Efficient Data Migration in Self-managing Storage Systems

Vijay Sundaram and **Prashant Shenoy**
Department of Computer Science,
Univeristy of Massachusetts,
Amherst MA 01003.

Abstract

Self-managing storage systems automate the tasks of detecting hotspots and triggering data migration to alleviate them. This paper argues that existing data migration techniques do not minimize data copying overhead incurred during reconfiguration, which in turn impacts application performance. We propose a novel technique that automatically detects hotspots and uses the bandwidth-to-space ratio metric to greedily reconfigure the system while minimizing the resulting data copying overhead. We implement our techniques into the Linux kernel and conduct a detailed evaluation using simulations and our prototype. Our results show a factor of two reduction in data copying overhead when compared to other approaches for a variety of system configurations. Our prototype successfully identifies system hotspots and eliminates them by incrementally computing a new configuration and without causing any noticeable degradation in application performance.

1 Introduction

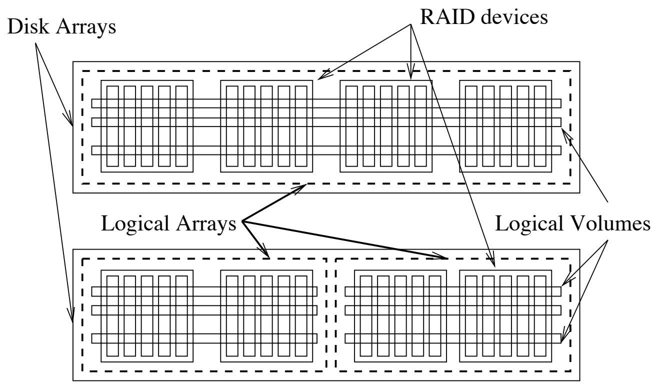
Recent trends in storage indicate that the cost per megabyte of hard drives continues to drop, while the appetite for storage capacity in enterprises continues to spiral. As information is increasingly created, processed, and manipulated in digital form, the role of large-scale enterprise storage systems becomes increasingly important. Enterprise storage systems are complex entities that comprise a large number of RAID arrays with one or more data partitions mapped to each array. As storage needs and I/O workloads evolve over time, managing, configuring, and continual tuning of such systems becomes a complex task [3, 4]. Consequently, design of self-managing storage systems is an appealing option; such a system performs automated mapping of data partitions to RAID arrays, monitors the workload on each array, and transparently triggers data migration if hotspots or imbalances are detected in the system.

Until recently, these tasks were performed manually by administrators using sophisticated tools to analyze the load on the system [1]. Such tools collect performance data, summarize loads on arrays, and predict the performance impact of moving a data partition from one array to another. Despite their sophistication, the decision process is manual and prone to human error. To address this drawback, recent efforts have focused on automating the task of detecting hotspots and triggering data migration to alleviate them [6, 8, 10]. Such a remapping of data partitions to RAID arrays involves a system reconfiguration that either results in downtime or reduced performance during the migration. Consequently, it is essential to minimize the reconfiguration overhead by *minimizing the volume of data moved*, and thus, the time needed to do so. Unfortunately, recently proposed approaches do not focus on minimizing data copying, resulting in potentially larger overhead than is necessary.

This paper focuses on the design of automated data migration algorithms that minimize the total data copying overhead incurred during reconfiguration in self-managing storage systems. We also propose automated techniques to detect hotspots and trigger our data migration algorithm. Our work has resulted in several contributions.

First, we propose a novel technique to minimize data copying overhead incurred during a system reconfiguration to alleviate hotspots. Our technique uses *bandwidth-to-space (BSR)* ratio as a guiding metric to maximize the load reduction per unit of data moved. The technique greedily displaces excess load from overloaded to underloaded disk arrays using two different strategies: (i) *displace*, which moves a data partition from one array to another, and (ii) *swap*, which swaps data partitions between underloaded and overloaded arrays.

Second, we describe a measurement-based approach that automatically detects hotspots in the storage system and invokes the above algorithm to alleviate them. Third, we implement our techniques into the Linux kernel and evaluate their efficacy using both simulation and prototype evaluation. Our results show a factor of two reduction in the data copying overhead for a variety of system configurations when compared to other approaches. Our prototype evaluation also shows the efficacy of our technique in correctly identifying hotspots and triggering data migration.



The figure shows two disk arrays with four RAID devices each. Each RAID device has five disks. Logical volumes are striped across all RAID devices on the first array, while each volume is striped across two RAID devices in the second array.

Figure 1: An enterprise storage system.

2 Problem Formulation

Consider an enterprise storage system as shown in Figure 1. Such a system comprises multiple disk arrays; arrays may be heterogeneous in terms of their storage capacities and bandwidth. Each array consists of one or more RAID devices; a RAID device is constructed by combining a set of disks from the array using RAID techniques [13]. For instance, a disk array of 16 disks may be partitioned into two RAID-5 devices, each comprising 8 disks of the array. Data partitions—also referred to as logical volumes—are then mapped on these RAID devices. Each RAID device can contain one or more data partitions. Interestingly, it is possible for a partition to span multiple RAID devices as well; when a partition spans multiple RAID devices, the set of underlying disks form a logical RAID (comprising multiple physical RAID devices). In this paper, we use the term logical array (or array for short) to collectively refer to the set of disks across which a data partition is striped. The terms data partitions and logical volumes are used interchangeably.

Assuming such a model, the data migration problem is defined as follows. Let the enterprise storage system consist of n arrays, $A_1, A_2, \ldots A_n$, and let m logical volumes $L_1, L_2, \ldots L_m$ be mapped onto these arrays. Each array A_j is assumed to have a certain storage capacity $\mathcal{S}_j$ and a certain bandwidth capacity $\mathcal{B}_j$. Similarly, each data partition/volume L_i has a certain storage requirement s_i and bandwidth need b_i. Clearly, the total storage needs of all partitions mapped onto an array can not exceed its storage capacity:

$$\Sigma_i s_i \cdot c_{ij} \leq \mathcal{S}_j \tag{1}$$

where c_{ij} is a binary variable that takes value 1 if volume i is mapped onto array j and 0 otherwise. An array is said to be load-balanced if the total bandwidth utilization of the array is smaller than a threshold τ_j:

$$\frac{\sum_i b_i \cdot c_{ij}}{\mathcal{B}_j} \leq \tau_j \tag{2}$$

While the above storage constraint is hard and can not be violated, the bandwidth constraint can see occasional violations if the cumulative bandwidth needs of partitions increase. In such an event, the array is said to be overloaded. Thus, a hotspot is said to be present in the system if Equation 2 is violated for one or more arrays. The *magnitude* of the hotspot is defined to be the cumulative overload $\sum_{j \in \mathcal{O}} (\Sigma_i b_i c_{ij} / \mathcal{B}_j - \tau_j)$ where $\mathcal{O}$ is the set of overloaded arrays.

Alleviating the hotspot requires some logical volumes to be moved from overloaded arrays to underloaded ones. The cost of such a system reconfiguration is defined to be the total amount of data moved.

$$Cost = \Sigma_i \Sigma_j |c_{ij}^{new} - c_{ij}^{old}| * s_i/2 \tag{3}$$

where c_{ij}^{new} and c_{ij}^{old} denote the new and old set of mappings of volumes to arrays, respectively. Observe that if a volume is unchanged, it does not contribute to the cost, whereas any moved volume L_i causes $\sum_j |c_{ij}^{new} - c_{ij}^{old}| = 2$ and contributes a cost that equals its storage capacity. Thus, the above equation captures the total amount of data moved for all remapped volumes. [1]

The goal of our work is to (i) automatically detect hotspots when they occur in the system (i.e., whenever Equation 2 is violated), (ii) determine a new configuration of the system, which involves moving or swapping volumes, so that the data copying cost as defined in Equation 3 is minimized. As argued earlier, minimizing data copying cost reduces the time needed for a reconfiguration and reduces the overall performance impact on running applications. We assume that such reconfigurations in self-managing storage systems occur in the background *without* incurring any system downtime—the system remains fully operational during the reconfiguration.

3 Cost-aware Object Remapping

This section first provides a brief overview of techniques to reconfigure a system upon formation of a hotspot. We then present our technique to minimize data copying overhead during reconfiguration.

3.1 Background

Consider a self-managing storage system that detects a hotspot. The system must then determine a new map-

[1] An optimization is to only move the used storage space within a volume rather than the entire volume, in which case the cost as defined in Equation 3 is adjusted accordingly.

ping of data partitions to arrays to alleviate the hotspot. Several techniques can be employed for this purpose:

Bin Packing: One possible approach is to consider the new storage and bandwidth requirement of each logical volume and determine a new mapping of volumes to arrays from scratch using bin packing heuristics [3]. Bin packing can be randomized, where a random permutation of volumes is generated and objects are assigned randomly to arrays in permutation order. The process is repeated until a valid assignment is obtained (a valid assignment satisfies both Equations 1 and 2). An alternate bin packing heuristic is to use *best-fit*, where a random permutation is generated and volumes are assigned to an array using best-fit. Observe that both heuristics compute a new mapping from scratch and are oblivious of the *current* mapping of volumes to arrays; consequently, they may result in significantly larger data copying overhead than is necessary to alleviate the hotspot.

BSR-based Approach: *Bandwidth-to-space ratio (BSR)* has been used as a metric for video placement [5, 7]. It is based on knapsack heuristics where objects are chosen based on value per unit weight. In our case, volumes are ordered in decreasing order of their bandwidth to storage ratio. Underloaded arrays are ordered by *spareBSR*, which is the ratio of spare bandwidth to spare storage space. Volumes are chosen in BSR order and assigned to arrays with the highest spareBSR. Choosing volumes in BSR order ensure better utilization of the system bandwidth per unit storage space and leads to a tighter packing. Like before, the heuristic determines a mapping from scratch and is oblivious of the current mapping.

Randomized reassignment: While the previous two approaches are *cost-oblivious*, we can modify the bin packing approach to take the current mapping into account; the approach incrementally modifies the current configuration until the hotspot is eliminated. The current configuration is assumed to be the initial configuration; a random permutation of volumes on only overloaded arrays is considered and these objects are assigned to underloaded arrays in permutation order. The process continues until sufficient load “shifts” from overloaded to underloaded arrays to remove the hotspot. A limitation of the approach is that it picks volumes based on a random permutation and does not explicitly attempt to reduce data copying overhead. A second limitation is that it only moves volumes from overloaded arrays to underloaded ones and does not consider the possibility of *swaps*, where two volumes are swapped. Thus, an entire set of possible solutions is ignored by the approach.

Next, we present a new approach that addresses this drawback.

3.2 Displace and Swap

Displace and Swap (Dswap) is a greedy data migration technique that alleviates hotspots by (i) using the current configuration to incrementally determine a new load-balanced configuration, (ii) using bandwidth-to-space ratio as a guiding metric to determine which volumes to move and where, and (iii) considering both volume moves and swaps as possible solutions for determining the new configuration. The key idea is to move one or more volumes from overloaded to underloaded arrays or swap heavily loaded volumes from overloaded/ arrays with less loaded volumes on underloaded arrays. BSR is used to guide the selection of volumes and maximize the reduction in overload per unit data moved (which in turn reduces data copying overhead).

Our approach involves two key steps: (i) *displace* where volumes from overloaded arrays are simply moved to free space on underloaded ones, and (ii) *swap*, where volumes between overloaded and underloaded arrays are swapped. The displace step is always considered before considering swaps, since one-way moves (displace) typically involves less data copying than two-way swaps of volumes. [2] Next we present the details of the displace and swap steps.

3.2.1 Displace

The displace step attempts to move volumes from overloaded arrays to unused storage space on underloaded ones. All arrays that see a hotspot (i.e., violation of Equation 2) are considered. Any underloaded array with non-zero unused storage space is a potential destination for overloaded volumes. Overloaded arrays are considered, one at a time, in decreasing order of overload—the magnitude of the overload is given by $(\Sigma_i b_i c_{ij} - \tau_j * \mathcal{B}_j)$. Within each overloaded array, volumes are considered for possible relocation in decreasing order of their BSR. Finally, the set of possible destination arrays are considered in decreasing order of spare BSR (spare bandwidth to spare storage space ratio).

The displace step works as follows. It picks the array with highest overload. Since the goal is to remove the excess load from the array while moving the least amount of data, it considers volumes with the highest BSR values for possible relocation. To do so, it first constructs a set $\mathcal{R}$ which comprises the k highest BSR volumes, such that relocating these volumes from the array eliminates the hotspot. The set is constructed by incrementally adding volumes to $\mathcal{R}$ in BSR order until the total load reduction causes the hotspot to disappear. The set of logical volumes in $\mathcal{R}$ are precisely the ones that must be moved to

[2] One way volume moves are also preferable to two way swaps since they do not require any scratch space. Swapping volumes between arrays may require use of temporary scratch storage space.

underloaded arrays. To determine a new mapping, volumes in $\mathcal{R}$ are considered in BSR order, and each volume is mapped to an underloaded array with the highest spareBSR, so long as neither the storage constraint nor the bandwidth constraint—Equations 1 and 2 —are violated on the underloaded array. Once a new mapping is determined for all volumes in $\mathcal{R}$, the algorithm moves on to the next most overloaded array and repeats the process. The actual data migration is triggered only once a new mapping is determined for volumes on *all* overloaded arrays (i.e., only after a complete solution is found).

In some cases, it may not be feasible to map all volumes in $\mathcal{R}$ to underloaded arrays (since sufficient spare storage space or bandwidth may be unavailable). Our technique also implement two optimizations where it considers additional volumes that were not initially present in the set $\mathcal{R}$ for possible relocation. A detailed discussion of these optimizations is omitted due to space constraints [16].

3.3 Swap

Displace succeeds only if underloaded arrays have sufficient unused storage to accommodate additional volumes. In the event that sufficient storage space is unavailable, our techniques must resort to volume swaps to alleviate the remaining hotspots. Swaps can be two-way involving only two arrays or can be multi-way involving a k-way swap between arrays. Due to the computational complexity of searching for multi-ways swaps, we only consider two ways swaps in our current work.

As in displace, *BSR* is used as the metric to guide the selection of volumes. The key idea is to choose the *highest BSR* volumes from the *most overloaded* array and attempt to swap them with the *lowest BSR* volumes on the *most underloaded* array. Doing so yields the maximum reduction in load per unit data moved and reduces data copying overheads. Choosing the most underloaded array as a destination increases the probability of finding valid swaps.

The swap step works as follows. Consider the most overloaded array and the most underloaded array in the system. First, volumes on the overloaded array are sorted in decreasing BSR order, while volumes on the underloaded array are sorted in increasing BSR order. The process then constructs two sets O and U for a possible swap. These sets are constructed by incrementally adding volumes from the two arrays, one at a time, in sorted order until the following constraints are satisfied:
Constraint C_1: There is at least a certain minimum amount of load reduction on the overloaded array. Swaps that yield less than a threshold reduction in overload are not useful.

$$b_O - b_U \geq \gamma \cdot \mathcal{B}_O \qquad (4)$$

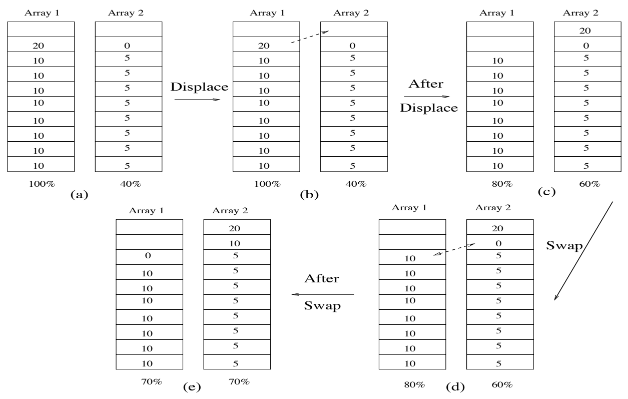


Figure 2: Illustration of Displace and Swap.

where b_O and b_U denote the total bandwidth needs of volumes in the two sets, and the difference represents the load reduction on the overloaded array. $\mathcal{B}_O$ is the load on the overloaded array and γ is a pre-defined threshold.
Constraint C_2: The swap should not violate the storage and bandwidth constraint on the underloaded array (Equation 1 and 2).
Constraint C_3 : The swap should not violate the storage constraint on the overloaded array.

If sets O and U satisfying the above three constraints are successfully found, then the corresponding volumes are marked for a possible swap. The swap step repeats the above process until the hotspot is completely removed from the overloaded array. The swap step then moves onto the next overloaded array and repeats the process.

Additional optimizations that increase the probability of finding valid swaps by skipping certain high storage volumes in the sorted BSR order are described in [16].

Example *The figure 2 illustrates how displace and swap works. Figure (a) shows two arrays with bandwidth utilizations of 100% and 40%, respectively. Each box with a number indicates a volume and an empty box indicates unallocated space. The number in a box indicates the bandwidth requirement of the volume. For simplicity, all volumes are assumed to be of unit size; so the bandwidth requirement of a volume is also its BSR. The bandwidth overload threshold τ is assumed to be 75% for both the arrays. As Array 1 is overloaded the* displace and swap *algorithm proceeds as follows.*

The displace step is invoked first as the underloaded array has one unit spare space.
Displace*: Figures (b) and (c) illustrate a volume being moved from Array 1 to Array 2. The volume selected is one with the maximum BSR.*

Since Array 1 is still overloaded after the displace step, the swap step is invoked.
Swap*: Figures (d) and (e) illustrate a volume with BSR*

10 being swapped with a volume with BSR 0. This, a high BSR volume gets swapped with a low BSR volume.

At this point, the hotspot is eliminated and the algorithm terminates.

4 Automated Hotspot Detection

In this section, we present an automated measurement technique to detect hotspots when they occur in the system; the technique automatically invokes our displace and swap algorithm upon hotspot detection. Our technique continually monitors the bandwidth utilization on each array and flags a hotspot if the bandwidth constraint is consistently violated on an array.

Load monitoring: Our technique uses bandwidth utilization on an array as an indicator of its load. The bandwidth utilization is computed as the average utilization of disks in the array. Since multiple volumes are typically mapped onto an array, each contributes a certain fraction of the total utilization, and the workload seen by each individual volume must be monitored to derive the total array utilization.

Consider a logical array with d disks. Let v volumes be mapped onto the array. Each volume is assumed to be striped across all disks of the array, and it is assumed that the RAID level of the array is known a priori. Let r_i denote the mean request rate seen at volume i over some measurement interval T. Let s_i denote the mean request size observed during the measurement interval. Further, let S_i denote the stripe unit size (or the block size) for volume i. Since each volume is striped, a request of size s_i will typically access $\delta = \lceil \frac{s_i}{S_i} \rceil$ disks.[3] Then the array utilization due to a single request is $\frac{\delta * (t_{seek} + t_{rot} + S_i * t_{xfr})}{D * T}$, where t_{seek}, t_{rot} and t_{xfr} denote the seek, rotational latency and transfer time per byte of an underlying disk.

Given a request arrival rate of r_i, volume i will see $r_i * T$ requests over the measurement interval T. Hence, the total array utilization is simply the summation of the utilization due to each volume and is given as

$$\sum_{i \in v} \frac{r_i * \delta * (t_{seek} + t_{rot} + S_i * t_{xfr})}{d} \tag{5}$$

Our monitoring module computes the utilization of each array during each measurement interval T. It maintains a history of utilizations observed on each array over a long period (e.g., a day or a week).

Observe that the load monitoring modules requires hooks in the OS kernel to measure the mean request rate and request size for each volume in the storage system, which we discuss further in Section 5.

Hotspot detection: Given a time series (history) of utilizations seen at each array, a hotspot is said to be present if the bandwidth constraint in Equation 2 is violated for a certain percentage of the observations. For instance, when the bandwidth constrain is violated for more than 75% of the observations. The threshold used to flag a hotspot is a configurable parameter; small values aggressively alleviate hotspots, while larger values require an overload to persist over a longer period before data migration is triggered. Upon hotspot detection, our displace and swap technique is invoked to determine a new mapping of volumes to arrays and data migration is triggered to actually move or swap volumes.

5 Implementation Considerations

We have implemented our techniques in the Linux kernel version 2.6.11. Our prototype consists of two components: (i) kernel hooks to monitor request sizes and request rates seen by each logical volume, and (ii) a reconfiguration daemon which uses statistics collected in the kernel to estimate bandwidth requirements, computes a new configuration if a hotspot is detected, and triggers volume migration.

Our prototype was implemented on a Dell PowerEdge server with two 933 MHz Pentium III processors and 1 GB memory that runs Fedora Core 2.0. The server contains an Adaptec 3410S U160 SCSI Raid Controller Card that is connected to two Dell PowerVault disk packs which comprised 20 disks altogether; each disk is a 10,025 rpm Ultra-160 SCSI Fujitsu MAN3184MC drive with 18 GB storage.

Our kernel implementation involves adding appropriate code and data structures to enable collecting statistics for each volume. The Linux 2.6 kernel uses `bio` as the basic descriptor for I/Os to a block device. Upon I/O completion, the `bio_endio` routine is invoked by the device interrupt handler; request-level statistics are gathered by modifying this routine. Since the Linux logical volume manager (lvm) creates a separate device identifier for each logical volume, our statistics routines use the device identifier to gather volume-specific statistics. Our implementation does not currently distinguish between hits in the array controller cache and requests that actually trigger disk I/Os; it is possible to infer cache hits by monitoring request completion times—hits see low completion times and can be intelligently filtered out when computing array utilizations.

Our reconfiguration daemon periodically makes a system call to query per-volume statistics from the kernel. These statistics are then used to compute per-array utilizations. We also provide two additional system calls

[3]In practice, the number of disks accessed by a request δ can not exceed d due to wrap-around and the actual RAID-level of the device will constrain it further (e.g., RAID-5 implies a wrap-around in $d-1$ disks due to the presence of parity; RAID-1 or mirroring uses only d/2 disks and so on).

to selectively enable and disable statistics collection for each volume.

If the daemon detects a hotspot, it invokes our displace and swap algorithm to determine a new configuration. Migration of volumes is then triggered using the *pvmove* tool of the Linux volume manager. This tool enables a volume to be migrated online, while being used. Thus, applications continue to work uninterrupted while migration is in progress, barring some minimal impact due to the ongoing data copying.

6 Experimental Evaluation

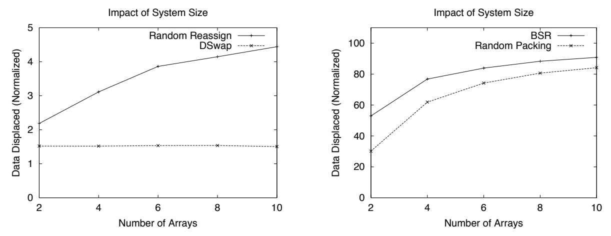
This section evaluates the efficacy of our techniques using simulations and prototype evaluation. Since our prototype is limited by the hardware configuration, we use simulations to evaluate the performance for a variety of system configurations.

6.1 Simulation Results

We constructed a simulator that implements all of the algorithms discussed in Section 3.1, in addition to our displace and swap technique. For each strategy, the simulator computes the cost of eliminating hotspots in terms of the data copying overhead.

The default storage configuration used in our simulations comprised four logical arrays, each with twenty 18 GB disks. Each logical array in the system is configured with an initial storage space and bandwidth utilization of 60% and 50%, respectively. To achieve a particular storage space utilization, volumes are assigned to an array until the desired utilization is reached. Volume sizes are chosen uniformly in the range [1 GB,16 GB] and are a multiple of 0.25GB. To achieve a particular bandwidth utilization, volumes are initially assigned bandwidth usage in proportion to their sizes. A random permutation of these bandwidth usage is then generated and mapped onto the volumes, yielding different BSR values for different volumes. The default system parameters result in an average of 25 volumes per array, and around 100 volumes in the system.

To create hotspots, the simulator increases the bandwidth utilization on half of the volumes on an array—this is done by randomly picking an array from the chosen half and increasing its utilization by a certain amount until the desired magnitude of overload is reached. For our experiments, the default bandwidth violation threshold τ was chosen to be 80%. Each experiment consists of 100 runs and the normalized data displaced over these runs is reported as the total data moved as a percentage of the total data in the system.



(a) Cost-aware (b) Cost-oblivious

Figure 3: Impact of system size.

6.1.1 Impact of System Size

Our first experiment studies the impact of system size on reconfiguration overhead. We vary the system size—the number of arrays—from two to ten (i.e., 40 to 200 disks) and determine the normalized cost of eliminating hotspots over 100 runs. Figures 3(a) depicts the data copying overheads for the Random Reassignment and our Dswap approach, both of which are cost-aware, while Figure 3(b) depicts the cost for Randomized bin packing and BSR, both of which are cost-oblivious and reconfigure the system from scratch.

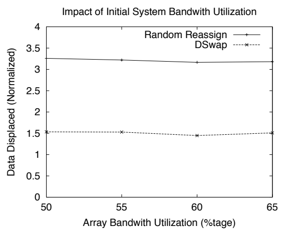
Figure 3(a) shows that *DSwap* outperforms *Random Reassignment* by factors of two to three. Moreover, the normalized reconfiguration costs for Dswap remains constant over a range of system sizes, while it increases for the latter. Since Dswap chooses volumes from overloaded arrays carefully based on BSR values, the normalized cost is not sensitive to system size (the data copying overheads increase in proportion to system size, resulting in constant normalized cost). In Random reassignment, however, increasing the system size increases the number of volumes to choose from, which also increases the likelihood of making sub-optimal decisions.

Figure 3(b) shows the cost of the reconfiguration for the cost-oblivious approaches. Since both approaches reconfigure the system from scratch, the cost of reconfiguration is higher by more than an order of magnitude when compared to that of the cost-aware approaches. Since the probability of a volume being moved to a new array increases with system size, the normalized data copying overheads increase with system size.

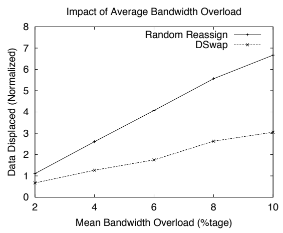
6.1.2 Impact of System Bandwidth Utilization

Next, we study the impact of the bandwidth utilization on the cost of reconfiguration. Figure 4(a) and 4(b) show the impacts of the initial array utilization and the magnitude of overload, respectively, on reconfiguration cost.

Figure 4(a) shows that as the initial array utilization is increased from 50% to 65%, the normalized cost remains unchanged for both approaches. This is because

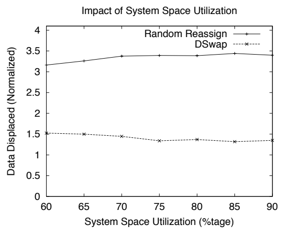


(a) Initial utilization

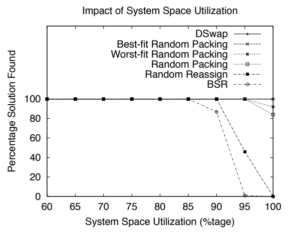


(b) Magnitude of overload

Figure 4: Impact of bandwidth utilization.



(a) Cost



(b) % Success

Figure 5: Impact of storage space utilization.

increasing the initial array utilization merely increases the initial bandwidth requirement of all the volumes proportionately. This reduces the fraction of volumes that can be reassigned to the underloaded array without any constraint violations. The normalized cost of reconfiguration, however, does not change significantly, as each object reassignment, on the average, now displaces more bandwidth. *DSwap* results in a lower cost as compared to the *Random Reassign* approach due to its careful BSR-based selection of volumes.

Figure 4(b) shows that an increase in the magnitude of overload from 2% to 10% increases the cost of reconfiguration for both approaches. This is because, on an average, more data needs to be displaced for higher overloads. Again Dswap performs better due to its BSR-based selection, which dissipates more load per unit data moved.

6.1.3 Impact of Storage Utilization

Next we study the impact of varying the storage utilization on the cost of reconfiguration. Figure 5(a) shows that the normalized cost of reconfiguration remains roughly unchanged for both Random Reassignment and Dswap even as the storage utilization is increased from 60% to 90%. A large storage utilization implies a larger number of volumes per array. Since the bandwidth utilization is fixed, this results in a smaller load per volume. While this may require more volumes to be relocated to eliminate the same overload, the corresponding increase in space utilization causes the normalized data copying cost to remain largely unchanged.

Figure 5(b) shows the success rates of various strategies in eliminating a hotspot with increasing storage utilization (and decreasing unused storage space). The BSR approach begins to fail when space utilization is around 85%, the Random reassignment begings failing at around 90% utilization. The latter assigns volumes from overloaded to underloaded arrays and a decrease in unused space reduces the likelihood of relocating volumes. Randomized bin packing and bin-packing based on worst-fit begin failing at around 97% utilization. Only Dswap and best-fit-based bin packing continue to be successful even at near 100% utilization (i.e., no free storage space in the system). Best-fit succeeds by mapping volumes to arrays based on their fit, while Dswap can resorts to swaps in the absence of free space; recall that prior results have already shown the lower cost of Dswap when compared to best-fit, which reconfigures the system from scratch.

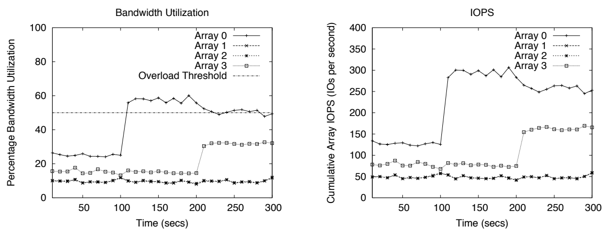
6.2 Prototype Evaluation

This section demonstrates the effectiveness of our approach using our Linux prototype. Our goal is two-fold (i) to demonstrate the efficacy of our kernel measurement techniques in identifying hotspots, and (ii) to demonstrate the effectiveness of the reconfiguration daemon in determining a new configuration, while minimizing data copying cost and performance impact on applications.

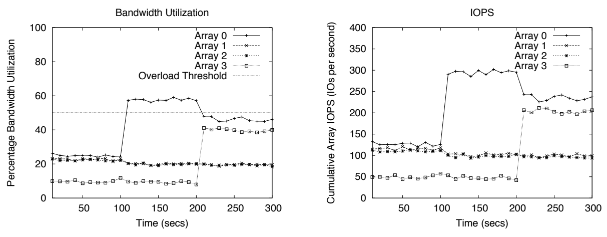
The characteristics of the host and the storage system used by our prototype were described in Section 5. We partition the twenty disks in the system into five logical arrays, each comprising four disks. The stripe unit size is chosen to be 16KB. We construct 14 partitions on each array, each of size 4GB. Logical volumes can then be constructed using one or more partitions on an array; the volume size is thus a multiple of 4GB. We configure four arrays in this fashion, and leave the fifth array unused as scratch space for swapping volumes.

We use a synthetic workload to control the load seen on each volume. The workload on a volume is defined using two parameters: the concurrency factor N and the mean think time IA. The concurrency factor defines the number of concurrent clients issuing requests to the volume, while the think-time determines the mean inter-arrival time between successive requests from a client; think times of requests are assumed to be exponentially distributed. Each request is assumed to access a random block on the volume and the request size is fixed at 16KB. Using such a synthetic workload allows us careful control over the load seen by each volume.

Our experiments use a utilization threshold τ of 50% to denote overload. The measurement interval for our



(a) Spare storage space.



(b) No spare storage space.

Figure 6: Uniform object size

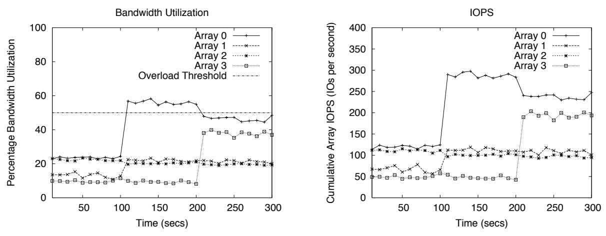


Figure 7: Variable volume size; no spare storage space

statistics collection was defined to be 10s. A hotspot is flagged if more than 70% of the measurements over some window show array utilizations exceeding τ. A hotspot invokes our Dswap routine, which in turn triggers volume migration. The performance impact during volume migration can be minimized by controlling the rate of data copying as explained in [11].

We conducted two sets of experiments, one where volume sizes are identical, and another where heterogeneous volumes are present in the system.

6.2.1 Homogeneous Volumes

We begin by considering homogeneous volumes sized 4GB each. We first consider a system with unused storage space and then consider one with no unused storage space; the former scenario triggers the displace step, while the latter requires volume swaps. For simplicity of experimentation, we assume the reconfiguration daemon checks for hotspots every 100s; in practice, hotspots would need to persist for significantly longer periods of hours or days before a reconfiguration is triggered.

Our first experiment configures the first three arrays with fourteen volumes each and the fourth array with only seven volumes (leaving half of its storage space unused). For the first 100s, all volumes are accessed by a workload with concurrency 2; interarrival time of requests was 400ms, 1s, 1s and 500ms on the four arrays, respectively. The initial bandwidth utilization of each array and the cumulative average I/O per second (IOPs) are shown in Figure 6(a). As can be seen, the interarrival times govern the array utilizations. The figure shows that the utilization measured by our kernel implementation matches the trend in the average I/Os per second issued by the workload generator, indicating that our kernel implementation correctly tracks the array workload.

At $t = 100s$, a hotspot is introduced on *array 0* by increasing the concurrency factor of seven volumes to nine. Workload of other arrays is kept unchanged. At $t = 200s$, our reconfiguration daemon correctly identifies a hotspot on array 0 and migrates two highest BSR volumes from array 0 to the free space on array 3, which helps eliminate the hotspot. As can be seen, once the migration is complete, the array utilization drops below the threshold level of 50%. We note that the Dswap algorithm makes the optimal choice by migrating the least number of volumes and minimizes the copying overhead.

Next we configure the system with no free space—all arrays are configured with 14 volumes each of size 4GB. The same scenario is repeated, where all arrays are initially underloaded and a hotspot is created on array 0 at $t = 100s$ (see Figure 6(b)). Like before, the reconfiguration daemon successfully detects the hotspot at $t = 200s$ and invokes the Dswap algorithm. The algorithm now triggers swaps and chooses to swap the highest BSR volumes on array 0 with those on the most underloaded array, namely array 3. Once the swap is complete, as shown in Figure 6(b), the array utilization drops below 50% and the hotspot is dissipated.

6.2.2 Heterogeneous Volume Sizes

Next we consider systems with heterogeneous volume sizes. Like in the previous experiment, we consider two scenarios, one where unused storage space is present and another where all arrays are full. The results for the former case are similar to that for homogeneous volume sizes and are omitted due to space constraints. For the scenario where no free space is available, we configure the first two arrays with six volumes each—two volumes each of size 4GB, 8GB and 16GB. The other two arrays are configured with 14 volumes, all of size 4GB each.

Initially the concurrency factor is set to two for all volume workloads. The request interarrival times are set to 300ms, 1s, 1s and 500ms, respectively, for the four arrays. As shown in Figure 7, all arrays are initially underloaded. Further, the utilization estimated by our measurement technique closely tracks the imposed workload measured in IOPs.

At $t = 100s$, we impose an overload on array 0 by increasing the concurrency factor to seven. The workload on array 1 is also increased slightly but not enough to cause a hotspot. At $t = 200s$, our prototype correctly identifies a hotspot on array 0 and invokes Dswap. The algorithm swaps two 4GB volumes from array 0 with two similar sized volumes on array 3, which is underloaded. As can be seem, the technique swaps the smallest size volumes from array 0, minimizing the copying overhead. Further, doing so successfully dissipates the hotspot, as indicated by the array utilization at $t = 300s$. The load on array 3 increases due to the swap but the array still remains underloaded.

6.2.3 Implementation Overheads

Our final experiment quantifies the overheads imposed by our kernel enhancements on application performance. As the number of volumes in the system increase, so do the number of statistics maintained by our in-kernel measurement module. To quantify the overhead of statistics collection, we varied the number of 4GB volumes on each of the four arrays from two to fourteen (i.e., from 8 to 56 volumes system-wide). The workload imposed on each volume had a concurrency factor of two and a think-time of zero (a zero think-time indicates that the next request is issued immediately after the previous one finishes).

Figure 8 plots the cumulative average IOPs seen at each array for varying number of volumes. We run the system with our statistics collection module enabled and repeat the experiment with statistics collection disabled. As can be seen, the mean I/Os per second seen for the two cases are nearly identical, indicating that our kernel implementation has negligible impact on application performance. Further, since the storage system is saturated due to the zero think-times, varying the number of volumes does not significantly impact the I/O completions per second. The slight drop in performance is due to the larger number of volumes, which increases the disk seek overheads (since the disk head needs to seek from one volume to another).

6.3 Summary of Experimental Results

Our results shows that for a variety of overload configurations the Dswap approach outperforms other approaches by a factor of two in terms of data copying overhead. Moreover, the larger the system size or the larger the overload, the greater the performance gap. Results from our prototype implementation show that our techniques correctly measures array utilizations and are effective at detecting hotspots and dissipating them, while imposing negligible overheads on the system.

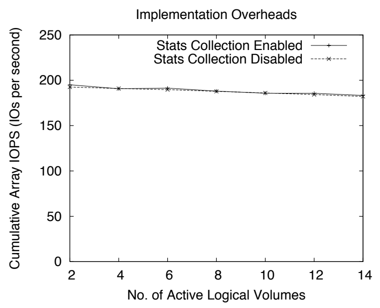


Figure 8: Impact on application performance.

7 Related Work

Algorithms for moving volumes from one configuration to another in the fewest time steps have been presented in [6, 8, 10]. It is assumed that the new final configuration is known *a priori*. In our work, we seek to identify a new final configuration with minimal data movement.

Techniques for initial storage system configuration have been presented in [2, 3]. Our work assumes that the storage system remains online, and presents techniques to reconfigure the system when workload hotspots occur with minimum data movement.

Load balancing at the granularity of files has been considered in [14]. The work assumes contiguous storage space is available on lightly loaded disks to migrate file extents from heavily loaded disks. Our work seeks to achieve load balancing at the granularity of logical volumes and makes no assumptions about the distribution of spare space in the storage system.

Techniques for moving data chunks between mirrored and RAID5 configurations within an array based on their load for improving storage system performance have been proposed in [17]. Our work seeks to achieve improved performance across the storage system by moving logical volumes between arrays.

Disk load balancing schemes for video objects have been presented in [18]. Video objects are assumed to be replicated and load balancing is achieved by changing the mapping of video clients to replicas. In our work, logical volumes are assumed to have no replicas across arrays and load balancing requires identifying a new mapping of data objects to arrays.

Request throttling techniques to isolate performance of applications accessing volumes on shared storage have been studied in [9, 12, 15]. These are complementary to our present work. Finally, techniques for controlling the rate of data migration to prevent any performance impact on foreground application are presented in [11]; however, unlike Dswap, this effort does not consider the issue of reducing the total copying overhead.

8 Concluding Remarks

In this paper, we argued that manual hotspot detection and storage system reconfiguration is tedious and error-prone and advocated the design of a self-managing system to automate these tasks. We argued that existing data migration techniques do not minimize data copying overhead incurred during a reconfiguration, which impacts application performance. We proposed a novel technique that automatically detects hotspots and uses the bandwidth-to-space ratio metric to reconfigure the system while minimizing the resulting data copying overhead. We implemented our techniques into the Linux kernel and conduct a detailed evaluation using simulations and our prototype. Our results showed a factor of two reduction in data copying overhead when compared to other approaches for a variety of system configurations. Our prototype successfully identified system hotspots and eliminated them by incrementally computing a new configuration and without causing any noticeable degradation in application performance. As future work, we plan to extend our techniques to volumes striped across heterogeneous arrays and to devise sophisticated swap techniques that consider multi-way swaps.

References

[1] EMC Symmetrix Optimizer. Available from http://www.emc.com/products/storage_management/symm_optimizer.jsp.

[2] G. Alvarez, E. Borowsky, S. Go, T. 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 (to appear)*, 2002.

[3] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running Circles Around Storage Administration. In *Proceedings of the Usenix Conference on File and Storage Technology (FAST'02), Monterey, CA*, pages 175–188, January 2002.

[4] E. Borowsky, R. Golding, P. Jacobson, A. Merchant, L. Schreier, M. Spasojevic, and J. Wilkes. Capacity Planning with Phased Workloads. In *Proceedings of WOSP'98, Santa Fe, NM*, October 1998.

[5] A. Dan and D. Sitaram. An Online video placement policy based on bandwidth and space ratio. In *Proceedings of SIGMOD*, pages 376–385, May 1995.

[6] E. Anderson et. al. An experimental Study of Data Migration Algorithms. In *Proc. of WAE- International Workshop on Algorithms Engineering, LNCS*, 2001.

[7] Yang Guo, Zihui Ge, Bhuvan Urgaonkar, Prashant Shenoy, and Don Towsley. Dynamic Cache Reconfiguration Strategies for a Cluster-based Streaming Proxy. In *Proceedings of the Eighth International Workshop on Web Content Caching and Distribution (WCW 2003), , Hawthorne, NY*, September 2003.

[8] J. Hall, J. Hartline, A. Karlin, J. Saia, and J. Wilkes. On Algorithms for Efficient Data Migration. In *Proceedings of ACM Symposium on Discrete Algorithms (SODA)*, 2001.

[9] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance Isolation and Differentiation for Storage Systems. In *Proceedings of IWQoS, Montreal Canada*, June 2004.

[10] S. Khuller, Y. Kim, and Y. Wan. Algorithms for data migration with cloning. In *Proceedings of ACM Symposium on Principles of database systems*, pages 27–36, New York, NY, USA, 2003. ACM Press.

[11] C. Lu, G. Alvarez, and J. Wilkes. Aqueduct: Online Data Migration with Performance Guarantees. In *Proceedings of the Usenix Conference on File and Storage Technology (FAST'02), Monterey, CA*, pages 219–230, January 2002.

[12] C. Lumb, A. Merchant, and G. Alvarez. Facade: Virtual Storage Devices with Performace Guarantees. In *Proceedings of FAST'03*, 2003.

[13] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Array of Inexpensive Disks (RAID). In *Proceedings of ACM SIGMOD'88*, pages 109–116, June 1988.

[14] P. Scheuermann, G. Weikum, and P. Zabback. Data Partitioning and Load Balancing in Parallel Disk Systems. *VLDB Journal*, 7(1):48–66, 1998.

[15] V. Sundaram and P. Shenoy. A Practical Learning-based Approach for Dynamic Storage Systems. In *Proceedings of IWQoS'03*, 2003.

[16] V. Sundaram and P. Shenoy. Efficient Data Migration for Load Balancing in Large-scale Storage Systems. Technical Report TR05-51, Dept. of Computer Science, Univ. of Massachusetts, 2005.

[17] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, Copper Mountain Resort, Colorado*, pages 96–108, Decmember 1995.

[18] J. Wolf, P. S. Yu, and H. Shachnai. DASD Dancing- A Disk Load Balancing Optimization Scheme for Video-on-Demand Computer Systems. In *Proceedings of ACM SIGMETRICS'95*, pages 157–166, 1995.