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 95014
{anderse,mjhobbs,kkeeton,suspence,uysal,aveitch}@hpl.hp.com

ABSTRACT

Enterprise-scale computer storage systems are extremely difficult to manage due to their size and complexity. It is difficult to generate a good storage system design for a given workload and to correctly implement the selected design. Traditionally, initial system configuration is performed by administrators who are guided by rules of thumb. Unfortunately, this process involves trial and error, and as a result is tedious and error-prone. In this paper, we introduce Hippodrome, an approach to automating initial system configuration. Hippodrome is an iterative loop that analyzes an existing system to determine its requirements, creates a new storage system design to better meet these requirements, and migrates the existing system to the new design. In this paper, we show how Hippodrome automates initial system configuration.

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 workload 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 very 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 the order in which 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 RAID1 if the workload is I/O intensive, and RAID5 otherwise. They then map the application data onto 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.

[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 RAID5 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.

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. Because of these difficulties, the results of this process are systems that take a long time to set-up, and yet are still either over-provisioned, so they are too expensive, or under-provisioned, so they have poor performance.

In this paper, 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.

We present a progression of increasingly complex systems for initial system configuration to explain the intuition behind Hippodrome's iterative loop. We start by describing the current, human-intensive, largely ad-hoc, manual methods. We analyze the process to determine which components are needed to convert the manual process into a simple, automatic loop. We then show that it is necessary to increase the sophistication of each component to overcome limitations in the initial automated loop. We increase the sophistication until the components are sufficiently advanced to handle most of the complexity of I/O workloads during initial system configuration.

The remainder of this paper is organized as follows. Section 2 presents the component requirements and loop progression that results in Hippodrome. Section 3 describes our experimental setup, methodology and workloads. Section 4 presents the results of applying Hippodrome to initial system sizing of synthetic workloads and the PostMark file system benchmark. Section 5 discusses related work and Section 6 summarizes the results and describes directions for future research.

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.

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 RAID1 if the workload is I/O-intensive, and RAID5 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 [23], 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.

They 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.

They then implement the proposed design by configuring newly purchased resources as described above, and by using

[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.

array tools and the LVM to migrate the stores to the appropriate target LUs. For instance, the HP XP512 [15] and EMC Symmetrix [10] 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 relatively well-understood applications, this process can take many weeks of time and effort (e.g., 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.

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 overprovision 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 [14], 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.

2.2 The iterative loop

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

- **Design new system**: Based on available inputs (typically previous observations of the system behavior), design a system that will (hopefully) better match the workload requirements.

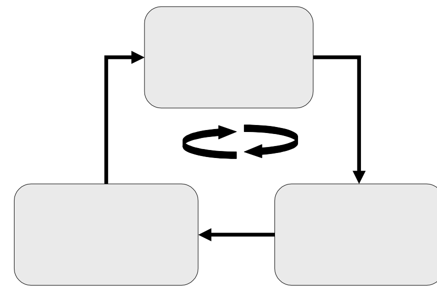


Figure 1: Three stages of the iterative loop.

- **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.

- **Analyze workload**: Once the system is running, analyze the system to determine its performance characteristics. This information can then be used to produce a better system design.

As described in the previous section and shown in Figure 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, for example 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 necessary to satisfy the workload's performance requirements. 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.

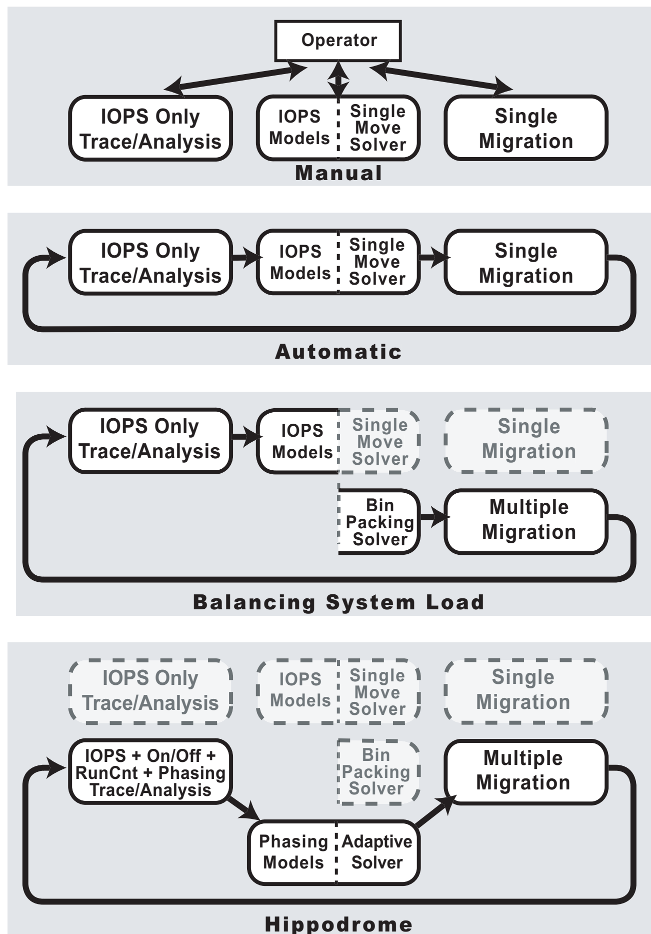


Figure 2: Loop progression. As the analysis, solver and migration components improve, so does the resulting loop approach. The automatic approaches are described in more detail in Section 2.3 through Section 2.5.

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 18GB in size, capable of performing 100 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 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. This progression is illustrated in Figure 2. 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 2.4 achieves a balanced load in the final system, whereas the one in Section 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.

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 build up enough expe-

rience 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 don't. Some tools provide some 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 2.

2.3.1 *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, the streams will capture many more properties.

2.3.2 *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, a disk can re-position in about 10ms, so an LU consisted of a single-disk can perform about 100 IOPS.

2.3.3 *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 doesn't 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.

[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.

2.3.4 *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 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.

2.3.5 *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, 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, 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 an extremely 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.

2.4 Balancing system load

As shown in Figure 2, 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.

2.4.1 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 general [11, 17, 19]. 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.

2.4.2 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 didn't 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 [13], 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 [2], 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.

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

2.4.3 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.

2.5 Hippodrome

Hippodrome, shown at the bottom of Figure 2, builds upon the previous approach by greatly improving the performance models and improving the design component to take advantage of them.

2.5.1 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, we add all of the attributes shown in Table 1.

We model an I/O stream as a series of alternating on/off periods. During an on period, we measure four parameters seperately for reads and writes. The first parameter is the *request rate*, which is the mean of the read (respectively write) 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

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.

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 [7].

2.5.2 *Improved performance models*

Hippodrome uses the table-based models described in [1], 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 [22]. 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 reordering.

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 [7]. The algorithms use queuing theory techniques so that the utilization of two streams is proportional to the fraction of time that they overlap.

2.5.3 *Improved design*

Introducing the more complex models violates the bin packing algorithms' assumption that individual stream utilizations are additive, as described in Section 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 [3] in the Hippodrome design stage builds on the best-fit approaches found in [11, 17, 19] 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

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

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.

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't 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 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.

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 Table 1 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. While 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. While the current models incorporated into Hippodrome have been validated over a wide range of workloads, we have encountered a few situations in our experiments where these behaviors occurred.

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.

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?

3.1 Workloads

Our evaluation is based on a variety of synthetic workloads and a modified version of the PostMark [18] benchmark. 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 2 with fixed-size, random requests; generating 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 4 requests.

We also used the PostMark benchmark [18], 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 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 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.

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

Table 2: Common parameters for synthetic workloads.

3.2 Experimental infrastructure

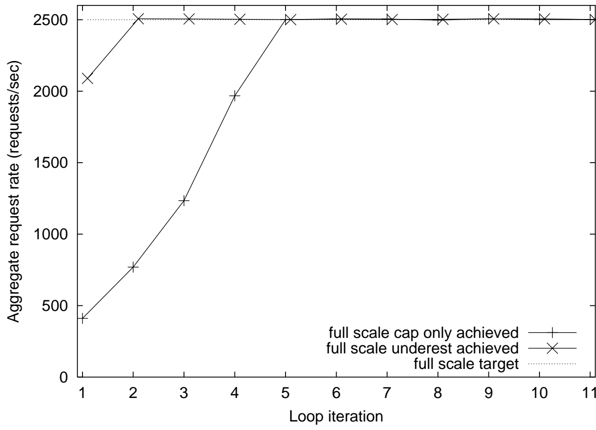
Our experimental infrastructure consists of an HP FC-60 disk array [16] and an HP 9000-N4000 server. The FC-60 array has 60 disks, each of which is a 36 GB Seagate ST136403LC disk, spread evenly across six disk enclosures. The FC-60 has two controllers in the same controller enclosure with one 40 MB/s 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.

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.

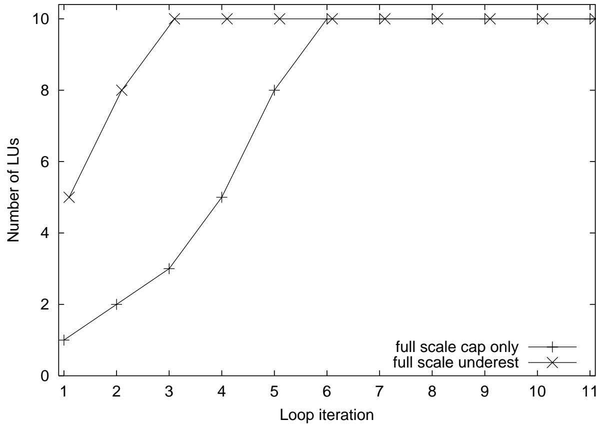
We have configured each of the LUs in the system as 6 disk RAID-5 with a 16 KB stripe unit size. The common configuration allows us to avoid the multi-hour reconfiguration time.

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.

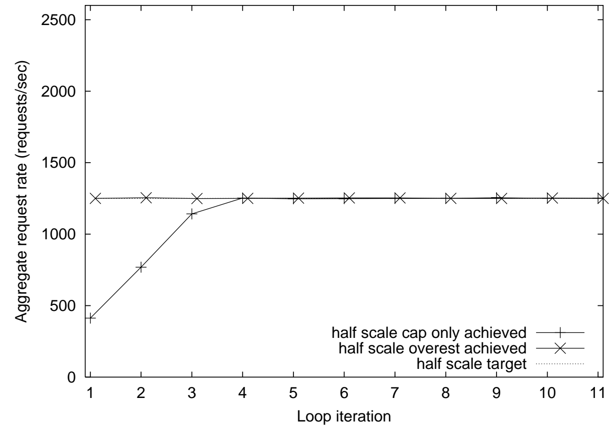


(a) Average request rate

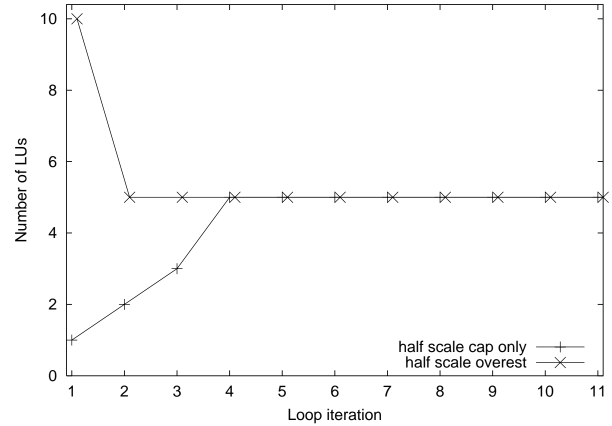


(b) Number of LUs.

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



(a) Average request rate



(b) Number of LUs

Figure 4: (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.

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, one where all streams are on at the same time (section 4.1.1), and one where streams have correlated on and off periods (section 4.1.2).

4.1.1 Always on workloads

Figure 3(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).

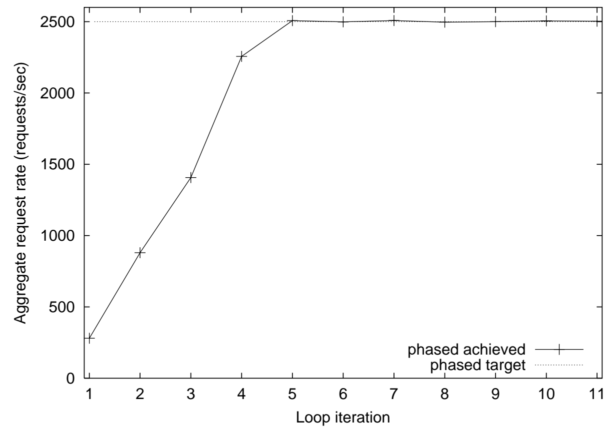
Figure 3(b) shows the number of LUs allocated by Hippodrome at each loop iteration to achieve the target I/O rate. The system converges in five loop iterations starting from only capacity requirements. In the first four iterations, the LUs are over-utilized, and Hippodrome allocates new LUs, increasing the system size to better match the target request rate. As more LUs are added, 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 underestimate 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 despite the application achieving its target rate (as discussed in Section 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 3 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.

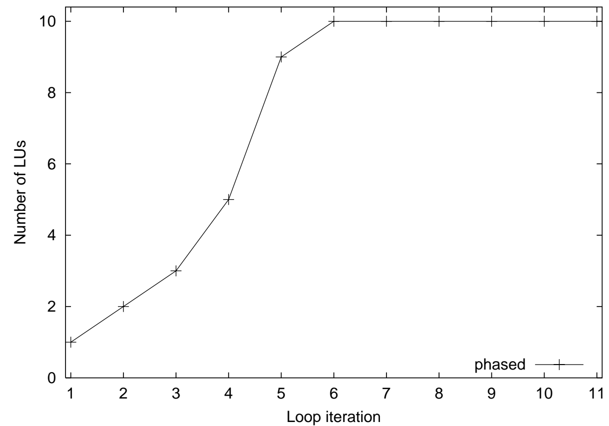
Figure 4 shows that Hippodrome uses the minimal number of resources necessary to satisfy the workload's performance requirements. The target request rate for both workloads is 1250 requests per second, which can be achieved using only five LUs. Given only capacity requirements as a starting point, the 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 in the first iteration shows that the resources are under-utilized, and Hippodrome scales back the system to use five LUs in the second iteration.

4.1.2 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 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.



(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 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.

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

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 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 would converge to a stable number of LUs, and not keep trying to indefinitely expand its resources, and second that the system converged to would be near the inflection point of the performance curve, i.e. increasing the number of LUs beyond this

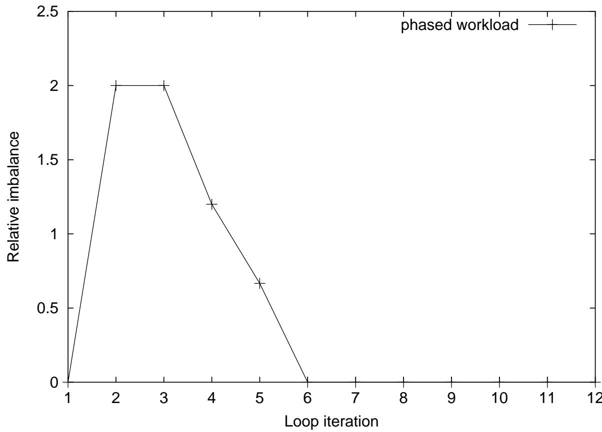


Figure 6: Relative imbalance of the two stream groups over the storage system for the phased workload.

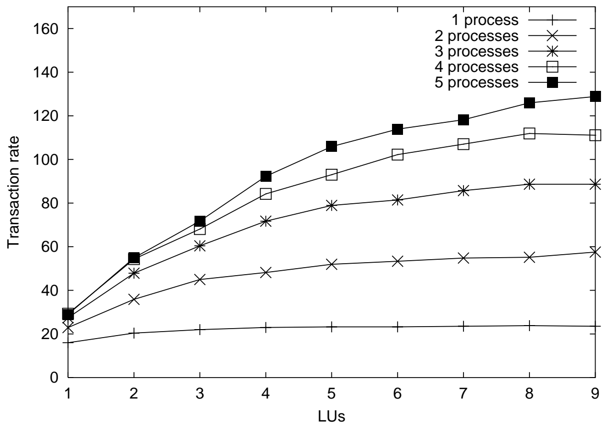


Figure 7: PostMark transaction rate as a function of number of LUs and processes used.

point would not result in 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 is indicative of the models under-predicting the utilization of the storage system, a problem discussed in Section 2.7. We determined that the PostMark benchmark had only 2.4 I/O's queued on average, whereas the models were calibrated with a minimum value of 16[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).

[7]This value is much more typical of the large, multiprocessor, I/O intensive workloads that we are targeting. Unfortunately, the low number of processes used in the PostMark benchmark result in concomitantly smaller queue lengths. **Reviewers: We will be extending our models to cover a wider range of workloads**

#processes	*headroom* 1.0		0.9	
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%

Table 3: 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.

Table 3 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.

4.3 Summary

The initial system configuration experiments show that, for all workloads, Hippodrome satisfies the three properties introduced in Section 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. 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 effi-

cient 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 Related Work

The EMC Symmetrix [10] and HP SureStore E XP512 Disk Arrays [15] 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 [24] 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 will correctly place data based on the usage patterns, and will expand the storage system if necessary to support increases in the workload.

Teradata [6] 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 [5] 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, Rivers requires modifying the application, and it makes short term load-balancing decisions, so long term changes in the workload are not handled. Conversely, Hippodrome makes long term decisions and does not require application modification.

IBM's work on capacity space management [20] 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 [8] 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 [4] 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 2.3 in its management of compute and network resources.

Existing solutions to the file assignment problem [9, 25] use heuristic optimization models to assign files to disks to get improvements in I/O response times. The work described on file allocation in [12, 21] 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 and future work

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 have several ongoing research projects that extend or rely upon Hippodrome. We plan to continue investigations

into how to build better loop components. In particular, we are working on new modeling techniques that we hope will allow for easier, and more accurate, models. We are also planning to conduct experiments on extremely large, complex enterprise-scale applications, operating on multiple, different arrays.

We are also experimenting with the use of 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.

7 Acknowledgements

We would like to thank Aaron Brown and David Oppenheimer for their comments on this paper, and Dave Patterson, Arif Merchant and Erik Riedel for their comments on a previous version of this paper.

REFERENCES

[1] E. Anderson. Simple table-based modeling of storage devices. Technical note, HPL-SSP-2001-4, HP Labs, July 2001.

[2] E. Anderson, J. Hall, J. Hartline, M.Hobbs, A. Karlin, R. Swaminathan, and J. Wilkes. An Experimental Study of Data Migration Algorithms. In *Proceedings of the 5th Workshop on Algorithm Engineering (WAE2001)*, University of Aarhus, Denmark, Aug. 2001.

[3] E. Anderson, M. Kallahalla, S. Spence, R. Swaminathan, and Q. Wang. Ergastulum: An approach to solving the workload and device configuration problem. Technical note, HPL-SSP-2001-5, HP Labs, July 2001.

[4] 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.

[5] 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 *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS'99)*, pages 10–22, May 1999.

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

[7] E. Borowsky, R. Golding, P. Jacobson, A. Merchant, L. Schreier, M. Spasojevic, and J. Wilkes. Capacity planning with phased workloads. In *Proceedings of the First Workshop on Software and Performance (WOSP'98)*, pages 199–207, Oct 1998.

[8] S. Chaudhuri and V. 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.

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

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

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

[12] P. Z. G. Weikum and P. Scheuermann. Dynamic File Allocation in Disk Arrays. In *Proceedings of the 1991 SIGMOD Conference*, pages 406–415, 1991.

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

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

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

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

[17] 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, Springer Verlag (Heidelberg, FRG and NewYork NY, USA)-Verlag*, 3, 1974.

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

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

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

[21] 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.

[22] M. Uysal, G. A. Alvarez, and A. Merchant. A modular, analytical throughput model for modern disk arrays. In *Proceedings of Ninth MASCOTS*, Aug. 2001.

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

[24] 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.

[25] J. 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.