Hippodrome: running circles around storage administration

Eric Anderson, Michael Hobbs, Kimberly Keeton, Susan Spence, Mustafa Uysal, Alistair Veitch
Storage and Content Distribution Department
Hewlett-Packard Laboratories
1501 Page Mill Road, Palo Alto, CA 94304
{anderse,mjhobbs,kkeeton,suspence,uysal,aveitch}@hpl.hp.com

Abstract

Storage system configuration, even at the enterprise scale, is traditionally undertaken by human experts using a time-consuming process of trial and error, guided by simple rules of thumb. Due to the complexity of the design process and lack of workload information, the resulting systems often cost significantly more than necessary, or fail to perform adequately.

Our solution to this problem is to automate the design and configuration process using a tool we call Hippodrome. It can explore the design space more thoroughly than humans, and implement the design automatically, thereby eliminating many tedious, error-prone operations.

Hippodrome is structured as an iterative loop: it analyzes a workload to determine its requirements, creates a new storage system design to better meet these requirements, migrates the existing system to the new design. It repeats the loop until it finds a storage system design that satisfies the workload's I/O requirements. This paper describes the Hippodrome loop and demonstrates that our prototype implementation converges rapidly to appropriate system designs.

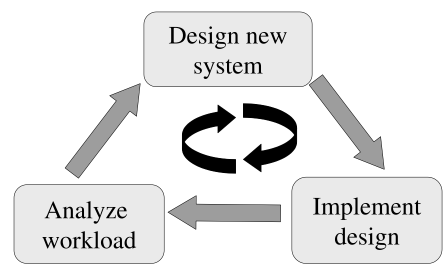


Figure 1: The stages of the iterative storage management loop. The loop can be bootstrapped using capacity information and optional performance estimates as input to the design stage.

1 Introduction

Enterprise-scale storage systems are extremely difficult to manage. The size of these systems, the thousands of configuration choices, and the lack of information about workload behaviors raise numerous management challenges. Users' demand for larger data capacities, more predictable performance, and faster deployment of new applications and services exacerbate the management problems. Worse, administrators skilled in designing, implementing and managing these storage systems are expensive and in short supply. It is estimated that the cost of managing storage is several times the purchase price of the storage hardware [1, 25]. These difficulties are beginning to cause enterprise customers to out-source their storage needs to Internet data centers and storage service providers, such as Exodus [18], who will lease networked storage. The growing importance of this storage model implies that the ability to accurately provision storage systems to meet workload needs will become even more critical in the future.

Storage management challenges include designing and implementing the storage system, adapting to changes in workloads and device status, designing the storage area network [32], and backing up the data [27]. In this paper, we concentrate on the important problem of *storage system configuration*: designing and implementing the storage system needed to support a particular workload, before the storage system is put into production use.

Given a pool of storage resources and a workload, we want to determine how to automatically choose storage devices, determine the appropriate device configurations, and assign the workload to the configured storage. These tasks are challenging because the large number of design choices may interact with each other in poorly understood ways. To make reasonable design choices, administrators need detailed knowledge of applications' storage behavior, which is difficult to obtain. Once a design has been determined, implementing the chosen design is time-consuming, tedious and error-prone. A mistake in any of the implementation operations is difficult to identify, and can result in a failure to meet the performance requirements of the workload.

Storage system configuration is naturally an iterative process, traditionally undertaken by human experts using "rules of thumb" gained through years of experience. They start with a first design based on an initial understanding of the workload, and then successively refine the design based on the observed behavior of the system. Figure 1 illustrates this iterative loop. Unfortunately, the complexities of the systems being designed, coupled with inadequate information about the true workload requirements, mean that the resulting systems are often over-provisioned so that they are too expensive, or under-provisioned so that they perform poorly.

In this paper, we describe Hippodrome, a system that automates the iterative approach to storage system configuration shown in Figure 1. Hippodrome analyzes a running workload to determine its requirements, calculates a new storage system design, and migrates the existing system to the new design. Hippodrome makes better design decisions by systematically exploring the large space of possible designs. Hippodrome decreases the chance of human error by automating the configuration tasks. As a result, Hippodrome frees administrators to focus on the applications that use the storage system.

We show that Hippodrome generates storage system configurations that employ near minimal resources to satisfy workload requirements, and that it converges to the final system design in a small number of iterations.

The remainder of this paper is organized as follows. Section 2 describes the automation of storage system configuration, including its goals and challenges. Section 3 introduces our solution, Hippodrome, and its components. Section 4 describes our experimental methodology and presents our results on random-access workloads and the PostMark filesystem benchmark. Section 5 discusses related work, Section 6 summarizes our results, and Section 7 describes directions for future research.

2 Problem statement

The iterative approach to system management shown in Figure 1 is applicable to many levels of the system, including the block-level array subsystem, the filesystem and the application itself. We focus on the block-level storage, as it provides a potential benefit to all applications that store data, including those that use the filesystem and those that use the raw block interface directly.

We define the three stages of the iterative storage management loop as follows:

- **Design new system**: Design a system to match the current workload requirements. This stage includes choosing which storage devices to use, selecting their configurations, and determining how to map the workload's data onto the configured devices. The requirements may come from observations of the workload behavior in previous iterations.
- **Implement design**: Configure the disk arrays and other storage system components, enable access to the storage resources from the hosts, and migrate the existing application data (if any) to the new design.
- **Analyze workload**: Analyze the running system to learn the workload's behavior. This information can then be used as input to the design stage in the next iteration.

We want to remove the human administrators from the loop as much as possible, by automating the iterative loop, to the point where all that is required at the beginning is workload capacity information. The loop will then learn the performance requirements across multiple iterations of the loop.

In order to be considered successful, the automated loop must meet two goals. First, it must converge on a viable design that meets the workload's requirements without over- or under-provisioning. Second, it must converge to a stable final system as quickly as possible, with as little input as possible required from its users.

2.1 Definitions

A *workload* is the set of requests observed by the storage system. A particular workload may be generated by one or more applications using the storage system. We describe a workload in terms of *stores* and *streams*. A store is a logically contiguous chunk of storage. A stream captures information about the I/O accesses to a single associated store, such as average request rate and average request size. Section 3.1 describes the characteristics of streams. Expressing a workload in terms of stores and streams decouples the specification of the workload from the application(s) that generate that workload. As a result, our workload specification and assignment techniques are applicable to a broad range of applications.

Disk array storage is divided into *logical units (LUs)*, which are logically contiguous arrays of blocks exported by a disk array. LUs are usually constructed by binding

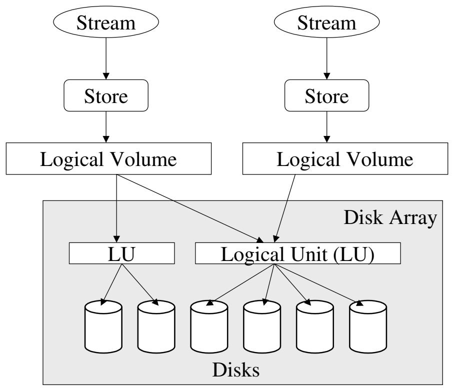


Figure 2: Storage and workload concepts. Stores and streams characterize the workload. LUs are containers for data. A store is implemented as a logical volume, which is used to map the store onto one or more LUs.

a subset of the array's disks together using RAID techniques. LU sizes are typically fixed by the array configuration, and so are unlikely to correspond to application requirements.

Logical volumes add flexibility by providing a level of virtualization that enables the server to split the (large) LUs into multiple pieces or to stripe data across multiple LUs. A logical volume provides the abstraction of a virtual disk for use by a filesystem or database table.

We implement a store as a logical volume in our system. Figure 2 illustrates the relationships between stores, streams, LUs and logical volumes.

3 Hippodrome

Hippodrome is our iterative design tool for storage systems. This section describes the components of the Hippodrome loop in more detail, and explains how the components interact to design a storage system iteratively.

The Hippodrome components are the most recent versions of our group's ongoing research into storage system modeling and configuration [2, 3, 4, 5, 6, 10, 26, 30, 31, 32]. The goal of this paper is to show how the different components can work together to automate storage management. Therefore, this section summarizes the techniques used in Hippodrome. The details on each component may be found in the paper on that topic.

Hippodrome uses four interdependent components to implement the iterative loop shown in Figure 1. The *analyze workload* stage summarizes a workload's behavior. This summary is used to predict the workload's requirements in the next iteration. Two components cooperate to implement the *design new system* stage: *performance models* for the storage devices, and a design engine, or *solver*. The performance models predict the utilization of storage devices under a candidate workload. The solver designs a new storage system using the performance models to guarantee that no device in the design is overloaded. The *implement design* stage migrates any existing system to the new design.

The different components share the responsibility for the correct operation of the loop. Collectively, they provide:

- **Accurate resource estimation.** The analysis and model components cooperate to accurately predict the utilizations of arbitrary candidate configurations. These utilizations are computed relative to the maximum performance capabilities of the devices, with utilizations at or above 100% indicating that the system is unable to support the desired performance of the workload, and utilizations under 100% indicating that the system can support the desired workload performance. The models and analysis encapsulate the physical performance characteristics of the devices and workloads, allowing the solver to concentrate on the optimization task.

- **Minimal designs.** The solver is responsible for choosing a design that uses the least resources among the set of candidate *valid designs*: ones that meet the performance requirements of the workload and ensure that all of the components in the design are under 100% utilized. It should perform this search quickly.

- **Balanced designs.** The solver is responsible for finding *balanced designs*: ones that have the minimal utilization variation across the storage system resources. Balanced designs allow the system to grow more quickly in each loop iteration, because they afford more opportunity for incorporation of new resources as the loop iterates.

- **Short migration time.** The migration component is responsible for converting the existing system into the new proposed system. It should perform this migration efficiently, without requiring significant temporary storage.

Accurate resource estimation and minimal designs result in correctly provisioned systems. Balanced designs and short migration time enable the loop to configure storage systems quickly.

We describe these components and their inputs/outputs in the sections below, focusing on how each component contributes to the operation of the Hippodrome loop.

3.1 Analysis component

The analysis component takes as input a detailed block-level trace of the workload's I/O references and a description of the storage system (LU and logical volume layouts). It outputs a summary of the trace in terms of stores and streams [31]. The analysis component captures enough properties of the I/O trace in the streams to enable the models to make accurate performance predictions.

The analysis component models an I/O stream as a series of alternating ON/OFF periods, where I/O requests are only generated during ON periods. More specifically, we define the minimum duration of an ON period, *minOnTime*, as 0.5 seconds, and the minimum duration of an OFF period, *minOffTime*, as at least two seconds of inactivity.

During an ON period, we measure six parameters for each stream: the mean read and write *request rates*; the mean read and write *request sizes*; the *run count*, which is the mean number of sequential requests; and the *queue length*, which is the mean number of outstanding I/O requests. Because streams can be ON or OFF at different times, we also capture inter-stream phasing and correlations using the *overlap fraction*, which is approximately the fraction of time that two streams' ON periods overlap. (The formal definition is slightly more involved and is described in [10].) Table 1 provides a summary of all of the stream attributes.

We choose to trace the I/O activity and analyze it later (or on another machine) to minimize the interference with the workload. Capturing I/O trace data results in a CPU overhead of 1-2% and an increase in I/O load of about 0.5%. Even day-long traces are typically only a few gigabytes long, which is a negligible storage overhead as the trace only has to be kept until the analysis is run. The duration of tracing activity is workload dependent, as it has to cover the full range of workload behavior. For simple workloads, a few minutes may be sufficient. For complex workloads, it may take a few hours.

3.2 Performance model component

The performance model takes as input a workload summary from the analysis component, and a candidate storage system design from the solver. The candidate design specifies both the parameters for the storage system and the layout of stores onto the storage system. It outputs the utilization of each component in the storage system. The model component needs to predict storage system performance quickly and accurately. We implement this component using table-based models [3]. The models use the stream information collected during the analysis stage to differentiate between sequential and random behavior, read and write behavior and ON-OFF phasing of disk I/Os. All of the properties shown in Table 1 are used, because we have found that ignoring any of them leads to inaccurate predictions. Models are used because simulating an I/O trace would be too slow for the solver to be able to examine a sufficient number of candidate configurations.

The performance models have three complementary parts:

1. **Inter-stream adjustments**. The input queue length and sequentiality are first adjusted to take into account the effect of interactions between streams on the same LU using the techniques described in [30]. For example, the sequentiality is decreased for two streams that are on simultaneously, because the overlap will cause extra seeks, while the queue length is increased because there will be more outstanding I/Os, which gives the disk array more opportunity for request re-ordering to improve performance.

2. **Single-stream prediction**. The utilization of each stream is calculated using a table of measurements [3]. 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. The utilization is the mean request rate of the stream divided by the maximum request rate.

3. **Utilization combination**. The model calculates the utilization of each LU by combining the estimated stream utilizations using the phasing algorithms found in [10]. The algorithms ensure that the utilization of two streams is proportional to the fraction of time that they overlap.

3.3 Solver component

The solver [5] reads as input the workload description generated by the analysis component, and outputs the design of a system that meets the workload's performance requirements. The output specifies a number of disk arrays, the configuration of those arrays (e.g., number of disks, LU configurations, controller and cache settings) and a mapping of the stores in the workload onto the disk arrays' LUs.

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	fraction of the "on" period when two streams are active simultaneously	fraction

Table 1: Workload characteristics generated by Hippodrome's analysis stage, and used by its models.

The solver efficiently searches the exponentially large space of storage system designs to find a balanced, valid, minimal design. The problem of efficiently packing a number of stores, with both capacity and performance requirements, onto disk arrays is similar to the problem of multi-dimensional bin packing. Since bin-packing is an NP-complete problem, exhaustive searches would take too long. Therefore our solver builds on the best-fit approaches found in [15, 21, 23] to produce initial solutions, and adds backtracking to help the solver avoid local minima in the search space of possible designs.

The solver algorithm has three phases:

1. **Initial assignment.** This phase attempts to find an initial, valid solution. It first randomizes the list of input stores, and then individually assigns them onto a growable set of LUs. It assigns each store onto the best available LU. Because the goal is to minimize the cost of the system, the best available LU is the one that is closest to being full after the addition of the store. If the store does not fit onto any available LU because the resulting utilization or capacity would be over 100%, the solver expands the storage system.
2. **LU re-assignment.** This phase attempts to improve on the solution found in the first phase. The solver uses randomized backtracking to avoid the local minima that can result from the first phase. It then randomly selects an LU from the current design, removes all the stores from it, and re-assigns those stores in a similar manner to the assignments done in the first phase. This operation is repeated until all of the LUs have been reassigned. At the end of this phase, we have a near-optimal but potentially unbalanced assignment of stores to LUs, using the minimum necessary storage resources.
3. **Store re-assignment.** This phase load-balances the best solution found in phase two. The load is measured as the utilizations of the components (e.g., LUs, disk-array controllers) predicted by the models. The solver repeatedly selects a store at random, removes it from the assignment and then reassigns it, but in this phase with the goal of producing a more balanced solution. The solver has already packed the stores tightly in the first two phases, and guarantees that the balanced solution does not increase in cost.

Experiments with this solver have found that it produces near optimal solutions. The optimal solution is a balanced valid design that meets the workload requirements with minimal set of resources. For most cases where we can prove optimality, the solver generates optimal solutions. We have also compared the solver to an exhaustive search algorithm on small cases, and again found that the solver finds optimal solutions. We have hand designed some pessimistic inputs (since the problem is NP-complete, these must exist), and found that the solver generates solutions about 10% worse than optimal on those inputs. In comparisons with other solver algorithms [2], we have found that our solver generates solutions that are as good or better. More details on the solver can be found in [5].

3.4 Migration component

The migration component takes as input the new design of the storage system, and changes the existing configuration to the new design. It configures storage devices, copies the data between old and new locations, and changes the parameters of the storage system to match the parameters in the new design.

Migration operates in two phases. First a plan is generated for the migration and then the plan is executed. The planning phase tries to minimize the amount of scratch space used and the amount of data that needs to be moved. The problem of migration planning for variable-sized objects is NP-complete, as it is reducible to subset sum [17]. We use a simple greedy heuristic that moves stores to their final location. If no store can be moved to its final location in a single step, the heuristic chooses a candidate store (or set of stores, if the underlying device needs to be reconfigured) and moves all of the stores blocking the move of the selected store into scratch space. The heuristic selects the candidate store to minimize the amount of scratch space needed. The result is a sequential plan for the migration.

If the underlying logical volume manager allows individual logical blocks to be moved, as opposed to an entire volume (store), then more advanced algorithms [4] that generate efficient parallel plans can be used.

Second, in the execution phase, the migration component copies the stores to their destinations as specified by the plan. The migration can be executed with the workloads either online or offline. Offline migration creates a new logical volume, copies the data there, and deletes the original volume. Online migration allows the workloads to continue executing. It uses the LVM to mirror the volume to its new location, and then splits the mirror, removing the old half. The techniques in [26] can be used to minimize the performance impact on the workload.

An alternative method that works during initial system configuration involves configuring the devices and then copying the data from a "master copy" of the stores to their final destinations. This approach works well if the design is changing substantially between iterations, but requires double the storage capacity to hold the master copy.

3.5 Putting it all together

We now consider how the Hippodrome components work together to find a storage system that supports the user's target workload so that the storage is not the bottleneck resource (i.e., the predicted utilization of all components is less than 100%). Since the performance of this target is unknown, Hippodrome iteratively estimates the target workload requirements by repeatedly generating and implementing storage designs. The estimation is performed by running the workload against the resulting system and monitoring and analyzing the I/O behaviors (as described in Section 3.1) to develop a new estimate of the workload requirements.

At each iteration of the loop, Hippodrome uses the new workload estimate to develop a storage system design to accommodate it. This design is created from scratch based only on the current estimate of the workload's requirements.

In searching the space of possible configurations, the solver will evaluate configurations with an increasing amount of resources. If it determines that all designs with the same amount of resources as in the current design would be over-committed, then we know that the storage system was the bottleneck, and the solver will find a design with more resources. If it determines that a design with fewer resources is sufficient, then we know that the storage system was under-utilized. Otherwise, it will find a configuration with the same amount of resources, and so we know that the loop has converged.

If the workload estimate was low because the storage system was over-committed, the newly designed system will contain more resources. The additional resources will allow the application(s) to increase their I/O performance, and hence increase the workload estimate. If the workload estimate is still too low, the process will repeat, until the workload appears to need no more resources.

For example, suppose the first iteration produces a design using 10 disk drives, based on only capacity information. Measurements from the first iteration might show the workload achieving 1000 I/Os per second (IOPS) on this configuration (because the bottleneck is the disk drives, which can perform 100 IOPS each). The second Hippodrome iteration might produce a design that incorporates 12 disk drives, because the Hippodrome models conservatively assume that the disk drives can only achieve 90 IOPS, and so 12 disk drives are required to support 1000 IOPS. With this configuration, the workload might then achieve 1200 IOPS, leading to a design with 14 disk drives. Finally, when running on the system with 14 disk drives, the workload might still run at 1200 IOPS, because the storage system is no longer the bottleneck. At this point Hippodrome has *converged*, and no longer increases the available resources.

The time to converge is determined by how many loop iterations must be performed and how long each iteration takes. 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 necessary to satisfy the workload's target performance requirements. Although Hippodrome performs well starting only with capacity requirements, Section 4 shows that Hippodrome can use an initial performance estimate to converge faster. 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. Conversely, analyzing the workload and generating the new design takes seconds to minutes.

The number of iterations required may be influenced by the explicit addition of *headroom* (resource slack) in each step. For example, Hippodrome's user can ask it to keep the utilization level below 85%, rather than 100%.

This guideline will increase the likelihood that there will be resource slack in the resulting system, thereby giving the workload a greater chance to express itself. This opportunity may result in increased application I/O performance and hence an increased workload estimate after the current iteration. In turn, this can reduce the number of iterations needed to get to the target workload. Explicit headroom can be used to compensate for optimistic errors in the performance models, where the models predict that available performance is higher than the actual performance (resulting in an under-estimate of the resources needed to support a workload). Headroom can also be implicitly added. If the models predict that the available performance is lower than the actual performance (i.e., they are pessimistic), the resulting designs implicitly include resource slack. In addition, solver designs that balance the load across the storage devices provide the maximum room for growth (over imbalanced designs), as no single part of the system will be nearer to its utilization limits than any other.

Once the load has stabilized and the configuration converged, retaining headroom may be of lesser value. However, it can still be used to accommodate short-term variations in the workload and to provide for future growth. Ultimately, providing headroom is part of a risk-cost decision: reducing the risk of a mis-configured system comes at the cost of additional resources. In what follows, we set the headroom to zero, because we wish to evaluate Hippodrome in the most stringent conditions, without any such resource slack.

We now turn to an evaluation of the Hippodrome system.

4 Experimental results

In this section we describe the experiments we ran to evaluate Hippodrome. Our experiments are designed to answer the following questions:

- Does Hippodrome converge? If so, how fast?
- Does Hippodrome allocate a reasonable amount of resources for a given workload?

4.1 Experimental workloads

Our evaluation is based on three variants of a simple fixed-size, random-access workload and a modified version of the PostMark benchmark [22]. The random-access 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 more realistic workload.

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 groups
Arrival process	4-bounded Poisson	4-bounded Poisson

Table 2: Common parameters for each stream in the random-access workloads.

In our experiments, we use the workloads shown in Table 2 with fixed-size, random requests; generating a load that ranges from 12.5 to 50 I/Os per second (IOPS) for each individual stream. We also use workloads that exhibit complex phasing behavior where groups of streams have correlated ON/OFF periods. We generate these workloads using a synthetic load generator capable of controlling the access patterns of individual streams. For each stream, it generates an access pattern from the given request rate, request size, sequentiality (specified by the run count), maximum number of outstanding requests and the duration of ON/OFF periods. We used a modified Poisson arrival process in the random-access workloads that restricted each stream to having no more than 4 requests outstanding at a given time.

We also use the PostMark benchmark, which simulates an email system, in our experiments. 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 have scaled the benchmark to use 40 sets of 10,000 files, ranging in size from 512B to 200KB. This provides both a large range of I/O sizes and sequentiality behavior. In order to vary the intensity of the workload, we run multiple identical copies of the benchmark simultaneously on the same filesystem. The data for the entire PostMark benchmark has been sized to fit within a single 50 GB filesystem. Hippodrome treats the filesystem as a single store, accessed by a single stream.

For each workload, we let Hippodrome generate an initial system design based solely on the capacity requirements and then iteratively improve the system design until it converges to support the workload. We do not expect the loop to converge in a single step, because the workload may not be able to run at full speed on the initial capacity-only design. However, we show that the loop converges quickly and that providing initial performance estimates can speed up the convergence.

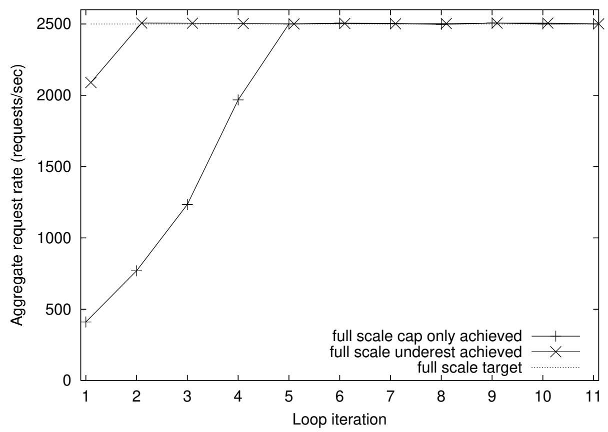
The migration step for the random-access workloads simply re-creates the logical volumes in the new locations. Those workloads accept arbitrary data in the logical volumes. The migration step for the PostMark workload copies the data from a master copy. Migrating the data would require reading and writing it from the same array since there is only a single store. Copying the data from the master speeds up our experiments because the array used for the experiments is only writing data during the migration.

4.2 Experimental infrastructure

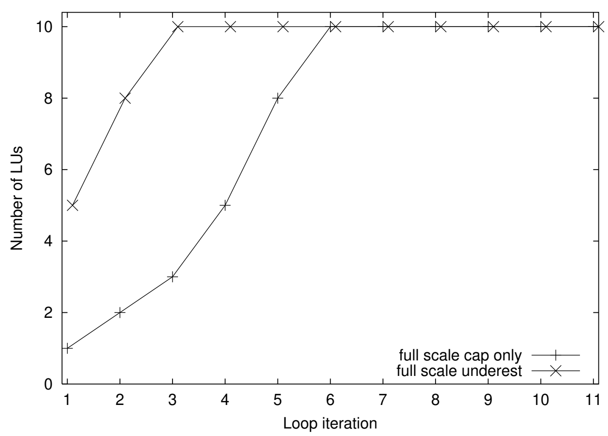
Our experimental infrastructure consists of an HP FC-60 disk array [20] and an HP 9000-N4000 server. The FC-60 array has sixty 36 GB Seagate ST136403LC disks, spread evenly across six disk enclosures. The FC-60 has two controllers in the same controller enclosure with one 40 MB/sec Ultra SCSI connection between the controller enclosure and each of the six disk enclosures. Each controller can access all of the SCSI buses, and has 512 MB of battery-backed cache (NVRAM). Dirty blocks are mirrored in both controller caches, to prevent data loss if a controller fails. Each controller of the FC-60 is connected to a Brocade Silkworm 2800 switch via a 1 Gb/sec FibreChannel link. A particular LU can only be efficiently accessed through a single controller at a time, although each controller can access all of the LUs.

Our HP 9000-N4000 server has eight 440 MHz PA-RISC 8500 processors and 16 GB of main memory, and runs HP-UX 11.0. It uses a separate FibreChannel interface to access each of the controllers in the disk array.

We configured each of the LUs in the system as a six disk RAID-5 LU with a 16 KB stripe unit size. This configuration allowed us to avoid a multi-hour array reconfiguration time during each iteration, at the cost of restricting the solver to a subset of the possible array configurations. Although Hippodrome is capable of allocating physical resources in smaller units (e.g., different numbers of disks in an LU), and it already considers controller and bus resource limitations in its allocation, the restriction to fixed-size LUs is convenient for experimental purposes. The restricted design space also helped us determine whether Hippodrome had found the correct configuration, as it allowed us to analyze all the possible designs. We report the resources Hippodrome allocates in units of LUs; the reader can also think of this as "groups of six disks".



(a) Average request rate



(b) Number of LUs.

Figure 3: (a) Target and achieved average request rates at each iteration of the loop for the random-access workloads with a target aggregate request rate of 2500 req/sec. (b) Number of LUs used during each iteration.

4.3 Random-access workloads

We start with fixed-size, random-access workloads so that it is easy to understand the behavior of the loop. We present two sets of results, one where all streams are ON at the same time (Section 4.3.1), and one where streams have correlated ON and OFF periods (Section 4.3.2). We compare our results to the target request rate to determine whether the system designed by Hippodrome has met the workload's requirements. However, this practice is for illustration only. Hippodrome has no knowledge of these target rates.

4.3.1 Always ON workloads

Figure 3(a) shows the target I/O rate and the achieved I/O rate for the random-access 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 an initial under-estimate of the performance (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 workload (2500 requests per second). We also see that the initial guess cuts the convergence time down to two iterations.

Figure 3(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. 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, a 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, even though the application could achieve its target rate without this.

Iterations for the random-access workloads are extremely quick. We run the workload generator for 5 minutes. The analysis and solver take at most a few minutes. The implementation takes a few minutes to re-build the logical volumes, but as the synthetic generator does not have any data, we skip the step of copying data onto the logical volumes.

These results show that Hippodrome can rapidly converge to the correct system design, using only capacity information as its initial input.

Figure 4 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, 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 be-

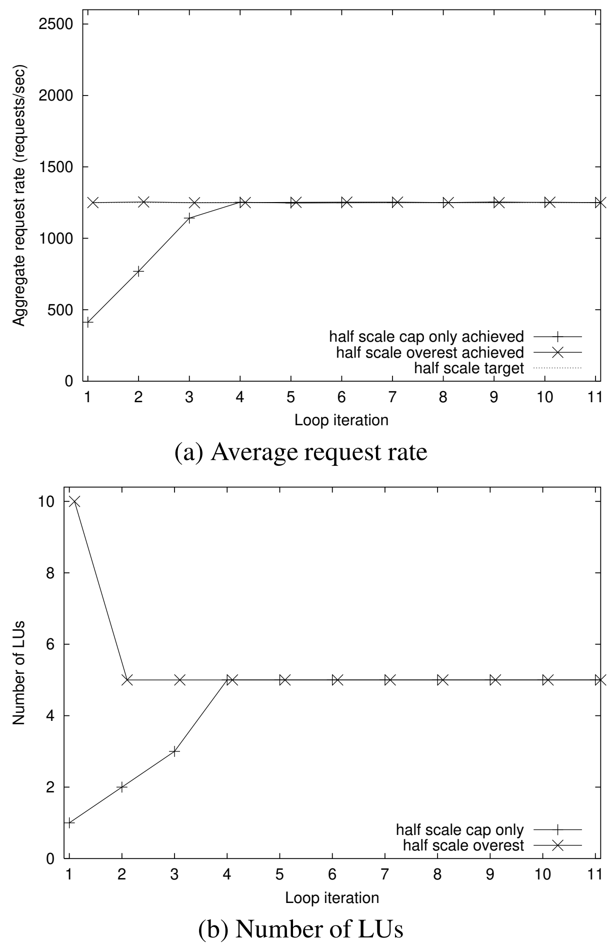
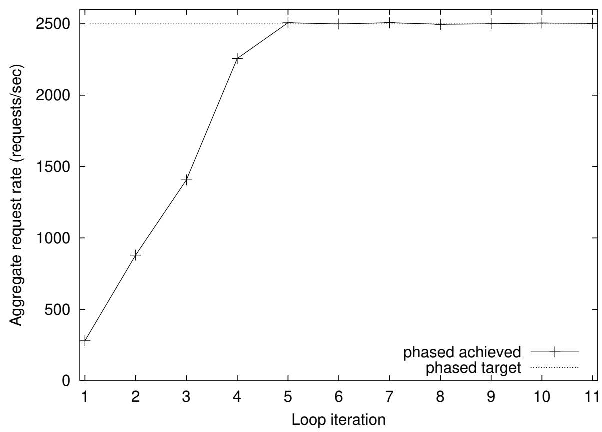


Figure 4: (a) Target and achieved average request rates at each iteration of the loop for the random-access workloads with a target aggregate request rate of 1250 req/sec. (b) Number of LUs used in each iteration.

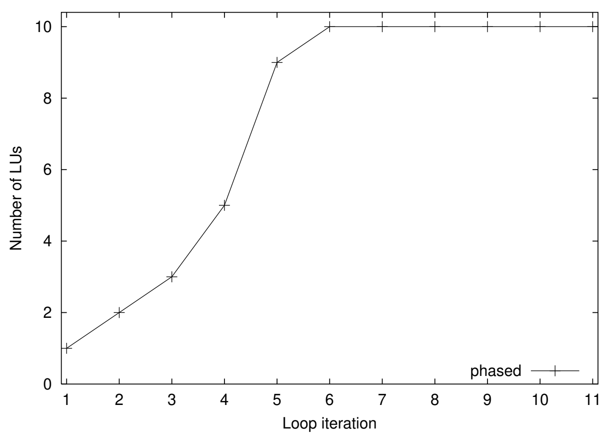
havior in the first iteration produces workload requirements that Hippodrome can accommodate with fewer resources, and Hippodrome scales back the system to use five LUs in the second iteration.

4.3.2 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



(a) Average request rate



(b) Number of LUs

Figure 5: (a) Target and achieved average request rates at each iteration of the loop for the phased random-access workloads with two correlated stream groups with a target aggregate request rate of 2500 req/sec. (b) Number of LUs used in each iteration.

for the storage system to support the workload. Figure 5 shows the average request rate achieved. We can see that the behavior of this workload is similar to 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 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 6 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

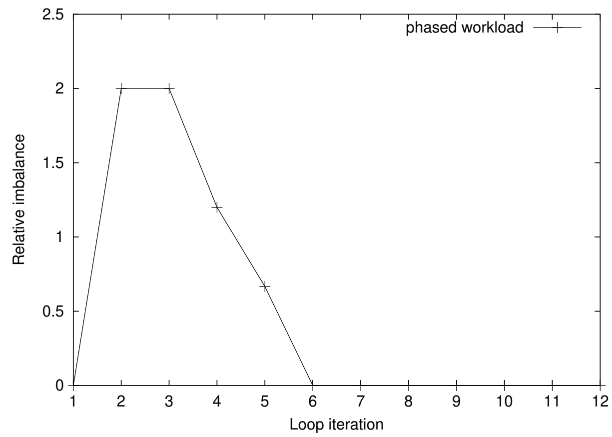


Figure 6: Relative imbalance of the two stream groups over the storage system for the phased workload.

LU for the phased workload; the imbalance goes to zero once the storage design has sufficient LUs.

4.4 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 previous workloads, which issued requests at a fixed rate when correctly provisioned, PostMark is designed to issue requests at the peak rate the I/O configuration can sustain, given sufficient resources at the clients. By using multiple PostMark processes, we can effectively simulate greater client resources, and a higher load. In order to determine what the achievable performance was in practice, we first ran a set of experiments with the PostMark filesystem split over a varying number of LUs. Figure 7 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 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 should converge to a stable number of LUs, and not keep trying to indefinitely expand its resources. Second, the final system should be near the inflection point of the performance curve: i.e., increasing the number of LUs beyond this point would not result in significant performance increases. Table 3 shows that Hippodrome satisfies both of these properties, converging in all cases to a system that has performance close to the maximum achievable, using a reason-

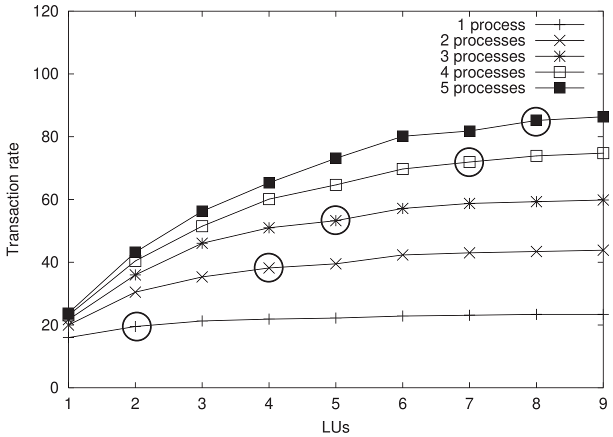


Figure 7: PostMark transaction rate as a function of number of LUs and processes used. The circles indicate the number of LUs/performance level to which Hippodrome converged.

number of processes	LUs	rate achieved	iterations	time
1	2	84%	2	1:13
2	4	87%	4	2:20
3	5	89%	5	2:53
4	7	96%	5	2:57
5	8	99%	5	2:59

Table 3: LUs, transaction rate achieved (as a percentage of the maximum observed for nine LUs), number of loop iterations and time to converge (hours and minutes) for the PostMark workload under varying numbers of processes.

able number of resources.

The wall clock time required for the loop to converge depends upon the number of iterations. Within each iteration, the majority of the time is spent copying the data from a master copy to the correct location in the new design, and takes between 20 and 40 minutes for the 50GB dataset, depending upon the number of LUs involved. The remaining time in each iteration is spent running the application, which takes roughly five to ten minutes. The analysis and the solver take a few minutes.

4.5 Summary

The experiments show that, for all workloads explored, Hippodrome satisfies the experimental goals. First, the system converges to the correct number of LUs in only a small number of loop iterations: at most four or five, and sometimes only one or two. Second, the designs that the system converges on are correctly provisioned; that is, the storage system contains the minimum number of LUs capable of supporting the offered workload. Finally, Hippodrome can leverage initial performance estimates (even inaccurate ones) to find the correct storage configuration more quickly.

These properties mean that Hippodrome can be used to perform storage system configuration automatically. 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, with the expectation that this configuration 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.

In the future, we would like to experiment with using more complex enterprise-scale workloads, such as a large database system. For such workloads, it is more difficult to tell if the loop did the "right thing", as we cannot easily *a priori* tell how good the design is, unlike the random-access and Postmark workloads. The applicability of Hippodrome to such systems is an ongoing research effort.

5 Related work

The EMC Symmetrix [14] and HP SureStore E XP512 Disk Arrays [19] 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 [33] supports moving data between RAID5 and RAID1. AutoRAID keeps current data in RAID1 (since it has better performance), and uses an LRU policy based on write rate and capacity to migrate infrequently accessed data to RAID5, which has higher capacity. Hippodrome correctly places data based on the usage patterns, and expands the storage system if necessary to support increases in the workload.

Teradata [9] 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.

River [8] is a cluster-based I/O architecture that uses credit-based back pressure and graduated declustering

(GD) to distribute work in a manner proportional to the speed of the recipient nodes. However, River requires modifying the application, and it makes short-term load-balancing decisions, and does not handle long-term changes in the workload. Conversely, Hippodrome makes long-term decisions and does not require application modification.

A few other, automated tools exist that are useful to administrators of enterprise class systems. The AutoAdmin index selection tool [12] 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.

LEO, IBM DB2's "learning optimizer" [29], uses a feedback loop to enhance query optimization performance estimates based on observed past performance. It monitors previously executed queries and compares the optimizer's cost estimates with the actual performance at each step in the query execution plan, and then adjusts the cost estimates and statistics that may be used in future query optimizations. Although it does not currently do so, Hippodrome could use such feedback from observed system performance to improve the quality of its storage device performance models.

Océano [7] 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 2 in its management of compute and network resources.

Muse [11] controls server allocation and energy-conscious, adaptive resource provisioning tool for Internet hosting centers. It is also based on an iterative loop, like Hippodrome, but it focuses on allocating computational resources. Its resource allocation framework is based on an economic model that factors in the trade-offs between the service quality and the cost.

Existing solutions to the file assignment problem [13, 34] use heuristic optimization models to assign files to disks to get improvements in I/O response times. The file allocation schemes described in [16, 28] 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.

6 Conclusions

In this paper we have introduced the Hippodrome loop, our approach to automating storage system configuration. 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 problem of storage system configuration, the Hippodrome loop satisfies two important properties:

- **Rapid convergence**: The loop converges in a small number of iterations to the final system design.
- **Correct resource allocation**: The loop allocates close to the minimal amount of resources necessary to support the workload.

We have demonstrated these properties using fixed-size, random-access workloads as well as the PostMark filesystem benchmark.

7 Ongoing and future work

We are currently extending Hippodrome to automatically manage the ongoing evolution of a storage system. Production systems evolve in time to handle device failures, changes in the workload, devices becoming obsolete, or new devices and workloads being added. Hippodrome should be able to detect and respond to these changes to keep the system appropriately provisioned and configured at all times. Preliminary results, using workloads similar to those described here, are promising. 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.

In the future, we plan to extend this work in several ways. First, we will investigate the sensitivity of the Hippodrome loop to the quality of its components. For instance, what information must be captured in the analysis stage to sufficiently specify the performance requirements of the workload [24]? What is the sensitivity of the loop to the quality of the model component's predictions or to the quality of the solutions generated by the design component? In addition, we will continue to investigate how to build better loop components, for example, higher accuracy models, and a solver that minimizes store motion across iterations.

Second, we plan to experiment with complex enterprise-scale applications, highly variable workloads with load spikes, and workloads with natural load cycles (e.g. daily backups or monthly reports). How can we tell how well Hippodrome does for more complex workloads, given the difficulty of provisioning such large systems? How should the analysis infrastructure differentiate transient load spikes from workload growth trends? Furthermore, how should Hippodrome incorporate information about the cyclic nature of the workload to support all operational modes, while not grossly over-provisioning the system?

Third, we plan to extend Hippodrome to manage very large scale, heterogeneous and widely distributed storage systems. The experiments in this paper explored workloads that could fit onto a single disk array. How well does Hippodrome perform for workloads that require multiple disk arrays? How well does Hippodrome handle multiple types of disk arrays?

Additional research questions include the following:

- How well does Hippodrome interact with optimizations at the application level, which may result in changes to the I/O workload? For instance, how would the automated storage loop interact with database systems that automatically create indices as needed [12] or tune query plans based on observed performance [29]?

- How does Hippodrome fit into an overall end-to-end optimization scheme? For instance, how should Hippodrome cooperate with other solutions for storage area network design [32] or quality of service-preserving online migration [26]?

- What are the right evaluation metrics for the effectiveness of the automated loop? In addition to convergence time, potential metrics might be the number of errors, the overall system cost, the system performance, and the cost/performance trade-off. An additional meta-level question is how to define "goodness" for each of these metrics.

8 Acknowledgements

We are grateful to our shepherd, Jeff Chase, for his invaluable comments, which improved our paper tremendously. We would like to thank John Wilkes for his comments on this paper, and Aaron Brown, David Oppenheimer, Dave Patterson, Arif Merchant and Erik Riedel for their comments on a previous version of this paper.

References

[1] N. Allen. Don't waste your storage dollars: what you need to know. Research note, Gartner Group, March 2001.

[2] G. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A. Veitch, and J. Wilkes. Minerva: an automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, Nov. 2001.

[3] E. Anderson. Simple table-based modeling of storage devices. Technical Report HPL–SSP–2001–4, HP Laboratories, July 2001. http://www.hpl.hp.com/SSP/papers/.

[4] E. Anderson, J. Hall, J. Hartline, M.Hobbs, A. Karlin, J. Saia, R. Swaminathan, and J. Wilkes. An experimental study of data migration algorithms. In *5th Workshop on Algorithm Engineering (WAE2001)*, University of Aarhus, Denmark, Aug. 2001.

[5] E. Anderson, M. Kallahalla, S. Spence, R. Swaminathan, and Q. Wang. Ergastulum: an approach to solving the workload and device configuration problem. Technical Report HPL–SSP–2001–5, HP Laboratories, July 2001. http://www.hpl.hp.com/SSP/papers/.

[6] E. Anderson, R. Swaminathan, A. Veitch, G. Alvarez, and J. Wilkes. Selecting RAID levels for disk arrays. In *Conference on File and Storage Technologies (FAST)*, Monterey, CA, January 2002.

[7] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. P. Pazel, J. Pershing, and B. Rochwerger. Océano – SLA based management of a computing utility. In *Integrated Network Management VII*, May 2001.

[8] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: Making the fast case common. In *6th Workshop on Input/Output in Parallel and Distributed Systems (IOPADS'99)*, pages 10–22, Atlanta, GA, May 1999.

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

[10] E. Borowsky, R. Golding, P. Jacobson, A. Merchant, L. Schreier, M. Spasojevic, and J. Wilkes. Capacity planning with phased workloads. In *1st Workshop on Software and Performance (WOSP'98)*, pages 199–207, Santa Fe, NM, Oct 1998.

[11] J. Chase, D. Anderson, P. Thakar, and A. Vahdat. Managing energy and server resources in hosting centers. In *18th ACM Symposium on Operating System Principles (SOSP'01)*, pages 103–116, Chateau Lake Louise, Banff, Canada, October 2001.

[12] S. Chaudhuri and V. Narasayya. AutoAdmin "What-if" index analysis utility. In *SIGMOD International Conference on Management of Data*, pages 367–378, June 1998.

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

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

[15] W. Fernandez and G. Lueker. Bin packing can be solved within 1+ε in linear time. *Combinatorica*, 1(4):349–55, 1981.

[16] P. Z. G. Weikum and P. Scheuermann. Dynamic file allocation in disk arrays. In *SIGMOD Conference*, pages 406–415, 1991.

[17] M. Garey and D. Johnson. *Computing and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

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

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

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

[21] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3(4):299–325, Dec. 1974.

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

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

[24] Z. Kurmas, K. Keeton, and R. Becker-Szendy. I/O Workload Characterization. In *Fourth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, Jan 2001.

[25] E. Lamb. Hardware spending spatters. *Red Herring*, pages 32–33, June 2001.

[26] C. Lu, G. A. Alvarez, and J. Wilkes. Aqueduct: online data migration with performance guarantees. In *Conference on File and Storage Technologies (FAST)*, Jan 2002.

[27] W. C. Preston. Using gigabit ethernet to backup six terabytes. In *12th Systems Administration Conference (LISA '98)*, pages 87–95, Boston, Massachusetts, 1998.

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

[29] M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO - DB2's learning optimizer. In *27th Intl. Conference on Very Large Data Bases (VLDB)*, 2001.

[30] M. Uysal, G. Alvarez, and A. Merchant. A modular, analytical throughput model for modern disk arrays. In *9th International Symposium on Modeling, Analysis and Simulation on Computer and Telecommunications Systems (MASCOTS 2001)*, Cincinnati, OH, Aug. 2001.

[31] A. Veitch, K. Keeton, M. Spasojevic, and J. Wilkes. The Rubicon workload characterization tool. Technical report, HP Laboratories, April 2001. http://www.hpl.hp.com/SSP/.

[32] J. Ward, M. O'Sullivan, T. Shahoumian, and J. Wilkes. Appia: Automatic storage area network fabric design. In *Conference on File and Storage Technologies (FAST)*, Monterey, CA, January 2002.

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

[34] J. Wolf. The placement optimization program: a practical solution to the disk file assignment problem. In *ACM SIGMETRICS Conference*, pages 1–10, Berkeley, CA, May 1989.