Scalable Load Balancing in Cluster Storage Systems

Gae-Won You, Seung-Won Hwang, Navendu Jain

Scalable Load Balancing in Cluster Storage Systems

Gae-won You[†], Seung-won Hwang[†], and Navendu Jain[‡]

[†]Pohang University of Science and Technology, Republic of Korea
[‡]Microsoft Research Redmond, United States
{gwyou,swhwang}@postech.edu,navendu@microsoft.com

Abstract. Enterprise and cloud data centers are comprised of tens of thousands of servers providing petabytes of storage to a large number of users and applications. At such a scale, these storage systems face two key challenges: (a) hot-spots due to the dynamic popularity of stored objects and (b) high reconfiguration costs of data migration due to bandwidth oversubscription in the data center network. Existing storage solutions, however, are unsuitable to address these challenges because of the large number of servers and data objects. This paper describes the design, implementation, and evaluation of Ursa, which scales to a large number of storage nodes and objects and aims to minimize latency and bandwidth costs during system reconfiguration. Toward this goal, Ursa formulates an optimization problem that selects a subset of objects from hot-spot servers and performs *topology-aware* migration to minimize reconfiguration costs. As exact optimization is computationally expensive, we devise scalable approximation techniques for node selection and efficient divide-and-conquer computation. Our evaluation shows Ursa achieves cost-effective load balancing while scaling to large systems and is time-responsive in computing placement decisions, *e.g.*, about two minutes for 10K nodes and 10M objects.
Keywords: Load balancing, storage, optimization, linear programming

1 Introduction

This paper describes a scalable data management middleware system Ursa that aims to improve load balancing for cloud storage services services in the spirit of web email data stores (*e.g.*, Hotmail or Yahoo! Mail) over utility storage systems (*e.g.*, Amazon S3 [1] and Windows Azure [2]).

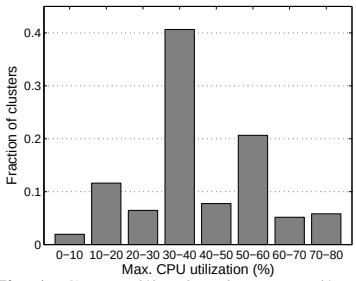


Fig. 1. CPU utilization in Hotmail clusters. CPU utilization is highly skewed to a few object partitions.

As cloud services continue to grow rapidly, large-scale cloud storage systems are being built to serve data to billions of users around the globe. The primary goal for these systems is to provide scalable, good performance, and high-availability data stores while minimizing operational expenses, particularly the bandwidth cost. However, diverse I/O workloads can cause significant data imbalance across servers resulting in hot-spots [14]. Fig. 1 shows the CPU utilization across clusters in Hotmail. From this figure, we can observe that I/O workloads are unbalanced over clusters and they are skewed to a few object partitions resulting in "hot

nodes," causing high delays to end users. As a result, these services typically shuffle terabytes of data per day to balance load across clusters [22].

The same challenge has been raised and actively discussed in the context of building database on clouds [5–7, 17, 20, 21]. However, all these works either do not address dynamic load reconfiguration or assume the source and target nodes of dynamic migration are known by an oracle. In contrast, we aim at achieving scalable and dynamic reconfiguration at the storage layer by identifying cost-optimal source-target pairs. Toward this goal, we highlight the following two key challenges for our system:

1. **Be scalable to large numbers of nodes and objects:** The system should scale to a large number of nodes storing data from participating users and applications. Cloud scale systems today have tens of thousands of nodes and billions of objects corresponding to petabytes of disk space, and these numbers will increase over time. Similarly, the system should support a broad range of applications which may track per-user data (*e.g.*, mail folders) or per-object access patterns (*e.g.*, video content), or perform analytical processing like MapReduce [8] on large datasets (*e.g.*, scientific traces and search index generation from crawled web pages).

2. **Minimize reconfiguration costs:** The system should aim to minimize the reconfiguration costs in terms of both bandwidth and latency. As workloads change over time, load balancing incurs bandwidth overhead to monitor the load of individual objects and servers to find hot-spots as well as to migrate data to alleviate hot-spots. Similarly, data movement risks a long reconfiguration time window, typically proportional to the migrated data volume, during which reads may incur a high delay and writes may need to be blocked to ensure consistency. Finally, reconfigurations may interfere with foreground network traffic risking performance degradation of running applications.

To our knowledge, all prior techniques have aimed to solve these challenges individually. To address load imbalance, many techniques perform dynamic placement of individual data objects [3, 25] or distribute objects randomly across the cluster (*e.g.*, based on hashing [18]). However, to adaptively re-distribute objects, we need to know load patterns for billions of objects. Optimizing reconfiguration costs for these patterns calls for offline solvers [11, 16] (*e.g.*, knapsack or linear programming based) to make migration decisions. However, as such optimization is inherently expensive, these approaches are suitable at small scale and less effective when systems grow to a large scale. Meanwhile, approaches trading effectiveness to achieve scalability, for instance, by using a greedy simulated annealing or an evolutionary algorithm [16], suffer from high reconfiguration costs; we discuss these approaches in detail in related work.

Our goal is to show that a simple and adaptive framework can significantly reduce reconfiguration costs for industry-scale load balancing in cloud storage systems. While designing Ursa, we keep it implementable with mechanisms in existing storage systems–meaning our design should avoid complexity, can be evaluated on physical hardware, and can be deployed to our data centers today.

Our approach Ursa is designed to address the challenges listed above, using the following two key ideas:

1. **Workload-driven optimization approach:** First, to minimize the reconfiguration costs, Ursa formulates the problem as a workload-aware integer linear programming (ILP) problem whose goal is to eliminate hot-spots while minimizing the data move-

ment cost. Based on our analysis of production workloads (Section 2.1), Ursa leverages the fact that the number of hot-spots is small compared to system size (*i.e.*, power law distribution) and hence optimizes load balancing for only the hot-spots rather than uniformly distributing the entire load across all servers.

2. **Scalable approximation techniques:** Second, to achieve scalability, Ursa applies an efficient divide-and-conquer technique to optimize for a subset of source/target nodes based on the workload and network topology. To further reduce migration costs under high bandwidth oversubscription prevalent in data center networks [13], Ursa migrates data in a topology-aware manner, *e.g.*, first finds placement within the same rack, then a neighboring rack, and so on. When a neighboring set of a hot-spot server overlaps with another, Ursa performs joint data placement of objects from hot-spots in the combined neighborhood set.

We have developed a prototype of Ursa and evaluated it on traces from Hotmail. Our evaluation shows that, compared to state-of-the-art heuristics [16], Ursa achieves scalable and cost-effective load balancing and is time-responsive in computing placement decisions (*e.g.*, about two minutes for 10K nodes and 10M objects).

2 Preliminaries

In Section 2.1, we first explain our target application scenarios, based on which we formally discuss our problem in Section 2.2.

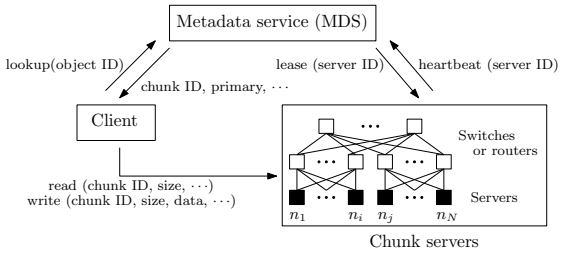


Fig. 2. Overall architecture; the data center network organized as a spanning tree topology [13]

2.1 Motivation

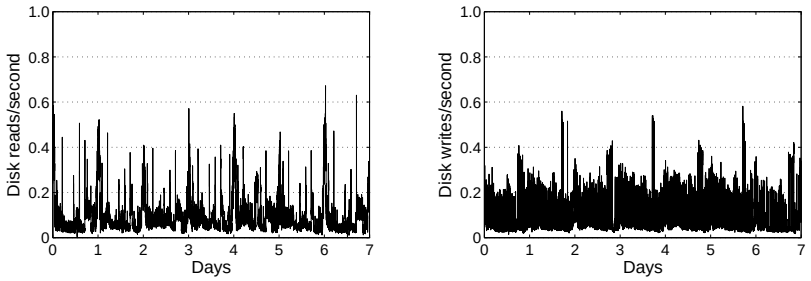
Our target system model for Ursa is that of a replicated, cluster-based object store similar to current systems such as GFS [12] and Azure blob store [2]. The object store is comprised of tens of thousands of storage nodes, typically commodity PCs, which provide read/write accesses to data. The nodes are inter-connected in a data center network which is organized as a spanning tree topology [13]. As a result, network distance between nodes can vary significantly *e.g.*, two nodes in the same rack or cluster have higher bandwidth connectivity than those in different ones, which affects migration costs between nodes. Fig. 2 illustrates the overall architecture.

Data is stored in units of chunks (or partitions). A chunk is of some fixed size, *e.g.*, 64KB in Google's Bigtable [4] and 64MB in GFS [12]. Each chunk has l replicas (1 primary and $l-1$ secondary replicas) to enable fault tolerance; a typical value of l is 3. At any time a storage server will be the primary for some of the chunks stored on it. For fault tolerance, each replica of a chunk is typically placed in a separate fault domain, *e.g.*, a node or a rack.

All client requests are first sent to a metadata service which maps a data object to its constituent chunks, and each chunk to its current primary. The metadata service periodically polls each server to track availability and uses leases to maintain read/write consistency. A read request is sent by the primary to the local replica. For a write request, the primary first determines request ordering, then sends the request to all replicas, and finally acknowledges 'accept' to the client when each of these steps is completed.

Load balancing is achieved by uniformly spreading chunks over storage servers and choosing primaries for each chunk randomly from the available replicas of that chunk. However, in balancing load under dynamic workloads, these storage systems face several challenges described using a case-study of a large cloud email service.

Case-study on Hotmail: Hotmail is a cloud service with millions of users and is comprised of tens of thousands of nodes storing petabytes of data (*e.g.*, emails and attachments). In Hotmail, most writes are message writes. When a message arrives, it is simultaneously written to multiple replicas. Hotmail is read-heavy, which is the source of disk latency, and its workload has a heavy tailed distribution with large size email attachments which result in significant disk I/O load.



(a) Reads (b) Writes

Fig. 3. Normalized Hotmail weekly loads

We observed the (normalized) volume of read and write requests for a typical server for a week, as reported in Fig. 3. First, we observed a significant variation in the request volume by about 10x. Second, the requests are imbalanced across partitions and correspondingly across storage nodes hosting them. For instance, recall from Fig. 1 that CPU utilization is highly skewed to a few partitions, which become hot-spots and bottlenecks for the entire system. We leverage these observations in designing our load balancing algorithm by spreading load away from hot-spots and minimizing reconfiguration costs to migrate data under dynamic load patterns.

2.2 Problem Statement

Based on these observations, we formally define the problem of load balancing. In particular, we show how to integrate load balancing in a cloud storage system.

We first define some notations to be used in this paper. Let $\{n_1, \cdots, n_N\}$ be a set of nodes, where N is the number of nodes and each node n_i contains replicas r_j's of some partitions. Each replica r_j has a workload L_j according to the request on the replica and each node n_i has the sum of loads of all replicas that it contains, *i.e.*, $L_{*i} = \sum_{r_j \in n_i} L_j$.

Fig. 4 evaluates the response time for a varying volume of requests in a VM on Windows Azure. We assume various block sizes, *i.e.*, 8-64MB, as the unit of each read

or write request. We measure the response time of the 99th percentile among 1,000 requests. As expected, we observe from Fig. 4 results that the response time increases significantly when the request load exceeds a threshold, *e.g.*, more than 10 seconds at 50 reads/sec in read requests and 20 writes/sec in write requests for 32MB block size.

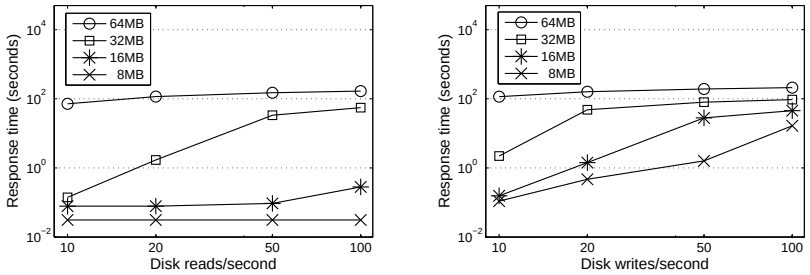


Fig. 4. Response time over the requests per second (the y-axis is on log scale)

This suggests that a "hot-spot," swamped with requests, incurs significantly higher response time and becomes a bottleneck, which motivates us to alleviate such hot-spots. Formally, hot-spots are defined as a node n_i with an aggregate load, *e.g.*, total I/O operations per second (IOPS) or bytes transferred per second for reads and writes, beyond some threshold C_i. We can define C_i empirically as illustrated in Fig. 4. Based on the observation, we define the following load capacity constraint:

Constraint 1 (Load Capacity (C1)) *For every node n_i in the storage system, keep the total load $L_{*i} \leq C_i$.*

Meanwhile, to provide fault tolerance, two replicas of the same object should not be in the same failure domain, *e.g.*, the same node or rack. We use the fault tolerance constraint for each node as follows:

Constraint 2 (Fault Tolerance (C2)) *Every node n_i should contain at most one replica of a given chunk (or partition).*

To achieve load balancing, our goal is to generate a sequence of operations satisfying C1 and C2, while optimizing the reconfiguration cost. The two available knobs for data migration are:

- **Swap:** This operation simply switches the role of the primary replica of a chunk with one of the secondary replicas.
- **Move:** This operation physically migrates the chunk data between two nodes.

For instance, to avoid some nodes being swamped with requests, Hotmail "moves" users between disk groups to maintain a high full watermark on each disk group, and servers "swap" primary and secondary roles.

This paper mainly focuses on optimizing moves, as they involve higher migration costs, which depends on network topology between servers, *e.g.*, the lowest cost to move data is within the same rack, relatively higher within neighboring racks, and more far away. To logically visualize this problem in 2D, Fig. 5 emulates bandwidth costs using an L2-metric, or Euclidean distance, based on inter-node distance (a circle of

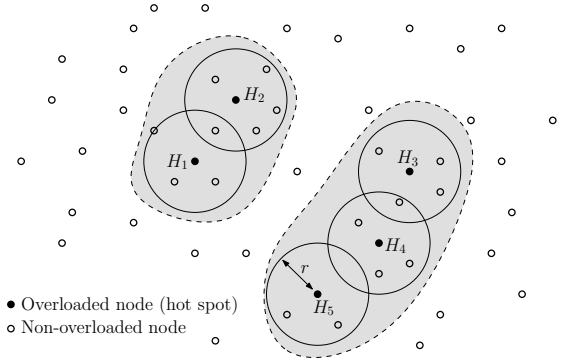


Fig. 5. Logical view of topology-dependent bandwidth costs for hot-spots H_1 to H_5

radius r denotes a logical partition of nodes, say a rack). In this figure, a smaller distance between two nodes is more desirable meaning it incurs lower migration cost.

To summarize, our load balancing problem can formally be defined as follows:

Definition 1 (Load Balancing). *Find the optimal placement $\mathcal{P}'$ satisfying C1 and C2 from initial placement $\mathcal{P}$ by generating a cost-optimal swap and move sequence.*

3 Algorithms for Load Balancing

In this section, we describe our optimization problem for load balancing more formally. Specifically, Section 3.1 formulates our goal and constraints as a linear programming (LP) problem [15], which is not scalable in our target scenario. Section 3.2 describes a two-phase approximation algorithm for near-optimal solutions.

While a joint-optimization of combining two available knobs, swap and move operations, seems attractive, we only consider the move operation for two reasons in this formulation. First, the number of decision variables will increase significantly making the computation prohibitive. Second, from a practical view, operators prefer to invoke cheap swap operations before they consider expensive data migrations. Thus, prior to applying the move operations, to cool down our entire system, we can iteratively swap the primaries in the hottest node and the secondaries in the cold node until no swap operation is applicable. After that, with respect to the remaining hot nodes, *i.e.*, on the swapped system, we apply our following strategies with only the move operations.

3.1 Exact Solution: ILP Formulation

LP is a mathematical model, populating "decision variables" with values achieving the best outcome (lowest migration cost in our problem). There can be multiple optimal solutions in terms of cost model, but our model is guaranteed to return one of such solutions. In our problem, decision variables are defined as a binary hyper-matrix Y representing optimal placement of the replicas. If the k-th replica of the j-th partition is best to be placed on the i-th node after migration, Y_{ijk} will be set to 1, and 0 otherwise. As decision variables are all integers, our problem is an integer linear programming (ILP) problem.

Table 1 summarizes all notations. In particular, current placement A_{ijk} can be represented in the same way as Y_{ijk}. With respect to A_{ijk} and Y_{ijk} placements, the bandwidth cost to move the k-th replica of the j-th partition from the node containing the replica to the i-th node can be defined. Note that $\overline{A}$ is a complement representation of A that converts 0 to 1 and 1 to 0. $\overline{A}_{ijk}Y_{ijk} = 1$ means the movement of the k-th replica of the j-th partition to the i-th node. Our goal is to find Y minimizing the total cost for such movements, which can be formally stated as an objective function in Eq. (1).

Table 1. Notations for optimal ILP model

Notation	Description
Y_{ijk}	1 if k-th replica of j-th partition is on i-th node after move, 0 otherwise
A_{ijk}	1 if k-th replica of j-th partition is on i-th node before move, 0 otherwise
B_{ijk}	Bandwidth cost to move k-th replica of j-th partition from the node containing the replica into i-th node
C_i	Load capacity of i-th node
L_{jk}	Load of k-th replicas of j-th partition
l	Number of replicas
F_q	Set of nodes with the same fault domain index q

Constraints C1 and C2 discussed in Section 2 can be formally stated as Eq. (2) and (5) respectively.

$$\textbf{minimize} \quad \sum_i \sum_j \sum_k \overline{A}_{ijk} Y_{ijk} B_{ijk} \tag{1}$$

$$\textbf{subject to} \quad \forall i : \sum_j \sum_k Y_{ijk} L_{jk} \leq C_i \tag{2}$$

$$\forall j, \forall k : \sum_i Y_{ijk} = 1 \tag{3}$$

$$\forall j : \sum_i \sum_k Y_{ijk} = l \tag{4}$$

$$\forall q, \forall j : \sum_{i \in F_q} \sum_k Y_{ijk} \leq 1 \tag{5}$$

$$\forall i, \forall j, \forall k : Y_{ijk} = 0 \text{ or } 1 \tag{6}$$

In the above ILP formulation, Eq. (5) can incorporate placement constraints for fault domains (*e.g.*, a node, rack, or cluster) where replicas of the same partition should be placed in different fault domains to ensure high availability. Note that our LP model does not have to keep track of whether a replica is a primary or a secondary. The model incorporates only move operations because the primary-secondary swaps are already completed before applying the model.

Our formulation suggests that obtaining an exact solution is not practical, especially in our target scenario of 10M partitions, 3 replicas, and 10K nodes. Y_{ijk} holds 300 billion variables, which is prohibitive in terms of both computation and memory usage. This observation motivates us to develop an efficient approximation in the following section.

3.2 Approximation

In this section, we present two complementary approaches for efficient computation: (1) reducing the number of decision variables and (2) speeding up the computation.

Reducing Number of Decision Variables: Two-Phase Approach To reduce the number of decision variables Y_{ijk}, representing all possible moves of the k-th replica of the j-th partition on the i-th node, we decide to restrict the moves (by restricting source and target nodes) to yield an approximate yet efficient solution, based on two key observations; we empirically evaluate the efficiency of our approach with respect to the offline optimal in Section 5.

First, as observed in Section 2, loads are skewed to a few hot-spots, *e.g.*, power law distribution. This suggests that we can safely localize the **source node** of a move to hot-spots. Second, only if we knew the maximal migration cost r_{max} of a single object a priori, we could easily eliminate any **target node** incurring higher costs than r_{max} from the model without compromising the optimality. However, as r_{max} is obtained after optimization, we start with its lower bound estimation r and iteratively increase the value by Δr until it converges to r_{max}, *i.e.*, a linear search, as in 'phase 2' described below. As over-estimating the value incurs high LP computation cost, a binary or an evolutionary search was not suitable for this purpose.

To reflect these two observations, our two-phase approach consists of (1) selecting a set of objects to move from all hot-spots, and (2) deciding a set of target nodes to which the selected objects are to be migrated.

Phase 1: Restricting Source Nodes/Objects

As discussed above, we restrict moves to those from hot-spots (with loads higher than C_i). From each hot-spot i, we then need to decide which set S_i of objects to move in order to reduce the loads below C_i. There can be various strategies as follows:

- Highest-load-first: Starting from the object with the highest request load, add objects to S_i in descending order of loads until the load is below the C_i threshold.
- Random: Selecting random objects from node i and adding to S_i until the load is below the C_i threshold.

Between these two strategies, we take the highest-load-first to minimize the number of objects to move. In Section 5, we show that this approach works well in practice.

Table 2. Notations for approximate ILP model

Notation	Description
X_{ij}	1 if i-th object moves to j-th node, 0 otherwise
C_j	Load capacity of j-th node
B_{ij}	Bandwidth cost for moving i-th object into j-th node
S_j	Set of objects selected to move from j-th node
L_i, L_{*j}	Load of i-th object, total load of j-th node
$R(i, r)$	Nodes within radius r from the node i-th object belongs to
G_p	Set of selected objects with partition index p to move from the overloaded nodes
F_q	Set of nodes with the same fault domain index q
I_{iq}	0 if q-th domain contains the same partition as i-th object, 1 otherwise

Phase 2: Target Node Selection

As discussed above, we can safely restrict our search for destination nodes to those with a migration cost less than r_{max}. As this value is unknown, we perform a linear search starting with a small r, *e.g.*, the cost of migrating into the nodes in the same rack, and run the ILP model. If a feasible solution exists, we stop searching. Otherwise, we expand the radius further $r + \Delta r$, and resume running the ILP model.

We now discuss how to formulate the ILP model from the revised strategy. Table 2 summarize decision variables and all constants. Based on these, we revise the ILP model in the previous section as follows:

$$\textbf{minimize} \quad \sum_i \sum_j X_{ij} B_{ij} \tag{7}$$

$$\textbf{subject to} \quad \forall j : \sum_i X_{ij} L_i - \sum_{i' \in S_j} \sum_{j'} X_{i'j'} L_{i'} + L_{*j} \leq C_j \tag{8}$$

$$\forall i : \sum_{j \in R(i,r)} X_{ij} = 1 \tag{9}$$

$$\forall i, \forall j \notin R(i,r) : X_{ij} \leq 0 \tag{10}$$

$$\forall p, \forall q : \sum_{i \in G_p} \sum_{j \in F_q} X_{ij} \leq 1 \tag{11}$$

$$\forall i, \forall q : \sum_{j \in F_q} X_{ij} \leq I_{iq} \tag{12}$$

$$\forall i, \forall j : X_{ij} = 0 \text{ or } 1 \tag{13}$$

In this revision, the decision variables are defined as a binary matrix X representing optimal movements of the replicas. If the i-th object moves to the j-th node, X_{ij} will be set to 1, and 0 otherwise. Using this representation, an objective function for minimizing such movement cost is formally stated in Eq. (7).

Table 2 describes all notations for representing the following constraints–Eq. (8) represents the capacity constraint for each j-th node. In this constraint, the first two terms represent the incoming load to the j-th node and outgoing load from the j-th node, respectively. Eq. (9) and (10) are the constraints for restricting the target nodes within the search radius r. Eq. (11) and (12) are the constraints to ensure fault tolerance by preventing objects with the same partition from being placed on the same fault domain similarly to Eq. (5) in the optimal ILP formulation. In particular, G_p denotes the group of all object replicas belonging to partition index p in the set of selected objects. Thus, Eq. (11) imposes a constraint such that candidate objects having the same partition index cannot be placed on the same fault domain while Eq. (12) imposes a constraint such that a candidate object cannot be placed on a fault domain which already has a replica from the same partition. Note that these two constraints are useful when a set of candidate objects is merged from nodes with overlapping search regions. For a single node, these constraints hold trivially as at most one copy of a replica partition can be hosted on the node.

In this formulation, we can significantly reduce the number of decision variables compared to the previous formulation, because we only consider the selected objects in hot-spots and constrain the search space within a specific radius. For instance, in the same scenario of 10M partitions, 3 replicas, and 10K nodes, if the number of the

selected objects in hot-spots is 10K×30 (1% of 3K objects per node on average) and the number of nodes within a typical rack is 40, the number of variables is about 12M, which is significantly less than the 300 billion in the previous model.

Speeding-up Computation We next describe how to speed up the computation using the following two approaches: (1) Divide-and-Conquer (D&C), and (2) relaxation to the LP model.

(1) Divide-and-Conquer: We formulate the ILP model to move all selected objects from the hot-spots to the candidate nodes within a specified cost radius. However, we observe that the computation for a hot-spot can be separated if the neighborhood defined by its cost radius does not overlap with that of others. That is, we can divide the given problem into sub-problems of optimizing moves for "disjoint" groups of hot-spots with overlaps, and merge the solutions without loss of accuracy, which shows significantly higher performance empirically. Fig. 5 illustrates the D&C approach for two disjoint groups of overlapping hot-spots. Unlike running ILP once for all hot-spots, D&C runs the ILP model independently and in parallel for two grey groups on $\{H_1, H_2\}$ and $\{H_3, H_4, H_5\}$.

Algorithm 1: Two-phase approximation: cost optimization for load balancing

```
1  M ← {}; // Initialize movement set
2  r ← r_0; // Initialize search radius
   // Merge the objects to move whose search regions are
   // overlapped
3  {S'_1, ··· , S'_n'} ← MergeObjects (S_1, ··· , S_n, r);
4  for k ← 1 to n' do
       // Find target nodes within r from the nodes
       // containing each group k
5      TN_k ← FindTargetNodes (k, r);
6      {X_ij | ∀i ∈ S'_k, ∀j ∈ TN_k} ← SolveLP (S'_k, TN_k, r);
7      if LP solution is feasible then
8          foreach i ∈ S'_k do
9              Let {X_i1, ··· , X_im} be a sorted list s.t. X_ij > 0;
10             for j ← 1 to m do
11                 if L_i + L_*j ≤ C_j then
12                     M ← M ∪ {(i, j)};
13                 break;
14 if ∃S'_k, LP solution is infeasible then r ← r + Δr;
15 if there still exist object(s) to move then goto line 3;
```

(2) Relaxation to LP: The ILP model, which requires decision variables to be integers, is more expensive than LP. We thus relax the model by changing the decision variables from binary to real numbers between 0 and 1, *i.e.*, by using constraint $\forall i, \forall j : 0 \leq X_{ij} \leq 1$ instead of Eq.(13). However, such relaxation can often return fractional values between 0 and 1 for some target nodes, *e.g.*, $X_{31} = 0.5$, $X_{32} = 0.3$, and $X_{33} = 0.2$. We thus use these scores to prioritize the target nodes by picking the one with the highest score. Meanwhile, if the movement causes violation of the constraints C1 and C2, we have to bypass it. In this example, we first check whether moving the object id 3 to the

node id 1 (as represented by X_{31}) is feasible to satisfy the constraints. If it is feasible, we move it. Otherwise, we iteratively check the next movement on X_{32} with the second highest score, and so on.

In summary, Algorithm 1 formally explains the optimization. There are three functional modules: (1) `MergeObjects`, merging the objects to move whose search radii overlap, (2) `FindTargetNodes`, finding the target nodes for each overlapping group, and (3) `SolveLP`, running the LP model for each overlapping group.

4 Implementation of Ursa on Windows Azure

This section describes our implementation of Ursa by using APIs provided by a real cloud system Windows Azure for deploying multiple servers in a distributed cloud environment. Our system consists of three main modules: (1) a resource manager that performs load balancing, (2) a load generator/trace replayer, and (3) a set of storage nodes. Fig. 6 illustrates the overall architecture. The resource manager periodically

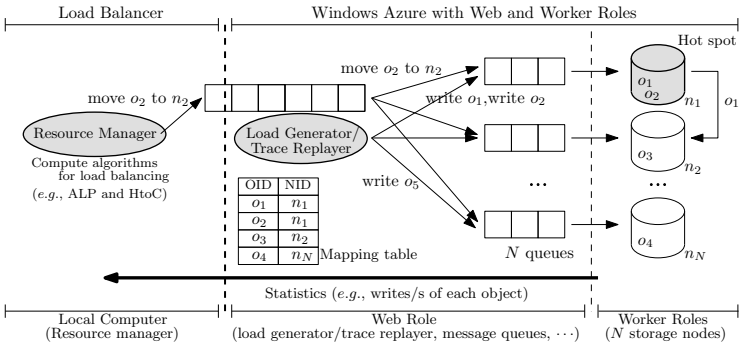


Fig. 6. Overall architecture of Ursa's prototype implementation on Windows Azure

collects load statistics of objects requested to the storage nodes and their location information, computes load balancing decisions by applying different algorithms such as ILP/LP models, and finally it generates messages to actuate the data migration operations and conveys them through a message queue to individual storage nodes. The load generator outputs storage requests by generating read and write messages for objects in the storage nodes according to Hotmail read/write requests logged. Similarly, the trace replayer takes as input a log of the read and write requests for running applications (*e.g.*, Hotmail) and issues I/O requests as per their temporal patterns. The location information for all objects is maintained in a mapping table corresponding to the MDS lookup table in Fig. 2, which is used to assign each request message to the corresponding storage node. The storage nodes receive the requests through their corresponding message queues. After processing the requests, the storage nodes send request acknowledgements which are used to compute load statistics for the resource manager. Each storage node handles both the read/write requests and data migration operations. Alternatively, our architecture can support a separate queue per storage node for data migration operations, similar to the principle of separating control and data plane in distributed systems. To actuate data migration between nodes, the source node transfers the corresponding objects to the destination node over a TCP channel; the mapping table is updated after

migration is completed. For correctness, write requests on objects being migrated are blocked until the migration operations are completed.

We implement the resource manager as a module running on a local computer, the load generator as a web role instance, and the storage nodes as worker role instances, where the web role and the worker role compose the application of Windows Azure. The web role instance is the interface to pass the migration operations decided by the resource manager and the request messages generated from the load generator/trace replayer to the storage nodes. We instantiate the web role as a large VM with 4 CPU cores, 7 GB memory, and 1,000 GB disk because it interacts with multiple worker role instances. The worker role instances execute write/read requests on data objects delivered from their queues, or move their objects to other worker role instances as specified by the resource manager when hot-spots occur. We instantiate them as small VMs with 1 CPU core, 1.75 GB memory, and 20 GB disk.

5 Experiments

Section 5.1 validates the effectiveness of our load balancing approach on a simulator over large-scale datasets up to ten thousand nodes and thirty million objects. Section 5.2 then evaluates our framework on the real system implemented on Windows Azure.

5.1 Evaluation on Simulator

This section evaluates the effectiveness of the load balancer by synthetically generating load statistics based on Hotmail traces. We first report our experimental settings and then validate the effectiveness and the scalability of our approach over synthetic datasets.

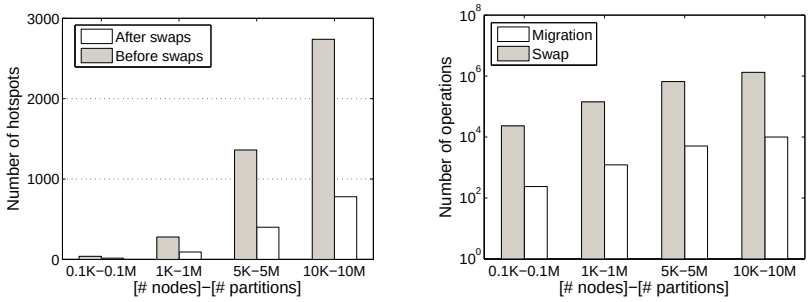
Settings. We synthetically generate several datasets (denoted as the naming convention of [number of storage nodes]-[number of partitions]) from 0.1K-0.1M to 10K-10M to closely emulate the statistics aggregated from storage servers in Hotmail clusters. More specifically, given the parameter pairs X-Y, we generate the dataset to mimic Hotmail traces by the following procedure:

1. Generate Y partitions where every partition has $l = 3$ replicas, and assign workload values following the power-law distribution into partitions. We assume that the workload of primaries is two times higher than their secondaries corresponding to the read to write ratio observed in our case-study workload.
2. Randomly distribute X nodes in two-dimensional Euclidean space. We attempt to reproduce the bandwidth cost for passing a message between two servers that may be hosted either on the same rack, neighboring racks, or topologically far apart so that it is proportional to the Euclidean distance.
3. Randomly distribute all replicas into nodes satisfying constraints such that no node can include two replicas corresponding to the same partition.
4. Assign load capacities C_i's to X nodes. In particular, when the total workload of the node n_i is L_{*i} by all the replicas assigned to n_i, the distribution of the ratio L_{*i}/C_i over X nodes follows the distribution illustrated in Fig. 1 under the assumption that the total workload of the node is proportional to the CPU utilization. The nodes with L_{*i}/C_i higher than 0.7 are regarded as hot-spots.

We then measure (1) the running time (the placement decision time without the reconfiguration time for actually passing the messages) and (2) the migration cost (sum of the distances between two nodes to move objects) and (3) the number of migration operations. Specifically, we measure the reconfiguration cost as the cost of migrations (and not swaps), because we apply the same swap optimization strategy for all algorithms, namely iteratively swapping the primary in the hottest node and the secondary in a cold node until no swap operation is applicable. In particular, we apply the swap operations with 90% success probability to emulate a small fraction of out-of-date secondaries. The swap operations for the out-of-date secondaries may require more expensive cost than the migration operations to synchronize their state. That is, we regard the swap operations for such secondaries as the failed operations. As we consider only migrations, all the message sizes are the same and thus the number of migration operations can also be interpreted as the bandwidth cost, which would be directly proportional to these numbers. The bandwidth cost is an important metric for reconfiguration costs as data center networks tend to have high bandwidth oversubscription [13].

We compare our best approach (especially the approximation, denoted as ALP (for Approximate LP), in Section 3) with a natural baseline called HtoC (for Hottest to Coldest), which iteratively eliminates hot-spots until no hot-spots exist by moving the hottest objects in a hot-spot node into the coldest node. (Note that, though we use a single machine, ALP can be easily parallelized for each non-overlapping group of the search regions.) HtoC is a simplified variation of the greedy algorithm suggested by Kunkle and Schindler [16], as the proposed algorithm "as is" cannot apply to a larger scale setup because of its complex objective function of minimizing an imbalance. As another baseline, we adopt a metaheuristic optimization algorithm using a simulated annealing technique called SA (for Simulated Annealing), similar to an evolutionary algorithm approach in [16]. SA randomly moves the objects into other nodes to minimize the load variance. We note that SA has eight parameters to tune and failing to optimize these parameters may lead SA to generate excessive migrations. For our evaluations, we empirically tune these parameters including setting the maximum number of migrations as 10K and the maximum running time as 30 minutes.

Results. We first motivate the necessity of migration operations, then evaluate our approximation techniques to speed up our ILP formulations, and finally compare our best approach with the baselines.

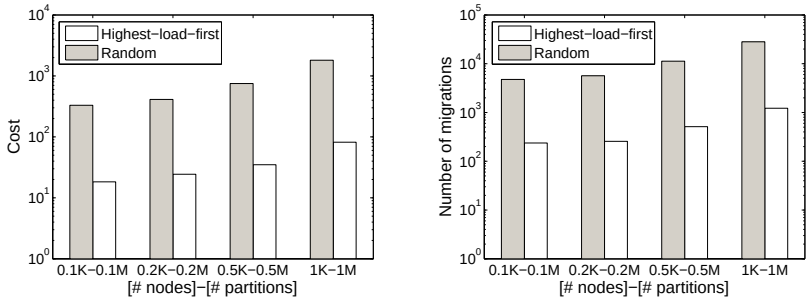


(a) Number of hot-spots (b) Number of operations

Fig. 7. Effect of swap and migration operations

To validate the necessity of the migration operations, we first evaluate the number of remaining hot-spots after applying only swap operations. Fig. 7 shows such results over various scales of datasets. Observe from Fig. 7(a) that there are still about one third of the initial hot-spots even after swaps, and Fig. 7(b) shows the numbers of swap and migration operations respectively until all hot-spots are eliminated.

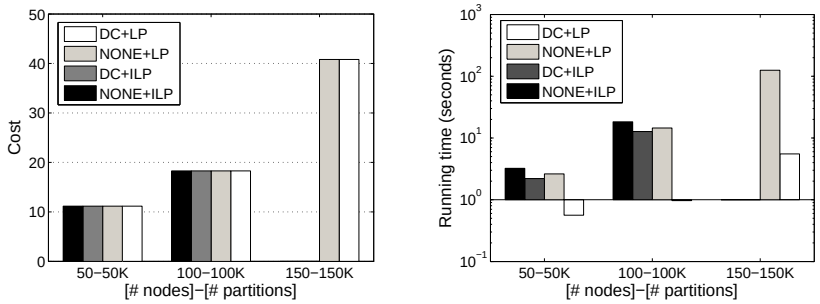
Fig. 8 shows the effect of the object selection strategies in our two-phase approach for reducing the number of decision variables. In particular, we compare two simple approaches, highest-load-first and random. Fig. 8(a) shows the total cost for migration operations. Observe that the highest-load-first strategy is much cheaper than the random in all settings, which is explained by the number of migration operations in Fig. 8(b). As the workload of objects follows the power-law distribution, the highest-load-first selects far fewer objects with the highest load than the random. Thus, in all experiments hereafter, we use the highest-load-first strategy in the first phase.



(a) Cost (b) Number of migrations

Fig. 8. Effect of selecting source objects

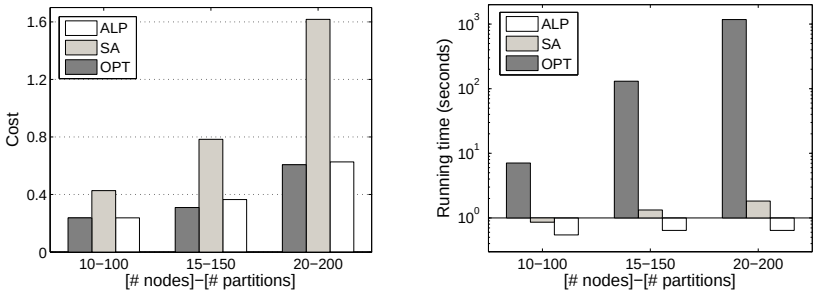
Fig. 9 shows the effect of our two speedup techniques suggested in Section 3.2, divide-and-conquer and relaxation from ILP to LP. We evaluate combinations of these two techniques (denoted as [NONE or DC]+[ILP or LP]) with respect to the migration cost and the running time in Fig. 9(a) and (b), respectively. Observe that, while all combinations show the same or almost the same cost, DC+LP applying both optimization techniques shows much shorter running time than the others. We do not plot the results of NONE+ILP and DC+ILP in 150-150K configuration because their running times become too large (more than two days).



(a) Cost (b) Running time

Fig. 9. Effect of speeding up computation

Fig. 10 compares the optimal cost computed by the optimal ILP model in Section 3.1 (denoted as OPT) with our best approach ALP, *i.e.*, DC+LP. Observe that ALP shows a similar cost to OPT in Fig. 10(a) while ALP exhibits a significantly higher scalability than OPT in the running time of Fig. 10(b). Note that we evaluate small-scale datasets due to the low scalability of OPT.



(a) Cost (b) Running time

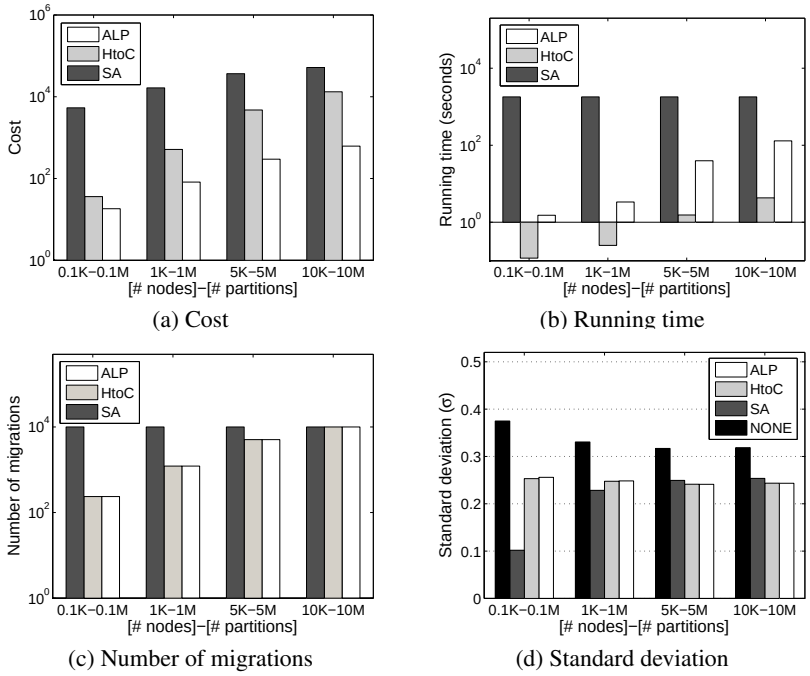
Fig. 10. Performance comparison of ALP with respect to SA and the optimal solution

Fig. 11 shows the effect over large-scale datasets: Fig. 11(a) and (b) show the total cost and the running time of ALP. Observe that ALP is a clear winner in all settings except for the running time. In terms of cost, ALP is 80-300 times cheaper than SA and 2-20 times cheaper than HtoC. Meanwhile, the running time of ALP is comparable to simple heuristics and remains reasonable, *i.e.*, 130 seconds, even in the largest setting of 10K-10M. Fig. 11(c) shows the effect over the number of migrations (proportional to the bandwidth cost). ALP and HtoC have the same number of operations, because both adopt the same method, highest-load-first, for selecting the objects to be migrated from the hot-spots. Fig. 11(d) shows the standard deviation σ over the loads of storage nodes. NONE depicts σ before the reconfiguration. SA has the lowest σ at small scale 0.1K-0.1M according to the goal of minimizing σ. However, as the scale increases, it becomes less effective. In particular, in 5K-5M and 10K-10M configurations, we observe that, when ALP and HtoC eliminate hot-spots, they result in slightly lower σ than SA.

5.2 Evaluation on Windows Azure

This section describes the evaluation of our prototype on Windows Azure described in Section 4. We first report our experimental settings and then discuss the results.

Settings. We simplify some environmental settings for convenience of evaluation. More specifically, we assume that all objects have the same block size, *e.g.*, 32MB, and the load generator only generates the write messages on those objects. (When the read and write requests are blended, we can easily quantify the capacity of the node as a weighted sum w_1·reads/s + w_2·writes/s.) Under these assumptions, before executing the above setup on Azure, we measure the load capacity C_i of the storage node in the unit of writes/s. We first find a threshold value T writes/s such that, for the storage node, the response time rapidly increases, and set $C_i = 0.7T$. We then generate T writes/s messages for the hot-spots (20% of all the nodes) and $0.4T$ writes/s messages for the other nodes. Meanwhile, in this setting, we do not distinguish between primary and



(a) Cost (b) Running time

(c) Number of migrations (d) Standard deviation

Fig. 11. Comparing (a) the cost, (b) running time, (c) number of migrations, and (d) standard deviation under different configurations for ALP, HtoC, and SA

secondary replicas in order to focus on the migration operation incurring significantly higher cost than the swap operation.

We perform evaluations in the following two scenarios–**Random** and **Controlled**. In **Random**, we leave Windows Azure to deploy nodes, and as a result, have nodes deployed randomly in the data center network (*i.e.*, similar migration costs between each source-destination pair). In contrast, in **Controlled**, we emulate data center scenarios where we have control over which nodes go to which racks. In these scenarios, migration costs can vary based on the underlying rack assignment. More specifically, in our system, the nodes of a data center network form a hierarchical structure of the spanning tree topology [13]. Thus, as traffic goes up through switches and routers, the oversubscription ratio increases rapidly, which means that movement of the data across rack or cluster causes significant delay compared to the movement within the same rack or cluster. As Windows Azure does not allow user-specified deployment of VMs to racks, we emulate arbitrary rack assignment by adding time delays.

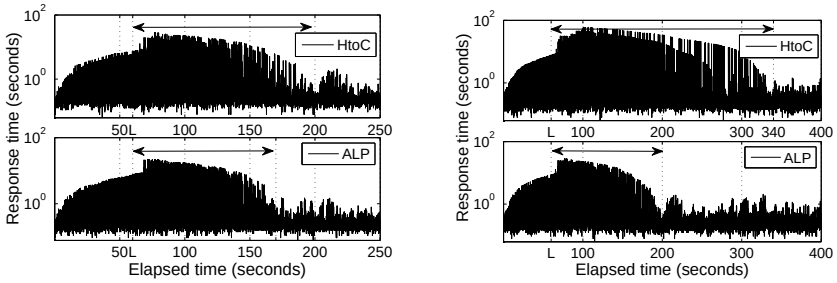
We evaluate the effectiveness of our approach over various scales of datasets, *e.g.*, 30-900 (30 storage nodes and 900 partitions). Note that we assume the number of replicas is three in all setups. Thus, 30-900 implies that the number of total objects is 2,700.

Running experiments in Windows Azure enables us to measure "elapsed time" which includes the time for running ALP and the time it takes for the system to stabilize (*i.e.*, until the system responds to a request within 1 second). We thus use two

metrics in this section: (1) the elapsed time and (2) the bandwidth cost (measured as the number of operations as discussed before.)

We then compare our proposed method with HtoC, a winner solution among the baseline approaches as shown in the experiment on our simulator.

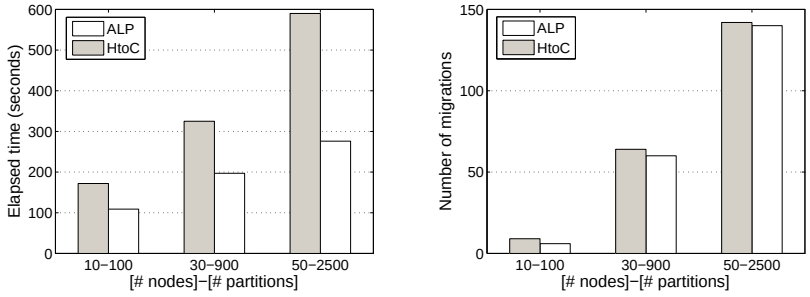
Results. Fig. 12 shows the response time of each request over elapsed time with the dataset 30-900. We observe that the response time rapidly increases before the load bal-



(a) Random (b) Controlled

Fig. 12. Response time over elapsed time

ancer is called at 60 second point marked as 'L' on x-axis. After the call, the response time starts to be stable passing through the reconfiguration by using two approaches. We argue **Random**, where migration costs are more or less the same, is not a favorable scenario for our proposed system, as a randomly selected node with no computation and the one selected from our optimization computation would incur similar migration costs (*i.e.*, little margin for optimization). However, even in this unfavorable setting, ALP achieves stable state about 30 seconds earlier than HtoC in Fig. 12(a). As expected, our performance gain is much clearer in **Controlled**, where migration costs differ significantly. As shown in (b), ALP is about 140 seconds faster than HtoC.



(a) Elapsed time (b) Number of migrations

Fig. 13. Elapsed time and number of migrations for reconfiguration in Controlled setting

Fig. 13 presents our results in varying settings of **Controlled** scenarios, *i.e.*, 10-100, 30-900, and 50-2500. We observe from Fig. 13(a) that the performance gap between the two approaches increases as the data scale increases, *e.g.*, two times faster in 50-2500, which suggests the scalability of our approach. Note from Fig. 13(b) that two approaches have a similar number of migrations because both select the objects to move from the hot-spots using the highest-load-first.

6 Related Work

As cloud systems have become popular as cost-efficient and agile computing platforms, there have been various efforts to efficiently manage them. For instance, pMapper [24] performs cost-aware server consolidation and dynamic VM placement to minimize power. Rhizoma [26] leverages constraint logic programming (CLP) to minimize cost for deploying and managing distributed applications while satisfying their resource requirements and performance goals. However, these approaches do not address load balancing in cluster storage systems or perform migration in a topology-aware manner.

Meanwhile, while many existing methods for load balancing perform dynamic placement of individual data objects [3, 16, 25, 27], they aim at evenly distributing loads over all nodes. However, finding load balancing schemes with minimal re-configuration costs is a variant of the bin-packing or knapsack problem, which is inherently expensive and solvable offline [11]. As a result, existing solutions cannot scale to cloud-scale systems with more than ten thousand nodes incurring prohibitive reconfiguration costs. For instance, Kunkle and Schindler [16] tackle this problem by applying a greedy search, an evolutionary algorithm, and an optimal ILP model. However, these algorithms are deployed and evaluated in a small scale system, *e.g.*, six nodes, and it is non-trivial to scale these approaches for a larger scale.

In clear contrast, our approach aims to scale up to tens of thousands of nodes by eliminating hot-spots to minimize the reconfiguration cost. Similarly motivated by the challenge for scaling down peak loads, Everest [19] proposed to off-load the workload from overloaded storage nodes to under-utilized ones. A key difference is Everest targets to flash crowds with bursty peaks. To optimize for workload of this type, Everest off-loads writes to pre-configured virtual storage nodes temporarily then later reclaims to original nodes lazily when the peak subsides. Meanwhile, Ursa has a complementary strength of handling long-term changes in load and making topologically-aware migration decisions.

Ursa has advantages for read-heavy workloads, such as Hotmail workload studied in this paper. In Everest, only writes are off-loaded and such write offloading can increase read latency, as a write offloaded to multiple locations with data versioning, *i.e.*, N-way offloading, requires the following read request to find the latest write (which is bottlenecked by the slowest server). Another restrictive assumption of Everest for Hotmail workloads is that reads to off-loaded writes (recent data) are rare, while such pattern can be frequently observed in many applications, such as email where a user immediately checks for incoming mail.

Data migration challenge has been raised in the context of database systems built on cloud computing environments as well [5–7, 9, 10, 17, 20, 21]. These systems can be categorized into those using replication strategies [5, 17, 20, 21] or limited dynamic data migration [6, 7, 9, 10]. However, existing dynamic migration approaches assume some oracle to determine the source and target nodes of migration and leave the problem of identifying cost-optimal dynamic reconfiguration as future work, while we identify a scalable and efficient solution for such reconfiguration.

7 Conclusion and Future Work

We have presented a new middleware system Ursa for load balancing. Ursa formulates load balancing as an optimization problem and develops scalable optimization

techniques. Our evaluations using traces from Hotmail are encouraging: Ursa scales to large systems while reducing reconfiguration cost and can compute placement decisions within about two minutes for 10K nodes and 10M objects in cloud-scale systems.

As future work, we consider the following issues:

First, regarding **interference during reconfiguration**, reconfiguration does consume resources, CPU and bandwidth, that could otherwise be used to serve requests. To control interference, we can build a mechanism, such as TCP Nice [23] developed for background data transfer, to automatically tune the right balance between interference and resource utilization. For workloads with many sudden transient changes, *e.g.*, flash crowds on the Internet, interference and reconfiguration costs may become significant. One approach would be to specify a resource budget for reconfigurations to ensure that interference to applications is within acceptable bounds.

Second, regarding **the unit of migration**, in this paper, we regard an object as the unit of migration, but supporting a varying degree of granularity may enable further optimization. For instance, partitions or chunks may be collected into a group, such that a single MDS message is sufficient to migrate or swap the whole group. That is, adopting a coarser unit can reduce the number of MDS operations.

Lastly, regarding **trade-offs between reconfiguration cost and speed**, our framework does have hidden knobs controlling this trade-off. For instance, in divide-and-conquer computation, a small initial radius r would incur less computation cost for LP but may require more rounds of LP computation, while larger value requires a single but more costly LP computation with possibly high reconfiguration cost. Meanwhile, divide-and-conquer computation does not compensate accuracy if (1) regions do not overlap and (2) a region has an infeasible solution. Though divide-and-conquer can still be used for faster LP computation when these conditions are violated, the process will identify sub-optimal results. Also, when using the greedy highest-load-first strategy, we may pick a very hot object that can only be migrated to a cold node far away, whereas we could pick several relatively less hot objects incurring lower migration cost. A joint optimization minimizing the overall cost would avoid this case and thus reduce reconfiguration cost, but may incur higher LP running time.

Acknowledgments

We thank our shepherd Steven Ko and the anonymous reviewers for their feedback. We are grateful to Hua-Jun Zeng for providing us with an initial simulator code and Boon Yeap for helping us understand the Hotmail datasets. This research was supported by Microsoft Research and the MKE (The Ministry of Knowledge Economy), Korea, under IT/SW Creative research program supervised by the NIPA (National IT Industry Promotion Agency) (NIPA-2010-C1810-1002-0003).

References

1. Amazon S3. http://aws.amazon.com/s3/.
2. Windows azure. http://www.microsoft.com/windowsazure/.
3. M. Abd-El-Malek, W. V. C. II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: Versatile Cluster-based Storage. In *Proc. of FAST*, 2005.

4. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proc. of OSDI*, 2006.
5. C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. In *Proc. of VLDB*, 2010.
6. C. Curino, E. Jones, Y. Zhang, E. Wu, and S. Madden. Relational Cloud: The Case for a Database Service. Technical Report MIT-CSAIL-TR-2010-014, MIT, 2010.
7. S. Das, S. Nishimura, D. Agrawal, and A. E. Abbadi. Live Database Migration for Elasticity in a Multitenant Database for Cloud Platforms. Technical report, UCSB, 2010.
8. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of OSDI*, 2004.
9. A. Elmore, S. Das, D. Agrawal, and A. E. Abbadi. Who's Driving this Cloud? Towards Efficient Migration for Elastic and Autonomic Multitenant Databases. Technical report, UCSB, 2010.
10. A. Elmore, S. Das, D. Agrawal, and A. E. Abbadi. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *Proc. of SIGMOD*, 2011.
11. E. A. Eric, S. Spence, R. Swaminathan, M. Kallahalla, and Q. Wang. Quickly Finding Near-optimal Storage Designs. *ACM Transactions on Computer Systems*, 23:337–374, 2005.
12. S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. *SIGOPS Operating System Review*, 37:29–43, 2003.
13. A. G. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proc. of SIGCOMM*, 2009.
14. A. Gulati, C. Kumar, I. Ahmad, and K. Kumar. BASIL: Automated IO Load Balancing Across Storage Devices. In *Proc. of FAST*, 2010.
15. F. S. Hiller and G. J. Lieberman. *Introduction to Operations Research*. McGraw-Hill, 8th edition, 2005.
16. D. Kunkle and J. Schindler. A Load Balancing Framework for Clustered Storage Systems. In *Proc. of HiPC*, 2008.
17. W. Lang, J. M. Patel, and J. F. Naughton. On Energy Management, Load Balancing and Replication. *SIGMOD Record*, 38:35–42, 2010.
18. W. Litwin. Linear Hashing: A New Tool for File and Table Addressing. In *Proc. of VLDB*, 1980.
19. D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron. Everest: Scaling Down Peak Loads through I/O Off-loading. In *Proc. of OSDI*, 2008.
20. S. Savinov and K. Daudjee. Dynamic Database Replica Provisioning through Virtualization. In *Proc. of CloudDB*, 2010.
21. H. V. Tam, C. Chen, and B. C. Ooi. Towards Elastic Transactional Cloud Storage with Range Query Support. In *Proc. of VLDB*, 2010.
22. E. Thereska, A. Donnelly, and C. Narayanan. Sierra: A Power-proportional, Distributed Storage System. In *Technical Report MSR-TR-2009-153*, 2009.
23. A. Venkataramani, R. Kokku, , and M. Dahlin. TCP Nice: A Mechanism for Background Transfers . In *Proc. of OSDI*, 2002.
24. A. Verma, P. Ahuja, and A. Neogi. pMapper: Power and Migration Cost Aware Application Placement in Virtualized Systems. In *Proc. of Middleware*, 2008.
25. S. A. Weil, S. A. Brand, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A Scalable, High-performance Distributed File System. In *Proc. of OSDI*, 2006.
26. Q. Yin, A. Schüpbach, J. Cappos, A. Baumann, and T. Roscoe. Rhizoma: A Runtime for Self-deploying, Self-managing Overlays. In *Proc. of Middleware*, 2009.
27. L. Zeng, D. Feng, F. Wang, and K. Zhou. A Strategy of Load Balancing in Objects Storage System. In *Proc. of CIT*, 2005.