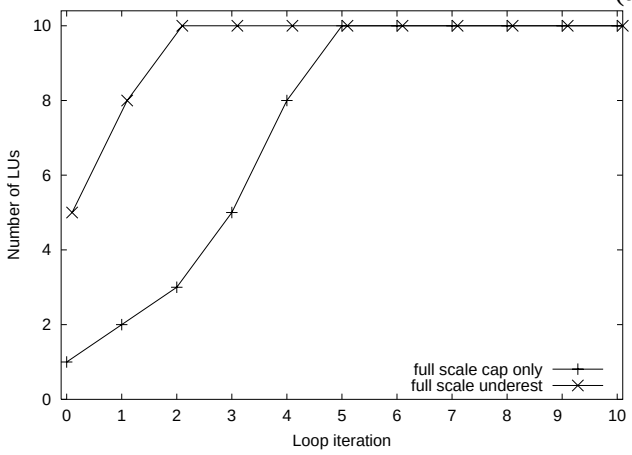


Figure 5.6: Experimental Infrastructure



(a) Average request rate



(b) Number of LUs.

Figure 5.7: (a) Target and achieved average request rates at each iteration of the loop for the synthetic workloads with a target aggregate request rate of 2500 req/sec. (b) Number of LUs used during each iteration.

5.4 Experimental results

In this section we discuss the results of our experiments using the synthetic workloads and the PostMark benchmark. For each workload, Hippodrome generates an initial system design based on the capacity requirements and then iteratively improves the system design until it converges to support the workload. We do not expect the loop to converge in a single step, because the workloads may not be able to run at full speed on the initial capacity-only design. We show that the loop converges quickly and that the system design remains constant once the loop converges. We also show for the synthetic workloads that providing initial performance estimates can speed up the convergence of the loop.

5.4.1 Synthetic workloads

We start with simple synthetic workloads so that it is easy to understand the behavior of the loop. We present two sets of results in this section, one where all streams are on at the same time, and one where

streams have correlated on and off periods.

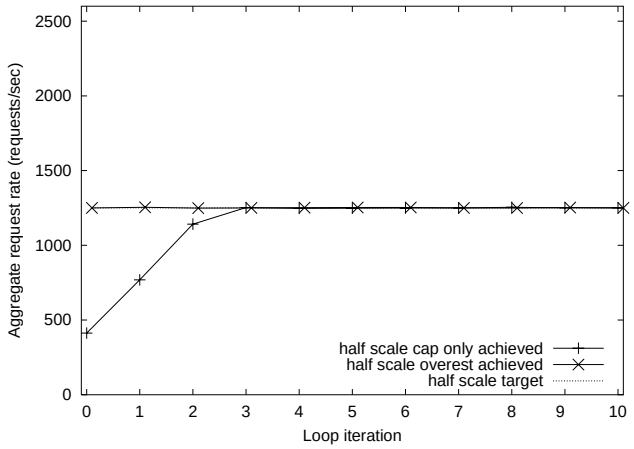
Always on workloads

Figure 5.7(a) shows the target I/O rate and the achieved I/O rate for the synthetic workloads at each iteration of the loop. The figure illustrates two sets of experiments with different input assumptions: one using only capacity information (labeled “cap only”), and one using initial performance information – an underestimate (labeled “underest”). For the capacity-only design, we see that Hippodrome’s storage system design converges within five loop iterations to achieve the target I/O rate of the synthetic workload (2500 requests per second).

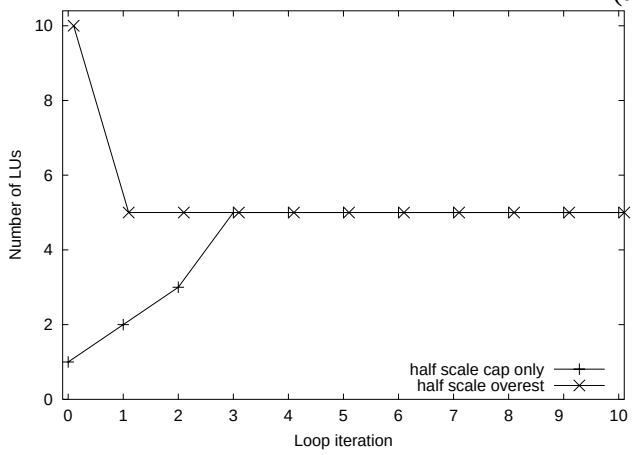
Figure 5.7(b) shows the number of LUs allocated by Hippodrome at each loop iteration to achieve the target I/O rate. The system converges in five loop iterations starting from only capacity requirements as shown in Figure 5.7(a). In the first four iterations, the LUs are over-utilized, and Hippodrome allocates new LUs, increasing the system size to better match the target request rate. As more LUs are added, smaller fraction of the LUs’ capacity is used for the workload’s data. As a result, the seek distances got shorter and the disk positioning times are reduced. However, our performance models were calibrated using the entire disk surface, and therefore slightly under-estimate the performance of the LUs when a fraction of an LU is used. As a result, Hippodrome allocates two more LUs at the fifth iteration in Figure 5.7(b) despite the application achieving its target rate (as discussed in Section 5.2.7). After convergence, however, the system design does not oscillate between successive loop iterations. These results show that Hippodrome can rapidly converge to the correct system design, using only capacity information as its initial input.

Figure 5.7 also demonstrates how Hippodrome can use initial performance estimates to allow the system to converge more rapidly. The system converges in a single iteration by taking advantage of the initial, conservative, but incorrect, performance estimate of 1250 requests per second.

Figure 5.8 shows that Hippodrome uses the minimal number of resources necessary to satisfy the workload’s performance requirements. The target request rate for both workloads is 1250 requests per second, which can be achieved using only five LUs. Given only capacity requirements as a starting point, Figure 5.8(a) shows the loop converges to the target performance and correct size in three iterations. Given an initial (incorrect) performance estimate that the aggregate request rate is 2500 requests per second (twice the actual rate), the loop initially over-provisions the system to use 10 LUs, easily achieving the target performance. The analysis of the actual workload in the first iteration shows that the resources are under-utilized, and Figure 5.8(b) shows Hippodrome scales back the system to use five LUs in the second iteration.



(a) Average request rate



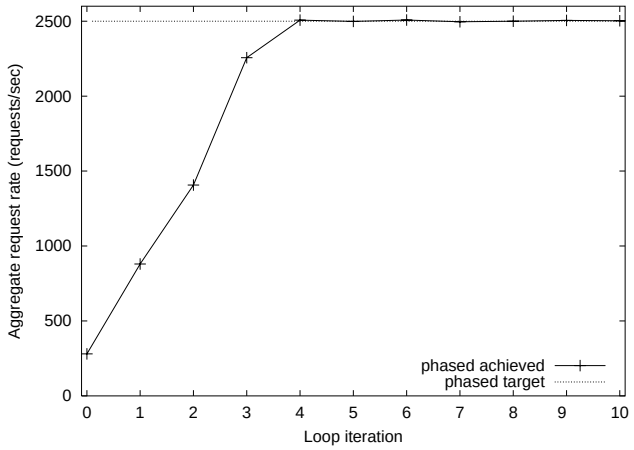
(b) Number of LUs

Figure 5.8: (a) Target and achieved average request rates at each iteration of the loop for the synthetic workloads with a target aggregate request rate of 1250 req/sec. (b) Number of LUs used in each iteration.

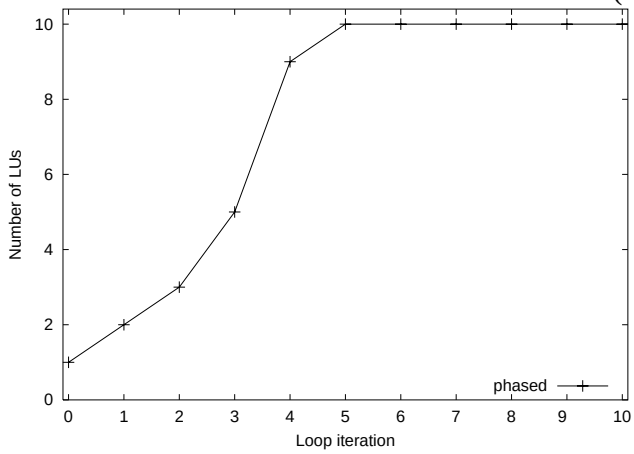
Phased workloads

We also ran experiments where groups of streams had correlated on/off periods. In these experiments, we used two stream groups, with all of the streams in the same group active simultaneously and only one group active at any time. Each group has an IOPS target of 2500 requests per second during its on period, requiring all 10 LUs available on the disk array. Clearly, the storage system could not support the workload if both of the stream groups were active at the same time, but since the groups become active alternately, it is possible for the storage system to support the workload. Figure 5.9 shows the average request rate achieved. We can see that Hippodrome worked very similarly to how it did for the earlier, always-on workload.

We now look at the distribution of the stores across the LUs. There are 100 stores in total; 50 in each group. What we expect is that each of the 10 LUs will end up containing 5 stores from group 1 and 5 stores from group 2. The imbalance of an LU is therefore the absolute value of the difference between the number of group 1 and group 2 stores on that LU. The *relative imbalance* over the entire storage system is



(a) Average request rate



(b) Number of LUs

Figure 5.9: (a) Target and achieved average request rates at each iteration of the loop for the synthetic workloads with two correlated stream groups with a target aggregate request rate of 2500 req/sec. (b) Number of LUs used in each iteration.

then the sum of the imbalance of each LU divided by the number of LUs. In a balanced system, this metric should converge to zero. Figure 5.10 illustrates the relative imbalance for the phased workload. This figure shows that the solver correctly puts an equal number of stores from each group on each LU for the phased workload; the imbalance goes to zero once the storage design has sufficient LUs.

5.4.2 PostMark

We ran the PostMark benchmark with a varying number of simultaneously active processes, which allows us to see the effect of different load levels on the behavior of the loop. Unlike the experiments performed with synthetic workloads, there is no predetermined goal for this system, except to provide “good” performance. In order to determine what “good” was in practice, we first ran a set of experiments with the PostMark filesystem split over a varying number of LUs. Figure 5.11 shows how the PostMark transaction rates change as a function of the number of LUs and processes used. As can be seen, the system is limited

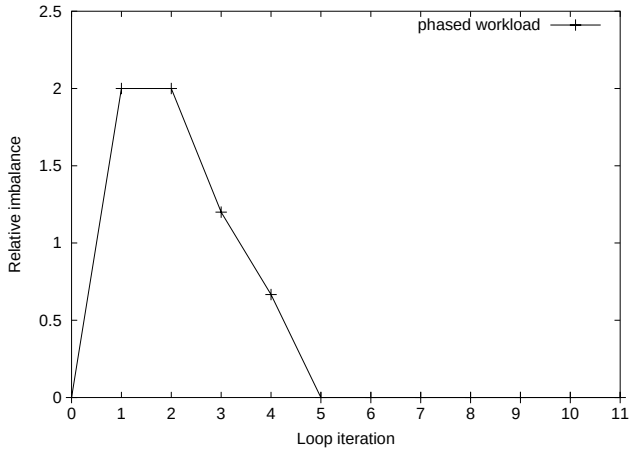


Figure 5.10: Relative imbalance of the two stream groups over the storage system for the phased workload.

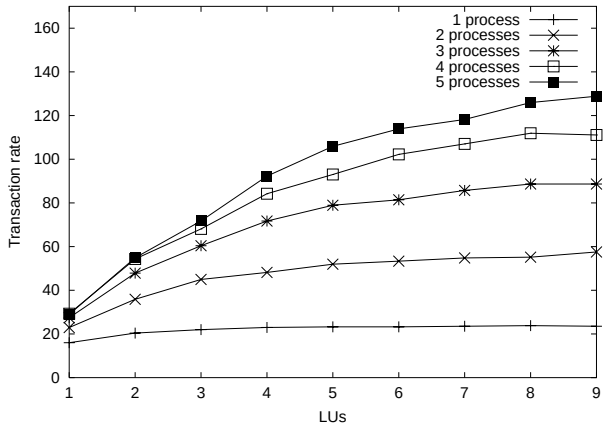


Figure 5.11: PostMark transaction rate as a function of number of LUs and processes used.

primarily by the number of LUs. In all cases, the performance continues to increase as resources are added, although with diminishing returns. We presume that the performance will eventually level off, due to host software limitations, but we did not observe this for any except the one process case. Ideally, Hippodrome would exhibit two properties with this workload. First, it would converge to a stable number of LUs, and not keep trying to indefinitely expand its resources. Second, the system it converged to would be near the inflection point of the performance curve, i.e. increasing the number of LUs beyond this point would not result in significant further performance increases.

When we first ran the PostMark system, we found that the system did converge, but to a system that was well below the achievable performance levels. This result is indicative of the models under-predicting the utilization of the storage system, a problem discussed in Section 5.2.7. We determined that the PostMark benchmark had only 2.4 I/O's queued on average, whereas the minimum value found in the table-based

#processes	*headroom*			
	1.0		0.9	
	LUs	% of max	LUs	% of max
1	2	87%	2	87%
2	2	61%	3	78%
3	3	68%	4	81%
4	4	76%	5	84%
5	5	82%	6	88%

Figure 5.12: LUs and transaction rate achieved (as a percentage of the maximum observed for any number of LUs) for various *headroom* values with the PostMark workload.

models were 16 I/O's queued on average[7]. This obvious disparity is easily detectable by the model software. The correct solution to this problem is to improve the models, but a workaround exists in the *headroom* parameter, which is used by the solver to adjust the maximum device utilization, and thus produce solutions which use more or less resources (for smaller and greater values of *headroom* respectively).

Table 5.12 shows, for various headroom values, the results achieved from running the PostMark benchmark with Hippodrome. As can be seen, with lower *headroom* values, the system will converge to a solution nearer to the maximum possible. A value of 0.9 works well for this workload, resulting in systems that provide about 85% of the maximum possible performance, while using substantially fewer resources – i.e. they find solutions that are well placed on the price/performance curve. In each case, Hippodrome converged in less than 6 loop iterations.

The wall clock time required for the loop to converge is roughly 2 1/2 hours. The first iteration, starting from the capacity-only design, takes about 40 minutes, and subsequent iterations take about 30 minutes. In each iteration, the application runtime is roughly five to ten minutes. Almost all the remaining time is spent copying the data from a master copy to the correct location in the new design. The overall size of the dataset was 50 GB.

5.4.3 Summary

The initial system configuration experiments show that, for all workloads, Hippodrome satisfies the three properties introduced in Section 5.3. First, the system converges to the correct number of LUs in only a small number of loop iterations, at most four or five iterations, and sometimes in only one or two. Second, the solutions are stable; they do not oscillate between successive loop iterations, but remain constant once the workload is satisfied. Third, the designs that the system converges on are not over-provisioned; that is, the storage system contains the minimum number of LUs capable of supporting the offered workload.

[7]This is because all of the measurements used to create the table based models were taken before we understood the range that we should be measuring.

Finally, Hippodrome can leverage initial performance estimates (even inaccurate ones) to more quickly find the correct storage solution.

These properties mean that Hippodrome can realistically be used to automatically perform initial system configuration. The system administrators need only provide capacity information on the workload, and can then let Hippodrome handle the details of configuring the rest of the system resources, in the expectation that this will happen in an efficient manner. In particular, administrators do not have to invest time and effort in the difficult task of deciding how to lay out the storage design; nor do they have to worry about whether the system will be able to support the application workload.

5.5 Related work

The EMC Symmetrix [EMC00] and HP SureStore E XP512 Disk Arrays [Hew00a] support configuration adaptation to handle over-utilized LUs. They monitor LU utilization and use thresholds, set by the administrator, to trigger load-balancing via data migration within the array. The drawback is that they are unable to predict whether the move will be an improvement. Hippodrome's use of performance models allows it to evaluate whether a proposed migration would conflict with an existing workload.

HP's AutoRAID disk array [WGSS96] supports moving data between RAID 5 and RAID 1. AutoRAID keeps current data in RAID 1 (since it has better performance), and uses an LRU policy based on write rate and capacity to migrate infrequently accessed data to RAID 5, which has higher capacity. Hippodrome will correctly place data based on the usage patterns, and will expand the storage system if necessary to support increases in the workload.

TeraData [Bal98] is a commercial parallel shared nothing database that uses a hash on the primary index of a database table to statically partition the table across cluster nodes. This data placement allows data parallelism and improves the load balance. In contrast, Hippodrome dynamically reassigns stores, based on observed device utilizations.

IBM's work on capacity space management [PM00] guides the re-balancing of existing storage (and other) resources using the life expectancy of the resource. Their approach, described for a Lotus Notes-based environment, uses historical usage data to predict when the resource will exceed a specified limit, and either extends the limits or moves the workload. In contrast to this historically-based predictive approach, Hippodrome monitors the current performance of the existing design, reconfiguring the system when necessary in response to the workload's actual behavior.

A few other, automated tools exist that are useful to administrators of enterprise-class systems. The AutoAdmin index selection tool [CN97] can automatically "design" a suitable set of indexes, given an input workload of SQL queries. It has a component that intelligently searches the space of possible indexes, similar to Hippodrome's design component, and an evaluation component (model, in Hippodrome terms) to

determine the effectiveness of a particular selection based on the estimates from the query optimizer. Océano [AFF+01] focuses on managing an e-business computing utility without human intervention, automatically allocating and configuring servers and network interconnections in a data center. It uses simple metrics for performance such as number of active connections and overall response time; it is similar in nature to the automatic loop in section 5.2.3 in its management of compute and network resources.

Existing solutions to the file assignment problem [DF82, Wol89] use heuristic optimization models to assign files to disks to get improvements in I/O response times. The work described on file allocation in [GWS91, SWZ98] will automatically determine an optimal stripe width for files, and stripe those files over a set of homogeneous disks. They then balance the load on those files based on a form of "hotspot" analysis, and swapping file blocks between "hot" and "cold" disks. Hippodrome can expand or contract the set of devices used, supports RAID systems, uses far more sophisticated performance models to predict the effect of system modifications, and will iteratively converge to a solution which supports the workload.

5.6 Conclusions

Due to their size and complexity, modern storage systems are extremely difficult to manage. Compounding this problem, system administrators are scarce and expensive. As a result, most enterprise storage systems are over-provisioned and overly expensive.

In this paper we have introduced the Hippodrome loop, our approach to automating initial system configuration. To achieve this automation, Hippodrome uses an iterative loop consisting of three stages: *analyze workload*, *design system*, and *implement design*. The components that implement these stages handle the problem of summarizing a workload, choosing which devices to use and how their parameters should be set, assigning the workload to the devices, and implementing the design by setting the device parameters and migrating the existing system to the new design.

We have shown that for the problems of initial system configuration, the Hippodrome loop satisfies three important properties:

- **Rapid convergence**: The loop converges in a small number of iterations to the final system design.
- **Stable design**: The loop solution remains stable once it has converged.
- **Minimal resources**: The loop uses the minimal resources necessary to support the workload.

We have demonstrated these properties using synthetic I/O workloads as well as the PostMark file system benchmark.

We can see how the automation (principle #2 from Chapter 1) of storage administration has been substantially improved. In addition, by increasing the consistency of performance during workload changes,

the dependability (principle #1) has also been improved. Because of limitations with the prototype, Hippodrome has only demonstrated limited scalability (principle #3). Furthermore, Hippodrome generates more complex configurations, and is itself somewhat complex, so the simplicity (principle #8) of the system has been reduced.

Hippodrome illustrates the research approach of "Sneak in-between." This approach has the advantage that it extends an existing system. As a result, it is much easier to test and deploy. Furthermore, it leverages previous research and development more directly. Unfortunately, to perform this type of research, there needs to be a hook, or indirection point available in the existing system. In our case, the indirection was through the logical volume manager. However, because this hook is used, the existing interfaces may limit the design choices, and may require the researchers to understand complex internal interfaces. The bottom line to this approach is that it is efficient, but may be limited.

Chapter 6

Future directions

System administration is a relatively new field for academic research. As a result, we spend this chapter describing future work in a number of directions. We describe these following the task breakdown found in Chapter 2, and leave the more generalized musings for the conclusions. We believe that examining tasks that are performed by administrators, and trying to solve them will help academic researchers understand the system administration problems, and then build the more general ideas.

We observe that there is a common pattern to research in system administration. First, there is the identification of a problem. This usually happens as a result of real experience, and is best done by talking with people in the field or from use of a system. Second, if the problem involves an existing system, there is a monitoring step where the solution is to understand the problems faster and better so that a human can fix them. Third, there are systems which automatically handle a sub-class of the problems that the monitoring solution could identify in order to reduce the workload of the human. Fourth, the system is extended to improve it along one of the principles usually without weakening it for the others. We can see this pattern of research both in our research and in the research of others as described in Chapter 2.

This discussion is not intended to be exhaustive. We discuss only the subjects which have enough research to justify a summary. Unfortunately, this leaves out the tasks which have recently become important. Furthermore, even if we included all of the tasks, ongoing changes in technology and computer usage constantly creates new problems, and hence new tasks. In this Chapter, we first cover the important topic areas discussed in Chapter 2, and then we cover the future work for Chapters 3-5.

6.1 Software installation: OS, application, packaging and customization

There have been a remarkable number of papers in this area, many of which seem like slight variations of each other. Closer examination indicates that each solution handles a slightly non-overlapping set of problems, and that the solutions can not be easily combined together because they are not separated

out. This lack of separation makes us wonder if the problem has been broken down poorly. We therefore propose decomposing the problem into the following five pieces: Packaging, Selection, Merging, Caching, and End-User Customization.

The distinction between installing applications and the operating systems is unnecessary and a historical artifact. Originally the work necessary to install an operating system was vastly different than work to install applications. Now most of the OS installation programs boot off the network or CD into memory, perform a little OS specific operations to setup the disks properly, and then just copy "OS-applications" onto the disk. Some of the OS installation papers supported some limited number of additional packages, and recent OS installation programs [Hoh99] can install most of the packages available on the net. However, "OS" installation programs tend use a blessed central location for files, and copy the files onto the local disk, whereas "application" installation programs support merging of multiple disparate directories into a functionally single whole.

6.1.1 Packaging

Software packaging, the process of collecting together the related files for an application, appears to be a mostly solved problem. There have been a few papers in the LISA conference on it, and the freely available Unix systems have associated packaging tools. Comparing these tools might pave the way to a single multi-platform tool.

Packaging usually binds pathnames into an application. This can limit how packages can be merged later (e.g. two versions both believe they own /usr/lib/package). Some packages allow environment variables to override pathname choices. Exploring the performance and flexibility of the different choices could help improve existing tools.

6.1.2 Selection

Package selection, the process of identifying the packages that are going to be installed, is part of all OS/Application installation tools. In previous tools, package selection has been fairly ad-hoc. The key pieces for a selection tool are the need for per-machine flexibility and the need to support multiple collections. Both programmatic and GUI interfaces should be supported so that the tool is both easier to use and script. The selection tool could then be integrated into some of the existing tools as a uniform front-end.

6.1.3 Merging

Merging packages, the process of resolving inter-package conflicts, remains a hard outstanding problem. Many tools just ignore the problem. A few have a configuration file to specify which package overrides another when conflicts occur. Merging is most difficult when packages are inter-related, as is the

case with Emacs, Perl and Tcl with their various separate extensions; Tex/LaTeX; X windows with various applications that add fonts and include files; and shared library packages.

One unsatisfying solution is to pre-merge packages during packaging so that there are no inter-relations between packages. A modular solution would need to handle merging of files, for example generating the top level Emacs info file, or the X windows font directory files. Some programs include search paths, which might make the merging easier to handle, others require the execution of a program in the final merged directory.

If multiple versions need to be supported simultaneously, there is a more substantial problem. Supporting the cross product of all possibilities is not practical. However, there is no clear easy solution. Quite a bit of thought will be needed to find an adequate solution.

6.1.4 Caching

Caching to the local disk is beneficial for both performance and for isolating clients from server failures. Caching is a semi-solved problem. Some file systems cache onto local disk to improve performance (e.g. AFS, CacheFS, Coda). In general, caching merely requires mounting the global repository somewhere different and creating symlinks or copies as appropriate. There have been tools written to do just this [Cou96, Bel96], and many of the general software installation tools have included support for caching [Won93]. Making the caching fully automatic and fine grained will probably require some amount of OS integration.

6.1.5 End-user customization

End user customization, the process of setting user-default parameters for applications, has been mostly ignored. A few tools help users dynamically select the packages they want to use [FO96]; most have fixed the choice on a per-machine basis. One old paper looked at how users customized their environment [WCM93]. It would be nice for this area to be resurrected for research. Programs are becoming increasingly complex, especially as they add GUI interfaces, but the ease of customizing the programs has not kept up. Work in this area would require a large amount of interviewing users to determine what they would like to customize.

6.2 Backup

Restores seem to be a somewhat overlooked part of the backup problem. Most backup papers deal in great detail with formats of dump tapes, scheduling of backups, streaming to tape. However, they usually only write a few paragraphs on the subject of restores, often ignoring the time taken to restore data.

The whole purpose of backup is so that when something goes wrong, restores can happen! We would like a discussion of restore difficulty and measurements of restore performance in future papers. When something fails, there is a cost in lost productivity in addition to the direct cost of performing the repair.

Examining technology trends and technology options would help identify future backup challenges before they occur. The technology involved has reasonably predictable future performance in terms of bandwidth, latency, and capacity. Somewhat weaker predictions can be made about the growth in the storage needs of users. Given this information, a prediction can be made about the required ratio of hardware in the future. In addition, alternatives to tape backup such as high capacity disks and writable cds/dvds may become viable in the future. One advantage of random access media is that data can be directly accessed off the backup media to speed up recovery.

Backup by copying to remote sites is very different from traditional approaches. A few companies are dealing with the possibility of a site failure by performing on-line mirroring to a remote site over a fiber connection. It may be possible to decrease the required bandwidth by lowering the frequency of the updates, so that this approach is practical for people unable to purchase a dedicated fiber.

Backups also present special security concerns. A backup is typically an unprotected copy of data. If anyone can get access to backup storage, they can read critical data. How can encryption be used to solve the security problem? Will encryption enable safe web backup systems?

Another interesting question is how to handle backup for long-term storage. Some industries have legal requirements to retain documents for a long time. There are two related problems. First, media needs to be found which is stable enough to last a long time. Second, it seems wise to rely on conversion to a common format because it is never clear what software will still work in 20-50 years. How can these two concerns be integrated into a backup solution?

6.3 Configuration: site, host, network, site move

The key to host configuration is having a central repository of information that is then pushed or pulled by hosts. Most of the LISA papers did some variant of this. Two areas remain to be refined: First, someone should analyze exactly what information should be in the central repository, and how it can be converted to the many different types of hosts in use. Second, someone should write a tool to automatically create the repositories so that the start-up cost to using a configuration tool is lower.

Site configuration tools vary widely, probably because of the different requirements at each site (e.g. a wall street trading firm vs. a research lab). One paper [Eva97] surveyed the current practices, and another [TH98] studied the best practices for certain environments. Combining these two directions by identifying the best practices based on the requirements of a site would help all sites do a better job of configuration.

Network configuration is a fairly recent topic, so proposing directions by analyzing the papers is risky. However, we can still look at analogies to previous work. First, we want to build abstract descriptions of the system. Second, the models should be customizable; early configuration tools did not support much customization, so later ones had to add it. Third, a survey paper, analogous to [Eva97] would help identify the problems in network configuration research.

6.4 Accounts

Surveying account creation practices would help identify why no tool has evolved as superior despite many papers on this subject. We believe this is because of unrecognized differences in the requirements at each site. With all the requirements explicitly described, it should be possible to build a universal tool.

A related topic is the examination of specific issues related to account creation. For example, many of the papers ignored the question of how to limit accounts to specific machines. Is a simple grouping as was done for host configuration sufficient, or is some sort of export/import setup needed? Sharing accounts across administrative boundaries within an organization will make this problem even more difficult.

Another specific issue is delegation of account creation. The one tool to do this [Arn98] assumed all the employees were trusted to enter correct account information. Clearly this solution will not work at all sites. There may be synergy with the secure root access papers that looked at delegation.

6.5 Mail

The biggest remaining mail problem is dealing with SPAM. The correct solution is probably dependent on trading off difficulty in being reached legitimately with protection from SPAM. Some possible approaches are: acceptance lists with passwords, a list of abusers that are automatically ignored (this is being done), a pattern matcher for common SPAM forms, and receive-only/send-only addresses. Finding a good solution will be challenging.

Scalability and security still need some work. Scalability of mail transport and mail delivery may be possible by gluing together current tools into a clustered solution. Both problems partition easily. Handling more types of security threats also remains open. Some initial work [BRW99] has done some initial work securing communications between mail transfer agents without losing backwards compatibility.

6.6 Monitoring: system, network, host, data display

There has been a lot of work on gathering data from specific sources, but in most cases, the overhead for gathering data has been high, so the interval is usually set in minutes. Reducing this overhead

is important for allowing finer grain monitoring [ABD+97]. In addition, we would like to vary the gathering interval so that the overhead of fine-grain gathering is only incurred when the data would be used. In addition to just gathering the data, having a standard form for storing the data efficiently would be very useful. Combining these two issues should lead to a nice universal tool with pluggable gathering modules.

Data analysis and data reduction have not received nearly the attention they deserve. The data collection techniques are only useful if the data can be used to identify problems. But beyond averaging time-series data, very little automated analysis has been done. An examination of methods for automated analysis, for example, looking at machine learning techniques, could prove fruitful.

Data visualization has started to get some examination in the system administration field. There is a vast amount of literature on various forms of visualization in the scientific computing field. We believe that a survey of existing techniques would lead to tools that allow visualization in system administration to be both more effective and more scalable.

6.7 Printing

Printing research seems to be in fairly good shape. Scaling print systems, debugging problems, and selecting the right printer are still challenging. Perhaps printer selection could be done by property (e.g. color, two sided). Finally, the path for getting information from printers back to users has not been well examined. A notification tool to tell users the printer's status, such as print job finished or out of paper, would be useful. The notification tool might also help in debugging printing problems.

6.8 Trouble tickets

There seems to be a fair amount of overlap in the research on trouble tickets. Many of the tools were created from scratch, only occasionally building on the previous research. Examining the existing tools should identify the different requirements that have led to all these systems and to a more general tool.

A second direction to extend trouble ticket systems would be to build in a knowledge of the request handling process. The process of handling problem reports has been examined[Lim99], but no tools exist to support that process. Creating those tools would be valuable.

6.9 Secure root access

As was described in the tasks section, and at the beginning of this section, there are many other problems in security, we discuss here only the problems in this sub-part of security. The unfortunate effect of having the research on secure root access split between local and remote security is that neither handles all

the problems easily. The remote tools are more flexible, but harder to configure, and do not support logging well. The local tools have a more natural interface, but do not have as much power to provide partial access. Combining these two paths of research should lead to a more powerful and flexible tool.

A second direction to consider is toward providing finer-grain access control. [GWTB96] did this by securely intercepting system calls. Further work could lead to having something like capabilities in the OS, allowing very precise control over the access granted to partially-privileged users.

6.10 Future work on CARD

Clearly the most important future work for CARD is re-implementation. This will enable experimentation with the system to determine how well it works in practice. In general, the additional work described in section 6.6 on monitoring is a good direction for future work.

The most interesting direction for work on CARD is automatic derivation of dependencies. The idea here is to use either machine learning [AL88, Kea93, BHL91, KL88, Lit89] or association rule mining [AIS93, AS94] techniques to automatically determine dependencies. This approach requires having some monitored values that indicate if a system is up or down. Then, if we can show that any time component 1 is down, component 2 is also down, but not the reverse, then it is likely that component 2 depends on component 1. If we continue this process, then we can build a graph of dependencies. At that point, we can suppress errors which are caused by another problem. For example, once we know that all services stop when the router fails, we can report only that the router has failed, rather than all the services.

Another direction for future work is in appropriate display of monitoring data. We showed in the CARD system one approach to visualization, primarily using strip-charts with additional information provided using color and shade. DEVise [LRM96, Liv97] looked at generic visualization from SQL databases by transforming the SQL columns into graphical objects; something similar tuned for system administration might be useful. Another direction to look in is 3-D visualization. For example, for visualizing wide-area monitoring data, it might be good to project the data onto a world map.

6.11 Future work on River

River demonstrated that the automatic adaptation techniques worked well for data-warehousing applications because they work over large amounts of streaming data. Later work [AH00] provided better support for join operations. The next direction is using the ideas in River for transaction processing, which is characterized by lots of random index lookups and updates. It would seem that both the distributed queue and graduated declustering ideas could be used to support those operations better.

Future work on integrating River into more applications would also be useful. Some work has

been done [AD99] to add some more operators into the River framework. However, there has not been the work to extend River to a full application.

Handling contention between different River instances is also an open question. River would adapt around memory contention, but it might be better to partition the two instances among the available nodes, or reduce the amount of memory each node uses in order to get better overall performance.

6.12 Future work on Hippodrome

There are numerous future directions for Hippodrome. One direction for future work is the interaction between applications that are adapting (such as River), and the Hippodrome system. It is unclear whether the two systems will cooperate or compete.

New modeling techniques need to be developed to allow for both easier to create and more accurate models. Experiments should be run with both more complex applications, and on larger storage systems consisting of multiple arrays to verify that the Hippodrome results hold on these more complex problems.

We expect that we will need to run the migration step in parallel in order to keep the total convergence time down. However, we are not substantially concerned about migration time taking too long. First, during initial system configuration, the migration is not impacting on a real workload, so there is little concern about the migration time. Second, the bandwidth for copying data scales linearly with the number of disks and scales partly with areal density. This scaling with the number of disks is the reason that the PostMark benchmark takes about 40 minutes in the first iteration and the subsequent iterations drop down to 30 minutes. In the future, we expect the time to perform a migration may increase as areal density is increasing faster than disk bandwidth. As that trend continues, the solver will need to be modified to minimize the amount of data which is migrated.

Another direction for future work is in using Hippodrome to manage the ongoing evolution of a storage system. We know that in practice real systems are changing, and Hippodrome should be able to respond to these changes to keep the system appropriately provisioned at all times. Preliminary results, using synthetic workloads similar to those described here, are promising. We anticipate that as long as the workload does not change faster than the migration component of the loop can copy the data from one configuration to the next, the system can rapidly adjust to both increased and decreased load. Using Hippodrome for on-line storage management also opens interesting research questions in controlling and/or maintaining quality of service, during both normal operation and while migration is taking place.

Chapter 7

Conclusions

System administration is a new area for academic research. Although work has been done to improve individual system administration tasks, little overall research has been done, and much of the research on individual tasks has been poorly analyzed. We conclude here by providing a summary of what we have done, a short discussion of the three research approaches we explored, and some general observations about the themes that we have observed in our research

7.1 Summary

We have presented a general set of principles by which a solution to a particular system administration task can be analyzed. The principles help to identify where a particular solution has improved the job of system administrators, and where the solution has made their job more difficult. Figure 7.1 shows how each of our different research efforts apply to each of the principles.

In addition to a set of principles, we have identified and classified a great deal of the previous work in system administration. This classification allows researchers to compare their solutions with the relevant previous work. The classification also identifies some of the better prior research in each of the different task areas that system administrators have to handle.

Our first attempt at improving system administration addressed monitoring. We developed a system which was more scalable and flexible than the previous work. However, as we deployed the system and experimented with it, we found that it was less automated than some of the previous work, and as a result was not a complete solution to the problem. We proposed a number of changes to our system which preserve the scalability and flexibility improvements without resulting in decreased automation.

After working with the monitoring system, we discovered that we really wanted a system which eliminated some of the problems administrators faced rather than giving them more transparency into the problems. We therefore developed the Rivers system, which does localized load-balancing in order to hide

	CARD	CARD+	River	Hippodrome
Dependability	-1	0	+1	+1
Automation	0	0	0	+1
Scalability	+1	+1	+1	?
Flexibility	+1	+1	0	0
Notification	0	0	0	0
Schedulability	0	0	+1	0
Transparency	+1	+1	-1	0
Simplicity	-1	0	-1	-1

Figure 7.1: All of the research efforts versus the principles. CARD+ is CARD with the modifications that we proposed to eliminate some of its weaknesses. Hippodrome gets a ? for scalability because while it is likely that the approaches will scale it has not yet been demonstrated. Although a wider range than -1 to +1 could have been used, we do not believe that adds any information.

the performance anomalies that are common in clustered systems. We developed techniques to load-balance the work among a number of consumers and to balance the overall performance of reading from disk.

We found that Rivers was a good solution to eliminating short-term performance problems, but that it worked poorly for long-term variation, and it did not identify when more resources were needed. We therefore developed the Hippodrome system, which identifies when more resources are needed, and is partially able to handle long-term variation.

7.2 Research approaches

We have illustrated three research approaches to tackling system administration problems: “Let the human handle it,” “Rewrite everything,” and “Sneak in-between.” Each approach has different strengths and weaknesses. Since the approaches are complementary, selecting the right approach must be done based on the individual problem.

“Let the human handle it” dramatically simplifies problems. It changes the goal from trying to eliminate a problem to trying to help the administrator manage the problem. The researcher then needs to provide two related solutions, a monitoring solution and tools to adjust the system. The first part of the solution lets the administrator learn about the problem, the second part helps them fix it when it occurs. In the end, we claim that for the really hard problems, we will always require a human to examine the interactions that are causing difficulties, and eliminate or work around the problem. This approach can therefore serve as an excellent first step. It will be necessary in the end, the tools may be useful for eliminating some of the problems, and it helps the researcher understand which problems occur in practice. However, this approach does not always make the administrator’s life easier. In fact, it could exacerbate their stress by presenting too many problems. Therefore, we can see that this approach is necessary and useful, but not sufficient.

"Rewrite everything" provides amazing flexibility in building a system to eliminate a class of problems. Under this approach to research, all of the legacy code with it's limitations and weaknesses vanish. Researchers are free to explore radically different system structures. They can just pick a problem, and develop a system in which the administrator will never have to deal with that problem. This approach therefore provides the greatest chance of a dramatic leap forward, and can be an excellent way to deal with a tricky, difficult problem. However, this approach makes generating the complete system much more difficult. The researcher has less to build on, and the lack of features may make their resulting system less useful, making it harder to test. Also, given the substantial changes, it is more difficult to verify that they have not introduced new, un-recognized problems. Therefore, we can see that this approach holds great promise, but is hard to validate.

"Sneak in-between" uses indirection layers to extend an existing system. Under this approach to research, additional tools are added in to the existing system to ameliorate some of the problems of the original system. Researchers can then build on all of the work that has gone into building the existing system. They can pick a problem, search for the hooks necessary to eliminate the problem, and then adapt the existing software until the problem is removed. This approach therefore generates an easily tested system because it is complete, and has all of the power of the original one. However, this approach requires a layer of indirection. If the existing system does not have one, researchers are faced with the difficult task of adding one, or finding some other hook. Moreover, the layer of indirection that is present may substantially limit the possible solutions that can be explored. Therefore, we can see that this approach is much easier to validate, but is potentially limited.

7.3 Themes

While working on the various problems that we explored, we found a number of common themes. The first theme is "deployment of a system helps to evaluate its success." We have partially deployed all of the systems that we described. For the CARD system, we discovered that the difficulties in maintaining the monitoring system reduced it's value as a solution. For the Rivers system, we used it to support an undergraduate class, which taught us that even with the Rivers tools, it was difficult to build performance-robust applications, and that we had not yet demonstrated that full applications could be built in the Rivers way. For the Hippodrome system, we are still in progress of deploying it more widely. We have already learned from trying it on a complex Data Warehousing workload that realistic workloads present challenges that were not present in our initial simple workloads.

The second theme is that solutions to part of a system administration problem may increase the difficulties elsewhere. Although we believe the deploying a system will help identify where difficulties have increased, we understand that deployment is difficult. We therefore identified a series of principles by which

a system administration solution can be examined. These principles will help identify both where a solution assists system administrators and where it hurts them.

The third theme is that techniques from other areas of computer science can be applied to system administration problems. We used techniques from databases and distributed systems in our monitoring work. We extended techniques from databases and parallel programming in the Rivers work. We combined techniques of bin-packing, analytic device modeling, and greedy approximations to build the Hippodrome iterative loop.

A fourth theme is that the human matters. Enabling the administrator to do a better job was central to all of the problems we tackled. CARD enabled the administrator to get a better understanding of a cluster's current state. Rivers reduced the need for administrators to explain variability in performance of applications. Hippodrome removed administrators from the capacity planning task. In some ways, a primary difference between doing research in system administration and most other fields of computer science is that system administration operates on the boundary between people and computers.

Having to do research that can include human subjects may be one reason why traditional systems researchers have shied away from system administration problems. Although this prejudice may have been acceptable in the past, many of the 20th century systems problems are well solved. Working in commonly accepted areas may be comforting, but rather than polishing an already smooth stone, we suggest that 21st century researchers need to tackle new system administration problems.

Bibliography

[ABD+97] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandervoorde, Carl A. Waldspurger, and William E. Weihl. Continuous Profiling: Where Have All the Cycles Gone? In *16th ACM Symposium on Operating Systems Principles*, 1997. `http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/abstracts/src-tn-1997-016.html`.

[ACPtNt95] Tom Anderson, David Culler, Dave Patterson, and the NOW team. A Case for NOW (Networks of Workstations). In *IEEE Micro*, pages 54–64, February 1995.

[AD99] Remzi H. Arpaci-Dusseau. *Performance Availability for Networks of Workstations*. PhD thesis, U.C. Berkeley, 1999.

[ADADC+97] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-Performance Sorting on Networks of Workstations. In *SIGMOD '97*, pages 243–254, May 1997.

[ADADC+98] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. Searching for the Sorting Record: Experiences in Tuning NOW-Sort. In *Symposium on Parallel and Distributed Tools (SPDT '98)*, pages 124–133, Aug. 1998.

[ADAT+99] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, Dave Patterson, and Katherine Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS'99)*, pages 10–22, May 1999.

[AFF+01] Karen Appleby, Sameh Fakhouri, Liana Fong, Germán Goldszmidt, Michael Kalantar, Srirama Krishnakumar, Donald P. Pazel, John Pershing, and Benny Rochwerger. Océano – SLA based management of a computing utility. In *Integrated Network Management VII*, May 2001.

[AFZ97] Swarup Acharya, Michael Franklin, and Stan Zdonik. Balancing Push and Pull for Data Broadcast. In *ACM SIGMOD Intl. Conference on Management of Data*, May 1997.

[AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: continuously adaptive query processing. In *SIGMOD: International conference on management of data*, pages 261–272, 2000.

[AHH+01] Eric Anderson, Joe Hall, Jason D. Hartline, Michael Hobbs, Anna Karlin, Jared Saia, Ram Swaminathan, and John Wilkes. An Experimental Study of Data Migration Algorithms. In *Proceedings of the 5th Workshop on Algorithm Engineering (WAE2001)*, University of Aarhus, Denmark, August 2001.

[AHK+02] E. Anderson, M. Hobbs, K. Keeton, S. Spence, and M. Uysal an d A. Veitch. Hippodrome: running rings around storage administration. In *Proceedings of the USENIX Conference on File and Storage Technol ogies (FAST)*, January 2002.

[AIS93] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, Washington, D.C., 26–28 1993.

[AkcTW96] Joel Apisdorf, k claffy, Kevin Thompson, and Rick Wilder. OC3MON: Flexible, Affordable, High Performance Statistics Collection. In *Proceedings of the Tenth Systems Administration Conference (LISA '96), Chicago, Illinois*, pages 97–112, 1996. `http://www.nlanr.net/NA/Oc3mon/`.

[AKS+01] Eric Anderson, Mahesh Kallahalla, Susan Spence, Ram Swaminathan, and Qian Wang. Ergastulum: An approach to solving the workload and device configuration problem. Technical note, HPL-SSP-2001-5, HP Labs, July 2001.

[AL88] Dana Angluin and Philip Laird. Learning from noisy examples. *Machine Learning*, 2(4):343–370, 1988.

[And95] Eric Anderson. Results of the 1995 SANS Survey. In *;login: October 1995, Vol20, No. 5*, 1995. `http://now.cs.berkeley.edu/Sysadmin/SANS95-Survey/index.html`.

[And01] Eric Anderson. Simple table-based modeling of storage devices. Technical note, HPL-SSP-2001-4, HP Labs, July 2001. http://www.hpl.hp.com/research/itc/scl/ssp/papers/.

[AP97] Eric Anderson and Dave Patterson. Extensible, Scalable Monitoring for Clusters of Computers. In *Proceedings of the Eleventh Systems Adminstration Conference (LISA '97), San Diego, California*, pages 9–16, 1997. `http://now.cs.berkeley.edu/Sysadmin/esm/intro.html`.

[Arn98] Bob Arnold. Accountworks: Users Create Accounts on SQL, Notes, NT, and UNIX. In *Proceedings of the Twelfth Systems Administration Conference (LISA '98), Boston, Massachusetts*, pages 49–61, 1998.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, 12–15 1994.

[Asa00] Satoshi Asami. *Reducing the Cost of System Administration of a Disk Storage System Built from Commodity Components*. PhD thesis, U.C. Berkeley, 2000.

[Bal98] Carrie Ballinger. Teradata database design 101: a primer on teradata physical database design and its advantages. Technical note, NCR/Teradata, May 1998.

[BBD+94] Brian Bershad, David Black, David DeWitt, Garth Gibson, Kai Li, Larry Peterson, and Marc Snir. Operating system support for high-performance parallel I/O systems. Technical Report CCSF-40, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.

[BBGS94] Tom Barclay, Robert Barnes, Jim Gray, and Prakash Sundaresan. Loading Databases Using Dataflow Parallelism. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(4):72–83, December 1994.

[BC91] Kenneth P. Birman and Robert Cooper. The isis project: Real experience with a fault tolerant programming system. *ACM Operating Systems Review, SIGOPS*, 25(2):103–107, 1991.

[BCF+95] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, and Charles L. Seitz. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[Bel96] John D. Bell. A Simple Caching File System for Application Serving. In *Proceedings of the Tenth Systems Administration Conference (LISA '96), Chicago, Illinois*, pages 171–179, 1996.

[BGJ+98] Elizabeth Borowsky, Richard Golding, Patricia Jacobson, Arif Merchant, Louis Schreier, Mirjana Spasojevic, and John Wilkes. Capacity planning with phased workloads. In *Proceedings of the First Workshop on Software and Performance (WOSP'98)*, pages 199–207, Oct 1998.

[BHL91] A. Blum, L. Hellerstein, and N. Littlestone. Learning in the presence of finitely or infinitely many irrelevant attributes. In *Proceedings of the Fourth Annual Workshop on Computational Learning Theory*, pages 157–166, Santa Cruz, California, August 1991. Morgan Kaufmann.

[BJK+95] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, July 1995.

[BKT92] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, Mar. 1992.

[BR98] Thomas Bushnell and Karl Ramm. Anatomy of an Athena Workstation, 1998.

[BRW99] Damien Bentley, Greg Rose, and Tara Whalen. ssmail: Opportunistic Encryption in sendmail. In *Proceedings of the Thirteenth Systems Administration Conference (LISA '99), Seattle, Washington*, pages 1–7, 1999.

[Bur00] Mark Burgess. *Principles of Network and System Administration*. John Wiley & Sons, 2000.

[CABK88] George Copeland, William Alexander, Ellen Boughter, and Tom Keller. Data Placement in Bubba. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 17(3):99–108, Sept. 1988.

[CAE+76] Donald D. Chamberlin, Morton M. Astrahan, Kapali P. Eswaran, Patricia P. Griffiths, Raymond A. Lorie, James W. Mehl, Phyllis Reisner, and Bradford W. Wade. SEQUEL 2: A unified approach to data definition, manipulation, and control. *IBM Journal of Research and Development*, 20(6):560–575, 1976. (also see errata in Jan. 1977 issue).

[Car87] Nicholas J. Carriero. *Implementation of tuple space*. PhD thesis, Department of Computer Science, Yale University, December 1987.

[CBH+94] Alok Choudhary, Rajesh Bordawekar, Michael Harry, Rakesh Krishnaiyer, Ravi Ponnusamy, Tarvinder Singh, and Rjeev Thakur. PASSION: parallel and scalable software for input-output. Technical Report SCCS-636, ECE Dept., NPAC and CASE Center, Syracuse University, September 1994.

[CDG+93] David E. Culler, Andrea Dusseau, Seth C. Goldstein, Arvind Krishnamurthy, Steve Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, 1993.

[CDI+95] Soumen Chakrabarti, Etienne Deprit, Eun-Jin Im, Jeff Jones, Arvind Krishnamurthy, Chih-Po Wen, and Katherine Yelick. Multipol: A Distributed Data Structure Library. Technical Report CSD-95-879, University of California, Berkeley, July 1995.

[CF96] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.

[CFSD90] Jeffrey D. Case, Mark Fedor, Martin Lee Schoffstall, and James R. Davin. A Simple Network Management Protocol (SNMP), 1990. Available as RFC 1157 from `http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc1157.html`.

[Cha92] D. Brent Chapman. Majordomo: How I Manage 17 Mailing Lists Without Answering =requestM̈ail. In *Proceedings of the Sixth Systems Administration Conference (LISA '92), Long Beach, California*, pages 135–143, 1992. `ftp://ftp.greatcircle.com/pub/majordomo.tar.Z`.

[Cis] Cisco. Cisco career certifications. `http://www.cisco.com/warp/public/10/wwtraining/certprog/`.

[Cis00] Cisco Systems Inc. Load Balancing: A Solution for Improving Server Availability, June 2000. White paper on LocalDirector.

[CKCS90] David A. Curry, Samuel D. Kimery, Kent C. De La Croix, and Jeffrey R. Schwab. ACMAINT: An Account Creation and Maintenance System for Distributed UNIX Systems. In *Proceedings of the Fourth Large Installation Systems Administrator's Conference (LISA '90), Colorado*, pages 1–9, 1990.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. In *ACM Transactions on Computer Systems*, volume 3, pages 63–75, February 1985.

[CL91] Luis-Felipe Cabrera and Darrell D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computing Systems*, 4(4):405–436, Fall 1991.

[CMRW96] Jeffrey D. Case, Keith McCloghrie, Marshall T. Rose, and Steven Waldbusser. Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2), 1996. Available as RFC 1907 from `http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc1907.html`.

[CN97] Surajit Chaudhuri and Vivek Narasayya. An efficient, cost-driven index selection tool for Microsoft SQL Server. In *Proceedings of the 23rd VLDB Conference*, pages 146–55, Athens, Greece, September 1997.

[Cod70] Edgar F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[Cod71] Edgar F. Codd. A Database Sublanguage Founded on the Relational Calculus. In *Proceedings of the 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control*, pages 35–68, Nov 1971.

[Coua] Alva Couch. Network administration class. `http://www.cs.tufts.edu/ couch/teaching.html`.

[Coub] Transaction Processing Council. TPC-D Individual Results, 1998. http://www.tpc.org/results/tpc_d.results.page.html.

[Cou96] Alva L. Couch. SLINK: Simple, Effective Filesystem Maintenance Abstractions for Community-Based Administration. In *Proceedings of the Tenth Systems Administration Conference (LISA '96), Chicago, Illinois*, pages 205–212, 1996. `ftp://ftp.cs.tufts.edu/pub/slink`.

[CW92] Wallace Colyer and Walter Wong. Depot: A Tool for Managing Software Environments. In *Proceedings of the Sixth Systems Administration Conference (LISA '92), Long Beach, California*, pages 153–162, 1992. `ftp://export.acs.cmu.edu/pub/depot/`.

[DC90] Stephen E. Deering and David R. Cheriton. Multicast routing in datagram internetworks and extended LANs. In *ACM Transactions on Computer Systems*, pages 85–110, May 1990.

[Det91] John F. Detke. Host Aliases and Symbolic Links -or- How to Hide the Servers' Real Name. In *Proceedings of the Fifth Large Installation Systems Administration Conference (LISA '91), San Diego, California*, pages 249–252, 1991.

[DF82] Lawrence W. Dowdy and Derrel V. Foster. Comparative models of the file assignment problem. *ACM Computing Surveys*, 14(2):287–313, June 1982.

[DG92] David DeWitt and Jim Gray. Parallel database systems: The future of high-performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[DGS88] David J. DeWitt, Shahram Ghandeharizadeh, and Donovan Schneider. A Performance Analysis of the Gamma Database Machine. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 17(3):350–360, Sept. 1988.

[dlVL81] Wenceslas Fernandez de la Vega and George S. Lueker. Bin packing can be solved within $1+\varepsilon$ in linear time. *Combinatorica*, 1(4):349–55, 1981.

[Dol96] HP Dolphin research project, 1996. Personal communication with author and some of the development group.

[dSlG93] James da Silva and lafur Gumundsson. The Amanda Network Backup Manager. In *Proceedings of the Seventh Systems Administration Conference (LISA '93), Monterey, California*, pages 171–182, 1993. `ftp://ftp.cs.umd.edu/pub/amanda`.

[EL92] Richard Elling and Matthew Long. user-setup: A System for Custom Configuration of User Environments, or Helping Users Help Themselves. In *Proceedings of the Sixth Systems Administration Conference (LISA '92), Long Beach, California*, pages 215–223, 1992. `ftp://ftp.eng.auburn.edu/`.

[EMC00] EMC Corporation. *EMC ControlCenter Product Description Guide*, 2000. Pub. No. 01748-9103.

[Eva97] Rmy Evard. An Analysis of UNIX System Configuration. In *Proceedings of the Eleventh Systems Adminstration Conference (LISA '97), San Diego, California*, pages 179–193, 1997.

[Ext] U.C. Extension. UNIX System Administration Certificate Program. `http://www.unex.berkeley.edu/cert/unix.html`.

[Fin97] Raphael A. Finkel. Pulsar: An extensible tool for monitoring large Unix sites. *Software Practice and Experience*, 10(27):1163–1176, 1997.

[FKK96] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL Protocol: Version 3.0, 1996. Internet draft available as `http://home.netscape.com/eng/ssl3/ssl-toc.html`.

[Fle92] Mark Fletcher. nlp: A Network Printing Tool. In *Proceedings of the Sixth Systems Administration Conference (LISA '92), Long Beach, California*, pages 245–256, 1992.

[FO96] John L. Furlani and Peter W. Osel. Abstract Yourself With Modules. In *Proceedings of the Tenth Systems Administration Conference (LISA '96), Chicago, Illinois*, pages 193–203, 1996. `http://www.modules.org/`.

[FS89] Raphael Finkel and Brian Sturgill. Tools for System Administration in a Heterogeneous Environment. In *Proceedings of the Workshop on Large Installation Systems Administration III (LISA '89), Austin, Texas*, pages 15–29, 1989.

[GCCC85] David Gelernter, Nicholas Carriero, Sharat Chandran, and Silva Chang. Parallel programming in Linda. In *1985 International Conference on Parallel Processing*, pages 255–263, 1985.

[GHN90] Tinsley Galyean, Trent Hein, and Evi Nemeth. Trouble-MH: A Work-Queue Management Package for a >3 Ring Circus. In *Proceedings of the Fourth Large Installation Systems Administrator's Conference (LISA '90), Colorado*, pages 93–95, 1990.

[GJ79] Michael R. Garey and David S. Johnson. *Computing and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

[GLP75] Jim Gray, Raymond A. Lorie, and Gianfranco R. Putzulo. Granularity of locks and degrees of consistency in a shared database. In *1st International Conference on VLDB*, pages 428–431, September 1975. Reprinted in Readings in Database Systems, 3rd edition.

[Gra90] Goetz Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 19(2):102–111, June 1990.

[Gra97] Jim Gray. What Happens When Processors Are Infinitely Fast And Storage Is Free?, November 1997. Invited Talk: 1997 IOPADS.

[Gro97] Gartner Group. A white paper on gartnergroup's next generation total cost of ownership methodology, 1997.

[Gro01] Hurwitz Group. Trends in e-business outsourcing and the rise of the managed hosting model. White paper, www.exodus.com, January 2001.

[GS93] Al Geist and Vaidy Sunderam. The Evolution of the PVM Concurrent Computing System. In *COMPCON*, pages 549–557, February 1993.

[GSC96] Seth Copen Goldstein, Klaus Erik Schauser, and David E. Culler. Lazy Threads: Implementing a Fast Parallel Call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, Aug. 1996.

[Gui] System Administrators Guild. General reference books. `http://www.usenix.org/sage/sysadmins/books/`

[Gui02] System Administrators Guild. Sage certification study guides, 2002. `http://www.sagecert.org/html/use.php?view=gen&story=study`.

[GWS91] Peter Zabback Gerhard Weikum and Peter Scheuermann. Dynamic File Allocation in Disk Arrays. In *Proceedings of the 1991 SIGMOD Conference*, pages 406–415, 1991.

[GWTB96] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications: confining the wily hacker. In *Sixth USENIX Security Symposium, Focusing on Applications of Cryptography*, 1996. `http://www.cs.berkeley.edu/ daw/janus/`.

[HA93] Stephen E. Hansen and E. Todd Atkins. Automated System Monitoring and Notification With Swatch. In *Proceedings of the 1993 LISA VII Conference*, pages 145–155, 1993.

[Hal99] Geoff Halprin. The system administrator's body of knowledge, 1999. Presentation at LISA 1999, summarized at `http://www.usenix.org/publications/library/proceedings/lisa99/summaries/summaries.html`.

[Har97] Robert Harker. Selectively Rejecting SPAM Using Sendmail. In *Proceedings of the Eleventh Systems Adminstration Conference (LISA '97), San Diego, California*, pages 205–220, 1997. `http://www.harker.com/sendmail/anti-spam`.

[HD90] Hui-I Hsiao and David J. DeWitt. Chained Declustering: A new availability strategy for multiprocessor database machines. In *Proceedings of 6th International Data Engineering Conference*, pages 456–465, 1990.

[HER+95] James V. Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A high performance portable parallel file system. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, Barcelona, July 1995. ACM Press.

[Hew00a] Hewlett-Packard Company. *HP SureStore E Auto LUN XP User's guide*, August 2000. Pub. No. B9340-90900.

[Hew00b] Hewlett-Packard Company. *HP SureStore E Disk Array FC60 - Advanced User's Guide*, December 2000.

[Hid94] Imazu Hideyo. OMNICONF Making OS Upgrads and Disk Crash Recovery Easier. In *Proceedings of the Eighth Systems Administration Conference (LISA '94), San Diego, California*, pages 27–31, 1994.

[Hil96] Brian C. Hill. Priv: Secure and Flexible Privileged Access Dissemination. In *Proceedings of the Tenth Systems Administration Conference (LISA '96), Chicago, Illinois*, pages 1–8, 1996. `ftp://ftp.ucdavis.edu/pub/unix/priv.tar.gz`.

[HM92] Darren R. Hardy and Herb M. Morreale. buzzerd: Automated Systems Monitoring with Notification in a Network Environment. In *Proceedings of the Sixth Systems Administration Conference (LISA '92), Long Beach, California*, pages 203–210, 1992.

[Hoh99] Dirk Hohndel. Automated installation of Linux systems using YaST. In *Proceedings of the Thirteenth Systems Administration Conference (LISA '99), Seattle, Washington*, pages 261–266, 1999.

[HSV] Hue, Saturation, and Value Color Model. `http://loki.cs.gsu.edu/edcom/hypgraph/color/colorhs.htm`.

[JDU+74] David S. Johnson, Alan J. Demers, Jeffrey D. Ullman, Michael R. Garey, and Ron L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing, Springer Verlag (Heidelberg, FRG and NewYork NY, USA)-Verlag*, 3:299–325, 1974.

[Kat97] Jeffrey Katcher. Postmark: a new file system benchmark. Technical report TR-3022, Network Appliances, Oct 1997.

[Kea93] Michael Kearns. Efficient Noise-Tolerant Learning From Statistical Queries. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, pages 392–401, May 1993.

[Ken96] Claire Kenyon. Best-fit bin-packing with random order. In *SODA: ACM-SIAM Symposium on Discrete Algorithms*, 1996.

[KL86] Brian Kantor and Phil Lapsley. Network News Transfer Protocol, 1986. Available as RFC 977 from `http://jakarta.apache.org/james/rfclist/rfc977.txt`.

[KL88] Michael Kearns and Ming Li. Learning in the presence of malicious errors. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, pages 267–280, Chicago, Illinois, May 1988.

[Kob92] David Koblas. PITS: A Request Management System. In *Proceedings of the Sixth Systems Administration Conference (LISA '92), Long Beach, California*, pages 197–202, 1992.

[Kol97] Rob Kolstad. Tuning Sendmail for Large Mailing Lists. In *Proceedings of the Eleventh Systems Adminstration Conference (LISA '97), San Diego, California*, pages 195–203, 1997.

[Kot94] David Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proceedings of the 1994 Symposium on Operating Systems Design and Implementation*, pages 61–74. USENIX Association, November 1994.

[KRM98] Steve Kubica, Tom Robey, and Chris Moorman. Data parallel programming with the Khoros Data Services Library. *Lecture Notes in Computer Science*, 1388:963–973, 1998.

[KVE+92] Steve Kleiman, Jim Voll, Joe Eykholt, Anil Shivalingiah, Dock Williams, Mark Smith, Steve Barton, and Glenn Skinner. Symmetric Multiprocessing in Solaris 2.0. In *Proceedings of COMPCON Spring '92*, page 181, 1992.

[Lim99] Thomas A. Limoncelli. Deconstructing User Requests and the 9-Step Model. In *Proceedings of the Thirteenth Systems Administration Conference (LISA '99), Seattle, Washington*, pages 35–44, 1999.

[LIN+93] Susan J. LoVerso, Marshall Isman, Andy Nanopoulos, William Nesheim, Ewan D. Milne, and Richard Wheeler. *sfs*: A parallel file system for the CM-5. In *Proceedings of the 1993 Summer USENIX Technical Conference*, pages 291–305, 1993.

[Lin98] Bruce Lindsey. SMP Intra-Query Parallelism in DB2 UDB, February 1998. Database Seminar at U.C. Berkeley.

[Lit89] Nick Littlestone. *Mistake bounds and logarithmic linear-threshold learning algorithms*. PhD thesis, U. C. Santa Cruz, March 1989.

[Liv97] Miron Livny. DEVise: an Environment for Data Exploration and Visualization, 1997. `http://www.cs.wisc.edu/ devise/`.

[LRM96] Miron Livny, Raghu Ramakrishnan, and Jussi Myllymaki. Visual Exploration of Large Data Sets. In *In Proceedings of the IS&T/SPIE Conference on Visual Data Exploration and Analysis*, January 1996.

[LRNL97] Tom Limoncelli, Tom Reingold, Ravi Narayan, and Ralph Loura. Creating a network for lucent bell labs research south. In *Proceedings of the Eleventh Systems Adminstration Conference (LISA '97), San Diego, California*, pages 123–140, 1997.

[Mal] Joseph Malcolm. personal communication.

[Mar97] Marimba Castanet, 1997. `http://www.marimba.com/`.

[MC96] Alan Mainwaring and David Culler. Active Message Applications Programming Interface and Communication Subsystem Organization. Technical Report CSD-96-918, University of California at Berkeley, October 1996.

[Met97] Rodney Van Meter. Observing the Effects of Multi-Zone Disks. In *Proceedings of the 1997 USENIX Conference*, pages 19–30, Jan. 1997.

[MFM95] Dan Mosedale, William Foss, and Rob McCool. Administering very high volume internet services. In *LISA IX: System administration conference*, pages 95–102, 1995.

[MHN+] Todd Miller, Dave Hieb, Jeff Nieusma, Garth Snyder, and et. al. Sudo: a utility to allow restricted root access. `http://www.courtesan.com/sudo/`.

[Mica] Microsoft. Reducing Total Cost of Ownership. `http://www.microsoft.com/mspress/business/TCO/`.

[Micb] Sun Microsystems. Server consolidation to reduce TCO. `http://www.sun.com/SunJournal/v3n1/IndustryTrends2.html`.

[Mil95] David Mills. Improved Algorithms for Synchronizing Computer Network Clocks. *IEEE/ACM Transactions on Networking*, 3(3), June 1995.

[Min97] Mini SQL 2.0, 1997. `http://hughes.com.au/`.

[MK92] Melissa Metz and Howie Kaye. DeeJay The Dump Jockey: A Heterogeneous Network Backup System. In *Proceedings of the Sixth Systems Administration Conference (LISA '92), Long Beach, California*, pages 115–125, 1992. `ftp://ftp.cc.columbia.edu/`.

[MRC+97] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas 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)*, pages 238–251, Saint-Malo, France, October 1997. ACM SIGOPS.

[Mur84] Gerald M. Murch. Physiological Principles for the Effective Use of Color. In *IEEE Computer Graphics and Applications*, pages 49–54, Nov. 1984.

[MWCR90] Kenneth Manheimer, Barry A. Warsaw, Stephen N. Clark, and Walter Rowe. The Depot: A Framework for Sharing Software Installation Across Organizational and UNIX Platform Boundaries. In *Proceedings of the Fourth Large Installation Systems Administrator's Conference (LISA '90), Colorado*, pages 37–46, 1990.

[Nem] Evi Nemeth. Unix system administration workshop. `http://www.cs.colorado.edu/ evi/Home.html`.

[Nit92] Bill Nitzberg. Performance of the iPSC/860 Concurrent File System. Technical Report RND-92-020, NAS Systems Division, NASA Ames, December 1992.

[NK96] Nils Nieuwejaar and David Kotz. The Galley parallel file system. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 374–381, Philadelphia, PA, May 1996. ACM Press.

[NSSH01] Evi Nemeth, Garth Snyder, Scott Seebass, and Trent R. Hein. *Unix System Administration Handbook, 3rd Ed.* Prentice Hall, 2001.

[Oet98] Tobias Oetiker. MRTG The Multi Router Traffic Grapher. In *Proceedings of the Twelfth Systems Administration Conference (LISA '98), Boston, Massachusetts*, pages 141–147, 1998. `http://ee-staff.ethz.ch/ oetiker/webtools/mrtg/3.0/`.

[PAB+98] Vivek S. Pai, Mohit Aron, Gaurav Banga, Michael Svendsen, Peter Druschel, Willy Zwaenepoel, and Erich M. Nahum. Locality-Aware Request Distribution in Cluster-based Network Servers. In *Architectural Support for Programming Languages and Operating Systems*, pages 205–216, 1998.

[Paca] Hewlett Packard. Hp openview. http://www.hp.com/openview/index.html.

[Pacb] Hewlett Packard. Web page discussing: return on investment. `http://www.hp.com/products1/unix/management/tco/ roi/infolibrary/roi.html`.

[PLC95] Scott Pakin, Mario Lauria, and Andrew Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of the 1995 ACM/IEEE Supercomputing Conference*, pages 1528–1557, San Diego Convention Center, San Diego, CA, December 1995. ACM Press and IEEE Computer Society Press.

[PM95] Patrick Powell and Justin Mason. LPRng An Enhanced Printer Spooler System. In *Proceedings of the Ninth Systems Administration Conference (LISA '95), Monterey, California*, pages 13–24, 1995.

[PM00] William G. Pope and Lily Mummert. Using capacity space methodology for balancing server utilization: description and case studies. Research report RC 21828, IBM T.J. Watson Research Center, 2000.

[Poi97] PointCast: the desktop newscast, 1997. `http://www.pointcast.com/`.

[Pre98] W. Curtis Preston. Using Gigabit Ethernet to Backup Six Terabytes. In *Proceedings of the Twelfth Systems Administration Conference (LISA '98), Boston, Massachusetts*, pages 87–95, 1998.

[RG95] Karl Ramm and Michael Grubb. Exu A System for Secure Delegation of Authority on an Insecure Network. In *Proceedings of the Ninth Systems Administration Conference (LISA '95), Monterey, California*, pages 89–93, 1995. `ftp://ftp.duke.edu/pub/exu`.

[RL91] Kenneth Rich and Scott Leadley. hobgoblin: A File and Directory Auditor. In *Proceedings of the Fifth Large Installation Systems Administration Conference (LISA '91), San Diego, California*, pages 199–207, 1991. `ftp://cc.rochester.edu/ftp/pub/ucc-src/hobgoblin`.

[RM94] John P. Rouillard and Richard B. Martin. Config: A Mechanism for Installing and Tracking System Configurations. In *Proceedings of the Eighth Systems Administration Conference (LISA '94), San Diego, California*, pages 9–17, 1994. `ftp://ftp.cs.umb.edu/pub/bblisa/talks/config/config.tar.Z`.

[Rue96] Craig Ruefenacht. RUST: Managing Problem Reports and To-Do Lists. In *Proceedings of the Tenth Systems Administration Conference (LISA '96), Chicago, Illinois*, pages 81–89, 1996. `ftp://ftp.cs.utah.edu/pub/rust`.

[SA95] Jeff Sedayao and Kotaro Akita. LACHESIS: A Tool for Benchmarking Internet Service Providers. In *Proceedings of the 1995 LISA IX Conference*, pages 111–115, 1995.

[SAG] SAGE. The System Administrators Guild. `http://www.sage.org/`.

[SAN] SANS. SANS 2000 Salary Survey Summary. `http://www.sans.org/newlook/publications/salary200`

[SB93] Gary Schaps and Peter Bishop. A Practical Approach to NFS Response Time Monitoring. In *Proceedings of the 1993 LISA VII Conference*, pages 165–169, 1993.

[Sch93] John Schimmel. A Case Study on Moves and Mergers. In *Proceedings of the Seventh Systems Administration Conference (LISA '93), Monterey, California*, pages 93–98, 1993.

[Sch97] Jürgen Schönwälder. Scotty Tnm Tcl Extension., 1997. `http://wwwsnmp.cs.utwente.nl/ schoenw/scotty/`.

[SCN+93] Michael Stonebraker, Jolly Chen, Nobuko Nathan, Caroline Paxson, and Jiang Wu. Tioga:providing data management support for scientific visualization applications. In *International Conference On Very Large Data Bases (VLDB '93)*, pages 25–38, San Francisco, CA, Aug. 1993. Morgan Kaufmann Publishers, Inc.

[Sco97] Peter Scott. Automating 24x7 Support Response To Telephone Requests. In *Proceedings of the Eleventh Systems Adminstration Conference (LISA '97), San Diego, California*, pages 27–35, 1997.

[SGI] SGI. Performance co-pilot. http://www.sgi.com/software/co-pilot/.

[Sim91] John Simonson. System Resource Accounting on UNIX Systems. In *Proceedings of the 1991 LISA V Conference*, pages 61–71, 1991.

[SL93] J. Schönwälder and H Langendörfer. How to Keep Track of Your Network Configuration. In *Proceedings of the 1993 LISA VII Conference*, pages 189–193, 1993.

[SMH95] Michael E. Shaddock, Michael C. Mitchell, and Helen E. Harrison. How to Upgrade 1500 Workstations on Saturday, and Still Have Time to Mow the Yard on Sunday. In *Proceedings of the Ninth Systems Administration Conference (LISA '95), Monterey, California*, pages 59–65, 1995.

[SNM] Sun Net Manager. Sun Solstice product.

[Spe96] Henry Spencer. Shuse: Multi-Host Account Administration. In *Proceedings of the Tenth Systems Administration Conference (LISA '96), Chicago, Illinois*, pages 25–32, 1996.

[SRC84] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems*, pages 277–288, November 1984.

[SS97] Margo I. Seltzer and Christopher Small. Self-Monitoring and Self-Adapting Systems. In *Proceedings of the 1997 Workshop on Hot Topics on Operating Systems*, pages 124–129, Chatham, MA, May 1997.

[SSU91] Avi Silberschatz, Michael Stonebraker, and Jeffrey D. Ullman. Database Systems: Achivements and Opportunities. *Communications of the ACM*, 34(10):110–120, 1991.

[SSU96] Avi Silberschatz, Michael Stonebraker, and Jeffrey D. Ullman. Database Research: Achievements and Opportunities into the 21st Century. Technical Report CS-TR-96-1563, Stanford Technical Report, 1996. `http://elib.stanford.edu/`.

[Sta98] Carl Staelin. mkpkg: A software packaging tool. In *Proceedings of the Twelfth Systems Administration Conference (LISA '98), Boston, Massachusetts*, pages 243–252, 1998. `http://www.hpl.hp.com/personal/Carl_Staelin/mkpkg`.

[Sun86] Remote Procedure Call Programming Guide, Feb 1986. Sun Microsystems, Inc.

[SW91] Carl Shipley and Chingyow Wang. Monitoring Activity on a Large Unix Network with perl and Syslogd. In *Proceedings of the 1991 LISA V Conference*, pages 209–215, 1991.

[SWZ98] Peter Scheuermann, Gerhard Weikum, and Peter Zabback. Data partitioning and load balancing in parallel disk systems. *VLDB Journal: Very Large Data Bases*, 7(1):48–66, 1998.

[TH98] Steve Traugott and Joel Huddleston. Bootstrapping an Infrastructure. In *Proceedings of the Twelfth Systems Administration Conference (LISA '98), Boston, Massachusetts*, pages 181–196, 1998.

[The93] The MPI Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing '93*, pages 878–883, November 1993.

[Tro96] Jim Trocki. PC Administration Tools: Using Linux to Manage Personal Computers. In *Proceedings of the Tenth Systems Administration Conference (LISA '96), Chicago, Illinois*, pages 187–192, 1996.

[UAM01] Mustafa Uysal, Guillermo A. Alvarez, and Arif Merchant. A Modular, Analytical Throughput Model for Modern Disk Arrays. In *To appear in Proceedings of the 9th International Symposium on Modeling, Analysis and Simulation on Computer and Telecommunications Systems (MASCOTS 2001)*, Cincinnati, OH, August 2001.

[vEBBV95] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 40–53, December 1995.

[vECGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Eric Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, May 1992. ACM SIGARCH and IEEE Computer Society TCCA.

[Ver00] Veritas Software Cororation. *Veritas Volume Manager Data Sheet*, July 2000. Pub. No. 90-00333-399.

[vRHB94] Robbert van Renesse, Takako Hickey, and Kenneth Birman. Design and performance of horus: a lightweight group communication system. Technical Report 94-1442, Department of Computer Science, Cornell University, 1994.

[VW99] Peter Valian and Todd K. Watson. NetReg: An Automated DHCP Network Registration System. In *Proceedings of the Thirteenth Systems Administration Conference (LISA '99), Seattle, Washington*, pages 139–147, 1999.

[Wal95] Rex Walters. Tracking Hardware Configurations in a Heterogeneous Network with syslogd. In *Proceedings of the 1995 LISA IX Conference*, pages 241–246, 1995.

[WCM93] Craig E. Wills, Kirstin Cadwell, and William Marrs. Customization in a UNIX Computing Environment. In *Proceedings of the Seventh Systems Administration Conference (LISA '93), Monterey, California*, pages 43–49, 1993.

[WGSS96] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[Wol89] Joel Wolf. The placement optimization program: a practical solution to the disk file assignment problem. In *Proceedings of the ACM SIGMETRICS Conference*, pages 1–10, May 1989.

[Won93] Walter C. Wong. Local Disk Depot Customizing the Software Environment. In *Proceedings of the Seventh Systems Administration Conference (LISA '93), Monterey, California*, pages 51–55, 1993. `ftp://export.acs.cmu.edu/pub/depot`.

[Woo98] Ben Woodard. Building An Enterprise Printing System. In *Proceedings of the Twelfth Systems Administration Conference (LISA '98), Boston, Massachusetts*, pages 219–228, 1998. `http://pasta.penguincomputing.com/pub/prtools`.

[ZSD90] Elizabeth D. Zwicky, Steve Simmons, and Ron Dalton. Policy as a System Administration Tool. In *Proceedings of the Fourth Large Installation Systems Administrator's Conference (LISA '90), Colorado*, pages 115–123, 1990.

[Zwi91] Elizabeth D. Zwicky. Torture-testing Backup and Archive Programs: Things You Ought to Know But Probably Would Rather Not. In *Proceedings of the Fifth Large Installation Systems Administration Conference (LISA '91), San Diego, California*, pages 181–189, 1991.

[Zwi92] Elizabeth D. Zwicky. Typecast: Beyond Cloned Hosts. In *Proceedings of the Sixth Systems Administration Conference (LISA '92), Long Beach, California*, pages 73–78, 1992. `ftp://ftp.erg.sri.com/pub/packages/typecast`.