UNIVERSITY OF CALIFORNIA
SANTA CRUZ

# DATA ALLOCATION APPROACHES FOR OPTIMIZING STORAGE SYSTEMS

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

**Christina Rose Strong**

June 2016

The Dissertation of Christina Rose Strong
is approved:

# Table of Contents

# List of Figures

# List of Tables

FOR PONY!

## Acknowledgments

It takes a village to raise a dissertation. That may not be *quite* how that quote is supposed to go, but it is true nonetheless. It is impossible to list everyone who contributed either in reading drafts, bringing me food, offering encouragement, or supplying much needed comic relief (and not just because of my memory). Thank you all who made this possible – you know who you are.

I would like to call out both the members of my advancement committee and dissertation committee. To my advisor, Darrell Long, who taught me to trust my instincts even in the face of adversity. To Ethan Miller, who could always be counted on to provide a voice of reason amongst the wild ideas. To Jehan-François Pâris, who passed along excellent advice from Ernest Hemingway and argued my approach for me. To Peter Alvaro, who may have come a bit late to the party, but was encouraging and insightful. Last but not least, to Ahmed Amer, without whom this may never have been completed. For responding to all the panicked phone calls, emails, and texts with an unwavering faith in my ability: from the bottom of my heart, thank you.

In addition to my committee members, I would be remiss if I did not mention my labmates. To those that came before, thank you for making me feel a part of the lab even before it was official. To those that were in the trenches with me, you all are what made the unbearable parts bearable. And to those that follow: always remember to fail often and loudly, and never be afraid to ask questions. You know how to get in touch with me.

A special thanks goes to Stephanie Jones, who has been my labmate, housemate, and friend. I will forever be grateful for the day you sat down next to me and declared that we were going to be friends. You have read drafts, listened to ideas and frustrations, reminded me to take care of myself, filled my life with laughter, and inspired me on a daily basis. And somehow you're still willing to write another paper with me once this is all over! I would not be where I am today without you, and I cannot find sufficient words to thank you.

Last but not least, I want to thank my parents. I may be biased, but you are the best parents anyone could hope, wish, dream, or ask for. You have always supported me in everything I do, even when it meant spending a decade pursuing a degree "just to see if I could do it". For all the drafts you've read even when you didn't completely understand it, for being my sounding board, just for being you: thank you. Never stop being awesome.

## Abstract

Data Allocation Approaches for Optimizing Storage Systems

by

Christina Rose Strong

As storage systems grow in size and complexity, the necessity for automatically managing them increases. One area of concern is system optimization, often managed by the administrator manually tweaking parameters to make the system as efficient as possible. The problem arises when there are multiple objectives to optimize, some of which may interfere with each other. Even when optimization is automated, it often requires either a decision to be made as to which objectives are more important than others, or if the objectives are combined into a linear function, this function (and the potential related weights) must be determined by the administrator. To this end, most systems optimize for only one objective, or when there are two objectives it is two that do not conflict. However, when all objectives are equally important, or it is unknown which objective is most important, the system must learn to balance multiple objectives.

One way to aid in automating system optimization is by looking at the file allocation problem. The file allocation problem considers a set of files (or tasks) that need to be allocated to some number of devices, so as to optimize an objective function. I propose extending the file allocation problem so that rather than optimizing a single objective function, the goal is to allocate data to a number of devices subject to a multi-objective optimization. This dissertation covers the three top-down theoretical approaches I developed as well as a bottom-up practical approach. In addition, I present a system design that incorporates the practical approach with two space saving modifications.

# Chapter 1

# Introduction

The length of this document defends it well against
the risk of its being read.

Winston Churchill

As storage systems grow in size and complexity, the necessity for automatically managing them increases. One area of concern is system optimization, often managed by the administrator manually tweaking parameters to make the system as efficient as possible. The problem arises when there are multiple objectives to optimize, some of which may interfere with each other. For example, a very simple optimization strategy for load balancing is to distribute data across the system such that the relative frequency of access is approximately the same for each device. In direct conflict with such a strategy is the simple optimization strategy for energy savings, which attempts to consolidate data on as few devices as possible.

Even when optimization is automated, it often requires either a decision to be made as to which objectives are more important than others, or if the objectives are combined into a linear function, this function (and the potential related weights) must be determined by the administrator. To this end, most systems optimize for only one objective, or when there are two objectives it is two that do not conflict. However, when all objectives are equally important, or it is unknown which objective is most important, the system must learn to balance multiple objectives.

One way to aid in automating system optimization is by looking at the file allocation problem. The file allocation problem considers a set of files (or tasks) that need to be allocated to some number of devices, so as to optimize an objective function. I propose extending the file allocation problem so that rather than optimizing a single objective function, the goal is to allocate data to a number of devices subject to a multi-objective optimization.

I focus on three objectives throughout this dissertation: load balancing, system responsiveness, and energy savings. I define load balancing to be the distribution of data over devices, such that frequently accessed data is evenly distributed across the system. I define system responsiveness to be the response time of a request. Finally, I define energy savings to be the decrease of power consumption of the system. I assume initially all devices are powered on and active; any reduction in the number of active devices is considered to be an increase in energy savings.

Two different approaches, a top-down view and a bottom-up view, are presented in this dissertation. The top-down view, discussed in Part I, consists of taking a theoretical approach to describe a multi-objective file allocation problem. I present a system design and three theoretical optimization methods: multi-objective optimization, bin packing, and queuing theory. Part II discusses the bottom-up view, which applies machine learning to help solve the file allocation problem and combines those results with constraint satisfaction to aid with system optimization.

I describe the simulated system in which I test the bottom-up approach, including an approach to workload generation that simulates realistic workloads without necessitating real world workload traces. The results of the bottom-up experiments show that using a regression tree to predict performance is not only possible, but a useful technique to aid in intelligent workload allocation. I explore two simplifications for the performance prediction step: a storage class model to replace individual device models, and a representative workload that summarizes the workloads currently active on a device. These simplifications decrease the storage and complexity overhead added by introducing machine learning. I conclude by presenting a system design that uses the performance prediction with discussed simplifications along with constraint satisfaction to maintain an optimal system state.

# Part I

# Theoretical Approach

# Chapter 2

# Defining Optimization

True genius resides in the capacity for evaluation of uncertain, hazardous, and conflicting information.

Winston Churchill

Since my goal is to extend the file allocation problem to satisfy multiple objective functions, I looked at the difference between single objective optimization problems and multi-objective optimization problems. In a single objective optimization problem, there is a single objective function that is being minimized (or maximized), expressed as in Equation 2.1. Here, $f(x)$ refers to the objective function to be minimized over $x$. While this is beneficial when that single objective is all that matters, if there are other objectives that are important, Equation 2.1 has no way to take those into account.

$$\underset{x}{\text{minimize}}\, f(x) \tag{2.1}$$

In a multi-objective optimization problem, there is often no single global solution. This is especially true when objectives conflict with each other: consider two objectives that are inversely proportional to each other. Optimizing with respect to a single objective may result in poor performance for another. For example, consider two committee members with drastically different writing styles. Member A prefers a paper to have a distinct structure and pattern; Member B considers that monotonous and would prefer a more fluidly written paper. There is no single solution that will optimize the style of a paper for both committee members.

The standard form of a multi-objective optimization problem is shown in Equation 2.2. This is extremely similar to Equation 2.1; in fact only one thing has changed. Instead of trying to optimize a single objective function $f(x)$, there is now a vector of objective functions, $F_i(x)$, to be minimized.

$$\underset{x}{\text{minimize}} \quad \mathbf{F}(x) = [F_1(x), F_2(x), \ldots F_k(x)] \tag{2.2}$$

There are many algorithms which can be used to solve this vector of objective functions; they are divided into three categories, depending on how the decision-maker chooses to articulate his preferences [40]. The first categorization is *a priori*, where preferences are determined prior to running the algorithm. The benefit to a priori algorithms is that often a single solution is found, rather than a set of multiple solutions; the downside is that the user must know the relative importance of the objectives in advance. The opposite approach is *a posteriori*, where the user can look at a set of mathematically equivalent solutions and choose one. This allows the user to see his/her options before making a decision, but can be more computationally intensive since a set of solutions must be generated rather than a single one. Finally, there are a set of algorithms which require no articulation of preference; these, however, are often simplified versions of the a priori algorithms.

One of the most common a priori methods is the weighted sum method, which assigns a weight to each objective and uses the sum of the weighted objectives to create an aggregate objective function. This approach, like all aggregate objective approaches, can be highly subjective due to the necessity of determining the objective weights in advance. There is an excellent summary of the different ways to calculate the weights, along with the difficulties of the weighted sum method in [40].

Often it is better to use an a posteriori method instead: to consider a set of solutions that satisfy the objectives and fit a predetermined definition of optimum, and make a decision from that set of solutions. The predominate concept used to define an optimal point is known as *Pareto optimality* [30]. Pareto optimality finds a set of solutions such that no element of the set can be replaced by another solution which improves an objective without worsening another. This provides flexibility in the way the data allocation algorithm behaves in different systems. If, for instance, a system prioritizes energy savings above any other objectives, it is possible to always chose an allocation from the set of solutions that favors energy savings. In this way, the system is optimized for multiple objectives while still placing priority on a particular objective when necessary.

## 2.1 Pareto Optimality

Pareto optimality, also known as *Pareto efficiency*, is a concept from economics. Given an initial allocation of goods, a *Pareto improvement* is an allocation that makes one person better off without worsening conditions for anyone else [38]. When no further Pareto improvements can be made, the allocation is said to be *Pareto optimal*, or *Pareto efficient.*

**Pareto Optimum.** *Given two optimization criteria $P(x)$ and $Q(x)$ to maximize, a point $z$ is a* Pareto optimum *if there is no point $x'$ such that $P(z) < P(x')$ and $Q(z) \leq Q(x')$ or $Q(z) < Q(x')$ and $P(z) \leq P(x')$*

**Table 2.1:** House rankings for each requirement on a scale of 1 to 5, with 5 being the best.

| | A | B | C |
|---|---|---|---|
| Size | 4 | 4 | 4 |
| School | 3 | 3 | 5 |

A typical example of Pareto optimality is looking to buy a house [38]. Consider a list of requirements about your dream house: that it is large enough to accommodate your family, is in a good school district, and is within a certain price range. Your real-estate agent shows you three different houses, A, B, and C, all within your price range. The decision then falls to the other requirements, so you visit each house and rank them as shown in Table 2.1. While house C is ranked the same in terms of size, it is ranked much higher than house A or B in terms of the school district.

Pareto optimality says that house C is the optimal choice; moving from C to B or from C to A would cause one of the requirements to suffer. Therefore, assuming that all requirements carry equal weight, *house C is Pareto optimal.* The concept of Pareto optimization can be applied to data allocation, where instead of determining an allocation of goods to people you are determining an allocation of data to devices. Thus, given an initial allocation of data, there is an optimal allocation that can be determined.

### 2.1.1 Pareto Frontier

Much work has been done looking at applying the concept of Pareto optimality to multi-objective optimization problems [10, 40, 68]. In the case of multi-objective

**Table 2.2:** House rankings for each requirement on a scale of 1 to 5, with 5 being the best.

| | A | B | C | D |
|---|---|---|---|---|
| Backyard | 5 | 4 | 3 | 2 |
| Size | 3 | 4 | 3 | 4 |
| School | 4 | 2 | 5 | 2 |
| Water Pressure | 4 | 5 | 3 | 4 |

optimization problems, there is often not a single allocation that is Pareto optimal, but rather a subset of allocations called the *Pareto frontier*. The Pareto frontier consists of the subset of optimal allocations with respect to the given objectives

Let us revisit the house example, except this time your dream house has a backyard and decent water pressure, as well as being large enough to accommodate your family, in a good school district, and within a certain price range. Your real-estate agent shows you four houses, all within your price range, and you rank them as shown in Table 2.2. House B and house D are equal in terms of school district and size, but house B is better in terms of backyard and water pressure. This allows us to remove house D from the decision, since house B would be a better choice. However, houses A, B, and C all have different tradeoffs. While house A is best in terms of backyard, house B has the best water pressure, and house C is in the best school district. Without any other information, it is impossible to tell which of the three remaining houses is "best".

Consider first what would happen if we combined the objectives into a single aggregate function. Let that function be the sum of the objectives, with the expectation that a higher sum is better. House A now has a value of 16, house B has a value of 15, house C has a value of 14, and house D has a value of 12. The obvious choice here according to our aggregate function is house A. Looking at the actual objective values, however, one might suggest that the size of the house is more important than the backyard. Let us revise our aggregate function to include a weight on the house size: let the size be twice as important as any other objective. The resulting values are house A is equal to 19, house B is equal to 19, house C is equal to 17, and house D is equal to 16. We have created a situation where we required extra knowledge (that the size is more important than the backyard) and we are still unable to determine which house is "best".

Pareto optimality says that A, B, and C are all optimal choices: while moving from A to B causes the backyard and school district to suffer, it increases the house size and water pressure. Similarly, moving from A to C would cause the backyard and water pressure to suffer, but causes no change in the house size and improves the school district. House D is not considered optimal because house B is equal to or better than house D in all objectives; house D is *dominated* by house B. Assuming all requirements carry equal weight, houses A, B, and C are Pareto optimal. *The Pareto frontier consists of houses A, B, and C*—while one allocation favors one objective over another, all are Pareto optimal. An allocation is on the Pareto frontier if it is not dominated by any other allocation.

# Chapter 3

# Objective Metrics

Research is to see what everybody else has seen,

and to think what nobody else has thought.

Albert Szent-Gyorgyi

In order to optimize for three different objectives, there needs to be a way to measure them. To this end I have defined the following three metrics: units that can be defined and measured, which influence an objective and are under the control of the file/object management system. This last point is necessary, as one could imagine adding more memory would influence system responsiveness–an effective solution, but not one that file allocation can achieve.

**Object Similarity.** *There is a body of work [1, 32, 56] that shows grouping similar data together has an effect on system responsiveness. There are different ways to define similarity: it could be as simple as comparing the metadata of two objects, or as complex as comparing the contents of two objects. I define object similarity is the probability that similar items will be accessed together. If the next item you want is near the item you just retrieved, the seek time will be decreased, resulting in increasing the system responsiveness.*

**Object Coverage.** *The number of active devices has a large influence on the power consumption of a system. The metric for energy savings has two parts: object coverage on a device, which is defined as the amount of similar data on a device, and device coverage in the system, which is defined as the number of similar devices in the system. In order to maintain the current number of active devices when an object is placed, object coverage identifies which active disk is most similar rather than spinning up an idle disk, even if it has better object coverage. Device coverage helps identify redundant disks, in order to decrease the number of disks necessary to guarantee the availability of the data.*

**Object Popularity.** *A well known influence on load balancing is the popularity of objects. While definitions differ among works, it is generally accepted that popularity relates either to the frequency with which an object is accessed or the recency with which an object was accessed. I define the popularity of an object as the frequency of access over a given period of time.*

These are not the only possible metrics for these objectives; other metrics could easily be used in place of the ones I have described. These metrics are not the only influences on the objectives I am exploring: system responsiveness, for example, is influenced by the amount of memory and speed of the CPU. In this section of the dissertation, however, I am focusing on the three metrics described, as they are good candidate metrics that have been selected for their suitability to the data set under consideration.

In order to develop these metrics, I worked with data sets from the Los Alamos National Laboratory (LANL). These data sets are anonymized, static metadata snapshots, with a subset of metadata fields shown in Table 3.1. I focused on the `anon-lnfs-fs4` data set, which consists of $163,267$ files over 12 nodes. For the purposes of these experiments, I looked at assigning files to nodes both sequentially

**Table 3.1:** Metadata fields in the Los Alamos National Laboratory data sets.

| **Fields Used** | **Fields Unused** |
|---|---|
| file permissions | unique identifier |
| file size (in bytes) | block size (in bytes) |
| user ID | path to file |
| group ID | |
| creation time | |
| modification time | |

and in a round robin manner. Sequential assignment simply assigned files in the order they appeared in the data set until a node filled; round robin distributed them across nodes.

## 3.1 Object Similarity

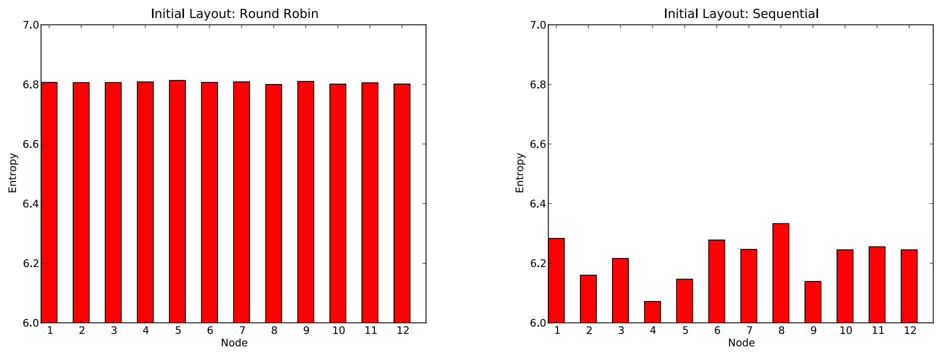
Many of the algorithms that optimize for system responsiveness require some sort of sorting [34, 69, 67], often by file size or access rate. This results in similarly accessed or similarly sized data being placed together. As mentioned previously, there is a body of work [1, 32, 56] that points out that grouping similar data is often used to help improve response time. Since the proximity of similar data has a well-researched influence on system responsiveness, I am using the similarity between objects as a metric for system responsiveness.

Object similarity depends on the probability that similar objects will be accessed together. When an object is accessed, there is (depending on the type of storage device) an initial seek time that is spent to get to that object. If the next object to be accessed comes immediately after it, that initial seek time is reduced. It is important to note that this benefit is only seen when working with small objects: objects that are larger than a track require a seek to the next object regardless of where it is. Despite the scale of the experiments, there are a limited number of large files[1].

There are three different ways to measure the similarity of two objects: comparing the contents, identifying similar metadata attributes, or when provenance information is available determining that the lineage is comparable. In practice, comparable lineage means that the input files are the same, the process to get from input file to the object is the same, or both the input files and the process are the same. This produces very different kinds of similarity. The first produces objects that all came from the same input file; the second produces all objects that were created using the same program. The third produces objects that had the same input file and were created using the same program. Consider a comma separated file of data that I want to manipulate using a Python script I have written. The first kind of similarity would give me all objects that came from that input file, the second would give me all objects created using that Python script, and the third would give me all objects created using the input file with the Python script. While these are all useful, it may be beneficial to keep the different types separate, as they may result in different access patterns.

For the purposes of this dissertation, I focused on identifying similar metadata attributes, since the data available was metadata information. Similarity is measured using Shannon Entropy, as seen in Equation 3.1. This means that given a specific metadata attribute, the probability of each value occurring on a node can be calculated. A low entropy means that there are many similar values, a high entropy means that there are many diverse values. Since the metadata used is

[1]As indicated by the metadata information from Los Alamos National Laboratory.



**(a)** Round Robin Assignment **(b)** Sequential Assignment

**Figure 3.1:** Similarity: Measured by entropy. $Y$-axis begins at 6 to show variation.

static, entropy works well as a metric, as shown by Parker-Wood et al. [50].

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_b p(x_i) \tag{3.1}$$

As you can see in Figure 3.1, similarity is dependent on the assignment strategy. The reason the entropy values in both assignment strategies are fairly large is due to the nature of the LANL data set. Despite the large values, it is still possible to see the difference between the two strategies.

When data is assigned sequentially, files in a directory are assigned to the same node, resulting in many of the metadata fields having the same value and lower entropy. When data is assigned in a round robin fashion, files in a directory are spread across the nodes, resulting in the metadata fields having diverse values on each node and higher entropy. This supports the findings of Parker-Wood et al. [50] that entropy is a good metric for comparing the similarity of files on a node.

## 3.2 Object Coverage

Coverage influences energy savings, and measures the percentage of data on the disk that is "covered" by other disks in the system. This allows for identification of redundant devices: devices that are 100% covered by the rest of the system can be spun down with few repercussions. In much of the related work, energy savings is defined as a reduction in power or energy consumed. Essary and Amer [17] looked at reducing the movement of the disk arm, thereby reducing the energy consumed by the arm mechanism. Others spin down the disk entirely or spin at lower speeds when the disk is idle [11, 8, 52] in order to reduce the energy consumed by the disk. For the purposes of this research, I define *energy savings* to mean *the power consumed is decreased.*

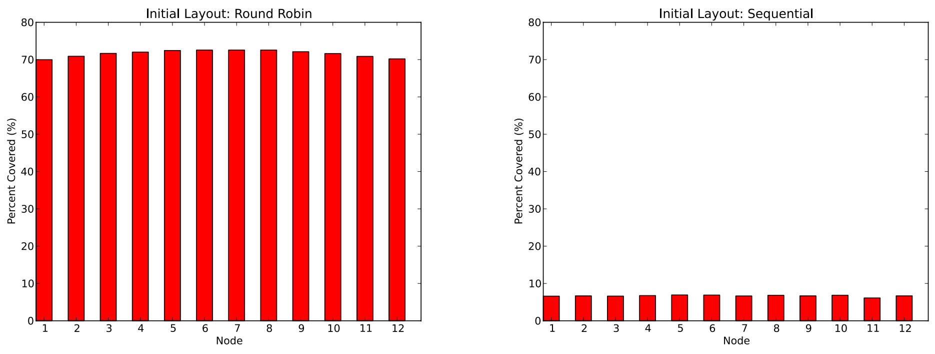
A "simple" solution for energy savings for write requests would be to write the data to whatever device is active and closest. Likewise, a "simple" solution for energy savings for read requests would be to write the data to the device that has the most similar data on it regardless of its state. This is based on the assumption that similar data is likely to be accessed together, as discussed previously. These approaches are very different, and result in very different data allocations. I propose a hybrid approach, building off the work on Rabbit [2]. Rabbit creates replicas of the data, and distributes the primary replica over $p$ nodes, guaranteeing the availability of the data as long as those $p$ nodes are active: the data is "covered" by those $p$ nodes.

Rather than using replicas to provide coverage, I use similarity. The more similar a piece of data is to the data on a device, the better it is "covered" by that device. This reflects the simple read request solution described in the previous paragraph. However, if a piece of data is best covered by an idle device, it will be placed on the next most similar device that is active. An extension of object coverage is device coverage, which looks at the coverage of the contents of a device in the system. Device coverage will be used to help identify redundant devices: devices which have content that is well covered elsewhere in the system and thus could be idle rather than active.

Given the measure of similarity that I am currently using is metadata, coverage is a somewhat flawed term. In this case, "approximate coverage" is a better term; just because the metadata is similar does not mean the data themselves are identical. Thus the term coverage can be misleading, and can lead to misleading results if not handled with care. However, in an environment where the metric similarity incorporates not only metadata, but also content and provenance as well, coverage can become a powerful metric.

The original design of the coverage metric was to compare the number of different values (on average) for each metadata attribute on each node. The first problem that arose was that this technique resulted in as many values as devices, when a single value was needed. To resolve this, I took the average of the values. The problem with that, however, was that if two nodes covered each other very well, but had little in common with the rest of the system, the average was highly skewed.

My next approach was to count the number of other nodes that covered the current node, but a simple count doesn't tell you which node is covered by which. Since the goal is to identify which nodes can be spun down with little to no loss of data, this was not a valid metric either. I needed a metric that rewarded skew, since that was a good indicator that there were two highly similar nodes, and I needed to know which node covered which. This resulted in the current metric for coverage, which is the percent of the node that is covered by the rest of the system. Nodes that have the exact same percentage as the current node are the



**(a)** Round Robin Assignment **(b)** Sequential Assignment

**Figure 3.2:** Coverage: Measured by the percent of node covered by other nodes.

ones that cover it.

As with similarity, coverage is highly dependent upon the assignment strategy, as can be seen in Figure 3.2. When all the files in one folder are stored on a single node, the coverage is poor as seen in Figure 3.2(b). However, when the files of a folder are spread evenly across the nodes, coverage increases by close to 65% (Figure 3.2(a)). This drastic difference, due to the assignment strategy, mimics what I would expect, and supports the coverage metric.

## 3.3 Object Popularity

One way to measure the popularity of a file or object is to use what is often called the "heat" of an object [12, 55]. Heat is often defined as the access frequency of a file or object over a period of time. Popularity influences successful load balancing, because it helps determine how to distribute frequently accessed data. If there is a specific set of data that are "hot", or very popular, then it is better as far as load balancing goes for that data to be distributed over multiple devices. While it is possible that an unbalanced system may result in better performance, this assumes that performance is your primary objective. I am addressing the case where there is not a primary objective, but multiple, equally important objectives.

Heat is often calculated by the product of the access rate and the service time, which requires either a workload that indicates access patterns or an assumption about the workload. In the case of web servers, for instance, the access rate can be successfully modeled by a Poisson process [34]. However, the same assumption may not be true in a file system. Rather than looking at frequency, an alternative approach is to consider recency: how recently the file was used.

Both recency and frequency on their own are somewhat incomplete. A better approach is to combine the two, similar to the algorithm **ARC** presented by Megiddo and Modha [41]. **ARC** is designed as a cache management policy, called Adaptive Replacement Cache. It continually balances between recency and frequency, in an online and self-tuning manner [42]. It does this by maintaining two lists, one with files seen at least once "recently" and the other with files seen at least twice "recently", implying high frequency. One major difficulty in adopting a combined approach like this is the dearth of dynamic data available for academic research. In addition, frequency is difficult to measure if the file has not been accessed yet.

Originally, since the data I have is static metadata and therefore there is no measure of frequency, and since using both recency and frequency is a better metric, I was going to try to estimate popularity by assuming that a more recent access meant more frequent access. However, having developed solid metrics for both system responsiveness (object similarity) and energy savings (object coverage), I decided to instead see if I could use the definition for "heat" found in [14, 34, 67, 69] to develop a better metric.

What I developed was an estimate for popularity, based on the definition of heat seen in Equation 3.2 where you're looking at the access rate and the expected service time. Disk accesses to a file can be modeled as a Poisson process with mean access rate of $\lambda$. Service time of a file is assumed to be fixed as $s_i$.

$$h_i = \lambda_i s_i \tag{3.2}$$

I started by looking at the access rate $\lambda_i$, which is the popularity weight times the aggregate access rate. The aggregate access rate can be seen in Equation 3.3, where $N$ is the number of files. I found that in some cases the aggregate access rate was fixed (either at 100 [14] or 200 [67]), and in other cases the aggregate access rate was varied (from 20 to either 240 or 1000 [67, 69]). For these experiments I present results for a fixed aggregate access rate of 200. Evaluation of varied rates is not considered for this work.

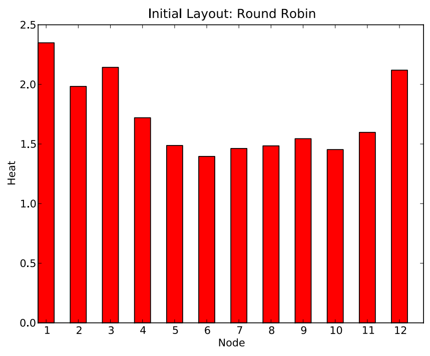
$$\sum_{i=1}^{N} \lambda_i \tag{3.3}$$

The other half of the access rate is the popularity weight, which provides an idea of the frequency of requests for that file. Equation 3.4 shows a definition for the popularity rate of a file, $p_i$, given a Zipf-like file distribution. Here, rank refers to the rank of the file (starting at 1), and $c = \dfrac{1}{H_N^{1-\theta}}$. The denominator of $c$ is the $N^{\text{th}}$ harmonic number of order $1-\theta$, which means that $c$ can also be written as seen in Equation 3.5. A common theme here is $1-\theta$, but there is no definition for $\theta$.

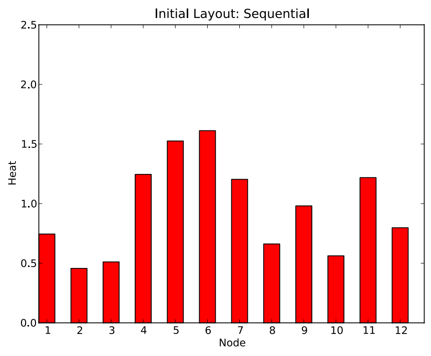
$$p_i = \frac{c}{\text{rank}_i^{1-\theta}} \tag{3.4}$$

$$c = \frac{1}{\sum_{k=1}^{N} \frac{1}{k^{1-\theta}}} \tag{3.5}$$

Most of the current work assumes that the file access request rate distribution is Zipfian with a skew of $\theta = \frac{\log \mathrm{A}/100}{\log \mathrm{B}/100}$, where A% of accesses are directed at B% of files. Given the Zipfian nature, generally the parameters are set as A$=70$ and B$=30$. However, since I am using a static data set, I don't actually know



**(a)** Round Robin Assignment



**(b)** Sequential Assignment

**Figure 3.3:** Popularity: Measured by the average heat of files on each node.

the access request rate distribution. That being said, the denominator of $c$ is incredibly similar to the denominator of the Zipf distribution (Equation 3.6). Here, $s$ is the value of the exponent characterizing the distribution, $N$ is the number of elements, and $k$ is the rank.

$$\frac{\frac{1}{k^s}}{\sum_{n=1}^{N} \frac{1}{n^s}} \tag{3.6}$$

The value of the exponent characterizing the distribution can be calculated, as $s$ is the slope of the log-log graph of the data set. However, calculating $s$ still requires knowing the file access request rates. So instead I make the common assumption that file sizes are inversely correlated with file access request rates. This allows me to say that the file size distribution is an inverse Zipf distribution with the same skew of *theta*. Since the file size distribution is known, as long as it follows an inverse Zipf distribution, the file access request rate can be calculated. I found that the file size distribution does indeed follow an inverse Zipf distribution, allowing me to calculate file access request rates.

However, if you recall Equation 3.2, the file access request rate is only one half of the equation. The other half is service time, which is comprised of seek time, rotational time, and transfer time. Seek time and rotational time are negligible compared to the transfer time, which is heavily influenced by the size of the file. I use an estimated service time metric equivalent to the size of the file in megabytes.

Since file popularity does not change regardless of where the file is placed, it is calculated before placement. Since I am dealing with simply static metadata to determine popularity and am using the size of the file in megabytes, the resulting values are extremely small. As a result, once the average heat of the files on each node is calculated, I have multiplied by $10^{13}$ for readability in Figure 3.3. As you can see, the results consistently vary depending on the assignment strategy as with our previous two metrics.

# Chapter 4

# System Design

Out of intense complexities, intense simplicities emerge.

Winston Churchill

While Pareto optimization provides us with a set of possible solutions, it does not actually make a decision as to where each object should be placed. Thus, I designed a data allocation algorithm called **JACK**, which stands for ***J**oining*, ***A**ggregating*, and ***C**ollocating **K**nowledge.* After identifying a set of Pareto optimal solutions, **JACK** uses these solutions to make a decision as to where each object should be placed. This decision is made based on information already calculated about the state of the system, as well as information given by the system administrator when necessary. **JACK** is designed to place data at any granularity; from a specific disk to an entire data center.

**JACK** consists of three stages: calculation, optimization, and decision. In the calculation stage, the metrics described in Chapter 3 are calculated to measure the objectives for which I am optimizing. The majority of the calculation stage can occur at any point in time prior to data needing to be placed, and in fact should occur in idle periods. Once the initial calculations have been made, this stage only needs to happen when new data has been added to the system.

The optimization stage occurs when an object needs to be placed initially or moved to a new location. This is when **JACK** calculates the Pareto frontier to determine the set of optimal devices on which the object could be placed. Phrasing the problem as a multi-objective optimization problem is attractive, because it provides flexibility to optimize for multiple objectives while still choosing the allocation from the Pareto frontier that favors a specific objective. Since I do not want the data allocation algorithm to be dependent on the user knowing what is important, using an a posteriori algorithm makes more sense.

One type of a posteriori algorithms that has been gaining popularity recently is genetic algorithms [30, 40, 58]. While many a posteriori algorithms determine the Pareto frontier one solution at a time (as the result of multiple single-objective optimizations), genetic algorithms are designed to discover the entire Pareto frontier. This is also expensive to compute: in the worst case, the complexity becomes $O(N^3)$, where $N$ is the population size [13]. For the purposes of this document, the population size is the number of devices in your system. In a small system, such as our development cluster of 16 nodes, $O(N^3)$ is not bad. In a real world situation, however, the complexity becomes orders of magnitude larger. Consider the high performance computer at Pacific Northwest National Laboratory, which is considered to be "small". It has over 2300 nodes, each with 8 local disks.

There are algorithms that can reduce the complexity to $O(N^2)$ in the worst case through careful bookkeeping and maintaining comparisons. However, the result of the bookkeeping is the increase of storage from $O(N)$ to $O(N^2)$. While this is a fairly high cost, it is still less costly compared to the approximate Pareto frontier found by running multiple single-objective optimizations. In order to alleviate this even further, I introduce a “pre-processing” step before the actual Pareto optimization. Based on the information calculated in the calculation step, **JACK** narrows down the number of possible devices for an object to be stored on, reducing the size of $N$.

Once the subset of optimal devices has been identified, the decision stage is invoked. **JACK** must make a decision as to which of these devices is the best for the given object. Ultimately, this decision will be based on input provided by the system administrator indicating which of the objectives is the most important. Recall the house buying problem discussed in Section 2.1.1, where houses A, B, and C were all optimal, given the requirements all carried equal weight. The user input allows the system administrator to indicate that one of the requirements has a higher weight. Just like the house buyer may decide that a good school system should count more than a nice backyard, the system administrator could decide that system responsiveness counts more than energy savings.

In the instance where the input indicates that all three objectives are of equal importance, or no input is given, a balanced decision strategy is necessary. A baseline approach is to have **JACK** make decisions in a round-robin fashion. The decision will favor each objective in turn, starting with energy savings, then system responsiveness, then load balancing. Knowing the prior two decisions will allow **JACK** to maintain this rotation as long as there is no user input. This is not an optimal solution, but it assures that every objective gets an equal opportunity.

There are two approaches to a more optimal balanced decision strategy: the simple approach and the intelligent approach. The simple approach takes the solution on the Pareto frontier which minimizes the difference among the objectives. In other words, it determines the Pareto optimal solution which is most balanced among the objectives. The intelligent approach requires **JACK** to take the state of the system into account when making a decision. This allows **JACK** to make intelligent decisions about whether to favor one objective over the others based on the state of the system. While this work is focused on using **JACK** to initially place data, it can also be used to relocate data. The relocation of data, however, is reserved for future work.

# Chapter 5

# Theoretical Approaches

Experience: that most brutal of teachers. But you learn, my God do you learn.

C.S. Lewis

The theory behind **JACK** is solid, with the minor exception of the concept of "heat" for the load balancing metric. The reason using a metric of heat for load balancing is so attractive is because it's "easy". Heat can be calculated by simply looking at how frequently the file or object is accessed, and then it can be used to optimize for load balancing by taking care not to put all the frequently accessed files on the same disk. The difficulty lies in finding data that reports, or allows you to calculate, the frequency of access to each file.

As discussed in Section 3.3, I used the definition for "heat" as found in [14, 34, 67, 69], shown in Equation 5.1.

$$h_i = \lambda_i s_i \tag{5.1}$$

When I started looking into tweaking the aggregate access rate ($\lambda_i$), however, I realized that these papers calculated a Poisson process for *each file*, something that is not feasible in any reasonably sized system.

With no clear direction for a metric for load balancing, I moved away from Pareto optimality and phrasing it as a multi-objective optimization problem. I started considering a more theoretical approach, while keeping in mind the desire to be able to optimize for multiple objectives. Two theoretical models arose from this: a partial bin packing model, and a model based on queuing theory. As with the Pareto optimality approach, a bin packing model nicely fits two out of the three objectives, and in fact I used concepts from the bin packing model in the next direction of my research. The queuing theory model is a purely hypothetical but solid theoretical model, and arose from my attempts to identify from whence Equation 5.1 was derived.

## 5.1 Bin Packing Model

In its simplest form, bin packing takes objects that have a size and places them in bins that have a capacity and minimizes the number of bins needed. Fairly obviously, in this work the objects correspond to files or objects and the bins correspond to the disks. This simple form of bin packing is the one dimensional bin packing problem. There exists a list $L$ of numbers between 0 and 1 which need to be assigned to unit capacity bins so that no bin receives numbers totaling to more than 1 and a minimum number of bins is used.

Clearly, however, there is more to it or this would not be a difficult problem. Indeed, rather than just placing files on disks, there is a set of files, each of which have a set of properties. This set of files needs to be placed on a set of disks such that the sum of the file properties does not exceed a threshold that has been set for the disk. This requires multi-dimensional bin packing, where the items being packed can be thought of as vectors $\langle R_1(x), R_2(x), \cdots, R_s(x)\rangle$ where $R_i$ is a function and the goal is to pack the vectors into bins such that bins are dominated by $\langle 1, 1, \cdots, 1 \rangle$.

Mathematically, $R_i : 1 \leq i \leq s$ is a set of functions $R_i : L \rightarrow [0, 1]$ where $L$ is the set of files. That is, $R_i$ maps a file to a value between 0 and 1, or rather $R_i$ identifies resource usages of the file. While the goal of multi-dimensional bin packing is to pack vectors into bins such that bins are dominated by $\langle 1, 1, \cdots, 1 \rangle$, it is possible to change those values. This allows one not only to set thresholds to maximum system responsiveness, but also provides a model for a heterogeneous system. It is not difficult to imagine that the thresholds for a solid state drive would differ from those of a hard disk drive; modifying multi-dimensional bin packing allows for this.

The problem arises, as with Pareto optimization, with load balancing. As I have pointed out, by modifying the threshold values I can maximize system responsiveness. Since the inherent goal of bin packing is to minimize the number of bins, the idea of reducing the number of disks used to maximize energy savings is already factored into the model. It may be possible to use the thresholds to also maximize load balancing, but ideally in that case one of the disk thresholds (and corresponding file property) would be how "hot" the file is, or how frequently it is accessed.

## 5.2 Queuing Theory Model

Studying the measure of heat in Equation 5.1 revealed that its origin lies in queuing theory. In determining this, I developed a theoretical model rooted in queuing theory, based on the concept that each disk in the system can be modeled as a queuing system.

I am using the standard three part descriptor of A/B/m that denotes an m-server queuing system where A is the inter-arrival time distribution and B is the service time distribution. A and B can take values from the following set of symbols: M (exponential), $E_r$ (r-stage Erlangian), $H_r$ (r-stage hyper-exponential), D (deterministic), and G (general). So an M/G/1 queue is a single-server system with Poisson arrivals and an arbitrary service time distribution denoted by $B(x)$, and a service time pdf denoted by $b(x)$ [29].

The key to an M/G/1 system is imbedded Markov chains, also known as semi-Markov processes. In an imbedded Markov chain, state transitions can happen at any time, causing the times between states to obey an arbitrary probability distribution, but at the instants of state transitions, the process behaves like an ordinary Markov chain. For the purposes of this model, state transitions occur when a customer departs, as the expected service time at those instances is 0 for the new customer in service (as the new customer just now entered).

The imbedded Markov chain then is the number of customers present in the system immediately following the departure. The distribution of time between state transitions is equal to the service time distribution $B(x)$ whenever the departure leaves behind at least one customer. If the departure leaves behind an empty system, the distribution equals the convolution of the inter-arrival time distribution (exponentially distributed) with $b(x)$.

Working under the assumption that a single node can be modeled as an M/G/1 queue, I can make use of the *utilization factor* to optimize load balancing. System responsiveness can be optimized by minimizing the total amount of time spent in the system, both waiting in the queue and the time in service. Lastly, I can take advantage of the busy-idle cycles and batch requests in such a way as to maximize busy times to optimize for energy savings.

## Load Balancing

The probability that the system is busy is defined as the utilization factor, $\rho$, and is equal to the expected number of arrivals per service interval.

$$\rho = \lambda\bar{x} \tag{5.2}$$

This is a very familiar looking equation, and indeed Equation 5.1 is derived from the utilization factor. Here, $\lambda$ is the arrival rate of customers and $\bar{x}$ is the average time spent in service.

The goal of load balancing is to spread your busy data across your system to keep from any one disk being thrashed. In order to do this, the model needs to minimize the utilization factor. To put it another way, it is necessary to keep the probability of any disk being busy from getting very high. This needs to be monitored at the system level, to keep the utilization factor approximately equal across all disks.

## System Responsiveness

Knowing that the service time for the model is the arbitrary service time distribution denoted by $B(x)$, it is possible to map files to this by assuming that each file has a service time along that distribution. Depending on the configuration of the system, the service time will be different for given nodes. The average time spent in the system can be expressed using $\rho$ as follows.

The average number of customers in an M/G/1 system can be written as seen in Equation 5.3.

$$\bar{q} = \rho + \rho^2 \frac{1 + C_b^2}{2(1-\rho)} \tag{5.3}$$

This is known as the Pollaczek-Khinchin (P-K) mean-value formula, where $\bar{q}$ is the average number of customers found at random. It is written in terms of the utilization factor $\rho$ and the squared coefficient of the service time variation, $C_0^2$. Little's law states that $\bar{N} = \lambda T$, which is the expected number of customers $\bar{N}$ in a system related to the arrival rate of customers $\lambda$ and their average time in the system $T$.

It is possible to apply Little's law to Equation 5.3 to obtain the average time spent in the system, both waiting in the queue and time in service. Since $\bar{q}$ is the average number of customers found at a random time, it can be stated that $\bar{q} = \bar{N}$, resulting in Equation 5.4.

$$\bar{N} = \rho + \rho^2 \frac{1 + C_b^2}{2(1-\rho)} = \lambda T \tag{5.4}$$

Solve for $T$ and the result is

$$T = \bar{x} + \frac{\rho\bar{x}(1 + C_b^2)}{2(1-\rho)} \tag{5.5}$$

This states that the average total time spent in the system is the average time spent in service plus the average time spent in the queue. The average queuing time can be expressed as follows

$$W = \frac{\rho\bar{x}(1 + C_b^2)}{2(1-\rho)} = \frac{W_0}{1-\rho} \tag{5.6}$$

where

$$W_0 \triangleq \frac{\lambda\bar{x}^2}{2}$$

$W_0$ is the average remaining service time for the customer in service at the time of a new arrival

In general, I actually expect that the total time $T = \bar{x} + W$ will be dominated by the average service time $\bar{x}$ and that the average queuing time $W$ is next to nothing. In order to maximize system responsiveness, this needs to be true; no request should be waiting on another.

## Energy Savings

Interestingly, the most difficult objective to optimize for in the theoretical queuing theory model is energy savings. In an M/G/1 queue, however, there is a concept of busy and idle periods. The expected length of a busy period depends only on the average service time $\bar{x}$ and the arrival rate $\lambda$, and is expressed as $\frac{1}{\bar{x}-\lambda}$.

It is possible to batch incoming requests in order to maximize the busy periods, optimizing for energy savings. Care must be taken to find an optimal batch size, even given a set of constraints defined in order to not adversely affect system responsiveness. Of course, once requests are arriving in a batched manner, it is no longer an M/G/1 queue.

# Part II

# Practical Approach

# Chapter 6

# Applied Machine Learning

Anyone who has never made a mistake has never tried anything new.

Albert Einstein

In order to approach the system optimization problem from the bottom up, I focused on the problem of finding intelligent ways to place the data. I changed the definition of data to be a workload, rather than attempting to identify behavior based on static metadata. I further defined a workload as a "logical volume", which could be anything from a single application to a virtual machine running several other virtual machines, each of which could be running several applications.

Approaching the problem from the perspective of placing workloads can be broken into two parts, modeling and mapping. First, a model of how a workload is going to behave must be created. This allows the prediction of how a workload will behave in the presence of other workloads. Second, that prediction must be able to be used to help map on which physical device the workload should be placed. This mapping is subject to practical issues such as whether the size of the workload added to the currently used space on the device will exceed the maximum capacity of the device.

The goals of modeling revolve around a single formula, specifically the following:

$$\text{Performance}_i = F(\text{Workload Characteristics}_i) \tag{6.1}$$

This says that the performance of a specific workload, $i$, is determined by some function $F$, the input to which is the set of characteristics that define the workload $i$. Clearly, a key problem is determining the function $F$, which I chose to approach with machine learning. The key to this challenge is determining whether the performance interference experienced by and caused by the workload should be part of $F$. The reason is that this becomes complicated, since the interference is dependent on other workloads as well as the properties of the physical disk. While an actual measure of performance interference is difficult, if not impossible, to determine, a prediction is not.

Mapping workloads to physical devices can be thought of as a constraint satisfaction problem. In this case, the problem is that there is a set of $n$ logical volumes to be placed onto $m$ physical volumes, with a constraint of either performance interference or service level agreements, such that the number of physical volumes used is minimized.

$$\sum_{i=1}^{b} w(S_i) \tag{6.2}$$

The goal is to find the allocation that maximizes the summation, where $S_i$ is a set of logical volumes and $b \leq m$ is the number of physical volumes. The major challenge here is to define the function $w$. This combination of predicting workload performance and constraint satisfaction resulted in an approach that drove the remainder of my dissertation.

# Chapter 7

# Predicting Workload Performance

To raise new questions, new possibilities, to regard old problems from a new angle, requires creative imagination and marks real advance in science.

Albert Einstein

The approach identified in Chapter 6 uses the concept of predicting workload performance and using these predictions along with constraint satisfaction in order to optimally place workloads as they enter the storage system. I argue that the important aspect for optimal placement is not the disk or the workloads, but how the workloads themselves interact with other workloads. Rather than attempt to mitigate the workload contention on disk, I model how the performance of one workload is affected given the presence of other workloads, and use this to make intelligent placement decisions. Thus, I care less about the actual predicted value of the performance of a workload, but rather on which disk the new workload performs best compared to the other disks in the system, given the existing workloads on the disk.

One of the primary benefits of approaching the workload placement problem in this manner is that since I am interested in how workloads interact with each other comparatively, I am not tied to models that reflect only how a specific device will behave. It also lifts the constraint of finding a precise value for performance. One could look at all possible combinations of workloads and determine the best division across the system, but such an approach would really only be viable at the initialization of a system. The best division could be constantly changing, and constantly moving workloads is not optimal. Instead, I observe what is already running on the system, and determine the best placement given the current state.

In order to predict which disk would be the best fit for a new workload, I needed to model the workloads currently running on each disk as well as the new workload. I have chosen to use workload characteristics as the input to the model, as they can be specified or observed. This allows for easy adaptation to a real system.

The performance of a specific workload, $i$, can be defined as the output of some function $F$, the input to which is the set of characteristics that define the workload $i$.

$$\mathrm{p}_i = F(\mathrm{wc}_i) \tag{7.1}$$

Here the workload $i$ is specified by its characteristics: $\mathrm{wc} = \langle bs, io, rw, pr \rangle$, where $bs$ is the block size, $io$ is the number of IO units in flight against a file, $rw$ is the read/write ratio of the workload, and $pr$ is the percentage of the workload that is random. I chose these characteristics because they can be specified, as I am doing in this work, but they can also be measured or observed in a real-world system, allowing the model to be easily adapted to a real system without needing to change how workloads are defined. Similarly, I measure performance as the

**Table 7.1:** Relationship between the number of possible values for each workload characteristic, given five workload characteristics, and the number of unique workloads.

| Number of Characteristic Values | Unique Combinations | |
|---|---|---|
| | One Workload | Two Workloads |
| 2 | 32 | 1024 |
| 3 | 243 | 59049 |
| 4 | 1024 | 1048576 |
| 5 | 3125 | 9765625 |

tuple of total bandwidth, total latency, and percentage of disk utilization.

It is, in theory, fairly straightforward to predict the performance of workloads using brute force methods by making $F$ a simple look-up table. In practice, however, this is not a realistic approach. If the number of characteristics used to define the workload are limited, $nc = 5$, and limit the number of possible values each characteristic can take on, $v$, the number of possible unique workloads can be calculated by $v^{nc}$. This assumes that the number of values for each characteristic is the same, which may not always be the case. The formula then becomes the product of the number of values for each characteristic.

$$\prod_{i=1}^{nc} v_i \tag{7.2}$$

This means that if I restrict the number of possible values each characteristic has to two, there are 32 unique workloads that can be created. Since it is possible to have multiple instances of the same workload running, when there is more than one workload on a device, the number of unique combinations is calculated by the following:

$$(\prod_{i=1}^{nc} v_i)^n \tag{7.3}$$

where $n$ is the number of workloads. In Table 7.1, I outline the growth that results when I vary the number of values possible for each characteristic for one and two workloads.

These values are a conservative estimate, as they do not take into consideration the fact that just because two workloads have the same set of characteristic values does not mean they will behave identically. For example, consider two workloads that have a read-write ratio of 25% read, 75% write. If workload $A$ reads first and workload $B$ writes first, the behavior is very different. The numbers in Table 7.1, then, are a conservative estimate to the number of possible unique workloads. It is clear, then, that $F$ can not be a simple look-up table. In order to intelligently place a workload in a reasonable amount of time, it is necessary to be able to predict the performance of the workload.

# Chapter 8

# Experimental Design

No idea is so outlandish that it should not be considered with a searching but at the same time a steady eye.

Winston Churchill

Real world workloads are notoriously difficult to obtain, as they contain identifying, and often sensitive, content and information about performance. In addition, they do not always translate well from the system on which they were obtained to another. Synthetic workloads attempt to alleviate the translation problem by creating statistical models or probability distributions of various workload characteristics [37], such as request size, request rate, and IO queue depth.

Rather than depend on real world workloads or statistical models, I developed a unique approach called **Bypass**. **Bypass** uses a workload generator, specifically the Flexible IO tester (*fio*) [4], to generate workloads using easily modified input parameters. Using the work summarized in [27, 26, 43, 21], further discussion of which can be found in Chapter 13.1.4, I identified significant workload characteristics and mapped them to input parameters in *fio*. By using reported and realistic values for these characteristics to generate workloads, I bypass the need for real world data while maintaining realistic workloads.

It is important to note that **Bypass** is not a workload generator or a benchmark, it is an approach to workload simulation. Benchmarks are designed to test the performance of a system [60], and thus often present an incomplete characterization of workloads. A benchmark designed to test the de-duplication capabilities of a system will be highly concerned with the content of the workload, while a benchmark to test the response time will emphasize relative timing.

Benchmarks and real world workloads that excel for specific system tests do not necessarily describe the workload completely, and report the performance of the system. I am interested in the performance of the workloads themselves; to limit the effects of interactions with layers of the software stack, I bypass the file system and perform IO directly on the disk. **Bypass** serves two purposes: it provides a more complete understanding of workloads themselves by using characteristics which define real world workloads, and it reports the performance of the workloads rather than the system.

## 8.1 Obtaining Workloads

Having decided on *fio* as the workload generator and identified the defining characteristics of real world workloads , I can now generate realistic workloads and run them. In order to generate workloads, *fio* takes the characteristics as either command line input or reads them from a job file. Once a workload, or set of workloads, has been generated, *fio* can run it and return performance information including bandwidth, latency, and IOPS.

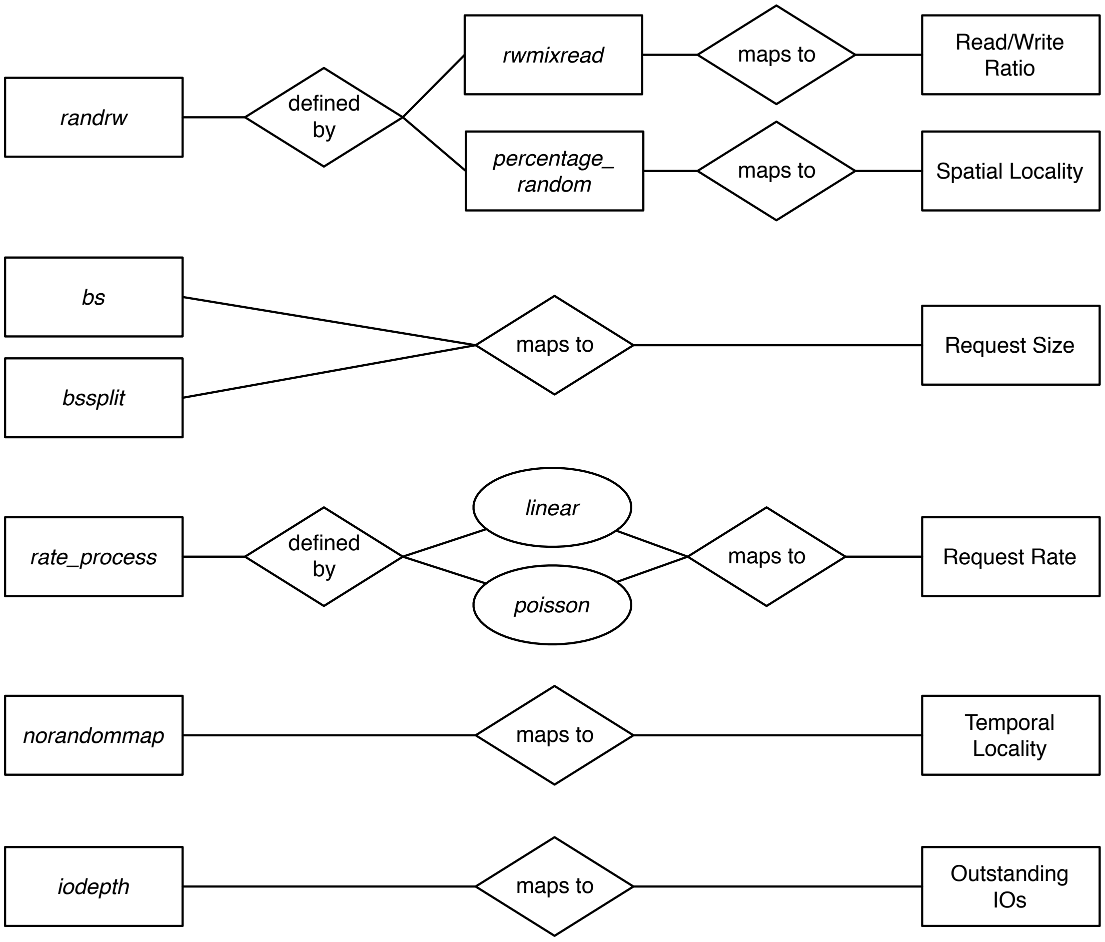


**Figure 8.1:** Input parameter to workload characteristic mapping.

### 8.1.1 Characteristics to Input Parameters

It is fairly straightforward to map the workload characteristics to input parameters in *fio*, despite the wide array of parameters *fio* allows you to specify. The formal mapping is shown in Figure 8.1, where rectangles indicate a *fio* parameter or a workload characteristic, ovals are a possible value for the parameter, and diamonds indicate the relationship.

I use three parameters to cover the read/write ratio and spatial locality. The first parameter, `rw`, indicates whether the job is read or write and random or sequential. For example, `rw = read` would indicate a workload that was composed only of sequential reads. Since I want more control over both the read/write ratio and the spatial locality, I designate `rw` to be `randrw`, indicating a mixed random read/write workload. This allows two more parameters to be set, `rwmixread` and `percentage_random`. The first, `rwmixread`, is a percent that represents how much of the workload should be read IO, with the inverse being the amount of write IO. Spatial locality is defined with `percentage_random`, indicating how much of the workload should be random IO and how much should be sequential.

The request size is handled by one of two parameters. Either `bs` is used to indicate a static block size for the entire workload, or `bssplit` is used to indicate a range of different block sizes. To use `bssplit`, the block size and the percent of the workload that should use that block size must be indicated. It is also possible to indicate different block sizes (or block size distributions) for reading and writing. Request rate is handled by the parameter `rate_process`, which controls how IO is submitted. I set this to *poisson*, as real world request rates are often modeled as a Poisson process.

I force *fio* to not keep track of where blocks are being written in a random workload using `norandommap`. This allows some blocks to be overwritten, while others never get touched. Another parameter that could be used for temporal locality is `dedupe_percentage`, which tells *fio* to generate that percent of identical buffers when writing. I currently only use `norandommap` to get temporal locality; `dedupe_percentage` would allow finer control of that characteristic.

In order to be able to account for outstanding IOs, I used the parameter `iodepth`. This defines the number of IO units in flight against the file at a time. I used the Linux native asynchronous IO engine (`libaio`) with non-buffered IO to ensure the IO depth is met. IO latency is not a parameter that I set, but rather one I collect for performance information. It is possible to set the parameter `latency_target` which will attempt to find the maximum performance for a workload while maintaining a latency less than the indicated target.

### 8.1.2 Assigning Values

Up to this point, I have not discussed what values to assign the input parameters. While one of the studies did not have actual values [27], the other three had at least average values. While I used the values from [43] as reference, they are averaged for specific vendor systems that were being tested, and are not useful for generating specific workloads.

This left me with the studies from Kavalanekar et al. [26] and Gulati et al. [21], both of which report values for the workloads they characterized. Kavalanekar et al. report a mix of averages and modes, while Gulati et al. provide a distribution of values when appropriate. I present the range of values from which I chose based on these studies; future studies could improve these to create even more realistic workloads.

Request size ranged from 2KB to 512KB in powers of two, with the most frequent sizes being 4KB, 8KB, 32KB, and 64KB. As mentioned in Section 8.1.1, I either keep the block size constant or create a distribution, with different values for read and write. For example, a workload based on the Microsoft Exchange Server data [21], could have a read block size of 8KB with a write block size being a distribution with 75% being 8KB, 15% are 16KB and the remaining 10% are 32KB.

Given the request rate in *fio* is limited to either a poisson distribution or a linear rate with fixed delays, and the interarrival time of requests varied quite widely (anywhere from 1 millisecond [21] to 500 milliseconds [26]), I chose to use the poisson distribution. Lambda is fixed at $10^6$/IOPS for a given workload. Outstanding IOs are allowed to range from 1 to 32, the maximum reported in Gulati et al.

One of the parameters I let vary quite widely is the one corresponding to spatial locality, `percentage_random`. Spatial locality is one of the easiest ways to observe how workloads interact; highly sequential and highly random workloads do not perform well when placed on the same disk. Another reason to vary spatial locality is that there is high variability in the real world workloads themselves. The Windows build server [26] for example is mostly random (only 4% of reads and 13% of writes are sequential), while the developer tools release server is more sequential (20% of reads and 32% of writes).

I let the read/write ratio range from 0% (meaning an all write workload) to 100% (an all read workload). In [26] the ratio ranges from essentially an all read workload to an all write workload and several ratios in between. Since the ratio is not recorded in the study done by Gulati et al. I used 50% when replicating those workloads and modified from there (for example a drop to 45% or an increase to 55%) to get slightly different workloads.

In order to collect data initially, I used the parameters specified in Table 8.1

**Table 8.1:** Workload Characteristics.

| FIO Parameter | Value |
|---|---|
| read percentage | 0 – 100 |
| percent random | 0 – 100 |
| iodepth | 1 – 64 |
| block size | 1k, 4k, 8k, 16k, 32k, 64k, 128k, 256k, 512k |

and randomly selected a value from the ranges shown for each workload. In this way, I was able to create a large number of workloads with unique IO characteristics. This was useful for the preliminary experiments that determined the viability of using performance prediction. In the experiments that tested using performance prediction to optimally place workloads, I used the more realistic characteristic values.

## 8.2 Modeling Workload Performance

Having identified the input parameters necessary to characterize a workload, and assigned realistic values to them, **Bypass** can use *fio* to run the generated workload and report performance information. As mentioned, *fio* is writing directly to either a hard disk drive or a solid state drive and bypassing the file system entirely. This is so that when I run multiple workloads on the same device, I can assert that changes in performance come from how the workloads are interacting. In the presence of a file system, changes in performance could be an artifact of the IO scheduler being used rather than the workload interaction.

Some of the initial tests collected IOPS (IOs per second), both read and write, as the measure of performance; most of the experiments also collected total completion latency and bandwidth. I used the measured performance information, along with the characteristics of the workloads, to create a model using machine learning to predict the performance of the workloads. I am using CART (Classification And Regression Tree) models since they allow for rapid iteration and are human readable, making it easy to identify behavior. I used the scikit-learn [51] modules for regression trees to generate the models.

In all of the experiments, I reserved 10% of the data for testing and used the remaining 90% for training and validation. I divided the training data into nine equal parts and used eight parts for training and one part for validation. This was repeated until all parts were used as validation. I chose to do 9-fold cross-validation, rather than 10-fold, because this way each validation part consisted of 10% of the initial data set. In each fold of validation, the mean relative error is calculated to give a measurement of goodness to the generated model. Mean relative error is calculated by comparing the predicted value to the real value using the following formula: $\frac{|\text{predicted}-\text{real}|}{\text{real}}$. I select the model with the lowest relative error, and that model is used with the test data to produce results.

# Chapter 9

# Pairwise Prediction

However beautiful the strategy, you should occasionally look at the results.

Winston Churchill

To successfully place workloads optimally by predicting the performance, I first needed to make sure that predicting the performance of workloads using my method of workload generation and modeling was possible with some degree of accuracy.

The initial set of experiments I designed were specifically to test whether predicting the performance of workloads was feasible. Each experimental run generated two workloads by randomly sampling from the values in Table 8.1 and wrote the characteristics to a file that could be read by *fio*. **Bypass** then ran the workloads and collected performance information as a tuple of total bandwidth, total completion latency, and total IOPS. I collected data from 3 000 experimental runs. The workload characteristics, as well as the performance of the first workload in isolation, were used as described in Section 8.2 to create a CART model.

In addition, I used a historical average as a naive predictor to compare how

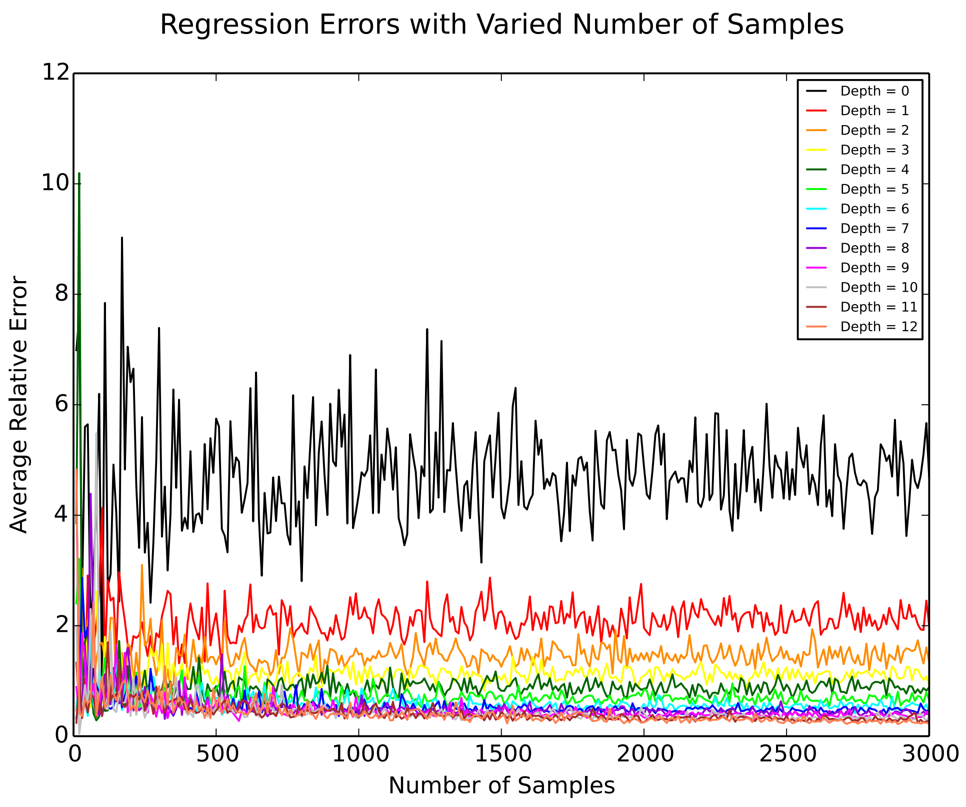


**Figure 9.1:** Predicting performance with varying tree depths and number of samples.

the CART model did at predicting performance. The historical average simply takes an average of the performance that has been seen previously and uses this as the predicted performance. I consider this a baseline for comparison purposes, to ensure that the CART model is successfully predicting the performance.

There are two parameters I chose to vary, the depth of the decision tree and the number of samples used, both of which represent a tradeoff in cost and benefit. A very shallow decision tree will be faster to compute and traverse, but will give less accurate predictions. The deeper the decision tree is allowed to go, the more accurate the prediction, but in addition to being more costly in computation and

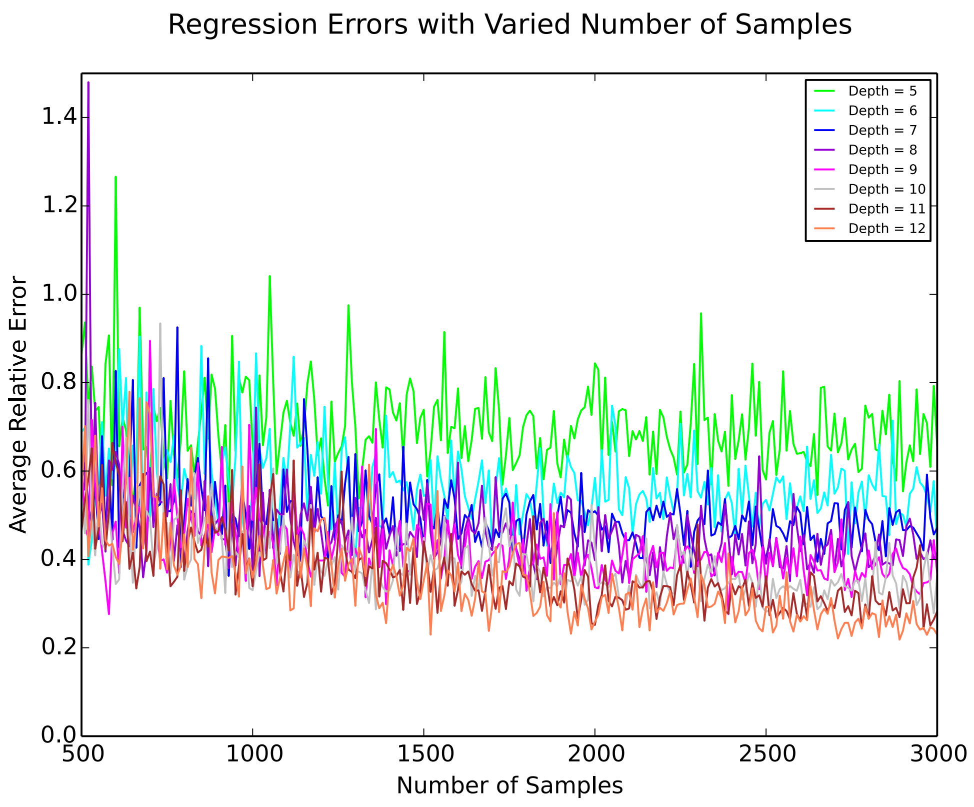


**Figure 9.2:** A closer look at the higher depths of the regression tree.

traversal, there is also the danger of overfitting. Similarly, the fewer samples that are needed the faster the model will be to compute, but a greater number of samples allows the model to begin to converge on a mean relative error.

The first thing to note in Figure 9.1 is that naive predictor (the black line) has a much higher mean relative error than any depth of the CART model. This is encouraging, since it means choosing to use machine learning was correct and that the CART model is better at predicting performance than the baseline. As expected, the mean relative error is very unstable with only a few samples; using only ten data points to create a predictive model isn't going to be very accurate. As the number of samples increases, the mean relative error begins to stabilize.

Based on Figure 9.1, a minimum of 500 samples seems necessary to create a stable model, though the more samples the better.

The other thing of interest in Figure 9.1 is that there appears to be a depth after which there is little to be gained by allowing the tree to go deeper. Figure 9.2 shows a closer look at the higher depths, with the $X$-axis starting at 500 to eliminate the variance in the mean relative error caused by too few samples. It is clear that while the mean relative error does improve as the depth increases, that improvement is only notable at the higher sample sizes. Additionally, the improvement past a depth of six is not significantly different (at 3000 samples, the difference in mean relative error between a depth of six and a depth of twelve is 0.204).

An example of one of the CART models generated is shown in Figure 9.3, where the depth is limited to three. The leaf nodes show the mean squared error, the number of samples at that leaf, and the predicted values for the tuple of bandwidth, latency, and IOPS. It is interesting to note that the top split is the latency of the first workload in isolation. Indeed, this remains constant across the different models built and different depths allowed. Performance is the best predictor of performance: that is, how the first workload performs on its own is a good predictor of how the two workloads will perform together. Another interesting thing to note is that the tree is not even; it is heavily weighted to the left. This is encouraging, since that means the majority of the workloads that were used to create the model had lower rather than higher latencies.

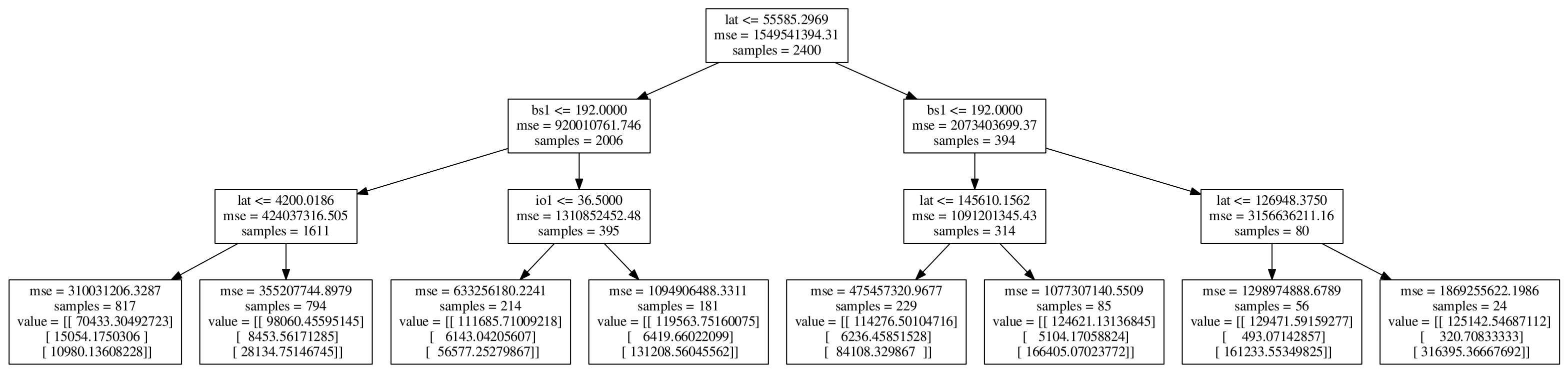


**Figure 9.3:** An example CART model with depth of three.

# Chapter 10

# Reducing Overhead

If we knew what it was we were doing, it would not be called research, would it?

Albert Einstein

While applying machine learning to the allocation problem allowed me to predict the performance of a workload with respect to other workloads, there is an associated, and often significant, overhead. In this particular problem, there is a time and space overhead in both creating and maintaining the models used for prediction. I looked at two different ways to decrease the overhead: one by creating a single model for each type of storage device rather than a model for every device in the system; the other by using a representative workload in place of keeping track of all the workloads running on a given device.

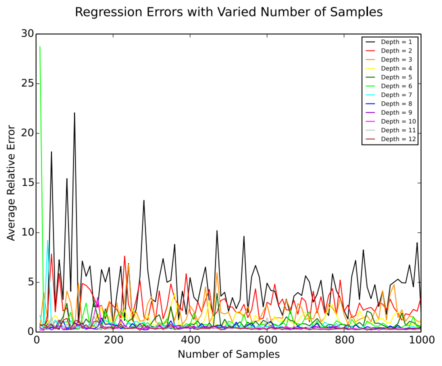
## 10.1 Device Class Models

Since modeling every single device in the system would swiftly become unwieldy in any decent sized system, I wanted to find a way to avoid individual models. As I am interested in the interaction among workloads, I decided to look at the model on a class basis rather than a device basis. Solid state drives are going to behave differently from hard disk drives, and this difference needs to be reflected so that workloads can be placed accordingly.

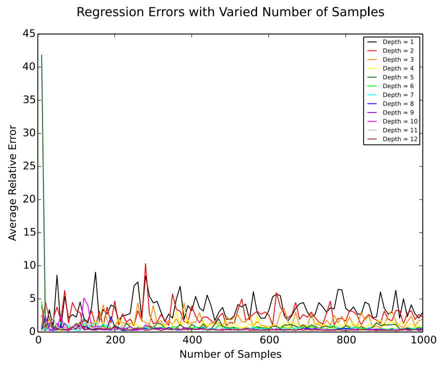
These experiments followed the general outline of collecting data as in Section 8.2, except I collected data on four individual devices, and then aggregated the data from those devices and sampled from the aggregate data to create the class model. There is no baseline in these experiments, and these results are all collected from SSDs.

The graphing is similar to the previous section's experiments, varying the number of samples and the depth of the tree, and measuring the mean relative error. The difference in these experiments is that I am interested in how the class model does compared to the individual device models. In Figure 10.1, it is clear that with only a few samples, the mean relative error varies wildly. In most of the device models, however, once the model gets above 200 samples and the depth is deeper than about four, the mean relative error starts to converge. The class model appears to fluctuate much more than the device models, though when the $Y$-axis is noted it becomes clear that it is not fluctuating more than most of the device models. In addition, the class model is sampling from a much larger pool of data, so the higher relative error at shallow depths makes sense. The larger the sample set, the harder it is to accurately predict based on only a few decision points.

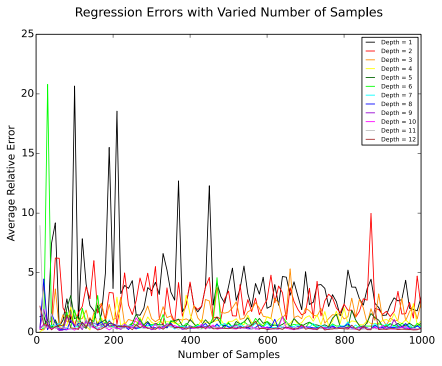
In order to do a more direct comparison, I took the data that was used for test for each device model and used it with the system model. While interesting, it is difficult to make any sort of statement about how the models compare with each other, since the class model fluctuates at different points than the device models.



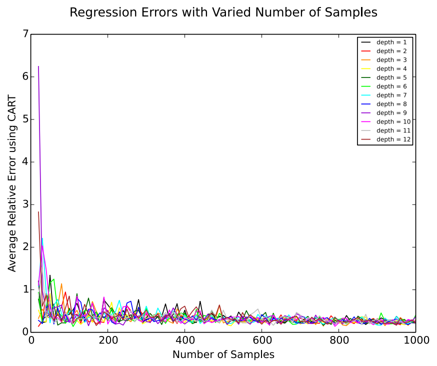
**(a)** Device A



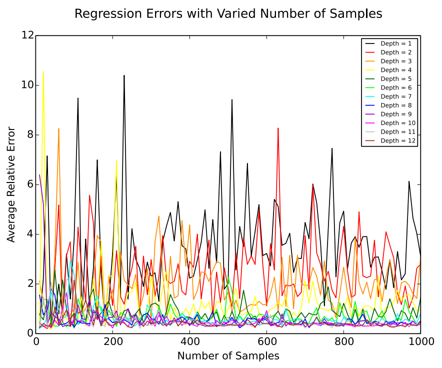
**(b)** Device B



**(c)** Device C

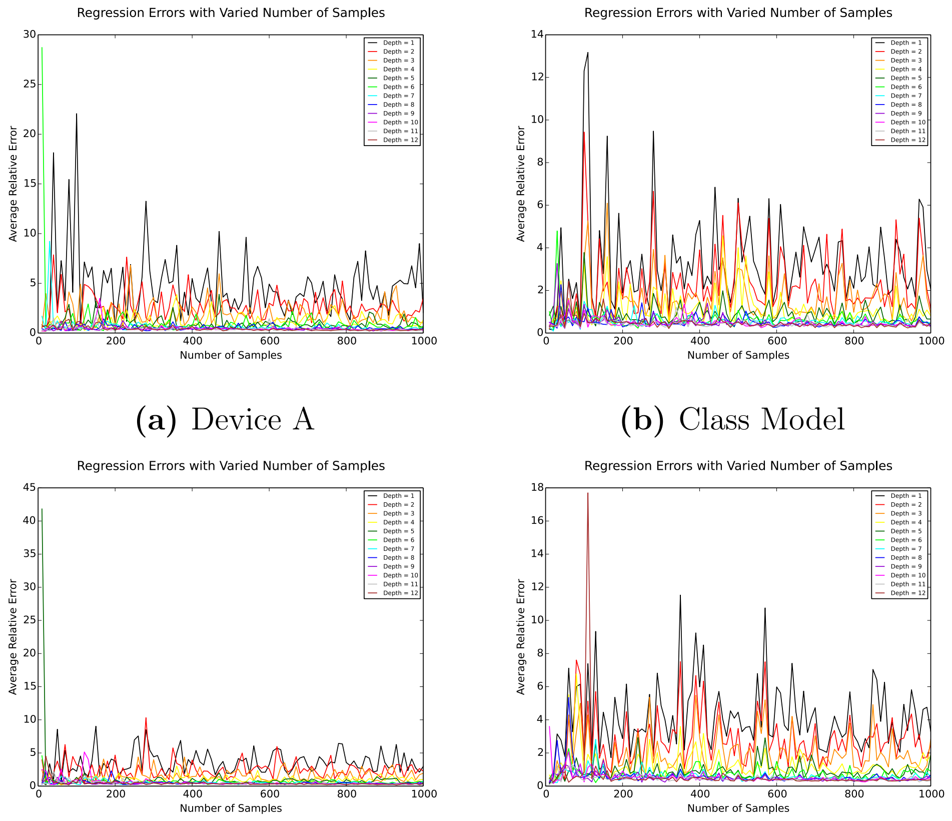


**(d)** Device D



**(e)** Class Model

**Figure 10.1:** Device and class models.



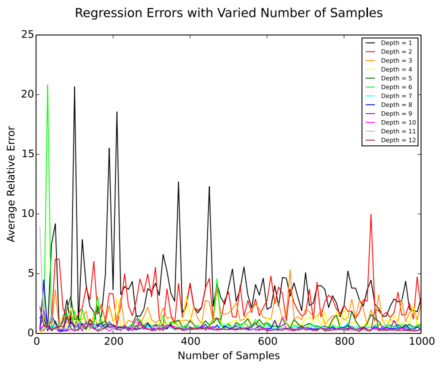
**(a)** Device A **(b)** Class Model

**(c)** Device B **(d)** Class Model

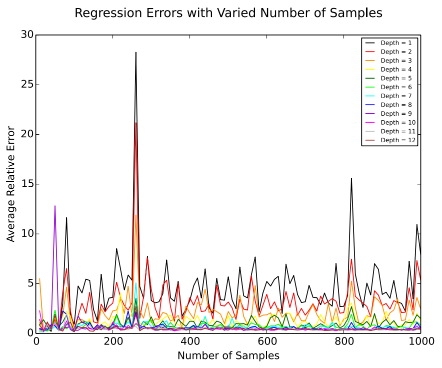
**Figure 10.2:** Direct comparisons.

I decided to give the device models the best advantage I could. From the graphs in Figure 10.2 and Figure 10.3 I determined the depth with the lowest mean relative error for each device, and then compared that with the same depth on the class model. As can be seen in Figure 10.4, while the class model has a some spikes in relative error where the device model does not, the class models start to mimic the device model as the number of samples increases.

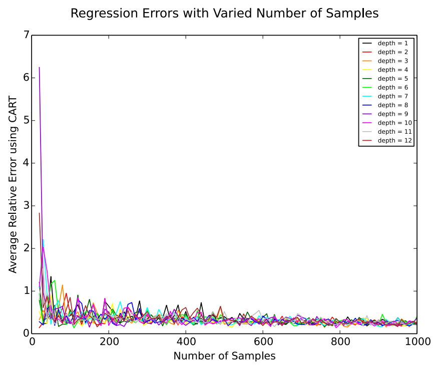
In order to replace a device model, however, what I really want to know is how accurate the class model is at predicting how the device model would predict. To



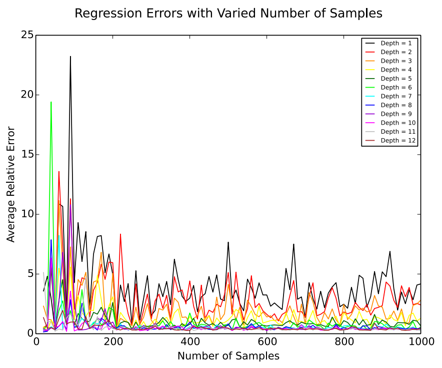
**(a)** Device C



**(b)** Class Model

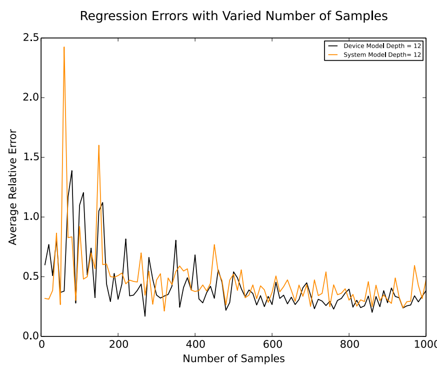


**(c)** Device D

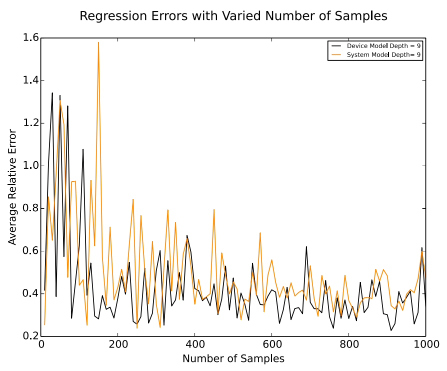


**(d)** Class Model

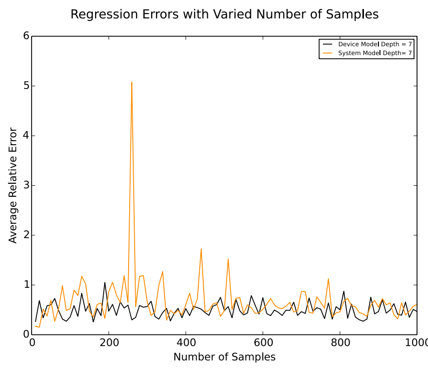
**Figure 10.3:** Direct comparisons.



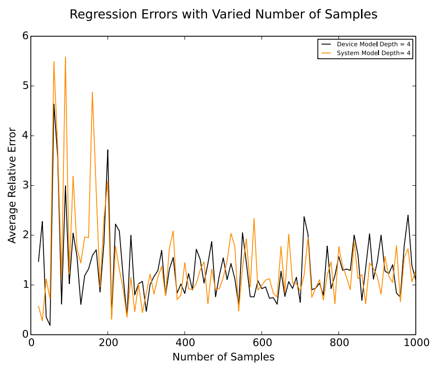
**(a)** Device A



**(b)** Device B



**(c)** Device C



**(d)** Device D

**Figure 10.4:** Best device depth.

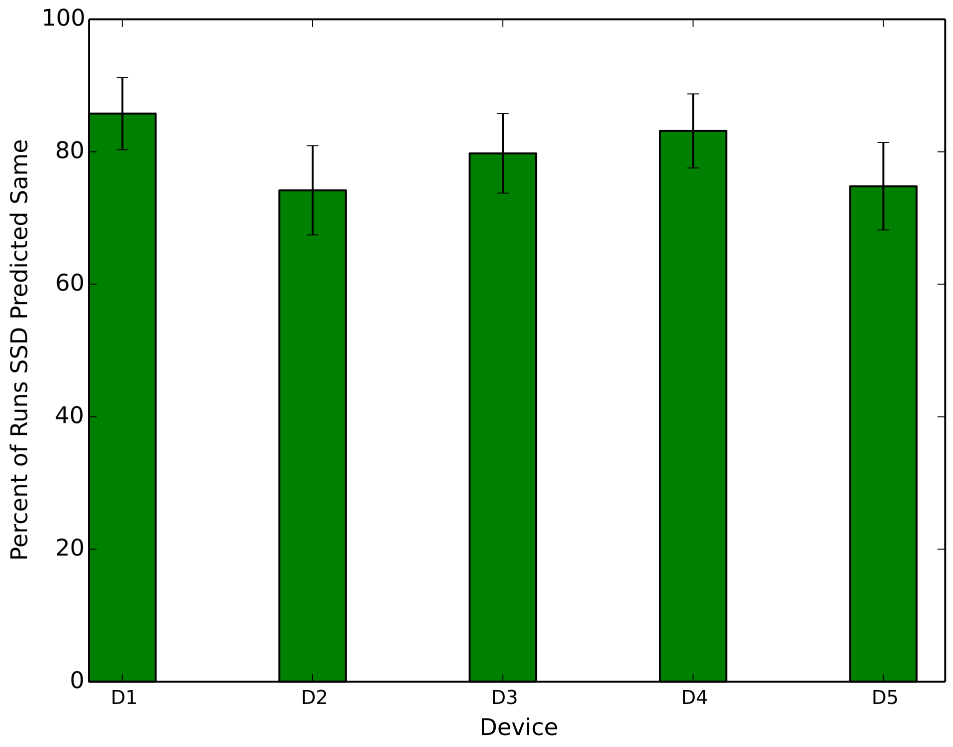


**Figure 10.5:** Class model accuracy in predicting device model.

determine this, I built a device model for each device as well as a class model and tested how the class model predicted with respect to how each device model predicted. As can be seen in Figure 10.5, the class model predicts the same as the device model between 65% and 85% of the time.

## 10.2 Representative Workloads

The other way I approached reducing the overhead caused by machine learning was to look at using a workload that was representative of all the workloads currently on a device, rather than keeping the information about all of them. Not only did this allow me to maintain a pairwise comparison, it also paves the way for collecting data directly from the device in the future.

The formula in Equation 7.1 determines the performance of a single workload, where a workload is defined by its characteristics: $wc = \langle bs, io, rw, pr \rangle$. However, it can be extended to calculate the performance of multiple workloads on a device $D$.

$$\mathrm{p}_D = F(\forall i \in D, \mathrm{wc}_i) \tag{10.1}$$

The most straightforward way is to take the characteristics of each workload on the device and use them all as inputs to the function $F$. The formula shown in Equation 10.1 defines the performance of a set of workloads $D$ running together on the same device. This, then, is the baseline I used when collecting data.

However, this requires keeping track of the specific characteristics of each workload currently running. In an effort to mimic what could be gathered from the device itself and reduce the amount of information needing to be kept, I took the characteristics of all the workloads running on a device and created a new workload representative of them. For each workload characteristic $c$, an average of the values from each workload is found, creating an averaged set of workload characteristics for the device, $\mathrm{wc}_m$.

$$\mathrm{wc}_m = \forall \mathrm{c} \in \mathrm{wc}, \frac{\sum_{i=1}^{m} \mathrm{c}_i}{m} \tag{10.2}$$

This allows me to collect information about what is currently running, without having to maintain a record of the specifics of each workload.

Using this representative summary workload rather than the individual characteristics results in Equation 10.3. This takes the characteristics of the new workload $n$ and the average characteristics of the workloads on the device $\mathrm{wc}_m$ as inputs. $\mathrm{p}_{m|n}$ is the performance of the workloads on the device if the new workload is placed there.

$$\mathrm{p}_{m|n} = F(\mathrm{wc}_n, \mathrm{wc}_m) \tag{10.3}$$

This is the equation used to collect data in order to compare against using Equation 10.1, which assumes all knowledge is known.

### 10.2.1 Experimental Setup

The setup was slightly different for these experiments than is described in Chapter 8. In the previous two sets of experiments, the focus was on whether it was possible. In these experiments, I shifted focus to see how well the two different methods did in predicting performance with respect to each other. The reason for this shift was twofold. First, the end goal is to predict performance and use it in optimally placing workloads. Second, and very related to the first point, a comparison of how accurately the performance was predicted would tell me less about whether the representative workload is viable than how accurately it can be used to predict where a new workload should be placed.

Experiments were run in sets of ten, where each instance in a set simulated a device in the system. In each set, one workload was held constant as the new workload being placed. The remaining workloads were varied to simulate a different set of workloads on each device. Every set was run twice, in order to calculate the performance using both Equation 10.1 and Equation 10.3. The workloads are the same across both equation runs; in the first run all the characteristics are used, in the second only the average characteristics are used. This allows us to see how well using an average does compared with knowing all the individual characteristics.

**Table 10.1:** Example of how IOPS is used to predict performance. The colored cells indicate the best possible measured and predicted IOPS for both the all-knowing experiment, where all workload characteristics are known and used, and the summary experiment, where an average of the workload characteristics is used.

| New Workload | Device Number | Workloads on Disk | All-Knowing | | Summary | |
|---|---|---|---|---|---|---|
| | | | Measured | Predicted | Measured | Predicted |
| 639232 | 1 | 1617429, 5566586 | 10129.0 | 12994.0 | 6407.0 | 6340.0 |
| | 2 | 1668095, 1026817 | 361.0 | 334.0 | 343.0 | 466.0 |
| | 3 | 2264879, 5840499 | 9534.0 | 16349.0 | 9394.0 | 9807.0 |
| | 4 | 2573259, 211665 | 568.0 | 714.0 | 466.0 | 298.0 |
| | 5 | 499374, 2973522 | 914.0 | 1154.0 | 830.0 | 553.0 |
| | 6 | 5169995, 1123477 | 6099.0 | 6215.0 | 3888.0 | 3508.0 |
| | 7 | 5655641, 6412728 | 20657.0 | 19300.0 | 16205.0 | 19076.0 |
| | 8 | 6208830, 4177327 | 15689.0 | 16006.0 | 10937.0 | 11125.0 |
| | 9 | 6415495, 4583967 | 18087.0 | 19976.0 | 13208.0 | 18413.0 |
| | 10 | 77175, 6343920 | 16425.0 | 18018.0 | 14964.0 | 15893.0 |

### 10.2.2 Comparison Method

Table 10.1 shows an example of one set of experiments with three workloads. The new workload to be placed, whose unique identifier is seen in Column 1, is held constant across all ten instances, and the remaining two workloads, the unique identifiers of which are in Column 3, are different on each disk. I measure the performance (in IOPS) using Equation 10.1, recorded in Column 4, and using Equation 10.3, in Column 6. I compare the measured performance of each device against the other devices in the system and identify the absolute best performing device as the device with the highest IOPS when the new workload is added, highlighted in green. In this set, Device 7 shows the best performance for both equations, though the actual IOPS measured are different.

In order to make a comparison of how well I can predict the best placement of a new workload, I created a CART model for each equation. One model, the *all-knowing* model, predicts $p_D$ where all workload characteristics were known and used, while the other model, the *summary* model, predicts $p_{m|n}$ where an average of workload characteristics on the device were used.

For each run, I used the all-knowing model to predict how well each device would perform, and then picked the best performing device based on the highest IOPS. I then used the summary model to predict how well each device would perform, and picked the best performing device from that data as well. In Table 10.1, the predicted IOPS are seen in Columns 5 and 7 for the all-knowing model and summary model respectively with the best predicted value highlighted in green. The all-knowing model predicted Device 9 to have the best performance and the summary model predicted Device 7.

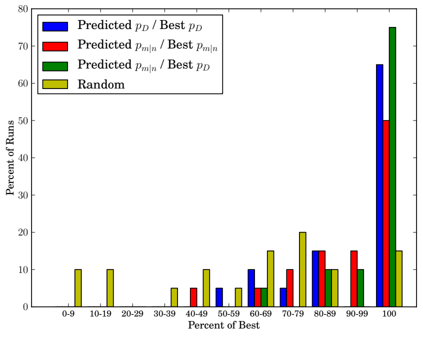
Since I care about where to place the new workload, I don't care about the actual value of IOPS beyond determining which device had the highest predicted value. Since the predictions aren't going to give a precise value, I determined the goodness of the prediction by taking the IOPS actually measured on the predicted device and expressing it as a percentage of the best IOPS actually measured, as in Equation 10.4.

$$\frac{\text{IOPS measured on predicted best device}}{\text{IOPS measured on actual best device}} \tag{10.4}$$

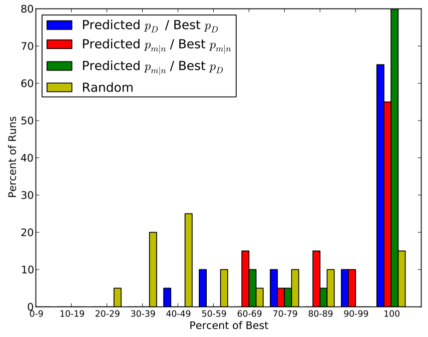
This tells me what percentage of the best performance the workload would achieve if I placed the new workload on the device predicted. If I predict the best device, the workload achieves 100% of the best performance.

In the example, to calculate how well the all-knowing prediction was, I took the all-knowing IOPS measured at Device 9 (the device I predicted would have the best performance) and divided it by the all-knowing IOPS measured at Device 7 (the device that had the actual best performance) ($\frac{18087}{20657} = 0.876$). This means that if I were to place the new workload based on the all-knowing prediction, it would achieve 87.6% of the IOPS it could have achieved. Similarly, I took the summary IOPS measured at Device 7 (the device predicted to have the best performance) and divided it by the summary IOPS measured at Device 7 (the device that had the actual best performance), giving $\frac{16205}{16205}$, or 100% of the IOPS the workload could achieve.

Finally, I used Equation 10.3 to predict the performance of Equation 10.1. To do this, I used the summary model to predict how well each device would perform, and chose the best performing device. In the example in Table 10.1, the summary model predicted Device 7 as the best. I follow the same procedure as before using Equation 10.4, except instead of finding the corresponding disk IOPS in Column 6 (the measured summary IOPS), I find it in Column 4 (the measured all-knowing IOPS). In the example, I take the IOPS from Device 7 for both the numerator and denominator ($\frac{20657}{20657}$), showing the predicted device would achieve 100% of the



(a) Three Workloads: one new, two on current device.



(b) Four Workloads: one new, three on current device.

**Figure 10.6:** Shows the percent of IOPS of the best possible placement option that the predicted value achieves, over 200 runs.

best performance.

Using the measured values from the all-knowing experiments gives me the actual performance on that device, rather than the performance of the representative workload. Predicting the performance using the representative workload allows me to see how well the summary model works to predict actual performance. This last approach gives me the most realistic view of the system, assuming I could only collect data from the device, as I would not have full knowledge.

### 10.2.3 Results

Figure 10.6a shows the results of adding a new workload to two workloads already running on device. The first bar, in blue, shows how well I can predict the all-knowing performance using the all-knowing model. The second bar, in red, shows how well I can predict the summary performance using the summary model. The fourth bar, in yellow, shows what the results would be if a random device placement were chosen each time, for comparison sake. Since there are ten disks, randomly selecting a device will result in the best placement about 10% of the time as we would expect.

The third bar, in green, shows the results of using the summary model to predict the all-knowing performance. It is interesting to note that I predict the best placement more often using this method than using the all-knowing model to predict all-knowing, or the summary model to predict the summary. Similar results are seen in Figure 10.6b, where I am adding a new workload to three workloads already running on a device.

There is more variation in the predictions in Figure 10.6b, but predicting $p_D$ using $p_{m|n}$ again outperforms the others. The cause for the decrease in accuracy can be explained by the increase in information due to an additional workload. As more workloads are added, the distance between the summary and the actual values will grow, causing the accuracy to decrease slightly. There may be a point at which the summary model is no longer effective due to the number and disparity of workloads; such a consideration is future work.

It is important to note that the graphs in Figure 10.6 were created without specifying the depth of the regression tree. When unspecified, the CART model will create a tree whose leaves each contain one and only one data point from the training data. With two hundred experiments, this created a tree of depth greater than 20. As can be seen in Table 10.2, higher accuracy can be achieved by limiting the depth of the tree. The most likely explanation for this is that it is an artifact of how the best device placement is selected.

When predicting the best placement, it is possible to end up with several devices with the same predicted IOPS. This is due to the nature of a decision tree: if two devices have workloads with similar characteristics, it is entirely possible that the predicted value will be the same, depending on the depth and complexity

**Table 10.2:** Trade-off between accuracy and precision with varying tree depths for three workloads, 200 runs.

| Tree Depth | Percent Predicted Best | Percent Predicted $>$ 75% of Best |
|---|---|---|
| 2 | 55.0 | 95.0 |
| 3 | 75.0 | 90.0 |
| 4 | 55.0 | 90.0 |
| 5 | 85.0 | 95.0 |
| 6 | 80.0 | 100.0 |
| 7 | 70.0 | 100.0 |
| 8 | 60.0 | 95.0 |
| 9 | 75.0 | 95.0 |
| 10 | 75.0 | 100.0 |
| 11 | 70.0 | 100.0 |
| 12 | 75.0 | 90.0 |

of the tree. When this happens, I randomly select one of the devices as the “best”, since I currently have no way to break such a tie. Thus, at smaller depths, the chance of a tie is much higher, resulting in a higher likelihood that I may select the best device. A more complex definition of performance may cause this to change.

# Chapter 11

# Simulation Design

I have nothing to offer but blood, toil, tears and sweat.

Winston Churchill

I designed a simulated system to test the feasibility of using the methods described in this dissertation as an online placement technique. Workloads are represented as a vector of characteristics, as seen in Chapter 7. When a new workload enters the system, it is sent to a "communications" device. An outline of what happens within the communications device is shown in Algorithm 1.

The communications device has several tasks, including performing constraint satisfaction and performance prediction. If the system is heterogenous, or if there is a service level agreement associated with the workload, the first task required by the communications device is to characterize the workload to determine which class of device is best suited to the workload's constraints and needs. This characterization identifies not only what type of device the workload should be run on, but also which class model should be used for predicting performance.

In order for the communications device to make placement decisions, it has

**Algorithm 1** Pseudocode for what happens when a new workload enters the system.

```
procedure PLACEWORKLOAD($newWorkload$)
    $type \leftarrow classify(newWorkload)$
    $currentStates \leftarrow$ current state of each device of $type$
    for all $device$ in $currentStates$ do
        if $device < deviceThreshold$ then
            if $device + newWorkload < deviceThreshold$ then
                $availableDevices \leftarrow newWorkload$
            end if
        end if
    end for
    for all $device$ in $availableDevices$ do
        $representative \leftarrow$ representative workload of $device$
        $prediction \leftarrow predict(representative, newWorkload)$
        $predictions \leftarrow [device, prediction]$
    end for
    $bestPerformance, bestDevice \leftarrow maximum(predictions)$
    while $bestPerformance > bestDeviceThreshold$ do
        $predictions.remove(bestDevice)$
        $bestPerformance, bestDevice \leftarrow maximum(predictions)$
    end while
    place $newWorkload$ on $bestDevice$
end procedure
```

the ability to ping the other devices in the system and collect the information necessary to create a representative workload for each device. In addition to the representative workload, the communications device also gets the current state of each device. Similar to workloads, devices are represented by two vectors: one representing the maximum threshold of what each device is capable of (or is reasonable for that device), and one with the current state of the device. Realistically, the maximum threshold should not be the absolute maximum of the device, to allow room for unexpected events. The current state of the device indicates how full the disk is and what the current bandwidth and latency are.

Before performance prediction happens, two constraint checks are performed. One to see if any devices are near their maximum thresholds, and the second to make sure that adding the new workload won't cause the device to exceed the maximum threshold. If either of these are true, the device is not considered to be available for the new workload. The representative workload for each available device is sent to the prediction model along with the new workload, and an optimal device is predicted. A second constraint check occurs at this point, to make sure that adding the workload's predicted performance will not exceed what the device is capable of, given the other workloads.

Once the workload has been placed, it is necessary to periodically check to make sure that the workloads are still performing together as expected, since the behavior of workloads can change over time. Rather than randomly checking, or checking too frequently, I have implemented a warning system. In order for the check to be triggered, each workload is set with a minimum allowable bandwidth and latency. When a minimum threshold is tripped, the workload that triggered it is removed from the device and treated as a completely new workload. Since this includes getting an updated current state of all the devices in the system, it also results in checking on everything else. The rebalancing of the system also gives the opportunity to reassign the triggered workload to a different device class.

# Part III

# Related Work

# Chapter 12

# Objective Optimization

The farther backward you can look,
the farther forward you can see.

Winston Churchill

The three objectives I focused on in this dissertation were load balancing, system responsiveness, and energy savings. Optimizing for load balancing and system responsiveness has been important since the early days of distributed systems [61]. Load balancing addresses the need to keep popular data evenly spread across the system, in order to not overload any device. It is closely related to system responsiveness, which is concerned with how quickly the system responds to a request. Energy savings is particularly an issue in backup systems, where disks can be spun down in order to save power and money [11, 8, 52], but is also beginning to be addressed in primary systems where spinning disks down is not always an option. However, very few optimization techniques go beyond optimizing for one objective; those that do only consider two and often in a very specific context.

## 12.1 Load Balancing

One common way to relieve the skew caused by load balancing is to replicate the popular data. A good example of this technique is CRUSH [63], which is a pseudo-random data distribution algorithm. CRUSH distributes object replicas across a storage cluster by mapping an input value to a list of devices on which to store object replicas. Distribution is pseudo-random in that there is no apparent correlation between resulting output from similar inputs or items stored on any device. Another example is MMPacking [28], a load and storage balancing method developed for distributed multimedia servers. MMPacking uses a combination of replication and a weighted scheduling algorithm in order to achieve load balancing. It produces at most $x - 1$ replicas of video streams in a system of $x$ servers, distributing the replicas among servers so that no server stores more than two video streams more than any other server.

MMPacking distributes identical size data items across multiple disks; the work done by Ma et al. [39] extends the work to apply to variable size data items. Called LSB_Placement, it is specifically designed for web applications and web servers and attempts to find a load balanced placement that also minimizes the required capacity for each disk. To achieve this, first best fit bin packing is performed, in order to place data items into approximately equal size bins. This reduces the problem such that the MMPacking method can be applied. The focus here is on finding the optimal bin capacity in order to minimize the required disk capacity.

Berenbrink et al. [6] use a parity strategy rather than a replication strategy. For blocks that are more frequently read than updated, each block is divided into $k$ equal subparts, and an extra subpart is created that is the `exclusive or` of the $k$ subparts. This allows a block to be read from any $k$ of the $k+1$ subparts, reducing the redundancy and space constraints that replication introduces. Subparts are distributed across disk arrays using a hash function and an assimilation function, to help remap the data should a disk array be added or removed. Short term load balancing is achieved using a *minimum game* strategy, where the $k$ necessary subparts are read from the $k$ least busy disk arrays.

Load balancing is also used in the file allocation problem, as a way to minimize disk utilization. One popular algorithm is the Greedy algorithm, which originated from the Longest Processing Time algorithm (LPT) [20]. The LPT algorithm is a simple greedy algorithm designed for multiprocessor load balancing and can operate on- or off-line. At each step, LPT greedily assigns a process to the processor that has the least accumulated load. When LPT is running online, processes are assigned in the order of their arrival; in offline mode, processes are ordered by their load and assignment is done in decreasing load order. When applied to the file allocation problem, the LPT algorithm becomes the Greedy algorithm [34], where the load of each file is defined as the product of the file access rate and the access service time.

## 12.2 System Responsiveness

System responsiveness refers to the amount of time it takes for the system to respond to a request, be it a query for data from the user or a simple *ls* to list the files in a directory. This is closely related to load balancing, as distributing the load often results in an increase of system responsiveness. This is not always true, however, as one of the best load balancing algorithms, the Greedy algorithm [20], results in very poor system responsiveness according to the comparisons performed by Lee et al. [34]. Likewise, the best algorithm for average response time, *Sort Partition*, does very poorly in load balancing.

Lee et al. [34] present two different algorithms, *Sort Partition* (SP) and *Hy-*

*brid Partition* (HP), and evaluate against the Greedy algorithm using average response time as a performance metric. SP tries to optimize the response time by minimizing the service time variance at each disk, but is an offline algorithm that requires complete knowledge of the files. HP is the online version of SP, and attempts to reconcile minimizing the load variance across disks and minimizing service time variance at each disk by giving priority to minimizing service time variance when overall disk utilization is low. This is because when disk utilization is low, the load imbalance does not have a significant effect on the response time.

Not content with trading optimal load balancing for response time, and vice versa, Zhu et al. [69] propose two algorithms that optimize both response time and load balancing in parallel I/O systems. The first, *Balanced Allocation with Sort* (BAS), is an offline algorithm for static file assignment and needs full knowledge of the service times and access rates for all files. The second, *Balanced Allocation with Sort for Batch* (BASB), is an online algorithm, which uses information about the coming batch of files and the previously assigned files rather than needing complete knowledge like BAS. Files in a batch are sorted in descending order of their service times, with no correlation between service times in other batches, and then assigned to disks based on the average disk load.

One of the problems with the algorithms above is that they were developed under the assumptions that file access rate obeys a Zipfian distribution and that file access frequency is inversely related to file size. These assumptions were based on early studies on web requests, but Xie et al. [67] point out that these assumptions are not strictly true, given other recent web proxy trace studies. Xie et al. propose an algorithm, called ***S**tatic R**o**und **R**obin* (SOR), which aims to minimize response time regardless of the workload assumptions. SOR does this by sorting the files in ascending size, so that files of the same size are placed near each other with large files placed separately on a dedicated disk. Each file is placed in a partial round robin manner, making sure that the load on each disk does not exceed the average disk load.

## 12.3 Energy Savings

Energy savings has been a concern with archival systems for some time: as systems add more storage capacity, the amount of power needed increases along with the cost of cooling. Since in some situations, disks and the power to cool them consume more energy than the rest of the system combined [5, 19], enabling more disks to remain idle is a big part of this research [11, 8, 52]. While high performance computing (HPC) workloads are not ideal candidates for techniques which exploit idle disks [8], there has been work which indicates that significant idle periods exist in enterprise workloads [45].

Grouping data on disk to help improve response time is a well-researched topic [1, 32, 56]. Building off this work, Essary and Amer [17] present a theoretical framework that aims to reduce energy consumption as well as improve response time. This is accomplished using a predictive grouping algorithm, called OE ME which stands for *optimized expansion, maximized expectation*. OE ME uses first order successors to build groups, combining breadth first and depth first expansion strategies to create a balanced expansion to find successors.

The work done by Wildani et al. [64] extends this by providing a realistic prediction mechanism and semi-permanent groupings, which reduces the need for constant prediction. Wildani et al. show that by grouping data likely to be accessed within a short period of time together, the number of times disks have to spin up is reduced. This is supported by other research showing that data arrangement can have an impact on energy savings in single-disk systems [15, 54, 17].

Much of this work, however, requires knowledge of access patterns and arranges data after the data has been placed. An alternative to this approach is Rabbit, a power proportional distributed file system proposed by Amur et al. [2], which uses a cluster-based storage data layout to provide ideal power-proportionality. This means that the performance-to-power ratio at all performance levels is equivalent to that at the maximum performance level. It does this through an *equal-work* data layout policy, which stores replicas of data on non-overlapping subsets of nodes. A primary replica is stored on $p$ nodes. The lowest power setting only requires those $p$ nodes be powered, in order to guarantee the availability of all the data.

Since my work is concerned with data allocation, there is no such knowledge: data that has not been stored cannot have access patterns associated with it. However, Pâris et al. [48] have shown that the access patterns over a large percent of files in most workloads is stable, so there may be some assumptions that can be made based on the characteristics of the file. The work that Wildani et al. have done could also be used in conjunction with my work, as they have shown that "working sets" (groups of data likely to be accessed together) can be identified from I/O traces gathered from real systems [65].

# Chapter 13

# Workload Characterization and Modeling

If you steal from one author it's plagiarism; if you steal from many it's research.

Wilson Mizner

Obtaining real world workloads is often prohibitively difficult in academic research, forcing many researchers to turn to synthetic workloads as a replacement. However, synthetic workloads often fail to catch nuances that exist in real world applications. In addition, current tools and techniques focus on using workloads to identify the performance of the system. Much of the work in predicting performance has been focused on modeling the devices in the system or the workloads that are being placed, rather than the interaction among them [35, 24, 23, 44, 62, 49]. My interest lies in identifying the performance of the workloads as they interact with each other.

## 13.1 Obtaining Workloads

An often overlooked key component of storage systems research is the data that is required to prove the theories and test the hypotheses. In order to make assertions about the performance and behavior of what is being proven, data is needed that is representative of what will be used in the system in the real world. Traditionally, there are two methods to obtain data: collecting real world workload traces, or creating workload data using synthetic models.

The debate between real world workloads and synthetic workloads is hardly new. While there will always be arguments for why real workloads are better for evaluating storage performance [66], there is also evidence that the choice of workload generation technique by itself does not significantly affect the performance of algorithms [37]. However, there is no real state-of-the art technique for evaluating the performance of the workloads themselves.

### 13.1.1 Real World Workloads

The term "real world workloads" refers to a workload *trace*, that is, a record of the timing of read and write requests that were issued and run on a system, along with the details of the requests. Replaying traces allows users to evaluate the performance of a system in the presence of real data, which is one reason real world workloads are difficult to come by.

Knowing how a workload behaves in a live system, even if the system is not the one that was originally traced, often results in knowing either about the information that is being read and written, or about the performance of a specific system, or sometimes both. It is easy to understand, then, why companies do not easily release this kind of information. Indeed, the most easily available block IO traces are the MSR Cambridge traces from 2007 [**?**].

Depending on the type of work being done, real workload traces are the only option. Tarasov et al. [59] built a system that converts traces into something that benchmarks can easily understand, shrinking the size of the trace in the process. Paragone [57] takes it a step further and suggests completely rethinking how traces should be modeled and replayed. It combines a number of existing data analysis techniques to preserve sequential patterns in a trace as well as IO bursts.

### 13.1.2 Synthetic Workloads

Although workload traces provide the highest level of realism, there are several drawbacks as well. It is difficult to change a single characteristic of the trace, in order to test the effect of that change. In addition, workload traces do not always translate well from one system to another. Synthetic workloads are one way to manage these drawbacks.

Synthetic workloads are generated by creating statistical models of all the characteristics of a workload trace and sampling from the models to create workload data. These models generally fall into two categories [18]: descriptive models, which describe how the workload behaves, and generative models, which attempt to describe how the workload was obtained to begin with.

The problem with synthetic workloads is that a workload trace is needed to create meaningful statistical models. While some models exist [18], often the model is tailored to something specific, such as SSD performance [36] or parallel workloads [16]. One could use "naive" models, which are simple statistical distributions, such as modeling inter-arrival times using an exponential distribution. However, the "naive" models are less realistic.

### 13.1.3 Workload Generators

Workload generators provide a way to create synthetic workloads without necessarily needing a statistical model of the workload characteristics. The Distiller [33] was designed to, using a target workload and a library of workload attributes, identify the key attributes necessary to create a synthetic workload that is representative of the target. In order for the synthetic workload to be representative of the target workload, the distribution of IO response times must be similar. Buttress [3], another workload generator, was designed to generate and replay workloads with a high level of timing accuracy. It was developed to address the issue of timing when replaying workload traces for IO benchmarking, though it can also generate workloads with microsecond accuracy.

Filebench [47] is a program for benchmarking file systems. It uses a workload model language to create "workload personalities", benchmarks which model real IO applications. Iometer [25] and fio [4] were both designed to generate workloads as well as record the performance of the workload and the impact on the system. Iometer emphasizes the generation of workloads for the purposes of stressing the system and was developed to run on Windows Server NT. With the goal of avoiding the need to write tailored programs for individual test cases, *fio* was designed to simulate workloads and is available on a variety of platforms.

Any of these workload generators could be used for workload simulation; I chose *fio* because it offered the widest range of options to generate realistic workloads and because it was designed to generate user specified workloads. In addition, *fio* can read and write directly to a disk drive without needing a file system. This enables observation of workload performance and interactions with other workloads without needing to determine how much of the performance is the workload itself and how much is outside influence from the system.

**Table 13.1:** Studies of Workloads and Their Defining Characteristics.

| | Keeton [27] | Mesnier [43] | Kavalanekar [26] | Gulati [21] |
|---|---|---|---|---|
| Read/Write Ratio | ✓ | ✓ | ✓ | |
| Request Size | ✓ | ✓ | ✓ | ✓ |
| Request Rate | ✓ | ✓ | ✓ | ✓ |
| Spatial Locality | ✓ | ✓ | ✓ | ✓ |
| Temporal Locality | ✓ | | ✓ | |
| Outstanding IOs | | ✓ | ✓ | ✓ |
| IO Latency | | ✓ | ✓ | ✓ |

### 13.1.4 Workload Characterization

In order to create realistic workloads, it is necessary to be able to characterize real world workloads. There are many different studies of workloads and how to characterize them; I have chosen a few to focus on and summarized them in Table 13.1. In the table, spatial locality refers to the sequentiality of the workload, while temporal locality refers to whether the IO is unique.

Keeton et al. [27] attempted to characterize the IO of commercial workloads with the goal of being able to describe a workload in sufficient detail to reproduce the workload on a different storage system. There are three characteristics that are not summarized in Table 13.1: burstiness, the correlation between accesses to different parts of the system, and phased behavior. They also emphasized that finding a distribution for the characteristics, rather than an average, was important.

Kavalanekar et al. [26] looked at twelve Windows production server traces and two database benchmark traces. While burstiness is not measured, they propose using "self-similarity" both in a spatial and temporal manner to calculate burstiness. They also track the number of outstanding IOs as well as the latency of IO requests. Gulati et al. [21] also includes outstanding IOs and IO latency in their list of workload characteristics. This study covers three enterprise applications using VMWare's ESX server hypervisor. A separate, detailed trace is used to observe burstiness in the workloads.

The approach of Mesnier et al. [43] is interesting. They start by generating workloads based on the read/write ratio, request size, and spatial locality. These workloads are run, and the characteristics in the table are measured, as well as the performance. In addition, the workload characterization done is for the purpose of building a relative fitness model to predict the difference in performance between two storage devices.

## 13.2 Modeling Approaches

Using machine learning enables a system to make predictions based on prior information [7]. The majority of my experiments used decision tree learning, which is a machine learning technique that uses a decision tree as a predictive model. The tree is built by successively splitting the data set into subsets based on attributes, and decisions are made by following these splits to leaf nodes.

Classification And Regression Tree (CART) analysis is a specialized version of decision tree learning that covers both classification trees, which predict the class to which the data belongs, and regression trees, which predict a real number. I chose CART analysis as the predictive method because they are human readable and provide comparable performance to similar techniques such as neural networks [53].

### 13.2.1 Device Modeling

Li and Huang [35] present a black box performance model for solid state drives (SSDs). They use a basic black box model with traditional workload characteristics, and collect training data on workload characteristics and device performance. A statistical machine learning algorithm is applied for model fitting, and performance is predicted as a function of workload characteristics. They extend this work by benchmarking an SSD and using this for training data [24].

BASIL [22] is a software system that automates virtual disk placement with a focus on IO load balancing. BASIL collects information pairs of outstanding IOs and average IO latencies observed, then uses data points of the form $\langle$OIO, Latency$\rangle$ over a period of time and computes a linear fit to minimize the least squares error. The resulting slope indicated the overall performance capability of the data store. Using the IO latency as a metric for modeling results in some amount of dependence on the underlying storage device and architecture.

PESTO [23] is a storage performance management system for virtualized data centers, providing IO load balancing. It builds off of the work done in BASIL, and extends it by including a cost-benefit analysis with the IO load balancing. Detailed statistics are collected on the way virtual disks are accessed at each host, in order to find a good placement that balances IO across available data stores. A workload injector is run during idle periods to generate performance models periodically.

All of these works end up creating a model that is dependent on the underlying storage device. The problem with this type of modeling is that when the system is modified due to upgrading or replacing devices, the model also must be rebuilt. My goal is to avoid this dependence by modeling the interaction among workloads based on the workload characteristics.

### 13.2.2 Workload Modeling

Wang et al. [62] build black box models based on CART models for storage devices. They predict device performance as a function of IO workloads. In this case, a workload is a sequence of disk requests (arrival time, LBN, request size in blocks, and read/write type). CART is used to approximate the function, working under the assumption that the model construction algorithm can feed any workload into the device to observe its behavior for a period of time for training. The main goal in this work is to accurately predict device performance, and could be used as complementary to my work.

Mesnier et al. [44] also uses a black box approach based on CART models to model the performance of storage devices, but they choose to use relative fitness. Rather than predicting the performance of a specific storage device, relative fitness predicts the differences between a pair of devices. The characteristics of a workload are measured on the device it is currently on, and then the relative fitness model is used to predict the consequences of moving the workload to a different device. This could be useful in conjunction with my work, in the case of workload migration.

Other workload models [31, 46, 9] are focused on the interference between workloads. This is important when workloads already exist on a device and the goal is to mitigate any interference that is occurring. While my work will minimize this interference by placing workloads intelligently, any of these models could be used to further eliminate any minor interference that may occur over time.

# Part IV

# Future Work and Conclusions

# Chapter 14

# Future Work

The outcome of any serious research can only be to make two questions grow where only one grew before.

Thorstein Veblen

There are many opportunities to extend the work presented in this dissertation. A highly impactful extension of this work would be to find or create realistic workloads that contain data content. This would allow the consideration of real system concerns such as replication and deduplication. It would be interesting to see if the prediction model could compensate for the idea of "the majority of the data this workload needs already exists on a certain device", or if there would need to be a weight of sorts added to reflect this information.

Another area of expansion that would be both insightful and useful is to work with this in a system at scale. Currently our simulated system only consists of ten devices. One potential issue is in the simulated system design; the communications device could very quickly become a bottleneck since all new workloads must go through it. One simple solution is to create multiple "pools" of devices, each with its own communications device. This could result in a very elegant solution: if the pools are created such that all devices in a pool have similar constraints and thresholds, entire pools could be eliminated at a constraint satisfaction check.

There are two interesting questions to explore with respect to the representative workloads. The first is concerned with a larger system, to see if there is a point at which the representative workload is no longer effective due to the number and disparity of workloads. It is conceivable that there is such a point, but finding that point has been left for future students. The second question has to do with how the representative workload is constructed. The characteristics that define a workload are values that can be either measured or calculated by collecting information from the device itself. This may also have interesting implications in defining the new workload as well, in the case where the workload characteristics are not known or not specified.

One other opportunity exists in the class models. The results presented in this dissertation were mainly collected on solid state drives (SSDs), though data from hard disk drives (HDDs) was also collected. I did not attempt to simulate a heterogenous system that has a mix of HDDs and SSDs, nor did I model Flash or shingled magnetic recording (SMR) disks. It would be interesting to see the effect of a heterogenous system with all types of storage devices.

Lastly, should all these questions be addressed[1], it would be fascinating to see this implemented in a real system to see how well it worked in a real environment. While I have pointed out areas for improvement and expansion, I would expect that there are other problems that will arise when this is no longer a simulated proof of concept. That being said, it is an excellent starting point for automatically optimizing the state of a system.

[1]If this happens (or really, if you have actually read this far) drop me an email at christina.strong@gmail.com

# Chapter 15

# Conclusions

If you have an important point to make, don't try to be subtle or clever. Use a pile driver. Hit the point once. Then come back and hit it again. Then hit it a third time – a tremendous whack.

Winston Churchill

I began by taking a top-down, theoretical look at solving automating system optimization by extending the file allocation problem to satisfy multiple objectives. The first theoretical model, a multi-objective optimization approach, was unsatisfactory in that the metric to measure load balancing was difficult to obtain with the data I had available (namely static metadata). The theoretical model based on bin packing suffered from the same difficulty. The third approach, based on queuing theory, resulted in a solid theoretical model, but became complex when attempting to implement.

Since each theoretical model had its own issues, I looked at a bottom-up approach of applying machine learning to assist in solving the file allocation problem combined with constraint satisfaction to maintain or achieve an optimal system state. Rather than attempting to predict a precise value for performance, I looked at predicting the best allocation. I introduced **Bypass**, a novel approach to workload simulation that generates realistic workloads based on real workload characteristics and values. I outlined how to map the workload characteristics to input parameters in *fio*, bypassing the system to perform IO directly on the disk and allowing me to evaluate the performance of the workloads themselves.

By turning prediction into a tool for workload allocation, I no longer needed a highly accurate value for performance. I needed an accurate prediction for a given allocation, or how the workload would behave on each device relative to the other devices. This allowed me to reduce the overhead of adding machine learning by creating a general model for each class of device, rather than requiring an individual model for each device in the system. I have shown that there a class model predicts the same as the device model between 65% and 85% of the time, and that the class model begins to mimic the device model (as measured by mean relative error) as the number of test samples increases.

Additionally, I decreased the amount of information I needed to keep track of by using a representative workload for each device. The representative workload was a summary of all the workloads currently running on a given device, which meant I didn't need to keep track of every workload in the system. I have shown that using the representative workload to create a model for predicting performance actually outperforms creating a model using information from all the workloads.

These two improvements make performance prediction a feasible option for determining an allocation. Combining performance prediction with constraint satisfaction, I have outlined a system design that achieves and maintains an optimal system state with minimal input from the system administrator.

# Bibliography

[1] Amer, A., Long, D., Pâris, J., and Burns, R. File access prediction with adjustable accuracy. *Proceedings of the International Performance Conference on Computers and Communication (IPCCC '02)* (2002).

[2] Amur, H., Cipar, J., Gupta, V., Ganger, G. R., Kozuch, M. A., and Schwan, K. Robust and flexible power-proportional storage. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (New York, NY, USA, 2010), SoCC '10, ACM, pp. 217–228.

[3] Anderson, E., Kallahalla, M., Uysal, M., and Swaminathan, R. Buttress: A toolkit for flexible and high fidelity I/O benchmarking. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies* (2004), USENIX Association, pp. 4–4.

[4] Axboe, J. Fio: Flexible IO tester. http://freecode.com/projects/fio.

[5] Barroso, L. A., and Holzle, U. The case for energy-proportional computing. *IEEE Computer 40*, December (2007), 33–37.

[6] Berenbrink, P., Brinkmann, A., and Scheideler, C. Design of the PRESTO multimedia storage network, 1999.

[7] Beygelzimer, A., Langford, J., and Zadrozny, B. Machine learning techniques – reductions between prediction quality metrics. In *Performance Modeling and Engineering*. Springer, 2008, pp. 3–28.

[8] Carrera, E. V., Pinheiro, E., and Bianchini, R. Conserving disk energy in network servers. In *Proceedings of the 17th Annual International Conference on Supercomputing* (New York, NY, USA, 2003), ICS '03, ACM, pp. 86–97.

[9] Casale, G., Kraft, S., and Krishnamurthy, D. A model of storage I/O performance interference in virtualized systems. In *Distributed Computing Systems Workshops (ICDCSW), 2011 31st International Conference on* (2011), IEEE, pp. 34–39.

[10] Censor, Y. Pareto optimality in multiobjective problems. *Applied Mathematics & Optimization 4* (1977), 41–59. 10.1007/BF01442131.

[11] Colarelli, D., and Grunwald, D. Massive arrays of idle disks for storage archives. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing* (Los Alamitos, CA, USA, 2002), Supercomputing '02, IEEE Computer Society Press, pp. 1–11.

[12] Copeland, G., Alexander, W., Boughter, E., and Keller, T. Data placement in Bubba. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1988), SIGMOD '88, ACM, pp. 99–108.

[13] Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. A fast elitist multi-objective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation 6* (2000), 182–197.

[14] Dong, B., Li, X., Xiao, L., and Ruan, L. A file assignment strategy for parallel I/O system with minimum I/O contention probability. In *Grid and Distributed Computing*, T.-h. Kim, H. Adeli, H.-s. Cho, O. Gervasi, S. S. Yau, B.-H. Kang, and J. G. Villalba, Eds., vol. 261 of *Communications in Computer and Information Science*. Springer Berlin Heidelberg, 2011, pp. 445–454.

[15] Douglis, F., Krishnan, P., and Marsh, B. Thwarting the power-hungry disk. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference* (Berkeley, CA, USA, 1994), WTEC'94, USENIX Association, pp. 23–23.

[16] Downey, A. B. A parallel workload model and its implications for processor allocation. *Cluster Computing 1*, 1 (1998), 133–145.

[17] Essary, D., and Amer, A. Predictive data grouping: Defining the bounds of energy and latency reduction through predictive data grouping and replication. *Transactions on Storage 4*, 1 (May 2008), 2:1–2:23.

[18] Feitelson, D. G. *Workload modeling for computer systems performance evaluation*. Cambridge University Press, 2015.

[19] Ganesh, L., Weatherspoon, H., Balakrishnan, M., and Birman, K. Optimizing power consumption in large scale storage systems. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems* (Berkeley, CA, USA, 2007), HOTOS'07, USENIX Association, pp. 9:1–9:6.

[20] Graham, R. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics 17*, 2 (1969), 416–429.

[21] Gulati, A., Kumar, C., and Ahmad, I. Storage workload characterization and consolidation in virtualized environments. In *Workshop on Virtualization Performance: Analysis, Characterization, and Tools (VPACT)* (2009).

[22] Gulati, A., Kumar, C., Ahmad, I., and Kumar, K. Basil: Automated IO load balancing across storage devices. In *FAST* (2010).

[23] Gulati, A., Shanmuganathan, G., Ahmad, I., Waldspurger, C., and Uysal, M. Pesto: online storage performance management in virtualized datacenters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (2011), ACM, p. 19.

[24] Huang, H. H., Li, S., Szalay, A., and Terzis, A. Performance modeling and analysis of flash-based storage devices. In *Mass Storage Systems and Technologies (MSST), 2011 IEEE 27th Symposium on* (2011), IEEE, p. 111.

[25] Intel. Iometer: The I/O Performance Analysis Tool for Servers. http://www.iometer.org/. Intel Open Source License.

[26] Kavalanekar, S., Worthington, B., Zhang, Q., and Sharda, V. Characterization of storage workload traces from production windows servers. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on* (2008), IEEE, pp. 119–128.

[27] Keeton, K., Veitch, A., Obal, D., and Wilkes, J. I/O characterization of commercial workloads. In *Proc. Third Workshop on Computer Architecture Evaluation Using Commerical Workloads (CAECW-00)* (2000).

[28] Kenyon, C. Best-fit bin-packing with random order. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadel-

phia, PA, USA, 1996), SODA '96, Society for Industrial and Applied Mathematics, pp. 359–364.

[29] Kleinrock, L. *Theory, Volume 1, Queueing Systems.* Wiley-Interscience, 1975.

[30] Konak, A., Coit, D. W., and Smith, A. E. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering & System Safety 91*, 9 (September 2006), 992–1007.

[31] Kraft, S., Casale, G., Krishnamurthy, D., Greer, D., and Kilpatrick, P. Performance models of storage contention in cloud environments. *Software & Systems Modeling 12*, 4 (2013), 681–704.

[32] Kroeger, T., and Long, D. D. E. Design and implementation of a predictive file prefetching algorithm. *Proceedings of the 2001 Annual USENIX Technical Conference* (January 2001), 105118.

[33] Kurmas, Z., Keeton, K., and Mackenzie, K. Synthesizing representative I/O workloads using iterative distillation. In *Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003. 11th IEEE/ACM International Symposium on* (2003), IEEE, pp. 6–15.

[34] Lee, L.-W., Scheuermann, P., and Vingralek, R. File assignment in parallel I/O systems with minimal variance of service time. *IEEE Transactions on Computers 49*, 2 (February 2000), 127–140.

[35] Li, S., and Huang, H. H. Black-box performance modeling for solid-state drives. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on* (2010), IEEE, pp. 391–393.

[36] Lingenfelter, D. J., Khurshudov, A., and Vlassarev, D. M. Efficient disk drive performance model for realistic workloads. *Magnetics, IEEE Transactions on 50*, 5 (2014), 1–9.

[37] Lo, V., Mache, J., and Windisch, K. A comparative study of real workload traces and synthetic workload models for parallel job scheduling. In *Job Scheduling Strategies for Parallel Processing* (1998), Springer, pp. 25–46.

[38] Luc, D. T. Pareto Optimality. In *Pareto Optimality, Game Theory and Equilibria*. Springer New York, 2008, pp. 481–515.

[39] Ma, Y.-C., Chiu, J.-C., Chen, T.-F., and Chung, C.-P. Variable-size data item placement for load and storage balancing. *Journal of Systems and Software 66*, 2 (May 2003), 157–166.

[40] Marler, R., and Arora, J. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization 26*, 6 (Apr. 2004), 369–395.

[41] Megiddo, N., and Modha, D. S. ARC: A self-tuning, low overhead replacement cache. *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)* (2003).

[42] Megiddo, N., and Modha, D. S. One Up on LRU. *login–The Magazine of the USENIX Association 28*, 4 (2003).

[43] Mesnier, M. P., Wachs, M., Sambasivan, R. R., Zheng, A. X., and Ganger, G. R. Modeling the relative fitness of storage. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and*

*Modeling of Computer Systems* (New York, NY, USA, 2007), SIGMETRICS 07, ACM.

[44] Mesnier, M. P., Wachs, M., Sambasivan, R. R., Zheng, A. X., and Ganger, G. R. Modeling the relative fitness of storage. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2007), SIGMETRICS 07, ACM, p. 3748.

[45] Narayanan, D., Donnelly, A., and Rowstron, A. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS) 4*, 3 (November 2008), 10:1–10:23.

[46] Noorshams, Q., Bruhn, D., Kounev, S., and Reussner, R. Predictive performance modeling of virtualized storage systems using optimized statistical regression techniques. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering* (2013), ACM, pp. 283–294.

[47] OpenSolaris. Filebench. http://filebench.sourceforge.net/.

[48] Pâris, J.-F., Amer, A., and Long, D. D. E. A stochastic approach to file access prediction. In *Proceedings of the International Workshop on Storage Network Architecture and Parallel I/Os* (New York, NY, USA, 2003), SNAPI '03, ACM, pp. 36–40.

[49] Park, N. *Statistical characterization of storage system workloads for data deduplication and load placement in heterogeneous storage environments.* PhD thesis, University of Minnesota, 2013.

[50] Parker-Wood, A., Strong, C., Miller, E., and Long, D. Security aware partitioning for efficient file system search. In *2010 IEEE 26th*

*Symposium on Mass Storage Systems and Technologies (MSST)* (May 2010), pp. 1–14.

[51] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[52] Pinheiro, E., and Bianchini, R. Energy conservation techniques for disk array-based servers. In *Proceedings of the 18th Annual International Conference on Supercomputing* (New York, NY, USA, 2004), ICS '04, ACM, pp. 68–78.

[53] Razi, M. A., and Athappilly, K. A comparative predictive analysis of neural networks (NNs), nonlinear regression and classification and regression tree (CART) models. *Expert Systems with Applications 29*, 1 (2005), 65–74.

[54] Rybczynski, J. P., Long, D. D. E., and Amer, A. Adapting predictions and workloads for power management. In *Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation* (Washington, DC, USA, 2006), MASCOTS '06, IEEE Computer Society, pp. 3–12.

[55] Scheuermann, P., Weikum, G., and Zabback, P. Data partitioning and load balancing in parallel disk systems. *The VLDB Journal — The International Journal on Very Large Data Bases 7*, 1 (February 1998), 48–66.

[56] Seltzer, M., Bostic, K., Mckusick, M. K., and Staelin, C. An implementation of a log-structured file system for unix. In *Proceedings of the USENIX Winter 1993 Conference* (Berkeley, CA, USA, 1993), USENIX'93, USENIX Association, pp. 3–3.

[57] Talwadker, R., and Voruganti, K. Paragone: What's next in block I/O trace modeling. In *IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)* (2013), IEEE, pp. 1–5.

[58] Tamaki, H., Kita, H., and Kobayashi, S. Multi-objective optimization by genetic algorithms: A review. In *Proceedings of IEEE International Conference on Evolutionary Computation* (1996).

[59] Tarasov, V., Kumar, S., Ma, J., Hildebrand, D., Povzner, A., Kuenning, G., and Zadok, E. Extracting flexible, replayable models from large block traces. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST '12)* (2012), vol. 12, p. 22.

[60] Traeger, A., Zadok, E., Joukov, N., and Wright, C. P. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage (TOS) 4*, 2 (2008), 5.

[61] Wah, B. File placement on distributed computer systems. *Computer 17* (1984), 23–32.

[62] Wang, M., Au, K., Ailamaki, A., Brockwell, A., Faloutsos, C., and Ganger, G. R. Storage device performance prediction with CART models. In *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.(MASCOTS 2004). Proceedings. The IEEE Com-*

*puter Societys 12th Annual International Symposium on* (2004), IEEE, p. 588595.

[63] Weil, S. A., Brandt, S. A., Miller, E. L., and Maltzahn, C. Crush: controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 2006), SC '06, ACM.

[64] Wildani, A., and Miller, E. Semantic data placement for power management in archival storage. In *Proceedings of the 5th Annual Petascale Data Storage Workshop (PDSW10)* (November 2010), pp. 1–5.

[65] Wildani, A., Miller, E. L., and Ward, L. Efficiently identifying working sets in block I/O streams. In *Proceedings of the 4th Annual International Conference on Systems and Storage (SYSTOR '11)* (New York, New York, USA, 2011), ACM Press.

[66] Wires, J., Ingram, S., Drudi, Z., Harvey, N. J., and Warfield, A. Characterizing storage workloads with counter stacks. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), pp. 335–349.

[67] Xie, T., and Sun, Y. A file assignment strategy independent of workload characteristic assumptions. *ACM Transactions on Storage (TOS) 5*, 3 (November 2009), 10:1–10:24.

[68] Yoo, S., and Harman, M. Pareto efficient multi-objective test case selection. In *Proceedings of the 2007 international symposium on Software testing and analysis* (New York, NY, USA, 2007), ISSTA '07, ACM, pp. 140–150.

[69] Zhu, Y., Yu, Y., Wang, W. Y., Tan, S. S., and Low, T. C. A balanced allocation strategy for file assignment in parallel I/O systems. In *Proceedings of the 2010 IEEE Fifth International Conference on Networking, Architecture, and Storage* (Washington, DC, USA, 2010), NAS '10, IEEE Computer Society, pp. 257–266.