

# A Load Balancing Framework for Clustered Storage Systems

Daniel Kunkle and Jiri Schindler

Northeastern University and NetApp Inc.

**Abstract.** The load balancing framework for high-performance clustered storage systems presented in this paper provides a general method for reconfiguring a system facing dynamic workload changes. It simultaneously balances load and minimizes the cost of reconfiguration. It can be used for automatic reconfiguration or to present an administrator with a range of (near) optimal reconfiguration options, allowing a tradeoff between load distribution and reconfiguration cost. The framework supports a wide range of measures for load imbalance and reconfiguration cost, as well as several optimization techniques. The effectiveness of this framework is demonstrated by balancing the workload on a NetApp Data ONTAP GX system, a commercial scale-out clustered NFS server implementation. The evaluation scenario considers consolidating two real world systems, with hundreds of users each: a six-node clustered storage system supporting engineering workloads and a legacy system supporting three email servers.

## 1 Introduction

The basic premise of clustered storage systems is to offer fine-grained incremental capacity expansion and cost-effective management with performance that scales well with the number of clients and workloads [1,2,3,4,5,6]. To address load imbalance, most previously proposed architectures either dynamically redistribute individual data objects and hence load among individual nodes in response to changing workloads [2,5], or use algorithmic approaches for randomized data allocation (e.g., variants of linear hashing [7]) to distribute workload across cluster nodes [4,6].

However, the first approach is not well suited for enterprise storage systems. First, deployed systems typically collect only cumulative statistics over a period of time [8,9], as opposed to detailed traces with per-request timings [10,11]. Yet, systems with data migration at the level of individual objects [2,5] typically use techniques that require detailed traces to make informed decisions [12]. Second, workloads do not always change gradually. They often do so in distinct steps, for example, during consolidation when an existing system inherits a legacy system workload.

A complementary approach to balancing load across system components is to use offline solvers [8,13]. They typically use a variant of the bin-packing or knapsack problem [14] to find a cost-efficient system configuration (solution). They require only a high-level workload description [15] and capacity or performance system model. While such solvers have been shown to be effective for building an enterprise-scale system from the ground up, they are less suitable when already deployed systems grow or experience workload changes over time. Previous work proposed to iteratively apply

constraint-based search with bin packing [16]. However, doing so does not take into account the *cost* of the system configuration change and the resulting impact of potential data movement on system performance.

To address the shortcomings of the existing solutions, we have developed a load-balancing framework with two primary objectives: (1) It should be modular and flexible, allowing for a range of definitions for *system load* and *imbalance*, as well as for many types of optimization techniques instead of using one specific algorithm. (2) The cost of reconfiguration should be a primary constraint, guiding which solutions are feasible and preferred. We use a combination of analytical and empirical estimates of cost, along with a measure of system imbalance, to define a multiobjective optimization problem.

Using this framework, we implemented a load balancing system tailored to the specifics of the NetApp Data ONTAP GX cluster [1]. Our approach is grounded in *real* features of our deployed systems. We are motivated to find *practical* solutions to problems experienced by real users of our systems; for example, how to best consolidate existing application workloads and legacy systems into one easier-to-manage cluster. We focus on balancing the load of an already operational system; a scenario more likely to arise in practice than designing a new system from the ground up. We also explore the use of more formal techniques in the context of production enterprise systems. This motivation has been recently echoed by a call to employ optimization methods already in use by the operations-research community in place of more ad-hoc techniques prevalent in the computer systems community [17]. We demonstrate the applicability of our approach using an internally deployed Data ONTAP GX cluster hosting engineering workloads and home directories. We examine a scenario of consolidating storage in a data center—rolling a legacy Data ONTAP 7G system with e-mail server workload into a Data ONTAP GX cluster supporting software development.

## 2 Load Balancing Framework Overview

The primary goal of our framework is to provide an abstract method for load balancing that is applicable to a wide range of workloads and systems, as well as allowing for many different policies and strategies. To facilitate this, we have divided the framework into four modules, each of which can be modified without requiring significant changes to any of the others. At a high level, the framework represents a canonical decision system with a feedback loop—a model that has previously been shown to work well for storage system configuration [16].

Figure 1 shows the general structure and components of our modular load-balancing framework. The Observe Load module records, stores, and makes available a set of statistics that characterize the load on the system. The Detect Imbalance module calculates the *imbalance factor* of the system—a measure of how evenly the load is distributed. If the imbalance factor passes some threshold, the Optimize Reconfiguration Plan module is invoked. It determines a set of system reconfigurations that will mitigate the load imbalance while minimizing the cost of those reconfigurations. The module Execute Reconfiguration Plan executes a series of system-level operations.

We now describe each module in greater detail. We first give a general definition of the module, followed by the details of how that module is applied to the specifics of the Data ONTAP GX cluster. The system architecture is detailed elsewhere [1].

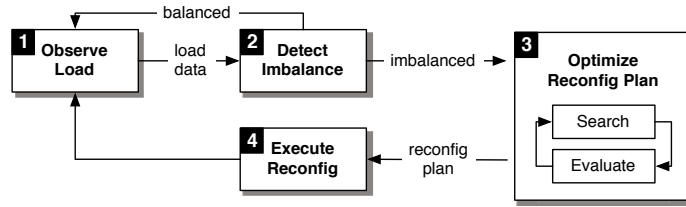


Fig. 1. Flow diagram of the load-balancing framework

## 2.1 Observe Load

We characterize load with two concepts: *element* and *load component*. An *element* is an object in the storage system that handles load, that is, something that can be *overloaded*. This can be a hardware component (e.g., a controller with CPU and memory), an architectural component, or a software module. A *load component* is an indivisible source of load that can be migrated, or otherwise reconfigured, to alleviate the overloading of an element. This can be a single data object (e.g., a file) or a logical volume.

An important factor for observing load is the frequency of data collection. In practice, this is driven by the constraints of the collection method used, for example, tracing every request versus having an agent periodically sample various counters. Another important factor is the time period over which the load is observed and decisions are made. If that time period is too short, the system may react too quickly, attempting to re-balance after a transitory spike in load. Conversely, if the time period is too long, the system may remain in an imbalanced state for an extended period of time. This period also depends on how long it takes to implement changes. In particular, it should be long enough to allow a new steady state to be reached and evaluated before acting again. Finally, load can be expressed by one or more variables, for example, we can collect both the throughput and latency of I/O operations over some time period.

*Application:* Load imbalance in the Data ONTAP GX cluster can be caused in two ways. First, nodes (Nblades) can be overloaded by client requests, which they must process and forward. This imbalance can be mitigated by directing client requests away from heavily loaded nodes by moving an existing virtual interface (VIF) from one Nblade to another. Second, a node (Dblade) may be overloaded if the data it contains is heavily requested. This imbalance can be mitigated by migrating data (i.e., volumes with separate file system instances), or by creating load-balancing volume mirrors on other nodes with lighter load. Dblades and Nblades constitute elements, each with their respective load components. We demonstrate their interrelationship with two scenarios.

**Scenario 1: Balancing Client-Request Handling.** An *element* is a networking component (Nblade), which handles client connections and routes requests to the target data component (Dblade). A *load component* is a virtual interface (VIF), through which clients requests are handled. Each VIF is assigned to a single Nblade. The system can be reconfigured by migrating a VIF from one Nblade to another; the new node will handle all future client requests through that VIF.

**Scenario 2: Balancing Data Servicing.** An *element* is a data component (Dblade), a compute node that receives requests from Nblades and serves the requested data to the client. A *load component* is a volume, a file system subtree, containing the directories and files in that subtree. The system can be reconfigured by migrating a volume from one Dblade to another or mirroring the volume on another Dblade.

For our evaluation in Section 3, we consider the second scenario, defining load in terms of operations per second contributed by each load component (volume).

## 2.2 Detect Imbalance

Module 2 compresses the multidimensional expression of load to a single value. This *imbalance factor* describes the extent to which the load on the system is evenly distributed. When this value passes some threshold, Module 3 is invoked to produce a set of possible reconfiguration plans. Even though this module reduces the load description to a single value to determine when rebalancing is appropriate, the full load description is still made available to the optimization methods in Module 3.

The imbalance factor computation uses two different functions to (i) compress the temporal load on each element to a single scalar value and (ii) calculate the final imbalance factor from the set of per element load values. We have developed a number of possible functions for each of these steps. By using different functions, one can choose under which conditions rebalancing occurs and which possible reconfigurations are optimal. For purposes of describing these functions, we define the following notation. Let  $u_t^e$  be the load on element  $e$  at time  $t$ . For example,  $u_t^e$  could be the number of I/O operations performed through element (node)  $e$  over some period of time (e.g. one minute). The values  $u_1^e, u_2^e, \dots, u_n^e$  would then define the number of I/O operations served by  $e$  over the last  $n$  minutes. Let  $L^e$  define the load on  $e$  with the temporal component removed.

**Reduction of Temporal Component.** First, we present three possible functions for removing the temporal component from the load observed on each element. Note that the functions are applicable regardless of the frequency of observations (i.e., the granularity of our measurements) or the period over which we apply the given function. The functions are defined in Table 1. The simple sum function adds the load on  $e$  at each time, placing equal weight on all possible load values. This is equivalent to a moving average of the load on an element.

The polynomial function emphasizes large utilization values. This can be useful in identifying situations where an element sees a large spike in utilization for a small period of time. With the simple sum function above, this spike would be lost. With this polynomial function, the spike is emphasized and corresponds to a larger increase in

**Table 1.** Reduction of temporal component functions

<i>Simple Sum</i>	<i>Polynomial</i>	<i>Threshold</i>
$L^e = \sum_{i=1}^n u_i^e$	$L^e = \sum_{i=1}^n (u_i^e)^\alpha$	$L^e = \sum_{i=1}^n (u_i^e \cdot T(u_i^e, k)); \quad T(u_i^e, k) = \begin{cases} 0 & \text{if } u_i^e < k \\ 1 & \text{if } u_i^e \geq k \end{cases}$

**Table 2.** Requirements for imbalance function

<i>Definition</i>	<i>Description</i>
$0 \leq f(\mathbf{L}) \leq 1$	The range of values is from zero to one.
$f(0, 0, \dots, 1) \rightarrow 1$	The maximum value is defined as a single element handling all load. Any load handled by more than one element should have a value less than one.
$f(1/n, 1/n, \dots, 1/n) \rightarrow 0$	The minimum value is defined as a perfectly balanced load. Any other load should have a value greater than zero.
$f(a, a + \epsilon, \dots) > f(a + \frac{\epsilon}{2}, a + \frac{\epsilon}{2}, \dots)$	Moving load from a more loaded element to a less loaded one reduces the value.
$f(\mathbf{L}) < f(0, \mathbf{L})$	Adding a new element with zero load increases the value.

$L^e$  than if that same load had been evenly spread over time. In the polynomial function definition  $\alpha > 1$ ; larger values of  $\alpha$  emphasize more high utilization values.

In some cases, we may care only about imbalances that cause elements to be overloaded. Imbalances that do not cause overloading can be more easily tolerated, and may not require reconfiguration (especially if that reconfiguration is costly). We define a threshold function,  $T(u_i^e, k)$ , where  $k$  is a parameter defining the threshold utility value at which we consider an element to be overloaded.

**Imbalance Function.** Once we have compressed the temporal component of the load description, we use another function to calculate the imbalance factor. Instead of choosing a function directly, we first construct a set of requirements for such a function (listed in Table 2). These requirements formally capture the intuitive notion of *balanced*. Given these requirements, we develop a function,  $f(\mathbf{L})$ , that satisfies them. Note that there are multiple functions that satisfy the necessary criteria, but we consider only one here.

First, we normalize all load values  $\lambda^e = L^e / \sum_{i=1}^n L^i$  and let  $\mathbf{L} = \{\lambda^1, \lambda^2, \dots, \lambda^n\}$  be the set of load values over all elements. None of the common statistical measures of dispersion satisfies all of the requirements, including range, variance, and standard deviation. One possible function that takes into account all of the properties in Table 2 is one based on entropy. We define the *normalized entropy* of a load distribution to be

$$f(\mathbf{L}) = 1 - \frac{\sum_{i=1}^n (\lambda^i \log \lambda^i)}{\log \frac{1}{n}}$$

where the numerator is the traditional definition of entropy, and the denominator is the normalizing factor. We orient the scale by taking the difference with 1.

*Application:* This module is independent of the target system. It acts only on the abstract *elements* and *load components* defined in Module 1.

### 2.3 Optimize Reconfiguration Plan

Once an imbalance has been detected, we determine a set of configuration changes that rebalance the load. The framework does not limit the kinds of configuration changes that are possible. Instead, the constraints are imposed by the system architecture. For

example, this could include migrating a data unit such as single volume from one node to another or creating a load-balancing volume mirror. Each of these changes could potentially increase or decrease the load observed on each element in the system. We call a set of configuration changes a *reconfiguration plan*.

The goal of Module 3 is to determine a reconfiguration plan that minimizes both the imbalance of the system and the cost of the reconfiguration. Because of these two competing objectives, there may not be a single reconfiguration plan that optimizes both. Instead, we discover a range of reconfiguration plans, which emphasize each of the objectives to a different degree. Module 3 is further broken down into two independent components: evaluation, which calculates the objectives and total cost of any possible reconfiguration plan; and search, which determines which of the many possible reconfiguration plans to evaluate. In any practical scenario, it is not feasible to evaluate all possible reconfiguration plans, and so the search component must be more intelligent than exhaustive search.

There are many possible search techniques we could apply—one of our goals is to compare a number of these methods. We choose three methods: greedy algorithms, evolutionary (or genetic) algorithms, and integer programming. The following paragraphs outline how we estimate reconfiguration costs and describe the details of each of the three optimization methods applied within our framework.

**Objectives and Costs.** The objective of a system reconfiguration is to mitigate a load imbalance. Specifically, we seek to minimize the imbalance factor of the resulting load, while simultaneously minimizing the cost of the reconfiguration. We define the resulting load to be what the load would have been had the reconfigurations been made before the current load was observed. Calculating the cost of a reconfiguration is specific to the target system and is covered at the end of this section.

**Greedy Algorithm.** Greedy algorithms, such as hill climbing, are a search technique that combines a series of locally optimal choices to construct a complete solution. We are optimizing with respect to multiple objectives, and so there is usually not just a single optimal choice at each step. Our approach is to randomly select one of the non-dominated possibilities at each step. A non-dominated solution is one for which there is no other solution with both a lower cost and lower imbalance factor. Algorithm 1 defines our greedy approach. It is specific to data migration but can be easily adapted to other reconfiguration options.

**Evolutionary Algorithm.** Evolutionary algorithms work by maintaining a *population* of possible solutions, and explore the search space by recombining solutions that are found to perform better. In this way, they are similar to natural selection. We use an algorithm based on the Strength Pareto Evolutionary Algorithm (SPEA) [18], a type of multiobjective evolutionary algorithm (MOEA).

The algorithm uses three primary input parameters: the population size,  $s$ ; the number of generations,  $n$ ; and the archive size,  $m$ . The population size is the number of possible solutions considered at one time. One *generation* is defined as the process of creating a new population of solutions, by recombining solutions from the current population. So the total number of possible solutions evaluated by the algorithm is the product of the population size and number of generations. In general, increasing either

---

**Algorithm 1.** Greedy algorithm for load balancing by data migration.

---

1. let  $s$ , the system state currently under consideration, be the original system state.
  2. initialize global solutions,  $S = \{\text{current state}\}$ .
  3. **repeat**
  4.   let  $e_{max}$  and  $e_{min}$  be the elements with maximum and minimum load (in  $s$ , respectively).
  5.   initialize list of locally non-dominated solutions,  $T = \emptyset$ .
  6.   **for** all load components  $\ell$  on  $e_{max}$  **do**
  7.     let  $t$  be the state of the system after migrating  $\ell$  from  $e_{max}$  to  $e_{min}$ .
  8.     calculate the imbalance factor of  $t$  and the cost of the migration.
  9.     if  $t$  is non-dominated with respect to  $T$  and  $S$ , add  $t$  to  $T$ .
  10.   **end for**
  11.   **if**  $T \neq \emptyset$  **then**
  12.     choose a random element  $r \in T$ .
  13.     add  $r$  to  $S$ .
  14.     update the current state  $s \leftarrow r$ .
  15.   **end if**
  16. **until**  $T = \emptyset$
- 

of these two parameters will improve the quality of the final solutions, in exchange for a longer running time. The *archive* is a collection of the best (non-dominated) solutions seen by the algorithm at a given time. The size of the archive determines the final number of solutions produced by the algorithm.

Given the three input parameters  $m$ ,  $n$ , and  $s$ , the algorithm generates as output archive  $A$ , a set of non-dominated solutions. At a high level, it works as follows:

1. **Initialize:** Create a population  $P$  of  $s$  possible reconfiguration plans, where each possible plan has a small number of random migrations specified.
2. **Evaluate:** Find the cost and imbalance factor for each solution in the current population  $P$ .
3. **Archive:** Add all non-dominated solutions from the population  $P$  to the archive  $A$ .
4. **Prune:** If the size of the archive  $A$  exceeds the maximum size  $m$ , remove some of the solutions based on measures of *crowding*. This is used to ensure the solutions take on the full range of possible values, in both imbalance factor and cost.
5. **Check Stopping Condition:** If the maximum number of generations has been reached, return  $A$ . Otherwise, continue.
6. **Select and Recombine:** Select individuals from the archive  $A$  and recombine them to form the next population  $P$ . A new solution is produced by combining two random subsets of migrations, each selected from an existing solution.
7. **Return** to Step 2.

**Integer Programming.** Another optimization technique used within our load-balancing framework is binary integer programming. Integer program solvers guarantee that the solution found will be optimal with respect to the given formulation. However, this method places several restrictions on such a formulation. The most important of these restrictions is that all of the equations, including the objective function, must be linear. That is, we cannot use arbitrary functions for the cost or imbalance functions.

Second, the method is not naturally multiobjective. To overcome this, we solve a series of integer-programming problems, with successively larger cost restrictions.

There are several variables and functions that describe our binary integer program. We use the following notation in its definition:

$N_e$	The number of elements	$w_i$	the weight of load component $i$
$N_l$	The number of load components	$c_{ij}$	the cost of migrating load component $i$ to element $j$
$y_{ij} = \begin{cases} 0 & \text{if load component } i \text{ was originally assigned to element } j \\ 1 & \text{otherwise} \end{cases}$		$C$	the maximum allowed cost of all migrations
		$U$	the target load on each element

**Decision variables**  $x_1, \dots, x_n$  are binary variables that the program will solve for.

$$x_{ij} = \begin{cases} 1 & \text{if load component } i \text{ assigned to element } j \\ 0 & \text{otherwise} \end{cases}$$

**Objective function** is a function of the form  $\sum_{i=1}^n c_i x_i$ , where  $c_i$  are any constants, and  $x_i$  are the decision variables. The goal is to maximize the total weight of all assigned load components:  $\max \sum_{i=1}^{N_l} w_i \cdot \sum_{j=1}^{N_e} x_{ij}$

**Constraint functions** are of the form  $\sum_{i=1}^m c_i x_i < C$ , where  $c_i$  are constants,  $x_i$  are some subset of the decision variables, and  $C$  is a constant.

Ensure that no load component is assigned to more than one element:  $\forall i \leq N_l : \sum_{j=1}^{N_e} x_{ij} \leq 1$

Ensure that maximum cost is not exceeded:  $\sum_{i=1}^{N_l} \sum_{j=1}^{N_e} x_{ij} y_{ij} c_{ij} < C$

Ensure that no element is overloaded:  $\forall j \leq N_e : \sum_{i=1}^{N_l} x_{ij} w_{ij} \leq U$

Note that it is possible that some load components will not be assigned to any element, either because doing so would exceed the target load or because it would exceed the maximum reconfiguration cost. These unassigned load components are assumed to remain on their original elements. The load components that the solver assigns to a new element make up the reconfiguration plan.

*Application:* We consider reconfiguration plans that consist of a set of volumes to be migrated from one Dblade to another (Scenario 2 from Section 2.1). The reconfiguration plans are evaluated with respect to two objectives: minimizing the imbalance factor calculated by Module 2; and minimizing the cost of the reconfiguration.

We define four functions for calculating cost, representing both linear and non-linear functions. The first function assigns a constant cost for each volume migration. The second function assigns a cost proportional to the total size of the volumes being moved. The third function uses empirically derived costs as encoded in a table-based model (see Table 3). The cost is the average latency of all operations while the migrations are taking place. This cost depends on both the number of volumes being simultaneously migrated and the workload being handled by the system at the time of reconfiguration. The fourth function is non-linear and estimates the total time of impairment.

Table 3 shows a sampling of (sanitized) values measured on a four-node cluster with midrange nodes running a SPECsfs benchmark [19]. The rows represent the relative

**Table 3.** Reconfiguration costs measured as average request latency in ms

Volume Moves	Load (ops/s)					Volume Moves	Load (ops/s)				
	Base (500)	2×	4×	8×	16×		Base (500)	2×	4×	8×	16×
0	0.5 ms	0.5	0.5	0.7	0.8	2	1.2 ms	1.0	1.0	1.2	1.5
1	1.0 ms	0.8	0.9	1.0	1.1	3	1.2 ms	1.2	1.3	1.5	1.9

costs when moving zero, one, two, or three volumes simultaneously. The columns represent the “load level” of the benchmark (in SPECsfs terms, the targeted number of operations per second). The baseline column corresponds to a very light load; the other columns represent a load-level that is a double, quadruple, and so on, of the baseline load. A performance engineering group generates similar tables for other cluster configurations, system versions and hardware types for other workload classes e.g., Exchange Server [20], during system development.

We implemented the hill-climbing and integer programming optimization methods directly in MATLAB. The Strength Pareto Evolutionary Algorithm (SPEA) is written primarily in Java<sup>TM</sup> and controlled by MATLAB. Hence, the SPEA runtime is dominated by interprocess communications. Because of this, we compare the efficiency of the methods by the number of possible solutions they evaluate, and not runtime.

The integer-programming method is restricted to using only linear cost and objective functions. It uses only a form of the simple sum function for calculating imbalance, and uses only the constant and linear cost functions. This corresponds to a form of traditional bin-packing problems. The greedy and evolutionary algorithms also correspond to traditional bin-packing when using the simple sum function and constant costs.

## 2.4 Execute Reconfiguration Plan

As described previously, Module 3 provides a set of (near) optimal reconfiguration plans, with a range of costs and resulting imbalance factors. The job of Module 4 is to select one of these possible solutions and perform a series of system-level operations to execute that plan. The choice of which of the nondominated solutions to choose depends primarily on the reconfiguration cost that the system can tolerate at the time, which can be specified by service level objectives (SLOs).

*Application:* In our case of performing load balancing through the migration of volumes, this module handles the details such as when to perform the migrations, in what order, and at what rate. In our experiments, we chose to perform all migrations simultaneously at the maximum rate. Our system performs this operation on-line with no client-perceived disruptions (though performance is affected, as described by Table 3).

## 2.5 Assumptions and Limitations

As presented here, the framework makes a few assumptions. First, we assume that the load on the system is sufficiently stable. That is, the load we have observed in the recent past is a good approximation of the load we will experience in the near future. If this were not the case, the steps taken to rebalance the system would likely be ineffective or

even detrimental. If a system does not meet this criterion, the load-balancing framework would need some other means for predicting future load in order to determine a viable reconfiguration. However, in practice, enterprise system workloads tend to experience cyclical workloads with daily and weekly periodicities.

Second, we assume that the system is composed of (nearly) homogeneous nodes. For example, our model does not take into account the amount of memory each node has. We believe that our framework is applicable without any loss of generality to non-homogeneous systems. In practice, this requires more performance characterization with larger cost-model tables.

### 3 Experimental Analysis

We evaluate our framework in two different ways. First, using a description of a week-long workload from an internally deployed system, we explore the various implemented functions and optimization techniques. Second, we examine the model in a real-world scenario—a recent hardware upgrade and consolidation effort in one of our data centers.

We compare optimization methods for various imbalance and cost functions using a data center storage consolidation scenario. In this scenario, we study the effects of taking an existing stand-alone system with e-mail workload from MS Exchange Servers and integrating it into the existing six-node system. The first step involves rolling the existing hardware with its data into a cluster. The second step, and the one targeted by our framework, involves moving data (volumes) between the nodes of the combined system to achieve a more balanced load.

#### 3.1 System and Workload Description

**Clustered system.** A six-node NetApp Data ONTAP GX system with 120 TB of storage across 794 disks and 396 total volumes stores the home directories of approximately 150 users, mostly engineers. Each user has a 200 GB primary volume with a remote site replica and a secondary RAID-protected volume with additional 200 GB of space that is not replicated. The home directory volumes are accessible by both NFS and CIFS protocols. The cluster is used predominantly for software development (compilation of source code) by individual users as well as build-server farms. There are several large volumes, one per cluster node, for a build- and regression-testing farm of a few dozen servers. We do not consider these volumes in our experiments because by manual assignment (and in practice), the load from these volumes is already “balanced.” It is this kind of manual load balancing that we aim to replace with our framework.

Figure 2 shows the load on the cluster over a one-week period. The load is characterized by the total number of file operations performed by the system (aggregated over 30 minute periods). There were a total of approximately 1.4 billion operations, or an average of 8 million operations per hour. The load has a strong periodic component, with large spikes occurring during workdays, both during business hours (software development) and at night (regression tests). The bars at the top show the daily cumulative per-node load. Although the cluster has six nodes, it currently uses only the first four—the other ones have been recently added for future system expansion. We show later on how our framework redistributes load and populates these nodes with data.

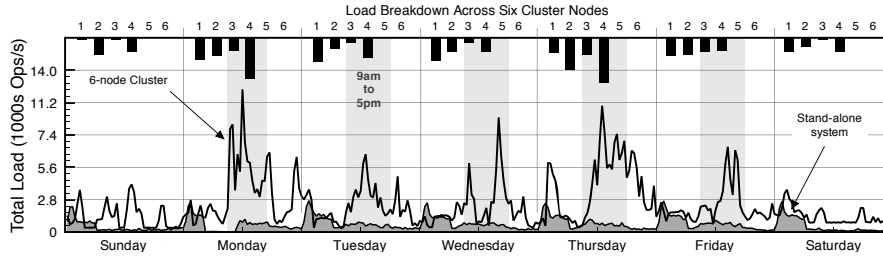


Fig. 2. Typical load profiles over a one-week period

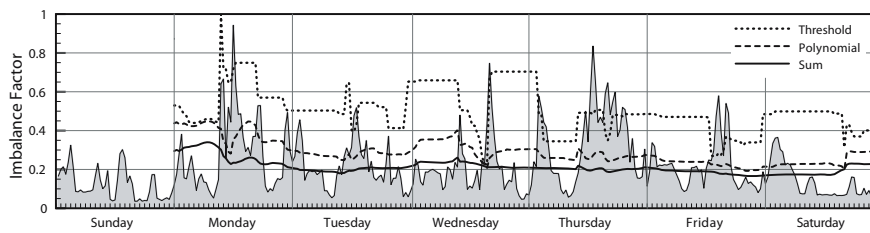


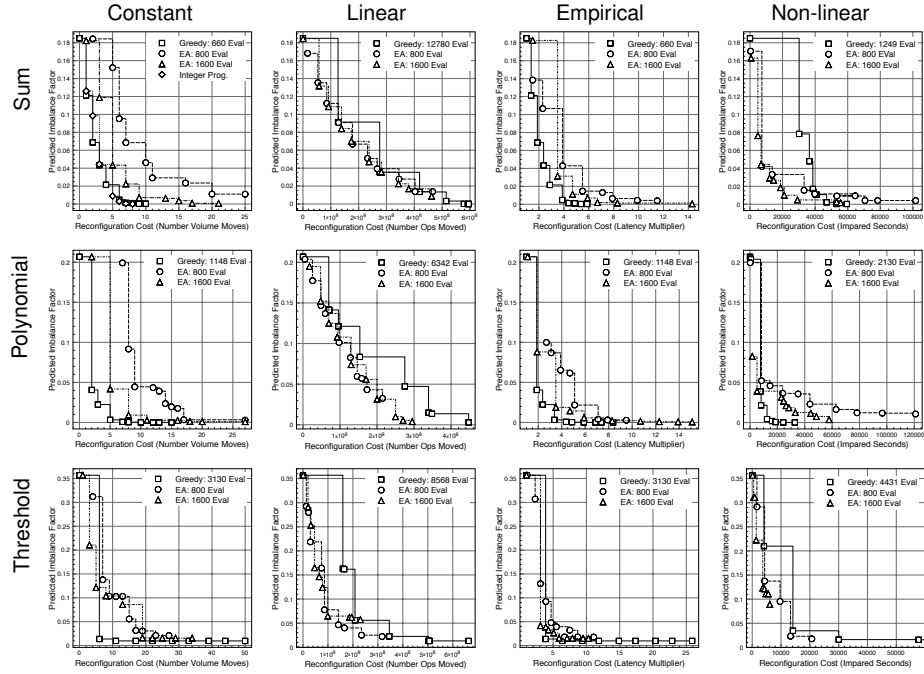
Fig. 3. Comparison of three load-flattening functions

**Stand-alone system.** There are 12 volumes supporting e-mail workload from three MS Exchange Servers with mailboxes for several hundred users. As shown in Figure 2, it also experiences load at night due to data consolidation and backup.

### 3.2 Results

**Load Imbalance Expression.** Figure 3 compares the three flattening functions from Module 2 with a 24-hour time window. Here, and for the remainder of this paper, we apply optimizations over a week-long period, since our workload has strong weekly periodicity. The input is the combined load from the two consolidated systems, shown in the background. The polynomial and threshold functions emphasize spikes in load more than the simple sum function, fulfilling their intended purpose. These two functions are also more variable in general, and require larger time windows to avoid rapid oscillations. Given our workload profile, a 24-hour window is sufficient to remove the daily load periodicity and more accurately reflect any underlying load imbalance.

**Comparison of Optimization Methods.** We use four cost functions—constant, linear, empirical, and non-linear; and three imbalance functions—sum, polynomial, and threshold, for our comparison of the different optimization methods in Figure 4. For the constant function, each volume move has a cost of 1. For the linear function, the cost of moving a set of volumes is the total number of bytes in all moved volumes. For the empirical function, we use the data from Table 3 and, more specifically, only the last column of the table with the base latency as 0.8 ms. Moving 1, 2, or 3 volumes increases the latency to 1.1, 1.5, or 1.9 ms, or respectively,  $1.375$ ,  $1.875$ , and  $2.375 \times$  the base.



**Fig. 4.** Comparison of three optimization methods using various measures of imbalance and re-configuration cost. The three rows correspond to the sum, polynomial, and threshold temporal-flattening functions. The four columns correspond to different cost models: constant (number of volumes moved); linear (number of bytes moved); empirical (latency increase); and, nonlinear (time period of impairment). The experiment using the sum-flattening function and constant migration cost compares all three optimization methods: greedy algorithm, evolutionary algorithm (EA), and integer programming. The others exclude integer programming because of their non-linear objectives. We use two cases for EA with 800 and 1600 solution evaluations respectively. For all cases, solutions closer to the origin are better.

When moving more than three volumes, we interpolate using the last two values. So each volume move beyond three adds latency with a  $0.5 \times$  latency multiplier. In practice, this linear penalty is too large for several volume moves. Therefore, we use data with approximate costs that are non-linear when moving more than three volumes simultaneously. The fourth column in Figure 4 shows the results for this approximation of a more realistic cost function.

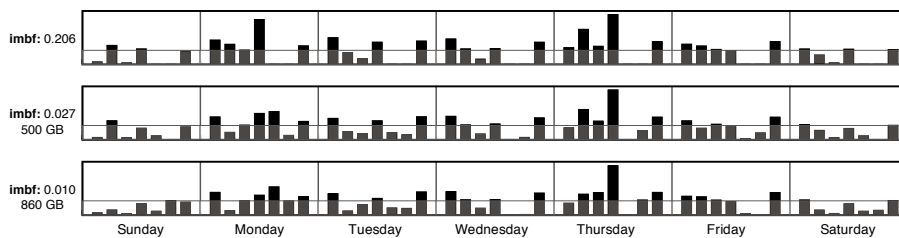
The greedy algorithm uses an unbounded number of solution evaluations, halting when no further improvements can be made. The graphs display a set of 10 intermediate points along the hill-climbing path. The evolutionary algorithm uses a constant number of solution evaluations based on input parameter settings. The total number of evaluations is equal to the product of the population size and the number of generations. We show two settings: population size of 40 and 20 generations (for 800 evaluations) and population size 50 and 32 generations (for 1,600 evaluations). The results of the

integer programming method are found in the experiment using only linear objectives (sum flattening function and constant cost).

We can draw the following general conclusions from the experimental results:

- The greedy algorithm is superior for the constant cost case. That is, it provides solutions with lower cost and lower imbalance factors.
- The evolutionary algorithm is superior for the linear cost case. With this more complex cost function, the greedy algorithm is more likely to be stuck in a local optimum early in the search, and requires more solution evaluations in total: in this case, up to 3,815 evaluations.
- The empirical cost function is nearly equivalent to the constant cost function. This is because each additional volume move adds a nearly constant latency multiplier, around  $0.5\times$ . Consequently, the experimental results of the third column are very similar to the first column. The greedy results are identical, with the same solution curves and number of evaluations. The evolutionary algorithm returns different solutions, due to the randomized starting point.
- The greedy algorithm tends to require more evaluations as the imbalance functions get more complex, moving from sum to polynomial to threshold.
- The integer-programming results are comparable with those found by the greedy algorithm. Some of its solutions perform slightly worse because the predicted imbalance of all solutions are evaluated using the normalized entropy function, but this nonlinear function is not used by the integer program.

**Executing a Reconfiguration Plan.** The optimization provides a set of nondominated solutions. The choice of which is most suitable depends on how much additional load (e.g., the expected increase in request latency) the system can tolerate during data migration. These are set as service-level objectives (SLOs) by a system administrator, allowing the reconfiguration plan to be executed automatically. If a single reconfiguration plan cannot reach a sufficiently balanced state without violating some SLOs, the framework iterates the process, as shown in Figure 1. By executing a small number of



**Fig. 5.** Comparison of predicted load imbalance after reconfiguration and plan execution. The top graph is the baseline with imbalance factor (imbf) 0.207 for the combined load from both systems. The middle graph shows the reconfiguration plan with total cost of 500 GB. and the bottom graph shows the “best” reconfiguration plan—860 GB of data moved. A single volume contributes to Thursday’s load on node 4; the spike cannot be flattened by data migration alone. Both graphs correspond to evolutionary algorithm solutions with the polynomial flattening function and linear cost function. The line at 1/3 graph height is a visual aid to more easily spot differences.

migrations over a longer time, or during off-peak hours, load imbalance can be eliminated without sacrificing system performance.

Figure 5 shows the effects of a subset of reconfiguration plans suggested by the evolutionary algorithm. These graphs illustrate how the load on each cluster node would change as a result of implementing a reconfiguration plan. The “best” plan at the bottom would be executed in about eight hours.

## 4 Related Work

Many components of our framework build upon previous work. Aqueduct, a tool for executing a series of data-migration tasks, with the goal of minimizing their effect on the current foreground work [21], is similar to Module 4 of our framework. The table lookup model in Module 3 is based on previous work of Anderson [22]. Our framework is similar in many aspects to Hippodrome [16], which iteratively searches over different system designs and uses Ergastulum to find the least-cost design in each step. Ergastulum is a system for designing storage systems with a given workload description and device model [13]. It uses a form of bin packing and solves the problem using a randomized greedy algorithm. Ergastulum was motivated by and improved upon Minerva [23], which uses a similar problem formulation but less-efficient search. Both of these systems focus on new system design for a specific workload. In contrast, our framework searches for load-balanced configurations of already-deployed systems, where reconfiguration cost is of importance. It is also suited for exploring what-if scenarios for system consolidation and upgrades. Stardust, which performs a function similar to Module 1, collects detailed per-request traces as requests flow through node and cluster components [11]. Other approaches mine data collected by Stardust for exploring what-if scenarios [10,24]. However, unlike our system, they use simple heuristics rather than optimization techniques. Our framework uses only high-level workload descriptions and performance levels similar to relative fitness models [25].

## 5 Conclusions

The modularity of our framework allows users to explore functions that best fit their workloads and systems. The use of multiple optimization methods *and* explicitly taking into account the cost of rebalancing when considering optimal configurations is one of the contributions of this work. Previous approaches have chosen a single method and designed their load-balancing systems around it.

To the best of our knowledge, we show the first application of evolutionary algorithms for optimizing storage system configurations. While not provably optimal, evolutionary algorithm is in our view the most general and versatile approach; it can leverage non-linear imbalance functions and empirical system models. Integer programming is most applicable with simple, that is, linear, cost and objective functions and with fewer elements and load components.

Our framework is practical in terms of (i) using high-level workload descriptions from periodic collections of performance data, (ii) its applicability to real-world scenarios for consolidating data center storage, and (iii) the use of high-level empirical

performance models. Generating detailed storage models is typically quite difficult. In contrast, collecting performance data for the table-based lookup model is “easy”, though it can be resource intensive and time consuming. System characterization under different system configurations, workloads, and operations (e.g., volume moves) is an integral part of system development similar to qualifications of a storage system against a myriad of client host controller cards, operating systems, and so on. Dedicated engineering teams across manufacturers of enterprise storage routinely undergo such tests.

## References

1. Eisler, M., Corbett, P., Kazar, M., Nydick, D., Wagner, C.: Data ONTAP GX: A scalable storage cluster. In: Proc. of the 5<sup>th</sup> Conf. on File and Storage Technologies, pp. 139–152. USENIX Association (2007)
2. Abd-El-Malek, M., et al.: Ursa Minor: versatile cluster-based storage. In: Proc. of the 4<sup>th</sup> Conf. on File and Storage Technologies, pp. 1–15. USENIX Association (2005)
3. Nagle, D., Serenyi, D., Matthews, A.: The Panasas ActiveScale storage cluster: Delivering scalable high bandwidth storage. In: Proc. of the ACM/IEEE Conf. on Supercomputing, Washington, DC, USA, p. 53. IEEE Computer Society, Los Alamitos (2004)
4. Hitachi: Archivas: Single fixed-content repository for multiple applications (2007), [http://www.archivas.com:8080/product\\_info/](http://www.archivas.com:8080/product_info/)
5. Weil, S., Brandt, S., Miller, E., Long, D., Maltzahn, C.: Ceph: a scalable, high-performance distributed file system. In: Proc. of the 7<sup>th</sup> Symposium on Operating Systems Design and Implementation, pp. 22–34. USENIX Association (2006)
6. Saito, Y., Frølund, S., Veitch, A., Merchant, A., Spence, S.: FAB: building distributed enterprise disk arrays from commodity components. In: Proc. of ASPLOS, pp. 48–58 (2004)
7. Litwin, W.: Linear Hashing: A new tool for file and table addressing. In: Proc. of the 6<sup>th</sup> Int’l Conf. on Very Large Data Bases, pp. 212–223. IEEE Computer Society, Los Alamitos (1980)
8. IBM Corp.: TotalStorage productivity center with advanced provisioning (2007), <http://www-03.ibm.com/systems/storage/software/center/provisioning/index.html>
9. NetApp Inc.: NetApp storage suite: Operations manager (2007), <http://www.netapp.com/products/enterprise-software/manageability-software/storage-suite/operations-manager.html>
10. Barham, P., Donnelly, A., Isaacs, R., Mortier, R.: Using Magpie for request extraction and workload modelling. In: OSDI, pp. 259–272 (2004)
11. Thereska, E., et al.: Stardust: tracking activity in a distributed storage system. SIGMETRICS Perform. Eval. Rev. 34(1), 3–14 (2006)
12. Thereska, E., et al.: Informed data distribution selection in a self-predicting storage system. In: Proc. of ICAC, Dublin, Ireland (2006)
13. Anderson, E., Spence, S., Swaminathan, R., Kallahalla, M., Wang, Q.: Quickly finding near-optimal storage designs. ACM Trans. Comput. Syst. 23(4), 337–374 (2005)
14. Weisstein, E.: Bin-packing and knapsack problem. MathWorld (2007), <http://mathworld.wolfram.com/>
15. Wilkes, J.: Traveling to Rome: QoS specifications for automated storage system management. In: Proc. of the Int’l. Workshop on Quality of Service, pp. 75–91. Springer, Heidelberg (2001)
16. Anderson, E., et al.: Hippodrome: Running circles around storage administration. In: Proc. of the 1<sup>st</sup> Conf. on File and Storage Technologies, pp. 1–13. USENIX Association (2002)

17. Keeton, K., Kelly, T., Merchant, A., Santos, C., Wiener, J., Zhu, X., Beyer, D.: Don't settle for less than the best: use optimization to make decisions. In: Proc. of the 11<sup>th</sup> Workshop on Hot Topics in Operating Systems. USENIX Association (2007)
18. Zitzler, E., Laumanns, M., Thiele, L.: SPEA2: Improving the strength pareto evolutionary algorithm for multiobjective optimization. In: Evolutionary Methods for Design, Optimisation and Control with Application to Industrial Problems (EUROGEN 2001), International Center for Numerical Methods in Engineering (CIMNE), pp. 95–100 (2002)
19. SPEC: SPEC sfs benchmark (1993)
20. Garvey, B.: Exchange server 2007 performance characteristics using NetApp iSCSI storage systems. Technical Report TR-3565, NetApp, Inc. (2007)
21. Lu, C., Alvarez, G., Wilkes, J.: Aqueduct: Online data migration with performance guarantees. In: Proc. of the 1<sup>st</sup> Conf. on File and Storage Technologies, p. 21. USENIX Association (2002)
22. Anderson, E.: Simple table-based modeling of storage devices. Technical Report HPL-SSP-2001-4 (2001)
23. Alvarez, G., et al.: Minerva: An automated resource provisioning tool for large-scale storage systems. ACM Transactions on Computer Systems 19(4), 483–518 (2001)
24. Thereska, E., Narayanan, D., Ganger, G.: Towards self-predicting systems: What if you could ask what-if? In: Proc. of the 3<sup>rd</sup> Int'l. Workshop on Self-adaptive and Autonomic Computing Systems, Denmark (2005)
25. Mesnier, M., Wachs, M., Sambasivan, R., Zheng, A., Ganger, G.: Modeling the relative fitness of storage. In: Proc. of the Int'l. Conf. on Measurement and Modeling of Computer Systems. ACM Press, New York (2007)