Efficient Data Migration in Self-managing Storage Systems

Vijay Sundaram
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003
vijay@cs.umass.edu

Timothy Wood
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003
twood@cs.umass.edu

Prashant Shenoy
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003
shenoy@cs.umass.edu

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 validate our technique with simulations and a prototype implemented into the Linux Kernel. Our prototype and simulations show our algorithm successfully eliminates hotspots with a factor of two reduction in data copying overhead compared to other approaches.**

I. Introduction

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 [2], [3]. 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]. Despite these tools' capabilities to collect performance data and predict the impact of moving data partitions from one array to another, the decision process remains 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 [5], [7], [8]. 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.

II. Problem Formulation

Consider an enterprise storage system as shown in Figure 1. Such a system comprises multiple, heterogeneous disk arrays, each consisting of one or more RAID devices. A RAID device is constructed by combining a set of disks from the array using RAID techniques [9]. Each RAID device can contain one or more data partitions (also called logical volumes) and it is possible for partitions to span multiple RAID devices, forming a logical RAID.

In this system, there is a hard constraint that the storage needs of all partitions mapped to an array cannot exceed the array's storage capacity. There is also a weak constraint that the total bandwidth utilization of an array be smaller than a threshold τ in order for an array to be considered load balanced. If the bandwidth utilization exceeds this threshold, the array is overloaded and a hotspot is said to be in the system.

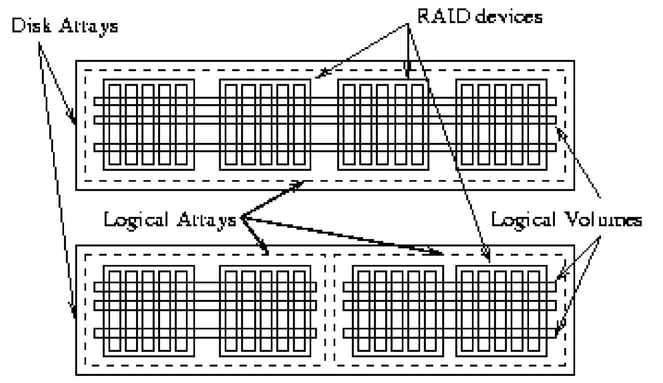
Alleviating the hotspot requires some logical volumes to be moved from overloaded arrays to underloaded ones. The cost of such a system reconfiguration is defined to be the total amount of data moved. If a new configuration does not change the mapping of a particular volume, then that volume does not contribute to the cost of the reconfiguration.

III. Cost-aware Object Remapping

A. Background

A self-managing storage system must be able to determine a new mapping of data partitions after detecting a hotspot. Several techniques can be employed for this purpose:

Bin Packing: One 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 [2]. Random permutations of volumes are generated and objects are assigned randomly to arrays until a valid assignment is obtained (a valid assignment satisfies



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.

Fig. 1. An enterprise storage system.

both the hard storage constraint and the bandwidth constraint, eliminating the hotspot).

BSR-based Approach: *Bandwidth-to-space ratio (BSR)* has been used as a metric for video placement [4], [6]. Using knapsack heuristics, 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. This ensures better utilization of the system bandwidth per unit storage space and leads to a tighter packing.

Randomized reassignment: The previous two approaches are *cost-oblivious*; they construct a new mapping of volumes to arrays without considering the current mapping. We can modify the bin packing approach to take the current mapping into account by incrementally modifying the current configuration until the hotspot is eliminated. The current configuration is assumed to be the initial configuration; bin packing is then used to assign objects from overloaded to underloaded arrays until sufficient load "shifts" to remove the hotspot. Although this algorithm is *cost-aware*, it does not explicitly attempt to reduce data copying overhead, nor does it consider the possibility of *swaps*, where two volumes are swapped. Thus, an entire set of possible solutions is ignored by the approach.

B. 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).

1) Displace: Our approach first considers a displace step which attempts to move volumes from overloaded arrays to unused storage space on underloaded ones. All arrays that see a hotspot (i.e., violation of the bandwidth constraint) 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. Within each overloaded array, volumes are considered for possible relocation in decreasing order of their BSR. This results in a set of logical volumes which when displaced to an underloaded array will sufficiently reduce the total load to the array to make the hotspot disappear. By selecting volumes by decreasing BSR order, we ensure that we are moving the minimum amount of data necessary to eliminate the hotspot.

The set of possible destination arrays are considered in decreasing order of spare BSR (spare bandwidth to spare storage space ratio) and selected so that neither the bandwidth nor storage constraints presented earlier will be violated. In the event that sufficient storage space is unavailable for a displace step, our techniques must resort to volume swaps to alleviate the remaining hotspots.

2) Swap: As in displace, *BSR* is used as the metric to guide the selection of volumes to be swapped between two arrays. 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 must determine a sets of overloaded and underloaded volumes such that the following constraints are satisified:

Constraint C_1: There is at least a certain minimum amount of load reduction on the overloaded array.

Constraint C_2: The swap should not violate the storage and bandwidth constraints on the underloaded array.

Constraint C_3 : The swap should not violate the storage constraint on the overloaded array.

If sets 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.

Additonal optimizations and the full details of the algorithm are described in [10].

3) Example: 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

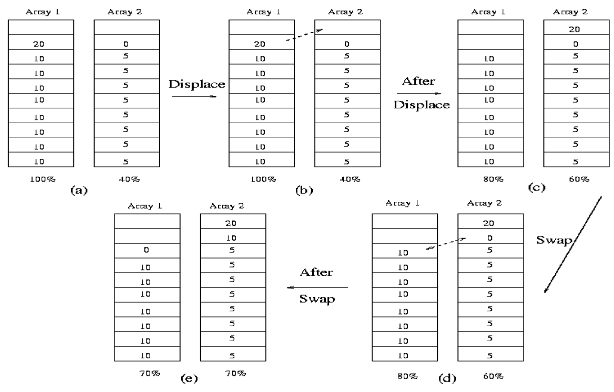


Fig. 2. Illustration of Displace and Swap.

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. Thus, a high BSR volume gets swapped with a low BSR volume.

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

IV. Automated Hotspot Detection

We detect overload by continually monitoring the bandwidth utilization on each array and flag 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.

Our monitoring module is implemented with hooks in the OS kernel which measure the mean request rate and request size for each volume. Knowing these as well as the seek latency, rotational latency, and transfer time per byte of the underlying disk, gives an indication of disk utilization. It maintains a history of utilizations observed on each array over a long period (e.g., a day or a week).

Hotspot detection: Given a time series (history) of utilizations seen at each array, a hotspot is said to be present if the bandwidth constraint is violated for a certain percentage 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.

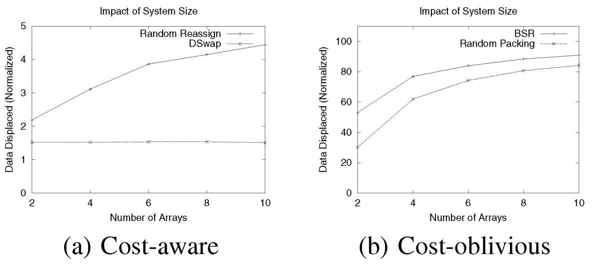


Fig. 3. Impact of system size.

V. Experimental Evaluation

A. Impact of System Size

We use simulations to evaluate the reconfiguration cost of our algorithm over a wide range of system configurations. The simulator allows us to study the impact of system size on reconfiguration overhead. We vary the system size—the number of arrays—from two to ten (each array holding 20 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.

B. Heterogeneous Volume Sizes

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

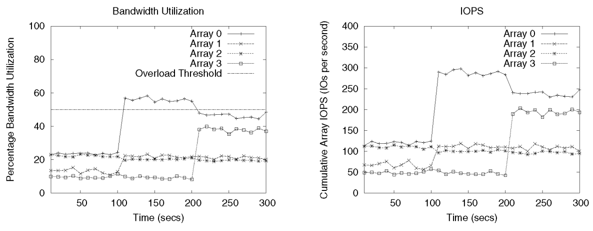


Fig. 4. Variable volume size; no spare storage space

We have used our prototype to evaluate systems with heterogeneous volume sizes. We consider two scenarios, one where unused storage space is present and another where all arrays are full. 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. 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 4, 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.

C. Summary of Other Results

We have used our simulator and prototype to examine how our Dswap algorithm works under a variety of overload configurations besides those mentioned here. In addition to system size, we have examined the impact of both bandwidth and storage utilization on the cost of reconfiguration. In all scenarios, we find that our algorithm performs reconfigurations cheaper, and is able to successfuly eliminate hotspots even in scenarios with very high storage utilization where the other methods fail to find a configuration which eliminates the hotspot. 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 measure array utilizations and are effective at detecting hotspots and dissipating them, while imposing negligible overheads on the system. Our full results are available in [10].

VI. Conclusion

In this paper, we argued that manual hotspot detection and storage system reconfiguration are 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. Evaluating our techniques within the Linux Kernel and through simulation showed a factor of two reduction in data copying overhead compared to other approaches without causing noticeable degradation in application performance.

VII. Acknowledgements

This research was supported, in part, by NSF grants EEC-0313747, CNS-0323597, and EIA-0080119. Thanks to our anonymous reviewers for their helpful comments on this paper.

References

[1] Emc symmetrix optimizer. Available from http://www.emc.com/products/storage_management/symm_optimizer.jsp.

[2] 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.

[3] 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.

[4] 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.

[5] E. A. et. al. An experimental study of data migration algorithms. In *Proc. of WAE- International Workshop on Algorithms Engineering, LNCS*, 2001.

[6] Y. Guo, Z. Ge, B. Urgaonkar, P. Shenoy, and D. 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.

[7] 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.

[8] 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.

[9] 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.

[10] V. Sundaram and P. Shenoy. Efficient data migration in self-managing storage systems. Technical Report TR06-21, Dept. of Computer Science, Univ. of Massachusetts, 2006.