Efficient Data Migration for Load Balancing Large-scale Storage Systems

Vijay Sundaram and **Prashant Shenoy**

Department of Computer Science
University of Massachusetts
{vijay,shenoy}@cs.umass.edu

Abstract

In this paper, we argue that techniques employed to load-balance large-scale storage systems do not optimize for the *scale of the reconfiguration*—the amount of data displaced to realize the new configuration. The scale of the reconfiguration determines the downtime for an offline reconfiguration, or the duration of performance impact on foreground applications for an online reconfiguration. We propose and experimentally evaluate algorithms which take object sizes and the current configuration into account to minimize the amount of data moved during load-balancing. Results from a simulation study suggest that for a variety of system configurations our novel approach reduces the amount of data moved to remove the hotspot by a factor of two as compared to other approaches. The gains increase for a larger system size and magnitude of overload. We also implement our techniques in the Linux kernel. We observe that our kernel measurement based techniques correctly identify workload hotspots. Furthermore, the kernel enhancements do not result in any noticeable degradation in application performance.

1 Introduction

1.1 Motivation

The configuration and management of large scale storage systems is a complex task. While storage has become cheap, the need for data storage in enterprise-scale storage systems continues to spiral. To handle their spiraling storage needs such systems employ multiple storage sub-systems and comprise a large number of storage devices. In these systems, object placement—the mapping of storage objects to storage devices—is crucial as it dictates the performance of the storage system. Consequently, extreme care is taken during capacity planning and initial configuration of such systems [3, 4].

Although the initial configuration may be load-balanced, over time, growth in storage space usage and changes in workload can cause load imbalances and workload hotspots. This in turn may necessitate a reconfiguration.

Hotspots in storage systems can occur for one of two reasons. Incorrect or insufficient workload information during storage system configuration may result in heavily accessed objects being mapped to the same set of storage devices thus resulting in hotspots. Long term workload changes or addition of a new object to a balanced system may also induce hotspots. Hotspots result in increased response times and a loss in throughput for applications accessing the storage system. When hotspots do occur, the mapping of objects to storage devices needs to be revisited to ensure that the bandwidth utilization of all devices is below a certain threshold so that applications see acceptable performance. Such a reconfiguration is undesirable because it is concomitant with a downtime or a potential performance impact on the applications accessing the storage system while the reconfiguration is in progress.

Sophisticated enterprise storage sub-systems come with tools to facilitate the process of load-balancing to address hotspots [1]. These allow for load-balancing to be either carried out manually or in an automated fashion. For manual reconfiguration, administrators use information from a *workload analyzer* component which collects performance data and summarizes the load on the component storage devices. The tool also provides the potential performance impact of moving an object so that the user can make an informed decision. The automated load balancing component, on the other hand, is self-driven, runs continuously, and uses the information from the workload analyzer to swap hot and cold objects when necessary.

Drawbacks of a manual process are that they require human oversight. Moreover, the procedure can be error-prone and human errors during the reconfiguration process may worsen performance. While an automated process addresses these drawbacks, a simple approach which swaps hot and cold objects will work all the time only if objects are of similar size. If objects are of different sizes then more sophisticated strategies are required. This motivates the need for more sophisticated approaches that *search* for a configuration with no hotspots.

Moving the system to a new configuration involves executing a *migration plan*, which is a sequence of object moves. The reconfiguration itself could be carried out either *online* or *offline*. In both cases, the scale of the reconfiguration i.e., the amount of data that needs to be displaced, is of consequence. While for an offline reconfiguration the scale of the reconfiguration determines the duration of the reconfiguration and hence

the downtime, for an online reconfiguration it determines the duration of performance impact on foreground applications.

Existing approaches do not optimize for the scale of the reconfiguration, possibly moving much more data than required to remove the hotspot. This motivates the need for a load-balancing approach that takes sizes of objects and their current mapping to storage devices into account. This is the subject matter of this paper.

1.2 Research Contributions

In this paper, we develop algorithms to minimize the amount of data displaced during a reconfiguration to remove hotspots in large-scale storage systems. This work has led to several research contributions.

Rather than identifying a new configuration from scratch, which may entail significant data movement, our novel approach uses the current object configuration as a *hint*; the goal being to retain most of the objects in place and thus limit the scale of the reconfiguration.

The key idea in our approach is to *greedily* displace excess bandwidth from overloaded to underloaded storage devices. This is achieved in one of two ways, (i) *displace*, which involves reassigning objects from overloaded devices to underloaded ones, and (ii) *swap*, which involves swapping objects between overloaded and underloaded devices. The swap step is useful when the spare storage space on the underloaded devices is insufficient to accommodate any additional objects, and an object reconfiguration, short of a reconfiguration from scratch, would have to entail a swapping of objects, or groups of objects, between storage devices.

To minimize the amount of data that needs to be moved we use the *bandwidth to space ratio* (*BSR*) as a guiding metric. For example, by selecting high *BSR* objects for reassignment in the displace step, we are able to displace more bandwidth per unit of data moved. Here, bandwidth (space) refers to the bandwidth (storage space) requirement of the storage object. We propose various optimizations, including searching for multiple solutions, to counter the pitfalls of a greedy approach.

We also describe a simple measurement-based technique for identifying hotspots and for approximating per-object bandwidth requirements.

Finally, we evaluate our techniques using a combination of simulation studies and an evaluation of an implementation in the Linux kernel. Results from the simulation study suggest that for a variety of system configurations our novel approach reduces the amount of data moved to remove the hotspot by a factor of two as compared to other approaches. The gains increase for a larger system size and magnitude of overload. Experimental results from the prototype evaluation suggest that our measurement techniques correctly identify workload hotspots. For some simple overload configurations considered in the prototype our approach identifies a load-balanced configuration which minimizes the amount of data moved. Moreover, the kernel enhancements do not result in any noticeable degradation in application performance.

The rest of the paper is structured as follows. In Section 2, we describe the problem addressed in this paper. Section 3 presents object remapping techniques for load-balancing large scale storage systems. Section 4 presents the methodology used for measuring object bandwidth requirements and for identifying hotspots. Section 5 presents the details of out prototype implementation and Section 6 presents the experimental results. Section 7 discusses related work, and finally, Section 8 presents our conclusions.

2 Problem Definition

2.1 System Model

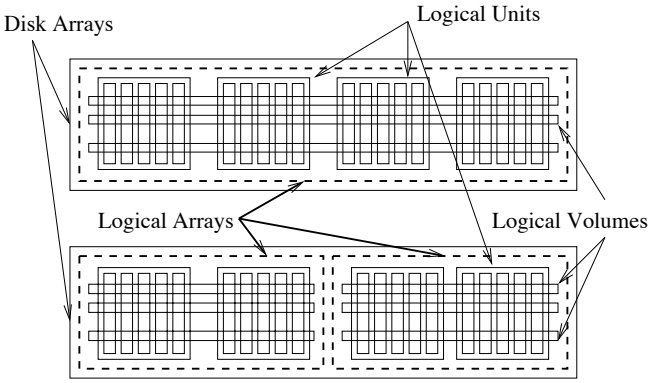
Large scale storage systems consists of a large number of disk arrays. We assume, as is typically the case, that each disk array consists of disks of the same type. Different disk arrays, however, could have disks of different types. The disks in a disk array are grouped into some number of *logical units* (LU); an LU is a set of disks combined using RAID techniques [12].

An object configuration indicates the mapping of storage objects to storage devices. Here, an object is an equivalent of a *logical volume* (LV), such as a database or a file system, and is allocated storage space by concatenating space from one or more LUs. From here on, we use the terms LV and object interchangeably.

In our model, we assume that all the LUs an LV is striped over are similar i.e., they have the same RAID level, and comprise disks of the same type. This is generally true in practice, since it ensures the same level of redundancy and similar access latency for all the stripe units of an LV. We further make the simplifying assumption that if any two LVs have an LU in common, they have all their component LUs in common. This assumption is also generally true in well-planned storage system configurations, as in such a configuration each object is subject to uniform inter-object workload interference on all of its component LUs. With this assumption, the set of LUs an LV is striped over can be thought of as a single *logical device* for load balancing purposes. From here on, we refer to such a logical device as a *logical array* or *array* for short. Figure 1 illustrates the system model.

2.2 Problem Formulation

Assuming the above system model, let us now formulate the problem addressed in this paper. Consider a storage system which consists of n arrays, $A_1, A_2....A_n$. There are m LVs, $L_1, L_2....L_m$ which populate the storage system. Each LV is mapped to a single array. Each array A_j, has storage capacity $\mathcal{S}_j$, and a bandwidth capacity $\mathcal{B}_j$. Similarly, each LV L_i has storage requirement s_i and bandwidth requirement b_i. For a *balanced configuration*, which is defined to be a configuration without any hotspots, it is required that the percentage bandwidth utilization of each array A_j, not exceed some threshold $\tau_j (0 < \tau_j < 1)$. The space and the bandwidth constraint on an



The figure shows two disk arrays each comprising four LUs. Each LU consists of five disks. The disk array on the top comprises of one logical array over which three LVs have been striped. The second disk array comprises of two logical arrays, each comprising of two LUs and three LVs striped over each logical array.

Figure 1: System model.

array A_j is given by the following equations:

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

$$\Sigma_i b_i c_{ij} \leq \tau_j * \mathcal{B}_j \tag{2}$$

Here, c_{ij} is a mapping parameter that denotes whether object i is mapped to array j—c_{ij} equals 1 if array j holds the object i, and is 0 otherwise. Although the space constraint is a hard constraint and cannot be violated, an array may observe a violation of the bandwidth constraint if the bandwidth requirements of the objects mapped to the array increase. If the bandwidth utilization on an array exceeds the corresponding bandwidth threshold, it is considered *overloaded*, otherwise it is *underloaded*.

Moving the system to a new configuration results in a change of mapping parameters. Let c_{ij}^{old} and c_{ij}^{new} denote the mapping parameter for LV i on array j in the old and new configurations, respectively. If $|x|$ denotes the absolute value of x, then we have $\Sigma_j |c_{ij}^{new} - c_{ij}^{old}| = 2$ if the mapping of store i has changed, and is equal to 0 otherwise. The *cost of the reconfiguration*, defined as the amount of data moved to realize the new configuration, is then given by:

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

Let $\mathcal{O}$ be the set of overloaded arrays in a configuration. The bandwidth violation for an overloaded array j is $(\Sigma_i b_i c_{ij}/\mathcal{B}_j - \tau_j)$. The cumulative bandwidth violation, defined as the sum of the bandwidth violation over all overloaded arrays, is then given by:

$$Overload = \Sigma_{j \varepsilon \mathcal{O}} (\Sigma_i b_i c_{ij}/\mathcal{B}_j - \tau_j) \tag{4}$$

Given an object configuration with some overloaded arrays and some underloaded arrays, the goal is to identify a balanced configuration which can be realized at the least cost. Given two new configurations, both of which satisfy the space and bandwidth constraints on all arrays, the one that can be realized at a lower cost is preferable.

For cases where a balanced configuration cannot be found, the goal of load balancing is a policy decision. One may require that $Overload$ (equation 4) be minimized, but when displacing excess bandwidth from overloaded arrays, the bandwidth constraint on the underloaded arrays should not be violated and that the utilization on an already overloaded array should not increase further. In some cases, absolute load balancing may be desirable, thus requiring that the maximum percentage bandwidth violation across all arrays be minimized. We refer to the approaches that adhere to the the former policy as *fair*, and the ones that conform with the latter as *absolute*. Another dimension in this context is the cost. Absolute load balancing may incur a significantly higher cost. A complete evaluation of the tradeoffs of gains (balance achieved) versus cost (amount of data moved) of these policies is beyond the scope of this paper.

In this paper, the goal is to design a reconfiguration algorithm for identifying a balanced configuration which has the least cost.

3 Object Remapping Techniques

There are two kinds of approaches to load balancing, (i) those that reconfigure from scratch, and (ii) those that start with the current configuration and aim to minimize the cost of reconfiguration. We refer to the former class of approaches as *cost oblivious* and the latter as *cost aware*.

In the following, an assignment of an object to an array is said to be *valid* if the new object could be accommodated on the array without any constraint violations (equations 1 and 2).

3.1 Cost Oblivious Object Remapping

In this section, we present two cost-oblivious object remapping algorithms to remove hotspots in large scale storage systems. We first present a randomized algorithm, and then another, which is deterministic in nature.

3.1.1 Randomized Packing

Heuristics based on best-fit bin-packing have been used in [3] for initial storage system configuration. There the goal was to identify a configuration which uses the least number of devices to meet the space and bandwidth requirements of a given set of objects. In our problem, the number of devices is a given, and the goal is to identify a valid packing which can be realized at the least cost. We first present a randomized algorithm and then present two variations of the same.

Initially, all the objects are unassigned. A random permutation of the objects is created, and the objects are assigned to arrays picked at random from the set of all arrays. All arrays may need to be tried for an object in the worst case. If all the objects could be validly assigned to some array, we have a balanced configuration. The procedure could be repeated multiple times, with different permutations of objects, and of multiple trials which result in a balanced configuration, one with

the least cost is chosen. Note that this makes the approach *semi cost aware*.

As opposed to the completely randomized approach, where both the objects and the arrays are chosen randomly, two partly randomized variants of interest are described next. In a best-fit version, of all possible valid assignments for an object, the object is assigned to the array such that new bandwidth utilization across all arrays as a result of this assignment is a maximum. A complementary approach is also possible, worst fit, where of all possible valid assignments, the array picked is such that the new bandwidth utilization across all arrays as a result of this assignment is a minimum. Whereas best-fit may fare better in finding a balanced configuration in bandwidth constrained scenarios, worst fit may yield a configuration with similar bandwidth utilization on all arrays. Consequently, in less bandwidth constrained scenarios, when the arrays have utilization values well below their corresponding threshold, worst fit may be advantageous, since with more headroom, arrays can absorb workload variations better.

3.1.2 BSR-based Approach

Bandwidth to Space Ratio (*BSR*) has been used as a metric for video placement [5]. These derive from the heuristics based on value per unit weight used for knapsack problems. The knapsack heuristic involves greedily selecting items ordered by their value per unit weight in order to maximize the value of the items in the knapsack. The approach described next uses *BSR* as a guiding metric, but as explained later, for slightly different reasons.

The *BSR* of an object is defined to be the ratio of its bandwidth requirement to its space requirement. We define the *spareBSR* of an array as the ratio of its spare bandwidth capacity to its spare space capacity. So, the *spareBSR* of an array is a dynamic quantity which depends on the objects currently assigned to it.

Initially, all the objects are unassigned. Objects are picked in order of their *BSR* from the set of all objects and assigned to arrays picked in order of their *spareBSR* from the set of all arrays. If a valid assignment is found for all the objects, we have a balanced configuration. Note that the *spareBSR* of the array an object is assigned to is updated appropriately after each valid assignment.

The intuition behind using *BSR* as a metric is that assigning high *BSR* objects to arrays with a high *spareBSR* possibly results in a better utilization of bandwidth per unit space in the system, and hence a tighter packing. A tighter packing increases the likelihood of finding a balanced configuration.

3.2 Cost-aware Object Remapping

In this section, we present two cost aware algorithms for searching a balanced configuration. The first of these is a randomized algorithm and the second is a deterministic greedy algorithm. Both approaches start with the current configuration and change the mapping of the objects incrementally until a balanced configuration is achieved. Thus, these approaches use the current configuration as a *hint*, and aim to retain most of the objects in place, possibly resulting in a lower cost of reconfiguration.

3.2.1 Randomized Object Reassignment

This approach is similar in principle to the randomized approach described in Section 3.1.1, except it starts with the current configuration. Given the current configuration, a random permutation of objects on all the overloaded arrays is created. These objects are then assigned, in order, to underloaded arrays picked at random from the set of all underloaded arrays. It is possible that all the underloaded arrays need to be tried before a valid assignment is found for an object. This is done until a fraction $frac$ of objects have been considered, or the system has reached a balanced configuration.

Once an overloaded array becomes underloaded, the objects on the now underloaded array are not considered for reassignment. The overloaded array is now considered as underloaded for load balancing purposes. This procedure could be repeated multiple times, with different permutations, and of multiple trials which result in a balanced configuration, the one with the least cost is chosen.

Again, as opposed to a completely randomized approach, there is a best-fit and a worst fit variant of the algorithm. The variants are similar to that described for the approach in Section 3.1.1.

Drawbacks

In Section 3.1 we presented two approaches which did not take the current configuration into account while searching for a balanced configuration and are typically associated with a large cost of reconfiguration. However, they are useful for initial storage system configuration. The randomized object reassignment approach described above starts with the current configuration. This approach, however, also has two drawbacks:

1. *Possibly high reconfiguration cost*: In this approach, the object which is to be reassigned, is picked at random. Since, the search is not exhaustive it can still result in a large amount of data being moved or may fail to find a balanced configuration.

   Even though an exhaustive search is not feasible, choosing an object as well as the array to which it is to be assigned carefully, taking into account their respective space and bandwidth attributes, could be beneficial.

2. *Simple reassignment*: If the storage system does not have the right combination of spare space and spare bandwidth on the constituent arrays, a simple reassignment of objects may not yield a balanced configuration. Barring a reconfiguration from scratch, which may entail a high cost, a low cost reconfiguration in such scenarios would have to involve *swapping* objects, or groups of objects,

between arrays. The diverse space and bandwidth requirements of the objects, coupled with diverse space and bandwidth constraints on arrays that comprise the storage system, makes this non-trivial.

In the following section, we present an approach to address these drawbacks.

3.2.2 Displace and Swap

The key idea in this approach is to *greedily* displace excess bandwidth from overloaded arrays to underloaded arrays. The goal is to identify a set of objects, while taking into account their sizes, that need to be moved from their original location in order to attain a balanced configuration. *BSR* is used as a guiding metric in order to minimize the amount of data that needs to be displaced. Here, by object size we refer to the storage space requirement of an object.

There are two basic steps which comprise this approach. The first is referred to as *displace* and involves reassigning objects from overloaded arrays to underloaded arrays. The second step, referred to as *swap*, involves swapping objects between overloaded and underloaded arrays. The second step is invoked only if the first step alone does not yield a balanced configuration. The goal is to first offload as much excess bandwidth on an overloaded array using one way object moves (displace), and if this does not suffice, search for two way object moves (swap). The intuition is that one way object moves, on the average, would require less data movement than a solution involving two way object moves. One way object moves are also preferable to two way object moves as they do not require any scratch space [1] to achieve the reconfiguration.

I. Displace

In this step, the goal is to use any spare space on the underloaded arrays to accommodate objects from overloaded arrays and thus offload excess bandwidth. Only underloaded arrays with spare space are considered as potential destinations during object reassignment.

Since the goal is to remove excess bandwidth from each overload array while moving the least amount of data, we consider objects from each overloaded array one by one. This allows us to optimize for the amount of data displaced from each overloaded array.

The overloaded arrays themselves could be considered in any order. To achieve a balanced configuration, the bandwidth utilization on all the overloaded arrays needs to be reduced below the corresponding threshold. So, we consider the overloaded arrays in descending order of the magnitude of bandwidth violation ($\Sigma_i b_i c_{ij} - \tau_j * \mathcal{B}_j$). This has the advantage that if the displace step is unable to identify a balanced configuration, there is less bandwidth that needs to be moved off each overloaded array, on the average, in the swap step.

[1] Swapping objects between arrays with little spare storage space may require using scratch storage space.

Finally, for a given overloaded array, objects on the array are considered for reassignment in descending order of their *BSR*. This is in order to minimize the amount of data displaced, as of all objects, the object with the maximum *BSR* displaces the most bandwidth per unit of data moved. The destination underloaded array for reassigning an object is chosen to be the one with the maximum *spareBSR*. The reason is similar to that for the approach in Section 3.1.2. This completes the essence of the displace step.

Object reassignments that remove the hotspot on an overloaded array could be single-object or multi-object. Any valid single object reassignment that can remove the hotspot on the overloaded array is referred to as a *soloSoln*. Any reassignment comprising multiple objects that removes the hotspot is referred to as a *grpSoln*. Any reassignment comprising one or more objects that is not able to remove the hotspot is referred to as a *semiSoln*. We refer to both *grpSoln* and *semiSoln* as *soln* for short.

It is possible that choosing objects for reassignment strictly in order of *BSR*, as described above, results in a *soloSoln* appearing as a part of a *grpSoln*. So, we identify all *soloSoln*s before searching for *grpSoln*s.

Identifying a soloSoln: Any object on the overloaded array that can be validly assigned to some underloaded array, and also removes the hotspot, classifies as a *soloSoln*. Any object that can be validly assigned, but does not remove the hotspot, is put in a set $\mathcal{R}$. The set $\mathcal{R}$, which is devoid of *soloSoln*s, is used to identify *grpSoln*s in the next step.

A minor optimization is possible here. If the set $\mathcal{R}$ consists of objects all of which are larger than the smallest size *soloSoln*, there is no need to execute the following. This is because any *grpSoln* would only have a higher cost.

Identifying a soln: In this step we search for a *grpSoln* using *BSR* as the guiding metric. Given a set $\mathcal{R}$ of objects, objects picked in descending order of *BSR* are assigned to underloaded arrays chosen in descending order of *spareBSR*. This is done until either all the objects on the overloaded array have been considered, or the set of reassignments so far is able to remove the hotspot. If the hotspot could be removed, we have a *grpSoln*, else we have a *semiSoln*.

In the above step, for identifying a *soln*, the objects were selected greedily based on their *BSR*. However, such a greedy approach could make some wrong choices. These could result in (i) a higher cost solution, or (ii) inability to remove the hotspot on the overloaded array. While an exhaustive search is infeasible, the following optimizations try to address at least some of the wrong choices.

These optimizations essentially involve questioning the choice of each object that comprises the *soln*. Any *soln* can be thought of as being comprised of two parts. One, the highest *BSR* object, referred to as the *root*. All the remaining objects

in the *soln*, if any, comprise the second part. Whereas the first optimization questions the choice of the root, the second questions the choice of each of the remaining objects that comprise the *soln*.

Also, while improving a *grpSoln* requires finding another with a lower cost, improving a *semiSoln* means finding a *grpSoln* or another *semiSoln* which displaces more bandwidth.

Optimization 1: Identifying multiple solns: In this optimization, we identify *soln*s with different elements in the set $\mathcal{R}$ as root. Note, that for a given root only objects with a lower *BSR* than the root are considered for reassignment. This optimization gives us multiple *soln*s. The number of such *soln*s equals the number of objects in the set $\mathcal{R}$.

Optimization 2: Backtracking: This optimization involves backtracking on the remaining objects that comprise a *soln*. We employ this optimization to improve each of the *soln*s identified in the optimization above. Each backtracking step involves searching for a new *soln* while not considering an object that is part of the *soln*. This is done for all the objects that comprise the *soln* excepting the root (the root has been optimized for in the previous step).

If backtracking results in a better *soln*, backtracking on the previous *soln* is discontinued and restarted for this new *soln*. It is possible that this procedure continues to yield successively better *soln*s. To limit the computational costs we explore only a constant number of these.

Note that the above optimizations result in a strategy which lies somewhere between a purely greedy approach and one that exhaustively considers every combination.

If the above results in multiple *grpSoln*s or *soloSoln*s, the one with the least cost is chosen[2]. If, however, the above only results in object reassignments which reduce the bandwidth violation on the overloaded array (i.e., only *semiSoln*s), the one which displaces the most bandwidth is chosen[3]. The mapping parameters (c_{ij}s) for the objects to be reassigned are adjusted appropriately. Note that this modified configuration serves as the starting configuration for the next overloaded array considered.

Once all the overloaded arrays have been considered, and the system is still not balanced, the swap step, which is described next, is invoked.

II. Swap

Displace works only when there is sufficient storage space on the underloaded arrays to accommodate objects from the overloaded arrays. In the absence of sufficient spare space, a low cost reconfiguration technique would require swapping objects between arrays. Such swaps could be two-way i.e., involve two arrays, or they could be multi-way. In this paper, we describe a strategy for identifying two-way swaps.

In this step, the goal is to identify valid swaps of objects, or groups of objects, between overloaded and underloaded arrays, such that the bandwidth utilization on the overloaded array is reduced. By successively identifying such swaps, we can remove the hotspot on an overloaded array.

BSR is again used as the guiding metric. By swapping high *BSR* objects on an overloaded array with low *BSR* objects on a underloaded array, maximum bandwidth is displaced per unit of data moved.

Swaps are searched for between a pair of an overloaded array and an underloaded array. Since, all arrays need to be underloaded for a balanced configuration, overloaded arrays are considered in descending order of the magnitude of bandwidth violation. For each overloaded array, underloaded arrays are considered in descending order of spare bandwidth. This is done so that possibly maximum bandwidth is displaced for each pair considered.

The diverse space and bandwidth attributes of the objects and arrays make identifying valid swaps non-trivial, so we use a simple greedy approach guided by the *BSR* of the objects. Before we describe how a swap is identified, let us define what classifies as a valid swap.

Valid swap: While a swap is valid if it does not violate the constraints on the underloaded array and decreases the bandwidth utilization on the overloaded array. It is not useful if this decrease is not significant. So, we define a parameter *ufrac* which quantifies the utility of a swap. Let bw_O and bw_U be the cumulative bandwidth requirement of the sets of objects from the overloaded and underloaded array, respectively, which are to be swapped. Then for a swap to be valid we require that:

$$bw_O - bw_U \geq ufrac * bw_U \tag{5}$$

In other words, the decrease in bandwidth on the overloaded array as a fraction of the bandwidth moved off the underloaded array should exceed a certain minimum.

We classify the constraints that need to be satisfied for a swap to be valid as follows:
Constraint $\mathcal{C}_1$: The swap should satisfy the bandwidth and space constraints on the underloaded array.
Constraint $\mathcal{C}_2$: The swap should have a certain minimum utility (equation 5).
Constraint $\mathcal{C}_3$: The swap should satisfy the space constraint on the overloaded array.
We now describe the approach for identifying a valid swap.

Identifying a valid swap: While simply considering all pairs of objects, one each from an overloaded array and an underloaded array at a time may not result in any valid swap, considering every combination of objects from the two arrays is infeasible. We present a simple greedy approach to swap

[2]Ties are broken by choosing the one which displaces the least bandwidth as this leaves more spare bandwidth on the underloaded array to accommodate future object moves.

[3]In this case, ties are broken by choosing the semiSoln with the least cost.

the equivalent of a high *BSR* object from the overloaded array with the equivalent of a low *BSR* object from the underloaded array. Identifying such a swap also displaces more bandwidth per unit of data moved.

To identify such a swap, objects on the overloaded and underloaded arrays are sorted in descending and ascending order of *BSR*, respectively, to give sets $\mathcal{L}_{lv}^{O}$ and $\mathcal{L}_{lv}^{U}$, respectively. First pairs of objects from these two ordered sets are considered. Each object in $\mathcal{L}_{lv}^{U}$ is considered, in order, for each object in $\mathcal{L}_{lv}^{O}$, in order. If a pair meets the constraints for a valid swap objects are swapped.

If after considering all pairs the array is still overloaded, we seek to identify contiguous sets of objects from these two ordered sets which constitute a valid swap. Note, that these contiguous sets of objects are the equivalent of a high *BSR* and low *BSR* object, respectively.

Ideally it is desirable that contiguous sets of objects from these two sets be identified. However, it is possible that no such sets can be identified that satisfy all the constraints for a valid swap. So, in the procedure described below we first (step 1) identify contiguous sets that satisfy two of the constraints; if these contiguous sets do not satisfy the third constraint, possibly non-contiguous objects are picked in order to meet the constraint (step 2).

Let $\mathcal{L}_{sw}^{O}$ and $\mathcal{L}_{sw}^{U}$ denote the sets of objects from the overloaded and underloaded arrays, respectively, that are to be swapped.

1. *Satisfy* $\mathcal{C}_1$ *and* $\mathcal{C}_2$: Contiguous elements from the ordered sets $\mathcal{L}_{lv}^{O}$ and $\mathcal{L}_{lv}^{U}$, respectively, are incrementally added to the sets $\mathcal{L}_{sw}^{O}$ and $\mathcal{L}_{sw}^{U}$, respectively, until $\mathcal{C}_1$ and $\mathcal{C}_2$ have been satisfied. This gives a valid swap if $\mathcal{C}_3$ is also satisfied.

2. *Satisfy* $\mathcal{C}_3$: If $\mathcal{C}_3$ has not been satisfied, additional objects from the set $\mathcal{L}_{lv}^{O}$, picked in order, are added to the set $\mathcal{L}_{sw}^{O}$; an object is added only if it does not result in a violation of $\mathcal{C}_1$ or $\mathcal{C}_2$. Objects are added until $\mathcal{C}_3$ has been satisfied. This may result in $\mathcal{L}_{sw}^{O}$ being comprised of non-contiguous elements from the ordered set $\mathcal{L}_{lv}^{O}$.

3. Given a valid swap, the ordered sets $\mathcal{L}_{sw}^{O}$ and $\mathcal{L}_{sw}^{U}$ are updated to reflect the swap.

4. If a valid swap was not found, the above steps are repeated but now with the second element in the ordered set $\mathcal{L}_{lv}^{O}$ as the first element added to set $\mathcal{L}_{sw}^{O}$, and so on.

5. Swaps are searched for until the hotspot on the overloaded has been removed.

If after executing this step, there are no overloaded arrays, we have a balanced configuration. Note that this simple greedy approach for swapping contiguous sets of objects between two arrays may be sub-optimal; however, the parameter *ufrac* allows some control over the utility of a swap.

Figure 2 and the following example together illustrate displace and swap.

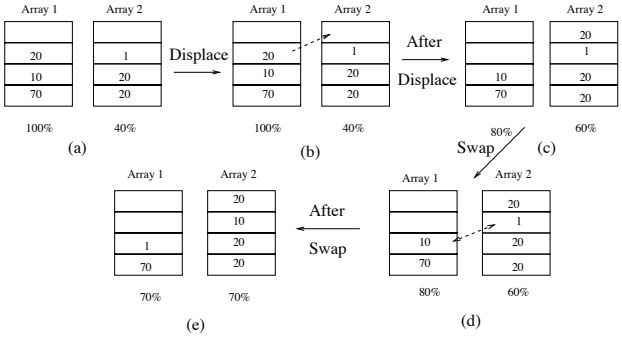


Figure 2: Illustration of Displace and Swap.

Example *The figure illustrates how displace and swap work. Figure (a) shows two arrays with bandwidth utilizations of 100% and 40%, respectively. Each box with a number indicates an object and an empty box indicates unallocated space. The number in a box indicates the bandwidth requirement of the object. For simplicity, all objects are assumed to be of unit size; so the bandwidth requirement of an object 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 an object being moved from Array 1 to Array 2. The object selected is one with the BSR of 20. The object with BSR 70 could not be accommodated on the underloaded array due to bandwidth constraints.*

After the displace step since Array 1 is still overloaded the swap step is invoked.

Swap*: Figures (d) and (e) illustrate an object with BSR 10 being swapped with an object with BSR 1. Note that first, pairs of objects are considered; the object with BSR 70 on Array 1 could not be swapped with any object on Array 2 without any constraint violations.*

Since, both the arrays are now underloaded the algorithm terminates.

4 Measuring Bandwidth Requirements and Detecting Hotspots

In the previous section, we presented techniques for identifying a balanced configuration. The techniques assume that the bandwidth requirement of the objects is known and so a hotspot can be identified. In this section, we describe techniques for measuring bandwidth requirements of objects and detecting hotspots in a real storage system.

Measuring Bandwidth Requirements

Whereas the space requirement of an object is fixed at object creation time[4], the bandwidth requirement of an object

[4]Note that the space requirement refers to the size of the corresponding logical volume and not the the actual storage space in use. Moreover, extending a logical volume is an infrequent operation and a consequent change in space requirement is easily accommodated.

depends on the current workload. Unless the workload access pattern to the object is well characterized and known *a priori* throughout the lifetime of the object, its bandwidth requirement needs to be inferred based on online measurements. We use a simple measurement-based technique to approximate the bandwidth requirement of each object.

Recall that each object is assumed to be striped across some number of LUs in a logical array. Given the request size (in sectors) and the first logical sector requested for each request, one can infer the number of disks accessed. Note that the number of disks accessed is upper bounded by the number of disks which comprise the logical array. This technique requires that the number of disks each object is striped over and the RAID level of the component LUs be known. Given the average latency and transfer rate for the underlying disk, if a request *req* results in $IOcount_{req}$ independent disk accesses and $SectorCount_{req}$ is the number of sectors requested, the percentage bandwidth utilization of a logical array over a time window *I* due to accesses to object L_i is given by:

$$\Sigma_{req\epsilon(I,L_i)}(IOcount_{req} * (t_{seek} + t_{rot}) + SectorCount_{req}/r_{xfr})/(I * numDisks) \quad (6)$$

Here, the summation is over $req \epsilon (I, L_i)$ i.e., requests that accessed object L_i and completed in the time window *I*. *numDisks* is the number of disks in the underlying logical array. t_{seek}, t_{rot} and r_{xfr} are the average seek time, average rotational latency and average transfer rate, respectively, for the underlying storage device. The above expression computes the diskhead busy time per unit time per disk due to requests accessing object L_i in a time duration *I*, thus giving the array utilization due to accesses to the object.

We use this utilization figure as a measure of the bandwidth requirement of a an object. Note that this is the perceived bandwidth requirement of the object and assumes that the workload accessing the object is able to express itself in the presence of inter-object interference i.e., accesses to other objects on the same logical array. Moving the object to a similar array with less load may result in a different bandwidth utilization.

A limitation of our approach is that it works only for similar logical arrays. An approach used in practice is the IOPS measure for characterizing object bandwidth requirements and array bandwidth capacity. A limitation, however, of such a characterization is that it implicitly assumes a basic transfer size or amount of data accessed for an IO. For objects with different stripe unit sizes mapped to the same array such a technique may not be accurate. Our approach does not have this drawback.

Identifying Hotspots

The above technique gives the bandwidth utilization on an array due to an object mapped to it. The bandwidth utilization of the logical array can be now be approximated as the summation of the bandwidth utilizations of all the objects mapped to the array. An array is overloaded if its bandwidth utilization exceeds a certain threshold (equation 2).

An approach which offers flexibility in defining a hotspot is one using percentiles. The bandwidth utilization is averaged over an interval *I*, and an overload is signaled if a percentile (*perc*) utilization over the samples $\lfloor \frac{W}{I} \rfloor$ in a time window *W*, exceed the threshold. Since the utilization for each logical volume is computed separately (equation 6), one can compute this percentile for each logical volume and use their summation as the measure of the bandwidth utilization of the array.

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 IO completions for each logical volume, and (ii) a user space reconfiguration module which uses statistics collected in the kernel to estimate bandwidth requirements, computes a new configuration if a hotspot is detected, and migrates the requisite LVs appropriately.

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.

The kernel portion of the code involved adding appropriate code and data structures to enable collecting statistics for each LV. The 2.6 kernel uses `bio` as the basic descriptor for IOs to a block device. On IO completion a routine `bio_endio` is invoked by the device interrupt handler. It is here that we do the bookkeeping for each LV separately. This is facilitated as each LV created using the Linux *logical volume manager* (LVM) has a separate device identifier; the device identifier for which the IO was performed is available in the `bio` descriptor.

The user space reconfiguration module makes a system call periodically to query the statistics from the kernel. The statistics are namely the *sectorCount* and *IOCount* (see Section 4) which are used to approximate the bandwidth requirement of an LV. The system call also automatically resets the kernel statistics. We also provide two additional system calls which allow selective enabling and disabling of statistics collection for an LV. Statistics collection is enabled by default for an LV when it is activated (in LVM terminology), and is thus registered with the kernel. Deactivating an LV automatically disables the statistics collection for the same. Finally, note that the implementation involved using appropriate kernel synchronization primitives since the same data structure is accessed by the user space reconfiguration module (via system calls) when querying statistics and by the device interrupt handler on an IO completion. A separate synchronization primitive was employed for each logical volume to improve concurrency.

If the reconfiguration module detects a hotspot, it invokes appropriate routines to identify a balanced configuration. If a

balanced configuration is found the logical volumes are migrated appropriately. We use tools provided by the Linux Logical Volume Manager(LVM), namely *pvmove*, to achieve data migration while the LVs are online and being actively accessed. The user application continues to work uninterrupted throughout the migration, except for possibly some performance impact while the reconfiguration is in progress.

Finally, since we collect statistics only for IOs actually issued to the block device, any hits in the buffer cache are transparently handled. Our current implementation does not account for hits in other caches (disk cache and controller cache).

It is possible that disk accesses for separate `bio` requests get merged at the disk level. This would mean that the value of *IOCount* would be overestimated. To account for this, in our implementation, separate `bio` requests which correspond to contiguous logical sectors and complete within a short time window are treated as one large request. This ensures that the *IOCount* estimate is more in tune with the actual value.

6 Experimental Evaluation

In this Section, we first compare different object remapping techniques using algorithmic simulations. We then present experimental results from the evaluation of our prototype implementation. Since, our prototype is limited by the hardware configuration, algorithmic simulations help exhaustively evaluate the performance of different approaches for a variety of system configurations.

6.1 Simulation Results

We used an algorithmic simulator to compare the different algorithms for object remapping described in Section 3. The simulator implements all the algorithms and when invoked for an imbalanced configuration reports the cost of reconfiguration for each.

We seek to study the performance of different algorithms as different system parameters are varied. The parameters varied were the system size, the initial system bandwidth and space utilization, and the magnitude of the bandwidth overload. We also study the impact of the optimizations developed for the *displace* algorithm.

The default storage system configuration in our simulations comprised four logical arrays, each with 20 18 GB disks. Each logical array in the system was configured to have an initial storage space and bandwidth utilization of 60% and 50%, respectively.

To achieve a specified storage space utilization on an array, objects were assigned to an array until the desired space utilization had been reached. The object sizes were assumed to be uniformly distributed in the range [1 GB,16 GB]; the object size was assumed to be a multiple of 0.25 GB. To achieve a specified bandwidth utilization, first bandwidth requirement values were generated, one for each object, and in proportion to the object size. A random permutation of these values was then generated and a value assigned to each object in the array. This procedure resulted in a configuration with the desired values of storage space and bandwidth utilization for each array and no correlation between object size and object bandwidth requirements. Note that the default system parameters resulted on an average 25 objects assigned to each logical array, and thus an average of 100 objects in the storage system (comprised of four arrays).

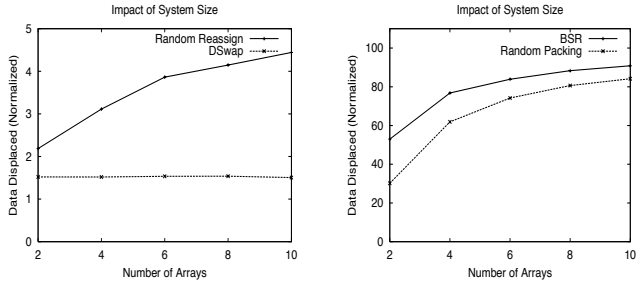
To generate an imbalanced configuration, we increased the bandwidth utilization on half the arrays in the system until a desired magnitude of overload had been reached. This resulted in a storage system with half the arrays overloaded and half with spare storage bandwidth. Here, *magnitude of overload* refers to the average of the bandwidth violation across all arrays in the system. To create an overload, we picked an object at random from one of the arrays that is to be overloaded, and increased its bandwidth requirement by an amount Δbw; for a given system configuration Δbw was chosen to be the bandwidth requirement of the least loaded object in the system. This procedure was repeated until the desired magnitude of overload had been attained. For our experiments, the default bandwidth violation threshold was chosen to be 80%, and the default magnitude of overload was fixed at 5%.

For each experiment, the performance figures reported correspond to an average over 100 runs i.e., correspond to the average cost of reconfiguration for 100 imbalanced configurations for the same choice of system parameters. The *normalized data displaced* figure reported in the following experiments is the total amount of data displaced (equation 3) as a percentage of the total data in the system i.e., the storage space allocated to all the objects put together.

6.1.1 Impact of System Size

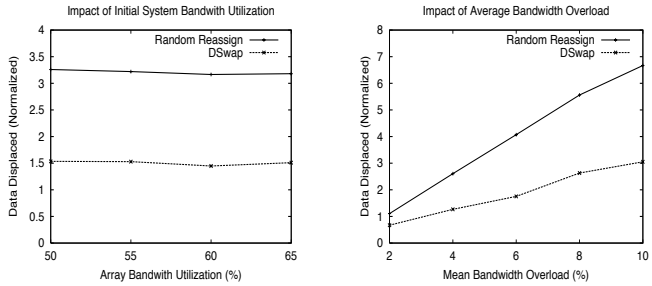
In this experiment, we study the impact of the system size on the cost of reconfiguration. We vary the system size i.e., the number of logical arrays, from 2 to 10. This resulted in systems with number of disks ranging from 40 to 200. Figures 3(a) and 3(b) show the performance of the cost-aware and cost-oblivious approaches, respectively, with varying system size. The graphs *Random Packing* and *BSR* correspond to the cost-oblivious approaches presented in Sections 3.1.1 and 3.1.2, respectively. The graphs *Random Reassign* and *DSwap* correspond to the cost-aware approaches presented in Sections 3.2.1 and 3.2.2, respectively.

Figure 3(a) shows that *DSwap* outperforms *Random Reassign*. Moreover, while the normalized reconfiguration cost with an increase in system size remains constant for the former, it increases for the latter. The higher cost of reconfiguration for the *Random Reassign* algorithm is because of the randomized nature of the algorithm. With increasing system size the number of possible objects to choose from goes up. The normalized cost remains constant for the *DSwap* algorithm as objects are chosen from an overloaded array carefully based on their *BSR* values, and so increasing the system size does not increase the cost. Note, however, that the absolute amount of data dis-



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

Figure 3: Impact of system size.



(a) Initial bandwidth utilization (b) Magnitude of overload

Figure 4: Impact of bandwidth utilization.

placed does increase with an increase in the system size.

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 significantly higher as compared to that of the cost-aware approaches. In both cases, the cost of reconfiguration increases with an increase in the system size because the probability that an object gets remapped to its original array decreases. *Random Packing* gives a cost lower than *BSR* because it is semi cost-aware (see Section 3.1.1). For the rest of the experiments described in this Section, the cost-oblivious approaches resulted in a similarly high cost of reconfiguration as compared to the cost-aware approaches and so we do not present the results for the same.

In our experiments, for the *Random Reassign* approach we set the fraction of objects *frac* (see Section 3.2.1) considered for reassignment from the set of objects on all the overloaded arrays to be 1.0 i.e., all the objects were considered. Also, for both the randomized algorithms, *Random Reassign* and *Random Packing*, the balanced configuration chosen was the one with the least cost from among 100 runs with different seed values.

6.1.2 Impact of System Bandwidth Utilization

In this experiment, we studied the impact of the bandwidth utilization on the cost of reconfiguration. Figure 4(a) and 4(b) show the impact of the initial bandwidth utilization and the magnitude of bandwidth overload, respectively.

Figure 4(a) shows that as the initial bandwidth utilization is increased from 50% to 65%, the normalized cost remains unchanged for both approaches. This is because increasing the initial system bandwidth utilization merely increases the initial bandwidth requirement of all the objects in the system proportionately. This reduces the fraction of objects 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. Note that *DSwap* results in a factor of two lower cost as compared to the *Random Reassign* approach due to reasons similar to that described in the previous experiment.

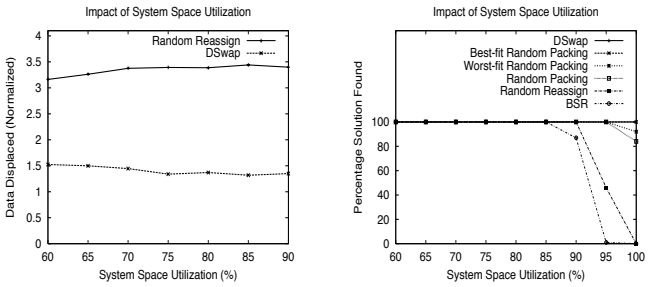
Figure 4(b) shows that with an increase in the magnitude of overload from 2% to 10%, the cost of reconfiguration increases for both approaches. This is because, on an average, more data needs to be displaced for a higher magnitude of overload. The rate of increase in the normalized cost is greater for *Random Reassign*, as compared to *DSwap*, as the objects are chosen for reassignment at random in the former approach, while in the latter approach objects are considered for reassignment based on their *BSR* values.

6.1.3 Impact of System Space Utilization

In this experiment, we studied the impact of varying system space utilization on the cost of reconfiguration. Figure 5(a) shows that the normalized cost of reconfiguration remains almost unchanged for both approaches as the the system space utilization is varied from 60% to 90%. This can be attributed to the fact that increasing the system space utilization increases the number of objects on each array. Consequently, for a fixed value of the initial bandwidth utilization the bandwidth requirement of the objects on an array decreases with an increase in the system space utilization. While this may require that more objects need to be reassigned to remove the same bandwidth overload, an increase in the system space utilization results in the normalized cost remaining largely unchanged. Note that *DSwap* results in a cost which is a factor of two less than *Random Reassign*.

We see a slight increase followed by a slight decrease in the cost for the *Random Reassign* approach. This is because an increase in the space utilization increases the number of objects to choose from for reassignment. The slight decrease that follows is because at higher space utilizations the number of objects that can be reassigned decreases as the space constraints on the underloaded arrays become a significant factor. The slight decrease in the cost for the *DSwap* approach is because with an increase in the system space utilization the fraction of objects on the overloaded arrays that can be accommodated on the underloaded arrays decreases.

Figure 5(b) shows the percentage of times a balanced configuration was identified for different imbalanced configurations generated for the same choice of parameters. The *BSR* approach, which reconfigures from scratch, fails to find a balanced configuration all the time when the system space utilization is 90% because of its deterministic nature. *Random Reassign* fails at 95% system space utilization, as there is little spare storage space on the underloaded arrays; recall that



(a) Cost (b) Percentage Solution Found

Figure 5: Impact of space utilization.

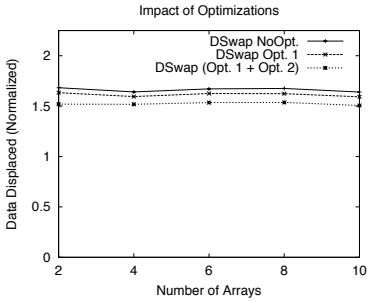


Figure 6: Impact of optimizations.

this approach only *reassigns* objects from overloaded arrays to underloaded arrays. The best-fit and worst-fit variants of this algorithm performed similarly, and so have not been shown in the figure.

At 100% system space utilization *Random Packing* and its variant *Worst-fit Random Packing* fail to find a balanced configuration all the time. Only, *DSwap* and *Best-fit Random Packing* approaches were able to find a balanced configuration for all overloaded configurations. As expected the best-fit variant of *Random Packing* is able to identify a balanced configuration in constrained scenarios (see Section 3.1.1). *DSwap* is able to identify a balanced configuration as it swaps objects between arrays. Note that in these experiments *ufrac* (equation 5) was chosen to be 0.50.

6.1.4 Impact of Optimizations

In this experiment, we study the impact of the optimizations developed for the *displace* step of our approach (see Section 3.2.2). Here, we present the results in the context of variation of system size. We conducted experiments to study the impact of the optimizations when various system parameters were varied; the results were similar to that for the system size experiment, and so we omit those to avoid repetition.

Recall that the first optimization involved choosing from among multiple possible groups of objects to remove the overload on a underloaded array. The second optimization used backtracking to improve the *soln* for each overloaded array.

Figure 6 shows the impact of the various optimizations as the system size is varied. As can be seen in the figure, *DSwap* without any optimizations (*NoOpt.*) has the maximum normalized cost. Introducing the first optimization (*Opt. 1*) results in a marginal improvement in the cost. The improvement is more pronounced when both the optimizations (*Opt. 1 + Opt. 2*) are employed. This is because while the first optimization questions only the choice of the *root* of the the *soln*, the second optimization uses backtracking to question the choice of each of the subsequent objects that comprise the *soln* thus resulting in more significant gains.

Note, however, that even this marginal improvement can be significant as the actual amount of data that needs to be displaced can be significantly different in the three cases as the system size is increased. Finally, the role of the optimizations can be particularly significant as compared to a purely greedy approach for some specific imbalanced configurations.

6.2 Prototype Evaluation

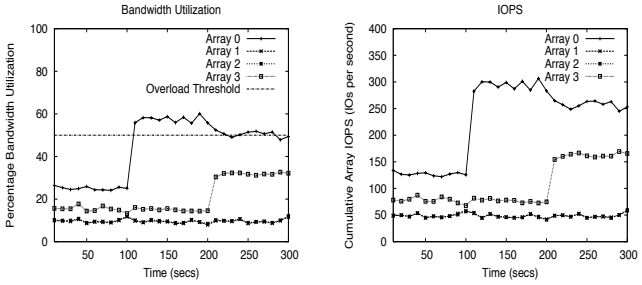
In this Section, we demonstrate the effectiveness of our approach by conducting experiments on our Linux prototype. The goal in these experiments was two-fold; (i) to show that the kernel measurement techniques are able to identify a hotspot, and (ii) to demonstrate that the reconfiguration module makes the correct choice when selecting objects and underloaded arrays to remove the hotspot.

In our experiments, we use a simple synthetic workload which provides us a great degree of control in imposing a desirable amount of IO load on the storage system. The workload for each logical volume was defined using two parameters, (i) concurrency N, and (ii) mean think time IA. A workload with concurrency N consists of N concurrent clients; each client issues a request and sleeps for a time interval exponentially distributed with a mean of IA on request completion before issuing a new request. The request sizes were assumed to be fixed and successive requests access random locations on the logical volume. The request size for client requests was fixed at 16KB in our experiments. Note, that while the think time provided control over the load imposed by each client, the client concurrency allowed us to independently control the load being imposed on an array due to accesses to an LV.

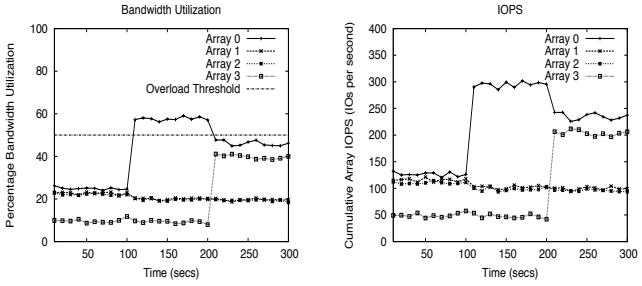
The characteristics of the host and the storage system in our prototype were as described in Section 5. We partitioned the 20 disks in the system to give five striped logical arrays each comprising four disks and with a stripe unit size of 16KB. Each array was partitioned into 14 partitions of size four GB each[5]. These partitions served as building blocks (in the form of LVM physical volumes) for the LVs created on each array; so, the LV size was a multiple of 4 GB. Of the five logical arrays, one array was configured without any logical volumes and was used as scratch space when swapping logical volumes between arrays.

We set the bandwidth overload threshold for all the arrays to 50% in our experiments. The interval I over which the

[5]Note that each striped array is visible as a SCSI drive on the host. Since we wanted to utilize maximum possible storage space on each logical array, and Linux allows only 14 usable partitions for a SCSI drive, we created 14 four GB partitions for a total of 56 GB allocatable storage space on each array.



(a) Spare storage space.



(b) No spare storage space.

Figure 7: Uniform object size

bandwidth utilization was approximated was chosen to be $10s$ and the window size W over which reconfiguration decisions were made was chosen to be $100s$. Note that these values are small, and were chosen to speed experimentation. Load-balancing involving object remapping is a long-term operation and is typically done only over periods of months or more. Finally, we used the $70th$ percentile of the samples accumulated in the time window W as a measure of the bandwidth utilization of an LV.

As mentioned before, the data migration could be achieved either online or offline. While our implementation supports online data migration, to speed up experimentation, we simply reconfigure the arrays for the new mapping of logical volumes. Techniques to control the rate of online data migration to mitigate the performance impact on foreground applications have been presented in [10].

We conducted two sets of experiments, one where the LV size or object size of all the objects on the system was the same, and another where the object sizes differed.

6.2.1 Uniform Object Size

For the case of uniform object sizes, we present results from two experiments, one were the system had spare storage space in the form of unallocated partitions, and another where the system had no spare space. While in the former, the *displace* step would be invoked, the latter would invoke the *swap* step. In both experiments, the size of any LV on an array was 4 GB.

In the first experiment, three arrays were configured with 14 LVs each, and the fourth array was configured to have seven LVs thus leaving half the array empty with seven allocatable partitions. For the first 100 seconds all the LVs in the system were accessed by a workload with a concurrency of two; the mean think time was fixed at 400ms, 1000ms, 1000ms and 500ms for the workload accessing LVs on the four arrays, respectively. Figure 7(a) shows the estimated average bandwidth utilization as well as the cumulative average IOPs across all the LVs for each array as a function of time. The average values reported are over 10 second intervals.

As can be seen the array utilizations are dictated by the mean think time values for the workload accessing the component LVs; lower mean think times mean higher utilization values. Also note that the average bandwidth utilization estimated using kernel measurements, and the average IOPs on each array based on measurements at the application level, follow the same trend. This indicates that our kernel measurements correctly track the application behavior.

At $t = 100s$ we increased the workload on *Array 0* by increasing the concurrency of half the clients to nine and that of the other half to four. This results in an increase in the bandwidth utilization on the array. Note that the bandwidth utilization on the remaining arrays remains unchanged. At $t = 200s$ the reconfiguration module detects that *Array 0* is overloaded, identifies a new balanced configuration, and triggers the appropriate reconfiguration. The reconfiguration involved moving two LVs from *Array 0* to *Array 3*.

The reconfiguration module correctly identified *Array 3* as the destination for the LVs, even though *Array 1* and *2* had a lower bandwidth utilization, as it was the only array with spare storage space. Moreover, of the LVs, it correctly chooses two of the seven logical volumes being accessed by a workload with concurrency of nine. This choice minimizes the amount of data displaced.

The graph from $t = 200s$ to $t = 300s$ shows the utilization of the arrays after the reconfiguration. As can be seen, the utilization of *Array 0* has decreased to a value close to the overload threshold. The utilization of *Array 3*, which now consists of two additional logical volumes, has increased appropriately. In our experiments, we allow for a soft threshold of 2% around the bandwidth violation threshold, and consequently, no more reconfigurations are triggered. This is done in order to avoid a reconfiguration for minor bandwidth violations.

For the second experiment, we configured the storage system with no spare space and each array comprised 14 LVs. The workload for the LVs on *Array 0* was the same as in the previous experiment. The mean think times for workloads on Arrays 1 through 3, however, were chosen to be 500ms, 500ms and 1000ms, respectively. The client concurrency of the workload was fixed at two. Figure 7(b) plots the bandwidth utilization and cumulative average IOPs for each array.

In this case, a reconfiguration is again triggered at $t = 200s$ as in the case above. The reconfiguration involved swapping three heavily accessed logical volumes with three logical volumes on *Array 3*, the array with the least load in the system. Note that despite the workload on *Array 0* being similar to that in the first experiment, a slightly different observed utilization due to peculiarities of a real system, result in three

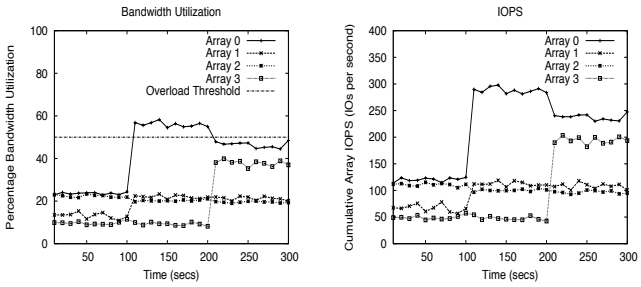


Figure 8: Variable object size; no spare storage space

logical volumes being swapped, as opposed to two in the first experiment. Consequently, the drop in bandwidth utilization is greater in this run. The reconfiguration results in a reduction in the bandwidth utilization on the array to a value below the overload threshold.

6.2.2 Variable Object Size

For the case of the storage system configured with LVs with variable size, we ran experiments for both the case when the system had spare space and when there was no spare space. The results for the case the system had spare space were similar to that in the previous experiment; when a hotspot occurred the heavily accessed volumes were chosen and moved to an array with spare space in order to minimize the amount of data displaced. To save space, we do not present the results from that experiment.

For the case the storage system had no spare space, the system configuration was as follows. *Array 0* and *1* were configured with six LVs each, two LVs each of size 4 GB, 8 GB and 16 GB, respectively. *Array 1* and *2* were configured with 14 LVs each, each of size 4 GB. The mean think times for the workload accessing the LVs on the four arrays were 300ms, 500ms, 500ms and 1000ms, respectively. For the first 100s of the experiment, the concurrency for the workload for all the LVs on the system was fixed at two. Figure 8 shows the bandwidth utilization and cumulative IOPs as a function of time.

At $t = 100s$ we increased the client concurrency for the workload accessing the LVs on *Array 0* and *1* to seven and four, respectively. As can be see in the figure, this results in an increases in the bandwidth utilization on both the arrays. However, only *Array 0* observes a violation of the bandwidth threshold. At $t = 200s$ the reconfiguration module detects a hotspot and triggers a reconfiguration. The reconfiguration involved swapping two 4 GB LVs with two LVs of the same size from *Array 3*. So, the reconfiguration module correctly identifies *Array 3* as the array with the least load. Also, since all the six LVs are configured with the same workload, the two LVs of size 4 GB are the one with maximum *BSR*.

The graph from $t = 200s$ to $t = 300s$ shows that after the reconfiguration the utilization on *Array 0* has decreased to a value below the threshold. *Array 3* with two new LVs with a heavier load observes an increase in the utilization.

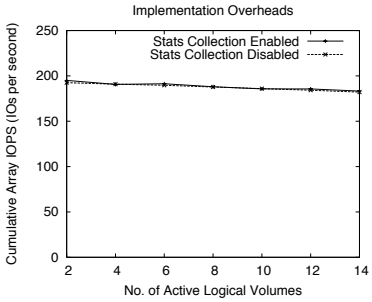


Figure 9: Impact on application performance.

6.2.3 Implementation Overheads

Our final experiment aimed to study the overheads introduced, if any, due to the kernel enhancements on application performance. While the computation involved in maintaining statistics was minimal, the synchronization primitives employed (Section 5) may introduce some overheads. So, in this experiment, we vary the number of logical volumes actively being accessed on each array and compare the application performance for the case statistics collection is enabled, to that of when statistics collection is disabled in the kernel. The storage system was configured with 14 LVs, each of size 4 GB, per array. The workload accessing each LV had a client concurrency of two and the mean think time was set to zero. Note, that with no think time, each client would issue a new IO request immediately after the previously issued request completes. Consequently, the storage system is saturated.

Figure 9 shows the cumulative average IOPs for one of the arrays as a function of the number of LVs being actively accessed; each point corresponds to an average value for a two minute run. The graph for the other arrays was the same. As can be seen, the average IOPs figure in both the cases is similar. So, there was not noticeable overhead on application performance. Since, the storage system is saturated, the average IOPs value does not change significantly as the number of active logical volumes is varied. The slight drop in the value in both cases, with an increase in the number of active LVs, is because with a larger number of LVs being accessed, the average seek latency increases. This is because, for each additional contiguous partition being accessed on an array, the disk heads on the component disks have to seek over a larger disk surface when servicing requests.

6.3 Summary of Experimental Results

Our experiments show that for a variety of system configurations our novel approach reduces the amount of data moved to remove the hotspot by a factor of two as compared to other approaches. Moreover, the larger the system size or the magnitude of overload the greater the performance gap. Results from our prototype implementation suggested that our kernel measurement techniques correctly track application behavior and identify hotspots. For simple overload configurations considered, our techniques correctly remove the hotspot while min-

imizing the amount of data displaced. Finally, the kernel enhancements do not result in any noticeable degradation in application performance.

7 Related Work

Algorithms for moving data objects from one configuration to another in as few time steps as possible have been presented in [6, 7, 9]. It is assumed that the new final configuration is known. In our work, we seek to identify the new final configuration which requires minimal data movement.

Techniques for initial storage system configuration have been presented in [2, 3]. Our work assumes that the storage system is 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 [13]. 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 [15]. Our work seeks to achieve improved performance across the storage system by moving logical volumes between arrays.

Disk load balancing schemes for video objects has been presented in [16]. 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 that identifying a new mapping of data objects to arrays.

Request throttling techniques to isolate the performance of applications accessing volumes on a shared storage infrastructure have been explored in [8, 11, 14]. We present algorithms to improve storage system performance by migrating entire logical volumes between arrays.

Finally, while [10] presents techniques for controlling the rate of data migration to mitigate the instantaneous performance impact on foreground applications during online reconfiguration, our work seeks to optimize for the scale of reconfiguration which dictates the duration of performance impact.

8 Concluding Remarks and Future Work

In this paper, we argued that techniques employed to load-balance large scale storage systems do not optimize for the *scale of the reconfiguration*—the amount of data displaced to realize the new configuration.

Reconfiguring the system from scratch can incur significant data movement overhead. Our novel approach uses the current object configuration as a *hint*; the goal being retain most of the objects in place and thus limit the scale of the reconfiguration. We also described a simple measurement-based technique for identifying hotspots and for approximating per-object bandwidth requirements.

Finally, we evaluated our techniques using a combination of simulation studies and an evaluation of an implementation in the Linux kernel. Results from the simulation study showed that for a variety of system configurations our novel approach reduces the amount of data moved to remove the hotspot by a factor of two as compared to other approaches. The gains increase for a larger system size and magnitude of overload. Experimental results from a prototype evaluation suggested that the measurement techniques correctly identify workload hotspots. For some simple overload configurations considered in the prototype our approach identified a load-balanced configuration which minimizes the amount of data moved.

Our current prototype implementation makes the simplifying assumption that all arrays in the storage system are similar. As future work we would like to develop techniques for quantifying the bandwidth requirements of objects, as well as the bandwidth capacities of arrays, for a storage system comprising heterogeneous arrays. Secondly, we plan develop optimizations for the swap algorithm in a manner similar to those developed for the displace algorithm. Finally, we plan to extend our kernel-measurement based technique to account for hits in the lower level storage system caches.

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*, 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] Eric Anderson et al. An experimental study of data migration algorithms. In *WAE: International Workshop on Algorithm Engineering*. LNCS, 2001.

[7] Joseph Hall, Jason D. Hartline, Anna R. Karlin, Jared Saia, and John Wilkes. On algorithms for efficient data migration. In *Symposium on Discrete Algorithms*, pages 620–629, 2001.

[8] Magnus Karlsson, Christos Karamanolis, and Xiaoyun Zhu. Triage: Performance isolation and differentiation for storage systems. In *Proceedings of the International Workshop on Quality of Service (IWQoS 2004), Montreal, Canada*, pages 67–74, June 2004.

[9] S. Khuller, Y. Kim, and Y. Wan. Algorithms for data migration with cloning. In *ACM Symp. on Principles of Database Systems (2003).*, 2003.

[10] C. Lu, G. Alvarez, and J. Wilkes. Aqueduct: Online data migration with performance guarantees. In *FAST'02*, pages 219–230, January 2002.

[11] C. Lumb, A. Merchant, and G. Alvarez. Facade: Virtual storage devices with performance guarantees. In *FAST'03*, 2003.

[12] 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.

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

[14] V. Sundaram and P. Shenoy. A practical learning-based approach for dynamic storage bandwidth allocation. In *IWQoS'03*, 2003.

[15] 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.

[16] 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.