Mirador: An Active Control Plane for Datacenter Storage

Jake Wires and Andrew Warfield, *Coho Data*

https://www.usenix.org/conference/fast17/technical-sessions/presentation/wires

This paper is included in the Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17).

February 27–March 2, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-36-2

Open access to the Proceedings of the 15th USENIX Conference on File and Storage Technologies is sponsored by USENIX.

Mirador: An Active Control Plane for Datacenter Storage

Jake Wires and Andrew Warfield
Coho Data

Abstract

This paper describes *Mirador*, a dynamic placement service implemented as part of an enterprise scale-out storage product. Mirador is able to encode multi-dimensional placement goals relating to the performance, failure response, and workload adaptation of the storage system. Using approaches from dynamic constraint satisfaction, Mirador migrates both data and client network connections in order to continuously adapt and improve the configuration of the storage system.

1 Introduction

In becoming an active resource within the datacenter, storage is now similar to the compute and network resources to which it attaches. For those resources, recent years have seen a reorganization of software stacks to cleanly disentangle the notions of control and data paths. This thrust toward "software defined" systems aims for designs in which virtualized resources may be provisioned on demand and in which central control logic allows the programmatic management of resource placement in support of scale, efficiency, and performance.

This paper observes that modern storage systems both warrant and demand exactly this approach to design. The emergence of high-performance rack-scale hardware [10, 17, 40] is amplifying the importance of *connectivity* between application workloads and their data as a critical aspect of efficient datacenter design. Fortunately, the resource programmability introduced by software defined networks and the low cost of data migration on non-volatile memory means that the dynamic reconfiguration of a storage system is achievable.

How is dynamic placement useful in the context of storage? First, consider that network topology has become a very significant factor in distributed storage designs. Driven by the fact that intra-rack bandwidth continues to outpace east/west links and that storage device latencies are approaching that of Ethernet round-trip times, efficient storage placement should ensure that data is placed in the same rack as the workloads that access it, and that network load is actively balanced across physical links.

A separate goal of distributing replicas across isolated failure domains requires a similar understanding of physical and network topology, but may act in opposition to the goal of performance and efficiency mentioned above. While placement goals such as these examples can be motivated and described in relatively simple terms, the resulting placement problem is multi-dimensional and continuously changing, and so very challenging to solve.

Mirador is a dynamic storage placement service that addresses exactly this problem. Built as a component within a scale-out enterprise storage product [12], Mirador's role is to translate configuration *intention* as specified by a set of *objective functions* into appropriate placement decisions that continuously optimize for performance, efficiency, and safety. The broader storage system that Mirador controls is capable of dynamically migrating both the placement of individual chunks of data and the client network connections that are used to access them. Mirador borrows techniques from dynamic constraint satisfaction to allow multi-dimensional goals to be expressed and satisfied dynamically in response to evolutions in environment, scale, and workloads.

This paper describes our experience in designing and building Mirador, which is the second full version of a placement service we have built. Our contributions are threefold: We demonstrate that robust placement policies can be defined as simple declarative objective functions and that general-purpose solvers can be used to find solutions that apply these constraints to both network traffic and data placement in a production storage system, advancing the application of optimization techniques to the storage configuration problem [1, 6–8, 49]. We show that for performance-dense storage clusters, placement decisions informed by the relative capabilities of net-

work and storage tiers can yield improvements over more static layouts originally developed for large collections of disks. And finally, we investigate techniques for exploiting longitudinal workload profiling to craft custom placement policies that lead to additional improvements in performance and cost-efficiency.

2 A Control Plane for Datacenter Storage

Mirador implements the control plane of a scale-out enterprise storage system which presents network-attached block devices for use by virtual machines (VMs), much like Amazon's Elastic Block Store [11]. A typical deployment consists of one or more independent storage nodes populated with performance-dense NVMe devices, each capable of sustaining random-access throughputs of hundreds of thousands of IOPS. In order to capitalize on the low latency of these devices, storage nodes are commonly embedded horizontally throughout the datacenter alongside the compute nodes they serve. In this environment, Mirador's role is to provide a centralized placement service that continuously monitors the storage system and coordinates the migration of both data and network connections in response to workload and environmental changes.

A guiding design principle of Mirador is that placement decisions should be ***dynamic*** and ***flexible***.

Dynamic placement decisions allow the system to adapt to environmental change. We regularly observe deployments of hundreds to thousands of VMs where only a small number of workloads dominate resource consumption across the cluster at any given time. Moreover, the membership of this set often changes as VMs are created and deleted or they transition through different workload phases. For these reasons, the initial choices made when placing data in the cluster may not always be the best ones; significant improvements can often be had by periodically re-evaluating placement decisions over time in response to changes in workload behavior.

Flexible placement decisions allow the system to articulate complex and multidimensional policy. Rather than trying to combine diverse and often conflicting goals in a single monolithic description, Mirador approaches system configuration as a search problem. Policies are composed of one or more *objective functions*, simple rules that express how resources should be allocated by computing numerical costs for specific configurations. A planning engine employs established constraint satisfaction techniques to efficiently search the configuration space for a minimal-cost solution.

In our experience, policies expressed as simple independent rules are substantially more perspicuous and robust than their monolithic alternatives. For example, after upgrading the customized planning engine that shipped in an early version of the product to a generic constraint solver, we were able to replace a load balancing policy originally defined in 2,000 lines of imperative Python with a similar policy composed of seven simple rules each expressed in less than thirty lines of code (see § 3.2.1 for examples). Much of the complexity of the original policy came from describing *how* it should be realized rather than *what* it intended to achieve. By disentangling these two questions and answering the former with a generic search algorithm, we arrived at a policy description that is equally efficient as the first version, yet much easier to reason about and maintain.

Mirador implements the configuration changes recommended by the planning engine by coordinating a cluster-wide schedule of data and network migration tasks, taking care to minimize the performance impact on client workloads. It communicates directly with switches and storage nodes to effect these migrations, continually monitoring system performance as it does so. In this way it actively responds to environmental and workload changes and results in a more responsive, robust system.

3 Mirador

Mirador is a highly-available data placement service that is part of a commercial scale-out storage product. Figure 1 presents a typical cluster composed of multiple storage *nodes*. Each node is a regular server populated with one or more directly-attached, non-volatile storage *devices*. Nodes implement an object interface on top of these devices and manage virtual to physical address translations internally. Objects present sparse 63-bit address spaces and are the primary unit of placement. A virtual block device interface is presented to clients. Virtual devices may be composed of one or more objects distributed across multiple nodes; by default, they are striped across 16 objects, resulting in typical object sizes on the order of tens to hundreds of GiB.

The storage cluster is fronted by a set of Software Defined Network (SDN) switches that export the cluster over a single virtual IP address. Clients connect to the virtual IP and are directed to storage nodes by a custom SDN controller. Nodes are connected in a mesh topology, and any node is capable of servicing requests from any client, allowing the mapping between clients and nodes to be modified arbitrarily.

One or more nodes in the cluster participate as a Mirador service provider. Service providers work together to monitor the state of the cluster and initiate *rebalance jobs* in response to topology and load changes. Rebalance jobs are structured as a control pipeline that generates and executes plans for dynamically reconfiguring

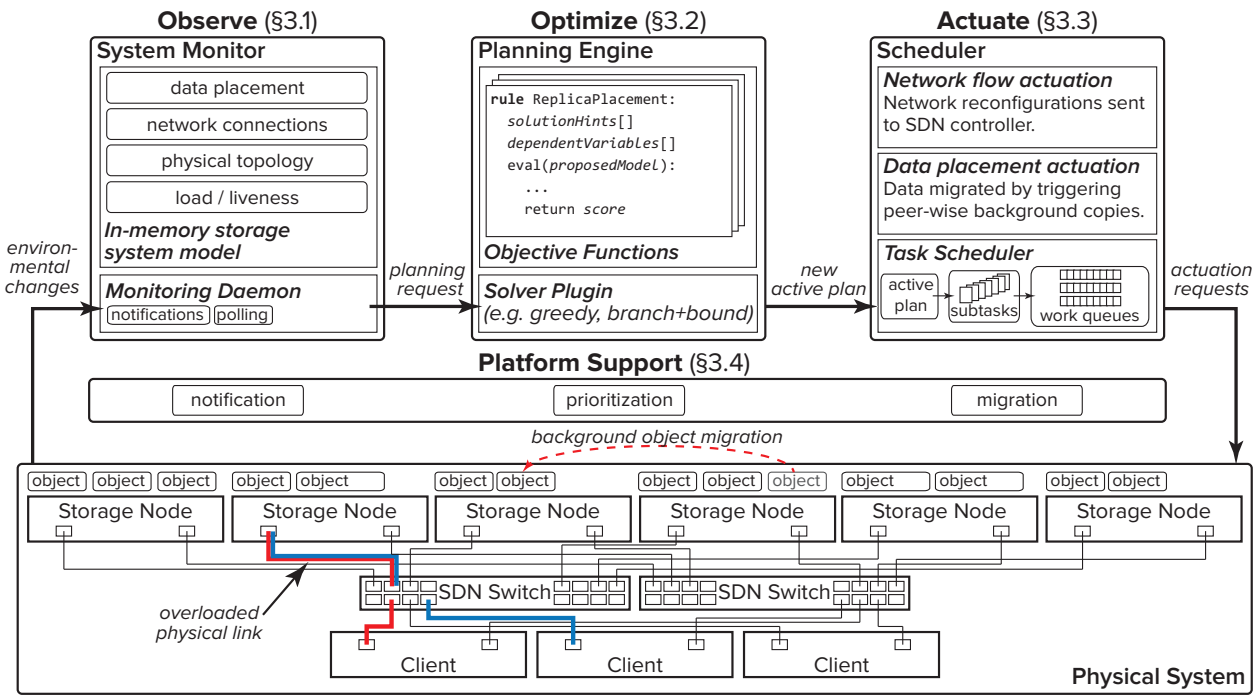


Figure 1: The storage system architecture (below) and the Mirador rebalance pipeline (above). The figure shows two examples of the system performing actuations in response to observed state. First, the fourth storage node has become disproportionately full relative to the other nodes. To balance capacity in the system, the rightmost object on that node is undergoing background migration to the third node. Second, the physical network link into the left side port of the second storage node has come under pressure from two high-volume flows from the first two clients. The system will observe this overload, and then chose one of the flows to migrate to a different physical link.

the placement of data and client connections in order to optimize for performance, efficiency, and safety. Job state is periodically checkpointed in a replicated state machine [28], providing strong resliency against failures.

The rebalance pipeline is composed of three stages:

Observation A *system monitor* collects resource metrics like device and network load along with detailed workload profiles to construct a model of the cluster.

Optimization A *planning engine* computes a numerical cost for the current configuration and searches for alternative configurations that would reduce or eliminate this cost. If a lower-cost arrangement is identified, a plan is constructed that yields the desired results.

Actuation A *scheduler* implements the plan by coordinating the migration of data and client connections.

3.1 Observation

The system monitor maintains a *storage system model* that captures all relevant properties of the physical system, including static features like cluster topology (e.g., the number of devices and nodes, the capacity of their network links, and user-defined failure domains) and dynamic features like the current free space and IO load of devices and the utilization of network ports.

The monitor also collects highly-compressed sketches of individual workload behavior [55]. These summaries are collected by a dedicated *workload analysis* service, and they include features such as *miss ratio curves* and *windowed footprints*. Unlike hardware utilization levels, this data cannot be computed from instantaneous measurements, but instead requires detailed profiling of workloads over extended periods of time.

The monitor synchronizes the model by polling the cluster; sampling frequencies vary from every few seconds for metrics like link load to tens of minutes for workload footprint measurements, while exceptional events such as device failures are signalled via special alerts.

3.2 Optimization

The planning engine implements the logic responsible for generating rebalance plans. Placement logic is encapsulated in one or more *objective functions* that specify

rules for how data and flows should be distributed across the cluster. The engine invokes a *solver* to search for new configurations that reduce placement costs, as defined by the objective functions.

The planning engine manipulates a copy of the storage model when considering alternative configurations. For example, if a decision is made to move an object from one device to another, the modelled free space and load of each device is adjusted to reflect the change.

Modelling data migration within the cluster is a challenging problem. While an object's size serves as a rough approximation of the cost of migrating it, the actual time required to move the data depends on many things, including the type and load of the source and destination devices, network contention along the migration path, and fragmentation of the data being migrated. This is important, however, because system resources like free space and bandwidth may be consumed at both the source and destination devices during migration, and the solver may make poor decisions if this usage is modelled incorrectly. For this reason, migrations initiated during the optimization stage are modelled conservatively by *reserving* space on the destination device at the beginning of operation and only releasing it from the source device once the migration has completed.

3.2.1 Objective Functions

Data placement is expressed as an optimization problem by representing objects and flows as variables and devices and links as the values these variables can take, respectively. Within this framework, objective functions model the cost (or benefit) of assigning a value to a given variable (e.g., placing a replica on a specific device). [1]

Mirador objective functions can assign arbitrary numerical costs to a given configuration. Hard constraints, implemented by rules imposing an infinite cost, can never be violated – any configuration with an infinite cost is rejected outright. Negative costs can also be used to express *affinities* for preferred assignments. An *optimal* configuration is one that minimizes the cumulative cost of all assignments; solvers employ various search strategies to find minimal-cost solutions. In the case that no finite-cost configuration can be found (e.g., due to catastrophic hardware failure), Mirador raises an alert that manual intervention is required.

Objective functions are expressed as simple Python functions operating on the storage system model described above. Listing 1 shows a rule designed to minimize load imbalances by stipulating that the spread between the most- and least-loaded devices falls within a given range. (Note that this formulation codifies a system-level notion of balance by assigning costs to *all* objects located on overloaded devices; moving just one such object to a different device may be enough to eliminate the cost for all the remaining objects.) During the optimization stage, the planning engine converts the storage model into an abstract representation of variables, values, and objectives, and computes the cost of each assignment by invoking its associated rules (see § 3.2.2).

A special annotation specifies the *scope* of the rule, indicating which components it affects (e.g., objects, connections, devices, links). Solvers refer to these annotations when determining which rules need to be re-evaluated during configuration changes. For example, the `load_balanced` rule affects `devices`, and must be invoked whenever the contents of a device changes.

Mutual objectives can be defined over multiple related objects. For instance, Listing 2 gives the implementation of a rule stipulating that no two objects in a replica set reside on the same device; it could easily be extended to include broader knowledge of rack and warehouse topology as well. Whenever a solver assigns a new value to a variable affected by a mutual objective, it must also re-evaluate all related variables (e.g., all other replicas in the replica set), as their costs may have changed as a consequence of the reassignment.

Rules can provide hints to the solver to help prune the search space. Rule implementations accept a *domain* argument, which gives a dictionary of the values that can be assigned to the variable under consideration, and is initially empty. Rules are free to update this dictionary with the expected cost that would be incurred by assigning a particular value. For example, the rule in Listing 2 populates a given replica's domain with the precomputed cost of moving it onto *any* device already hosting one of its copies, thereby deprioritizing these devices during the search. The intuition behind this optimization is that most rules in the system only affect a small subset of the possible values a variable can take, and consequently, a handful of carefully chosen hints can efficiently prune a large portion of the solution space.

A policy consists of one or more rules, which can be restricted to specific hardware components or object groups in support of multi-tenant deployments.

3.2.2 Solvers

The planning engine is written in a modular way, making it easy to implement multiple solvers with different search strategies. Solvers accept three arguments: a dictionary of *assignments* mapping variables to their current values, a dictionary of *domains* mapping variables to all possible values they can take, and a dictionary of

[1] For clarity of exposition, we use the terms *objective function* and *rule* interchangably throughout the paper.

```
@rule(model.Device)
def load_balanced(fs, device, domain):
   cost, penalty = 0, DEVICE_BALANCED_COST
   # compute load of current device
   # for the current sample interval
   load = device.load()
   # compute load of least-loaded device
   minload = fs.mindevice().load()
   if load - minload > LOAD_SPREAD:
      # if the difference is too large,
      # the current device is overloaded
      cost = penalty
   return cost
```

Listing 1: Load Balancing Rule

```
@rule(model.ReplicaSet)
def rplset_devices_unique(fs, replica, domain):
   cost, penalty = 0, INFINITY
   for rpl in replica.rplset:
      if rpl is replica:
         # skip current replica
         continue
      if rpl.device is replica.device:
         # two replicas on the same device
         # violate redundancy constraint
         cost = penalty
      # provide a hint to the solver that the
      # devices already hosting this replica set
      # are poor candidates for this replica.
      domain[rpl.device] += penalty
   return cost
```

Listing 2: Hardware Redundancy Rule

objectives mapping variables to the rules they must satisfy. Newly-added variables may have no assignment to start with, indicating that they have not yet been placed in the system. Solvers generate a sequence of *solutions*, dictionaries mapping variables to their new values. The planning engine iterates through this sequence of solutions until it finds one with an acceptable cost, or no more solutions can be found.

Mirador provides a pluggable solver interface that abstracts all knowledge of the storage model described abover. Solvers implement generic search algorithms and are free to employ standard optimization techniques like forward checking [24] and constraint propagation [36] to improve performance and solution quality.

We initially experimented with a branch and bound solver [44] because at first glance it fits well with our typical use case of soft constraints in a dense solution space [19]. A key challenge to using backtracking algorithms for data placement, however, is that these algorithms frequently yield solutions that are very different from their initial assignments. Because reassigning variables in this context may imply migrating a large amount of data from one device to another, this property can be quite onerous in practice. One way to address this is to add a rule whose cost is proportional to the difference between the solution and its initial assignment (as measured, for example, by its Hamming distance) [25]. However, this technique precludes zero-cost reconfigurations (since every reassignment incurs a cost) and thus requires careful tuning when determining whether a solution with an acceptable cost has been found.

We eventually adopted a simpler greedy algorithm. While it is not guaranteed to identify optimal solutions in every case, we find in practice that it yields quality solutions with fewer reassignments and a much more predictable run time. In fact, the greedy algorithm has been shown to be a 2-approximate solution for the related makespan problem [22], and it is a natural fit for load rebalancing as well [3].

Listing 3 presents a simplified implementation of the greedy solver. It maintains a priority queue of variables that are currently violating rules, ordered by the cost of the violations, and a priority-ordered domain for each variable specifying the possible values it can take. A pluggable module updates domain priorities in response to variable reassignments, making it possible to model capacity and load changes as the solver permutes the system searching for a solution. The current implementation prioritizes values according to various utilization metrics, including free space and load.

As described in § 3.2.1, objective functions can provide hints to the solver about potential assignments. The greedy algorithm uses these hints to augment the priority order defined by the storage system model, so that values that would violate rules are deprioritized. The search is performed in a single pass over all variables, starting with the highest-cost variables. First the rules for the variable are invoked to determine whether any values in its domain violate the prescribed placement objectives (or alternatively, satisfy placement affinities). If the rules identify a zero or negative-cost assignment, this is chosen. Otherwise, the highest-priority unconstrained value is selected from the variable's domain. The search yields its solution once all violations have been resolved or all variables have been evaluated.

Besides its predictable run time, the greedy algorithm generally yields low migration overheads, since only variables that are violating rules are considered for reassignment. However, if the initial assignments are poor, the algorithm can get trapped in local minima and fail to find a zero-cost solution. In this case, a second pass clears the assignment of a group of the costliest variables collectively, providing more freedom for the solver, but potentially incurring higher migration costs. We find that this second pass is rarely necessary given the typically under-constrained policies we use in production and is limited almost exclusively to unit tests that intentionally stress the planning engine (see § 5 for more details).

```
def greedy(assignments, domains, objectives):
  # rank variables according to cost
  queue = PriorityQueue(domains)

  while queue.cost() > 0:
    # select the highest-cost variable
    val = None
    var = queue.pop()
    cur = assignments.get(var)
    domain = domains[var]

    # retrieve the variable's current cost
    # and any domain hints provided by the
    # rules
    cost, hints = score(var, cur, objectives)
    if cost <=0:
      # current assignment is good
      continue

    if hints:
      # find the lowest-cost hint. NB: we
      # assume that typically, most values
      # are unconstrained, so this linear
      # scan adds a small constant overhead.
      try:
        val = min(
          v for v in hints
          if v in domain and v != cur
        )
      except ValueError:
        pass

    if val is None or hints[val] > 0:
      # if we have no hints, or the best
      # hints are costly, choose the lowest-
      # cost unconstrained value in the domain
      val = next(
        (
          v for v in domain
          if v not in hints and v != cur
        ),
        val
      )

    if val is None:
      # couldn't find a value
      c = infinity
    else:
      # compute cost of new value
      c, _ = score(var, val, objectives)

    if c >= cost:
      # no benefit to re-assigning
      continue

    # found a better assignment
    assignments[var] = val

    # recompute the cost of any mutually-
    # constrained variables that haven't
    # already been evaluated
    for v in rulemap(var, objectives):
      if v in queue:
        queue.reschedule(v)

  # we've arrived at a solution
  return assignments
```

Listing 3: Greedy Solver

3.3 Actuation

Mirador can migrate both data and client connections. The scheduler models the cost of data migration conservatively, and attempts to minimize the impact of such migrations on client performance whenever possible. Connection migrations are generally cheaper to perform and as such occur much more frequently – on the order of minutes rather than hours.

Optimally scheduling data migration tasks is NP-hard [31–33]; Mirador implements a simple global scheduler that parallelizes migrations as much as possible without overloading individual devices or links.

Data migrations are performed in two steps: first, a background task copies an object to the destination device, and then, only after the object is fully replicated at the destination, it is removed from the source. This ensures that the durability of the object is never compromised during migration. Client connections are migrated using standard SDN routing APIs augmented by custom protocol handlers that facilitate session state handover.

3.4 Platform Support

Mirador executes rebalance jobs in batches by (1) selecting a group of objects and/or client connections to inspect, (2) invoking the planning engine to search for alternative configurations for these entities, and (3) coordinating the migration tasks required to achieve the new layout. Batches can overlap, allowing parallelism across the three stages. Mirador attempts to prioritize the worst offenders in early batches in order to minimize actuation costs, but it guarantees that every object is processed at least once during every job.

Mirador is able to perform its job efficiently thanks to three unique features provided by the storage platform. First, the system monitor relies on a *notification* facility provided by the cluster metadata service to quickly identify objects that have been recently created or modified. This allows nodes in the cluster to make quick, conservative placement decisions on the data path while making it easy for Mirador to inspect and modify these decisions in a timely manner, providing a strong decoupling of data and control paths. Second, the planning engine makes use of a *prioritization* interface implemented at each node that accepts a metric identifier as an argument (e.g., network or disk throughput, storage IOPS or capacity) and returns a list of the busiest workloads currently being serviced by the node. Mirador can use this to inspect problematic offenders first when attempting to minimize specific objective functions (such as load balancing and capacity constraints) rather than inspecting objects in arbitrary order. Finally, the actuation scheduler implements plans with the help of a *migration* rou-

tine that performs optimized background copies of objects across nodes and supports online reconfiguration of object metadata. This interface also provides hooks to the network controller to migrate connections and session state across nodes.

4 Evaluation

In this section we explore both the expressive power of Mirador policies and the impact such policies can have on real storage workloads. Table 1 lists the rules featured in this section; some have been used in production deployments for over a year, while others are presented to demonstrate the breadth and variety of placement strategies enabled by Mirador.

§ 4.1 measures the performance and scalability of the planning engine, independent of storage hardware. § 4.2 shows how Mirador performs in representative enterprise configurations; storage nodes in this section are equipped with 12 1 TB SSDs, two 10 gigabit Ethernet ports, 64 GiB of RAM, and 2 Xeon E5-2620 processors at 2 GHz with 6 cores each and hyperthreading enabled. § 4.3 and § 4.4 highlight the flexibility of rule-based policies, as measured on a smaller development cluster where 2 800 GB Intel 910 PCIe flash cards replace the 12 SSDs on each node.

Client workloads run in virtual machines hosted on four Dell PowerEdge r420 boxes running VMware ESXi 6.0, each with two 10 gigabit Ethernet ports, 64 GiB of RAM, and 2 Xeon ES-2470 processors at 2.3 GHz with 8 cores and hyperthreading enabled. Clients connect to storage nodes using NFSv3 via a dedicated 48-port SDN-controlled Arista 7050Tx switch, and VM disk images are striped across sixteen objects.

4.1 Optimization

We begin by benchmarking the greedy solver, which is used in all subsequent experiments. Given rules that run in constant time, this solver has a computational complexity of $O(N \log N \log M)$ for a system with N objects and M devices.

We measure solver runtime when enforcing a simple load-balancing policy (based on the `device_has_space` and `load_balanced` rules, with the latter enforcing a `LOAD_SPREAD` of 20%) in deployments of various sizes. In each experiment, a simulated cluster is modelled with fixed-capacity devices (no more than ten per node) randomly populated with objects whose sizes and loads are drawn from a Pareto distribution, scaled such that no single object exceeds the capacity of a device and the cluster is roughly 65% full. For each configuration we present the time required to find a zero-cost solution as well as the number of reconfigurations required to achieve the solution, averaged over ten runs. Some experiments require no reconfigurations because their high object-to-device ratios result in very small objects that yield well-balanced load distributions under the initial, uniformly random placement; the runtimes for these experiments measure only the time required to validate the initial configuration.

As Table 2 shows, the flexibility provided by Python-based rules comes with a downside of relatively high execution times (more than a minute for a system with 100K objects and 1K devices). While we believe there is ample opportunity to improve our unoptimized implementation, we have not yet done so, primarily because rebalance jobs run in overlapping batches, allowing optimization and actuation tasks to execute in parallel, and actuation times typically dominate.

4.2 Actuation

In the following experiment we measure actuation performance by demonstrating how Mirador restores redundancy in the face of hardware failures. We provision four nodes, each with 12 1 TB SSDs, for a total of 48 devices. We deploy 1,500 client VMs, each running `fio` [18] with a configuration modelled after virtual desktop workloads. VMs issue 4 KiB requests against 1 GiB disks. Requests are drawn from an 80/20 Pareto distribution with an 80:20 read:write ratio; read and write throughputs are rate-limited to 192 KiB/sec and 48 KiB/sec, respectively, with a maximum queue depth of 4, generating an aggregate throughput of roughly 100K IOPS.

Five minutes into the experiment, we take a device offline and schedule a rebalance job. The `rplset_durable` rule assigns infinite cost to objects placed on failed devices, forcing reconfigurations, while load-balancing and failure-domain rules prioritize the choice of replacement devices. The job defers actuation until a 15 minute stabilization interval expires so that transient errors do not trigger unnecessary migrations. During this time it inspects more than 118,000 objects, and it eventually rebuilds 3053 in just under 20 minutes, with negligible effect on client workloads, as seen in Figure 2.

4.3 Resource Objectives

We now shift our attention to the efficacy of specific placement rules, measuring the degree to which they can affect client performance in live systems. We first focus on resource-centric placement rules that leverage knowledge of cluster topology and client configurations to improve performance and simplify lifecycle operations.

Name	Objective	Cost	Lines of Code
device_has_space	devices are not filled beyond capacity	∞	4
rplset_durable	replica sets are adequately replicated on healthy devices	∞	4
load_balanced	load is balanced across devices	70	13
links_balanced	load is balanced across links	20	13
node_local	client files are co-located on common nodes	60	30
direct_connect	client connections are routed directly to their most-frequently accessed nodes	10	14
wss_best_fit	active working set sizes do not exceed flash capacities	40	4
isolated	cache-unfriendly workloads are co-located	20	30
co_scheduled	competing periodic workloads are isolated	20	35

Table 1: Objective functions used in evaluation section; *cost* gives the penalty incurred for violating the rule.

Objects	Devices	Reconfigurations	Time (seconds)
1K	10	6.40 ± 2.72	0.40 ± 0.06
1K	100	145.50 ± 33.23	0.83 ± 0.08
1K	1000	220.00 ± 12.53	10.11 ± 0.49
10K	10	0.00 ± 0.00	1.61 ± 0.01
10K	100	55.70 ± 5.46	5.54 ± 0.37
10K	1000	1475.00 ± 69.70	16.71 ± 0.88
100K	10	0.00 ± 0.00	17.10 ± 0.37
100K	100	9.30 ± 4.62	22.37 ± 5.38
100K	1000	573.80 ± 22.44	77.21 ± 2.87

Table 2: Greedy solver runtime for various deployment sizes with a basic load-balancing policy; *reconfigurations* gives the number of changes made to yield a zero-cost solution.

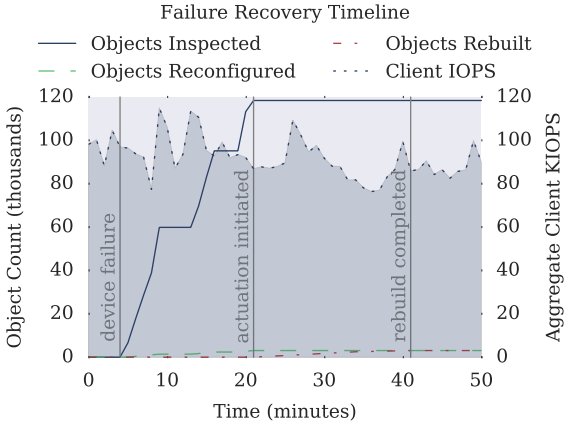


Figure 2: Rebuilding replicas after a device failure.

4.3.1 Topology-Aware Placement

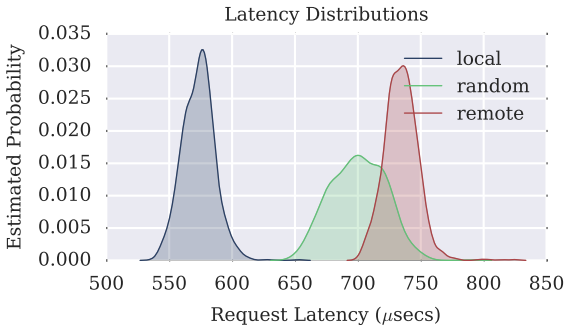
In this experiment we measure the value of topology-aware placement policies in distributed systems. We deploy four storage nodes and four clients, with each client hosting 8 VMs running a `fio` workload issuing random 4 KiB reads against dedicated 2 GiB virtual disks at queue depths ranging between 1 and 32.

Figure 3a presents the application-perceived latency achieved under three different placement policies when VMs issue requests at a queue depth of one. The *random* policy distributes stripes across backend devices using a simple consistent hashing scheme and applies a random one-to-one mapping from clients to storage nodes. This results in a configuration where each node serves requests from exactly one client, and with four nodes, roughly 75% of reads access remotely-hosted stripes. This topology-agnostic strategy is simple to implement, and, assuming workload uniformity, can be expected to achieve even utilization across the cluster, although it does require significant backend network communication. Indeed, as the number of storage nodes in a cluster increases, the likelihood that any node is able to serve requests locally decreases; in the limit, all requests require a backend RTT. This behavior is captured by the *remote* policy, which places stripes such that no node has a local copy of any of the data belonging to the clients it serves. The *local* policy follows the opposite strategy, placing all stripes for a given VM on a single node and ensuring that clients connect directly to the nodes hosting their data. Notably, all three policies are implemented in less than twenty lines of code, demonstrating the expressiveness of Mirador's optimization framework.

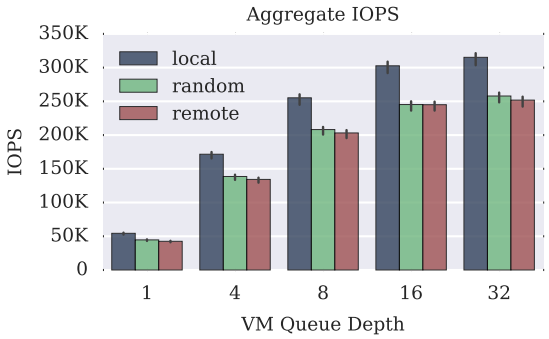
By co-locating VM stripes and intelligently routing client connections, the local policy eliminates additional backend RTTs and yields appreciable performance improvements, with median latencies 18% and 22% lower than those of the random and remote policies, respectively. Similar reductions are obtained across all measured queue depths, leading to comparable increases in throughput, as shown in Figure 3b.

4.3.2 Elastic Scale Out

In addition to improving application-perceived performance, minimizing cross-node communication enables linear scale out across nodes. While a random placement policy would incur proportionally more network RTTs as a cluster grows in size (potentially consuming oversubscribed cross-rack bandwidth), local placement strate-



(a) Latency distributions at queue depth of 1



(b) Mean throughput at various queue depths

Figure 3: Performance under three different placement strategies. The *local* policy yields a median latency 18% and 22% lower than the *random* and *remote* policies, respectively, resulting in an average throughput increase of 26%. (Error bars in Figure 3b give 95% confidence intervals.)

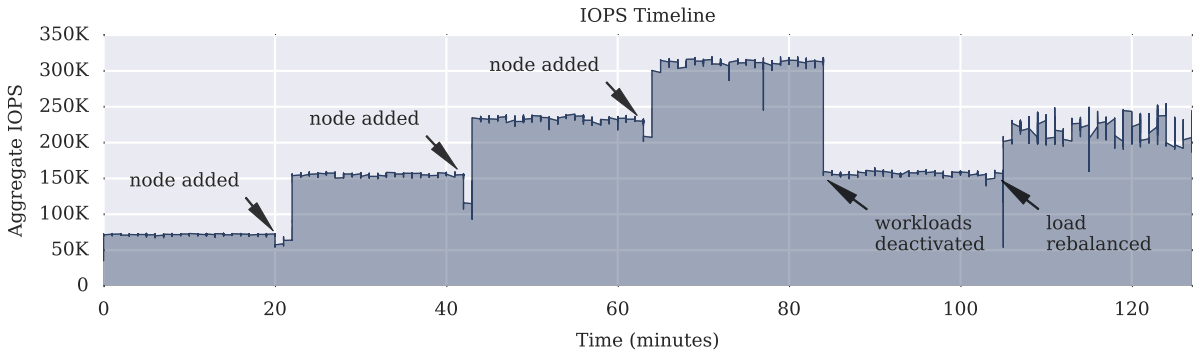


Figure 4: Mirador responds to changes in cluster topology and workload behavior. Data is immediately migrated to new storage nodes as they are introduced in 20 minute increments, starting at time t_{20}; the brief throughput drops are due to competition with background data copies. At time t_{85}, two of the four client machines are deactivated; the remaining client load is subsequently redistributed, at which point performance is limited by client resources.

gies can make full use of new hardware with minimal communication overhead. This is illustrated in Figure 4, which presents a timeline of aggregate client IOPS as storage nodes are added to a cluster. At time t_0 the cluster is configured with a single storage node serving four clients, each hosting 16 VMs issuing random 4 KiB reads at a queue depth of 32; performance is initially bottlenecked by the limited storage. At time t_{20}, an additional node is introduced, and the placement service automatically rebalances the data and client connections to make use of it. It takes just over two minutes to move roughly half the data in the cluster onto the new node. This migration is performed as a low-priority background task to limit interference with client IO. Two additional nodes are added at twenty minute intervals, and in each case, after a brief dip in client performance caused by competing migration traffic, throughput increases linearly.

The performance and scalability benefits of the local policy are appealing, but to be practical, this approach requires a truly dynamic placement service. While both local and random policies are susceptible to utilization imbalances caused by non-uniform workload patterns (e.g., workload 'hot spots'), the problem is exacerbated in the local case. For example, if all workloads placed on a specific node happen to become idle at the same time, that node will be underutilized. Figure 4 shows exactly this scenario at time t_{85}, where two clients are deactivated and the nodes serving them sit idle, halving overall throughput. After waiting for workload behavior to stabilize, the placement service responds to this imbalance by migrating some of the remaining VMs onto the idle storage, at which point the clients become the bottleneck.

4.4 Workload Objectives

Placement policies informed by resource monitoring can provide significant improvements in performance and efficiency, but they are somewhat *reactive* in the sense that they must constantly try to 'catch up' to changes in work-

load behavior. In this section we introduce and evaluate several techniques for improving data placement based on longitudinal observations of workload behavior.

The following examples are motivated by an analysis of hundreds of thousands of workload profiles collected from production deployments over the course of more than a year. The synthetic workloads evaluated here, while relatively simple, reflect some of the broad patterns we observe in these real-world profiles.

For these experiments, we extend the storage configuration described in § 4.3 with a disk-based capacity tier. The placement service controls how objects are assigned to flash devices as before; nodes manage the flash cards as LRU caches and page objects to disk in 512 KiB blocks. We artificially reduce the capacity of each flash device to 4 GiB to stress the tiering subsystem. While our evaluation focuses on conventional tiered storage, we note that the techniques presented here are applicable to a wide variety of hierarchical and NUMA architectures in which expensive, high-performance memories are combined with cheaper, more capacious alternatives, possibly connected by throughput-limited networks.

4.4.1 Footprint-Aware Placement

Many real-world workloads feature working sets (roughly defined as the set of data that is frequently accessed over a given period of time) that are much smaller than their total data sets [13, 56]. Policies that make decisions based only on knowledge of the latter may lead to suboptimal configurations. We show how augmenting traditional capacity rules with knowledge of working set sizes can lead to improved placement decisions.

We begin by deploying eight VMs across two clients connected to a cluster of two nodes. Each VM disk image holds 32 GiB, but the VMs are configured to run random 4 KiB read workloads over a fixed subset of the disks, such that working set sizes range from 500 MiB to 4 GiB. Given two nodes with 8 GiB of flash each, it is impossible to store all 256 GiB of VM data in flash; however, the total workload footprint as measured by the analysis service is roughly 17 GiB, and if carefully arranged, it can fit almost entirely in flash without exceeding the capacity of any single device by more than 1 GiB.

We measure the application-perceived latency for these VMs in two configurations. In the first, VMs are partitioned evenly among the two nodes using the *local* policy described in § 4.3.1 to avoid network RTTs. In the second, the same placement policy is used, but it is extended with one additional rule that discourages configurations where combined working set sizes exceed the capacity of a given flash card. The cost of violating this rule is higher than the cost of violating the node-local rule, codifying

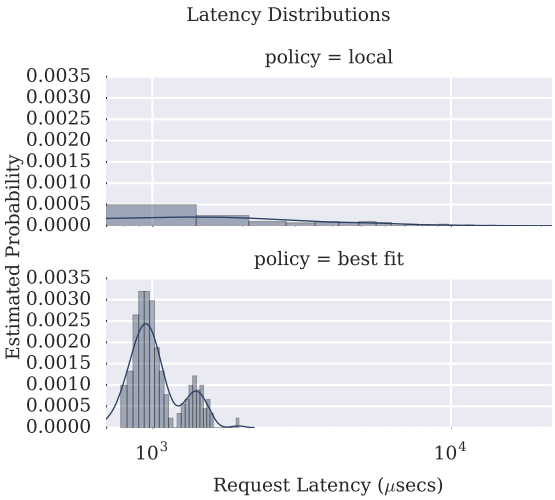


Figure 5: Fitting working sets to flash capacities ('best fit') yields a median latency of 997 μsecs, compared to 2088 μsecs for the 'local' policy that eliminates backend network RTTs but serves more requests from disk.

a preference for remote flash accesses over local disk accesses. The greedy solver is a good fit for this problem and arrives at a configuration in which only one flash device serves a combined working set size larger than its capacity.

As Figure 5 shows, the *best-fit* policy results in significantly lower latencies, because the cost of additional network hops is dwarfed by the penalty incurred by cache misses. The purely local policy exhibits less predictable performance and a long latency tail because of cumulative queuing effects at the disk tier. This is a clear example of how combining knowledge of the relative capabilities of network links and storage tiers with detailed workload profiling can improve placement decisions.

4.4.2 Noisy Neighbor Isolation

We next introduce four cache-unfriendly workloads each with 4 GiB disks. The workloads perform linear scans that, given 4 GiB LRU caches, are always served from disk and result in substantial cache pollution. These workloads make it impossible to completely satisfy the working set size rule of the previous experiment.

We measure the request latency of the original workloads as they compete with these new cache-unfriendly workloads under two policies: a *fair share* policy that distributes the cache-unfriendly workloads evenly across the flash devices, and an *isolation* policy that attempts to limit overall cache pollution by introducing a new rule that encourages co-locating cache-unfriendly workloads on common nodes, regardless of whether or not they fit within flash together. As Figure 6 shows, this lat-

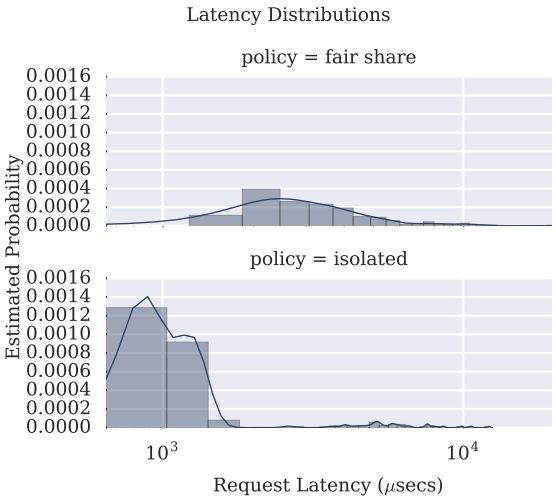


Figure 6: Isolating cache-unfriendly workloads on a single device yields a median latency of 1036 μsecs, compared to 3220 μsecs for the 'fair' policy that distributes these workloads uniformly across all devices.

ter policy exhibits a bimodal latency distribution, with nearly 48% of requests enjoying latencies less than one millisecond while a handful of 'victim' workloads experience higher latencies due to contention with cache-unfriendly competitors. The fair share policy, on the other hand, features a more uniform distribution, with all workloads suffering equally, and a median latency more than three times higher than that of the isolated policy.

4.4.3 Workload Co-scheduling

Finally, we introduce a technique for leveraging long-term temporal patterns in workload behavior to improve data placement. We frequently see storage workloads with pronounced diurnal patterns of high activity at key hours of the day followed by longer periods of idleness. This behavior typically correlates with workday habits and regularly scheduled maintenance tasks [16, 37, 46]. Similar effects can be seen at much smaller scales in CPU caches, where the strategy of co-locating applications to avoid contention is called 'co-scheduling' [50].

We present a simple algorithm for reducing cache contention of periodic workloads. The workload analysis service maintains an extended time series of the footprint of each workload, where footprint is defined as the number of unique blocks accessed over some time window; in this experiment we use a window of ten minutes. Given a set of workloads, we compute the degree to which they contend by measuring how much their bursts overlap. Specifically, we model the cost of co-locating two workloads W_1 and W_2 with corresponding footprint functions $f_1(t)$ and $f_2(t)$ as $\int \min(f_1(t), f_2(t))$. We use this metric

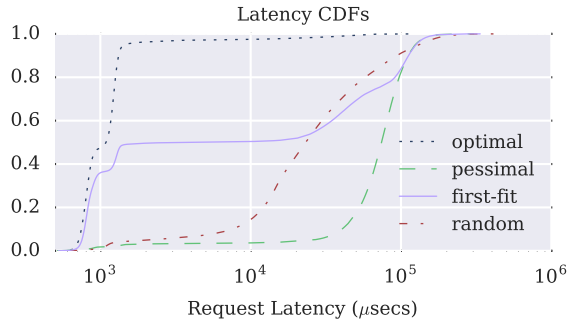


Figure 7: Co-scheduling periodic workloads.

to estimate the cost of placing workloads together on a given device, and employ a linear *first-fit* algorithm [14] to search for an arrangement of workloads across available devices that minimizes the aggregate cost. Finally, we introduce the `co_scheduled` rule which encodes an affinity for assignments that match this arrangement.

We evaluate this heuristic by deploying 8 VMs with 4 GiB disks across two storage nodes each with two 4 GiB flash devices. The VMs perform IO workloads featuring periodic hour-long bursts of random reads followed by idle intervals of roughly 3 hours, with the periodic phases shifted in some VMs such that not all workloads are active at the same time. The combined footprint of any two concurrent bursts exceeds the size of any single flash device, and if co-located, will incur significant paging. We measure request latency under a number of different configurations: *random*, in which stripes are randomly distributed across devices, *optimal* and *pessimal*, in which VMs are distributed two to a device so as to minimize and maximize contention, respectively, and *first-fit*, as described above.

Figure 7 plots latency CDFs for each of these configurations. The penalty of concurrent bursts is evident from the pronounced disparity between the optimal and pessimal cases; in the latter configuration, contention among co-located workloads is high, drastically exceeding the available flash capacity. The first-fit approximation closely tracks optimal in the first two quartiles but performs more like random in the last two, suggesting room for improvement either by developing a more sophisticated search algorithm or responding more aggressively to workload changes.

5 Experience

To see how Mirador performs in real-world environments, we sample logs detailing more than 8,000 rebalance jobs in clusters installed across nearly 50 customer sites and ranging in size from 8 to 96 devices. Figure 8 illustrates how time spent in the optimization stage scales

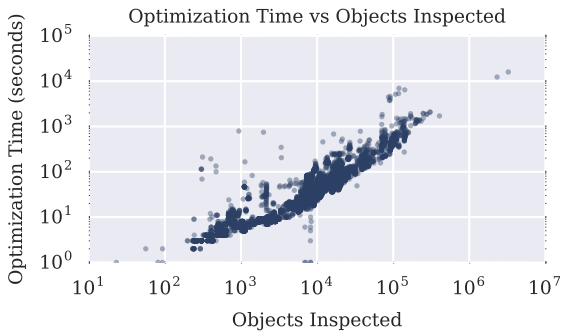


Figure 8: Optimization time vs. objects inspected.

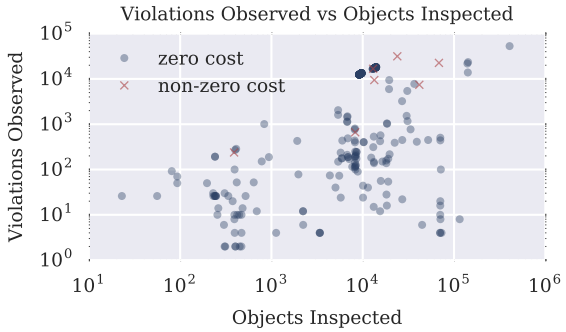


Figure 9: Violations observed vs. objects inspected (jobs where no zero-cost solution was found after a single optimization round are marked with a red x).

in proportion to the number of objects inspected; these measurements include rate-limiting delays imposed to prevent Mirador from impacting client workloads when reading metadata. Figure 9 plots the number of observed violations against the number of objects inspected per job, and highlights jobs that fail to find a zero-cost solution after a single optimization pass. This occurs in only 2.5% of sampled jobs in which objective functions are violated, and in 71% of these cases, no zero-cost solutions are possible due to environmental circumstances (some log samples cover periods in which devices were intentionally taken offline for testing or maintenance).

We have found Mirador's flexibility and extensibility to be two of its best attributes. Over the nearly 18 months in which it has been in production, we have adapted it to new replication policies and storage architectures simply by modifying existing rules and adding new ones. It has also been straightforward to extend Mirador to support new functionality: in addition to providing capacity balancing across storage devices and network links, it now plays a central role in cluster expansion, hardware retirement, failure recovery, health monitoring, and disk scrubbing features. For example, upon discovering an invalid data checksum, our disk scrubbing service simply marks the affected object as corrupt and notifies the placement service, where a custom rule forces the migration of marked objects to new locations, effectively rebuilding them from valid replicas in the process.

Our deployment strategy to date has been conservative: we ship a fixed set of rules (currently seven) and control how and when they are used. Assigning appropriate costs to rules requires domain knowledge, since rules often articulate conflicting objectives and poorly chosen costs can lead to unintended behavior. As an example, if solvers fail to identify a zero-cost solution, they yield the one with the lowest *aggregate* cost – if multiple rules conflict for a given assignment, the assignment which minimizes the overall cost is chosen. It is thus important to know which objective functions a replica set may violate so that high priority rules are assigned costs sufficiently large enough to avoid priority inversion in the face of violations of multiple lower-priority rules.

While objective functions neatly encapsulate individual placement goals and are relatively easy to reason about, comprehensive policies are more complex and must be carefully vetted. We validate rules, both in isolation and combination, with hundreds of policy tests. Declarative test cases specify a cluster configuration and initial data layout along with an expected optimization plan; the test harness generates a storage system model from the specification, invokes the planning engine, and validates the output. We have also built a fuzz tester that can stress policies in unanticipated ways. The test induces a sequence of random events (such as the addition and removal of nodes, changes in load, etc.) and invokes the policy validation tool after each step. Any cluster configuration that generates a policy violation is automatically converted into a test case to be added to the regression suite after the desired behavior is determined by manual inspection. Validating *any* non-trivial placement policy can require a fair amount of experimentation, but in our experience, the cost-based framework provided by Mirador provides knobs that greatly simplify this task.

In production, rebalance jobs run in two passes: the first enforces critical rules related to redundancy and fault tolerance, while the second additionally enforces rules related to load-balancing and performance. This is done because the planning engine must inspect objects in batches (batches are limited to roughly 10,000 objects to keep memory overheads constant), and we want to avoid filling a device in an early batch in order to satisfy low-priority rules when that same device may be necessary to satisfy higher-priority rules in a later batch.

Early testing revealed the importance of carefully tuning data migration rates. Our migration service originally provided two priorities, with the higher of these intended for failure scenarios in which replicas need to be rebuilt. In practice, however, we found that such failures place

additional stress on the system, often driving latencies up. Introducing high-priority migration traffic in these situations can lead to timeouts that only make things worse, especially under load. We have since adopted a single migration priority based on an adaptive queuing algorithm that aims to isolate migration traffic as much as possible while ensuring forward progress is made.

6 Related Work

Researchers have proposed a wide variety of strategies for addressing the data placement problem, also known as the file assignment problem [15]. Deterministic approaches are common in large-scale systems [38, 41, 48, 51, 53] because they are decentralized and impose minimal metadata overheads, and they achieve probabilistically uniform load distribution for large numbers of objects [43,45]. Consistent hashing [30] provides relatively stable placement even as storage targets are added and removed [21, 57]. Related schemes offer refinements like the ability to prioritize storage targets and modify replication factors [26, 27, 52], but these approaches are intrinsically less flexible than dynamic policies.

Non-deterministic strategies maintain explicit metadata in order to locate data. Some of these systems employ random or semi-random placement policies for the sake of simplicity and scalability [34, 39, 42], but others manage placement with hard-coded policies [20, 47]. Customized policies provide better control over properties such as locality and fault tolerance, which can be particularly important as clusters expand across racks [29].

Explicit metadata also make it easier to perform fine-grain migrations in response to topology and workload changes, allowing systems to redistribute load and ameliorate hot spots [35, 37]. Hierarchical Storage Management and multi-tier systems dynamically migrate data between heterogeneous devices, typically employing policies based on simple heuristics intended to move infrequently accessed data to cheaper, more capacious storage or slower, more compact encodings [4, 54].

Mirador has much in common with recent systems designed to optimize specific performance and efficiency objectives. Guerra et al. [23] describe a tiering system that makes fine-grain placement decisions to reduce energy consumption in SANs by distributing workloads among the most power-efficient devices capable of satisfying measured performance requirements. Janus [5] is a cloud-scale system that uses an empirical cacheability metric to arrange data across heterogeneous media in a manner that maximizes reads from flash, using linear programming to compute optimal layouts. Volley [2] models latency and locality using a weighted spring analogy and makes placement suggestions for geographically distributed cloud services. Tuba [9] is a replicated key-value store designed for wide area networks that allows applications to specify latency and consistency requirements via service level agreements (SLAs). It collects hit ratios and latency measurements and periodically reconfigures replication and placement settings to maximize system utility (as defined by SLAs) while honoring client-provided constraints on properties like durability and cost. Mirador supports arbitrary cost-function optimizations using a generic framework and supports policies that control network flows as well as data placement.

Mirador also resembles resource planning systems [6, 8] like Hippodrome [7], which employ a similar observe/optimize/actuate pipeline to design cost-efficient storage systems. Given a set of workload descriptions and an inventory of available hardware, these tools search for low-cost array configurations and data layouts that satisfy performance and capacity requirements. Like Mirador, they simplify a computationally challenging multidimensional bin-packing problem by combining established optimization techniques with domain-specific heuristics. However, while these systems employ customized search algorithms with built-in heuristics, Mirador codifies heuristics as rules with varying costs and relies on generic solvers to search for low-cost solutions, making it easier to add new heuristics over time.

Ursa Minor [1] is a clustered storage system that supports dynamically configurable *m-of-n* erasure codes, extending the data placement problem along multiple new dimensions. Strunk et al. [49] describe a provisioning tool for this system that searches for code parameters and data layouts that maximize user-defined *utility* for a given set of workloads, where utility quantifies metrics such as availability, reliability, and performance. Utility functions and objective functions both provide flexibility when evaluating potential configurations; however, Mirador's greedy algorithm and support for domain-specific hints may be more appropriate for online rebalancing than the randomized genetic algorithm proposed by Strunk et al.

7 Conclusion

Mirador is a placement service designed for heterogeneous distributed storage systems. It leverages the high throughput of non-volatile memories to actively migrate data in response to workload and environmental changes. It supports flexible, robust policies composed of simple objective functions that specify strategies for both data and network placement. Combining ideas from constraint satisfaction with domain-specific language bindings and APIs, it searches a high-dimension solution space for configurations that yield performance and efficiency gains over more static alternatives.

Acknowledgements

The authors would like to thank our shepherd, Kim Keeton, for multiple rounds of thorough and constructive feedback. The paper benefited enormously from Kim's help. We would also like to thank John Wilkes for some very frank and direct comments on an early version of the paper, and Mihir Nanavati for all of his help and feedback along the way.

References

[1] ABD-EL-MALEK, M., II, W. V. C., CRANOR, C., GANGER, G. R., HENDRICKS, J., KLOSTERMAN, A. J., MESNIER, M. P., PRASAD, M., SALMON, B., SAMBASIVAN, R. R., SINNAMOHIDEEN, S., STRUNK, J. D., THERESKA, E., WACHS, M., AND WYLIE, J. J. Ursa minor: Versatile cluster-based storage. In *FAST* (2005), G. Gibson, Ed., USENIX.

[2] AGARWAL, S., DUNAGAN, J., JAIN, N., SAROIU, S., AND WOLMAN, A. Volley: Automated data placement for geo-distributed cloud services. In *NSDI* (2010), USENIX Association, pp. 17–32.

[3] AGGARWAL, G., MOTWANI, R., AND ZHU, A. The load rebalancing problem. *J. Algorithms 60*, 1 (2006), 42–59.

[4] AGUILERA, M. K., KEETON, K., MERCHANT, A., MUNISWAMY-REDDY, K.-K., AND UYSAL, M. Improving recoverability in multi-tier storage systems. In *DSN* (2007), IEEE Computer Society, pp. 677–686.

[5] ALBRECHT, C., MERCHANT, A., STOKELY, M., WALIJI, M., LABELLE, F., COEHLO, N., SHI, X., AND SCHROCK, E. Janus: Optimal flash provisioning for cloud storage workloads. In *USENIX Annual Technical Conference* (2013), USENIX Association, pp. 91–102.

[6] ALVAREZ, G. A., BOROWSKY, E., GO, S., ROMER, T. H., BECKER-SZENDY, R. A., GOLDING, R. A., MERCHANT, A., SPASOJEVIC, M., VEITCH, A. C., AND WILKES, J. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Trans. Comput. Syst. 19*, 4 (2001), 483–518.

[7] ANDERSON, E., HOBBS, M., KEETON, K., SPENCE, S., UYSAL, M., AND VEITCH, A. C. Hippodrome: Running circles around storage administration. In *FAST* (2002), D. D. E. Long, Ed., USENIX, pp. 175–188.

[8] ANDERSON, E., SPENCE, S., SWAMINATHAN, R., KALLAHALLA, M., AND WANG, Q. Quickly finding near-optimal storage designs. *ACM Trans. Comput. Syst. 23*, 4 (2005), 337–374.

[9] ARDEKANI, M. S., AND TERRY, D. B. A self-configurable geo-replicated cloud storage system. In *OSDI* (2014), J. Flinn and H. Levy, Eds., USENIX Association, pp. 367–381.

[10] ASANOVIC, K. Firebox: A hardware building block for 2020 warehouse-scale computers. Keynote presentation, 2014 USENIX Conference on File and Storage Technologies (FAST).

[11] BARR, J. Amazon EBS (Elastic Block Store) - Bring Us Your Data. `https://aws.amazon.com/blogs/aws/amazon-elastic/`, August 2008.

[12] CULLY, B., WIRES, J., MEYER, D. T., JAMIESON, K., FRASER, K., DEEGAN, T., STODDEN, D., LEFEBVRE, G., FERSTAY, D., AND WARFIELD, A. Strata: scalable high-performance storage on virtualized non-volatile memory. In *FAST* (2014), B. Schroeder and E. Thereska, Eds., USENIX, pp. 17–31.

[13] DENNING, P. working set model of program behavior. *Communications of the ACM* (1968).

[14] DÓSA, G. The tight bound of first fit decreasing bin-packing algorithm is FFD(i) <= 11/9OPT(i) + 6/9. In *ESCAPE* (2007), vol. 4614 of *Lecture Notes in Computer Science*, Springer, pp. 1–11.

[15] DOWDY, L. W., AND FOSTER, D. V. Comparative models of the file assignment problem. *ACM Comput. Surv. 14*, 2 (1982), 287–313. comments: ACM Computing Surveys 15(1): 81-82 (1983).

[16] ELLARD, D., LEDLIE, J., MALKANI, P., AND SELTZER, M. I. Passive nfs tracing of email and research workloads. In *FAST* (2003), J. Chase, Ed., USENIX.

[17] EMC. DSSD D5. `https://www.emc.com/en-us/storage/flash/dssd/dssd-d5/index.htm`, 2016.

[18] Flexible io tester. `http://git.kernel.dk/?p=fio.git;a=summary`.

[19] FREUDER, E. A sufficient condition for backtrack-free search. *Communications of the ACM 29*, 1 (1982), 24–32.

[20] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system. In *ACM SIGOPS operating systems review* (2003), vol. 37, pp. 29–43.

[21] GOEL, A., SHAHABI, C., YAO, S.-Y. D., AND ZIMMERMANN, R. SCADDAR: An efficient randomized technique to reorganize continuous media blocks. In *ICDE* (2002), IEEE Computer Society, pp. 473–482.

[22] GRAHAM, R. L. Bounds on multiprocessing anomalies and related packing algorithms. In *Proceedings of the May 16-18, 1972, spring joint computer conference* (New York, NY, USA, 1972), AFIPS '72 (Spring), ACM, pp. 205–217.

[23] GUERRA, J., PUCHA, H., GLIDER, J. S., BELLUOMINI, W., AND RANGASWAMI, R. Cost effective storage using extent based dynamic tiering. In *FAST* (2011), USENIX, pp. 273–286.

[24] HARALICK, R., AND ELLIOT, G. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence 14*, 3 (1980), 263–313.

[25] HEBRARD, E., HNICH, B., O'SULLIVAN, B., AND WALSH, T. Finding diverse and similar solutions in constraint programming. In *AAAI* (2005), M. M. Veloso and S. Kambhampati, Eds., AAAI Press / The MIT Press, pp. 372–377.

[26] HONICKY, R. J., AND MILLER, E. L. A fast algorithm for online placement and reorganization of replicated data. In *IPDPS* (2003), IEEE Computer Society, p. 57.

[27] HONICKY, R. J., AND MILLER, E. L. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In *IPDPS* (2004), IEEE Computer Society.

[28] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference* (2010), P. Barham and T. Roscoe, Eds., USENIX Association.

[29] KANDULA, S., SENGUPTA, S., GREENBERG, A., PATEL, P., AND CHAIKEN, R. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference* (New York, NY, USA, 2009), IMC '09, ACM, pp. 202–208.

[30] KARGER, D., LEHMAN, E., LEIGHTON, T., PANIGRAHY, R., LEVINE, M., AND LEWIN, D. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing* (New York, NY, USA, 1997), STOC '97, ACM, pp. 654–663.

[31] KASHYAP, S. R., KHULLER, S., WAN, Y.-C. J., AND GOLUBCHIK, L. Fast reconfiguration of data placement in parallel disks. In *ALENEX* (2006), R. Raman and M. F. Stallmann, Eds., SIAM, pp. 95–107.

[32] KHULLER, S., KIM, Y.-A., AND MALEKIAN, A. Improved approximation algorithms for data migration. *Algorithmica 63*, 1-2 (2012), 347–362.

[33] KHULLER, S., KIM, Y. A., AND WAN, Y.-C. J. Algorithms for data migration with cloning. *SIAM J. Comput. 33*, 2 (2004), 448–461.

[34] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., CZERWINSKI, S., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. OceanStore: an architecture for global-scale persistent storage. *SIGPLAN Not. 35*, 11 (2000), 190–201.

[35] LIN, L., ZHU, Y., YUE, J., CAI, Z., AND SEGEE, B. Hot random off-loading: A hybrid storage system with dynamic data migration. In *MASCOTS* (2011), IEEE Computer Society, pp. 318–325.

[36] MACKWORTH, A. K. Consistency in networks of relations. *Artificial Intelligence 8*, 1 (1977), 99–118.

[37] NARAYANAN, D., DONNELLY, A., THERESKA, E., ELNIKETY, S., AND ROWSTRON, A. I. T. Everest: Scaling down peak loads through i/o off-loading. In *OSDI* (2008), USENIX Association, pp. 15–28.

[38] NIGHTINGALE, E. B., ELSON, J., FAN, J., HOFMANN, O. S., HOWELL, J., AND SUZUE, Y. Flat datacenter storage. In *OSDI* (2012), USENIX Association, pp. 1–15.

[39] ONGARO, D., RUMBLE, S. M., STUTSMAN, R., OUSTERHOUT, J. K., AND ROSENBLUM, M. Fast crash recovery in RAMCloud. In *SOSP* (2011), ACM, pp. 29–41.

[40] PETERSEN, C. Introducing Lightning: A flexible NVMe JBOF. `https://code.facebook.com/posts/989638804458007/introducing-lightning-a-flexible-nvme-jbof/`, March 2016.

[41] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001* (2001), Springer, pp. 329–350.

[42] SAITO, Y., FRÃŸLUND, S., VEITCH, A. C., MERCHANT, A., AND SPENCE, S. FAB: building distributed enterprise disk arrays from commodity components. In *ASPLOS* (2004), ACM, pp. 48–58.

[43] SANTOS, J. R., MUNTZ, R. R., AND RIBEIRO-NETO, B. A. Comparing random data allocation and data striping in multimedia servers. In *SIGMETRICS* (2000), ACM, pp. 44–55.

[44] SCHIEX, T., FARGIER, H., AND VERFAILLIE, G. Valued constraint satisfaction problems: Hard and easy problems. In *IJCAI (1)* (1995), Morgan Kaufmann, pp. 631–639.

[45] SEO, B., AND ZIMMERMANN, R. Efficient disk replacement and data migration algorithms for large disk subsystems. *TOS 1*, 3 (2005), 316–345.

[46] SHAMMA, M., MEYER, D. T., WIRES, J., IVANOVA, M., HUTCHINSON, N. C., AND WARFIELD, A. Capo: Recapitulating storage for virtual desktops. In *FAST* (2011), G. R. Ganger and J. Wilkes, Eds., USENIX, pp. 31–45.

[47] SHVACHKO, K., KUANG, H., RADIA, S., AND CHANSLER, R. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (Washington, DC, USA, 2010), MSST '10, IEEE Computer Society, pp. 1–10.

[48] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *ACM SIGCOMM 2001* (Aug. 2001).

[49] STRUNK, J. D., THERESKA, E., FALOUTSOS, C., AND GANGER, G. R. Using utility to provision storage systems. In *FAST* (2008), M. Baker and E. Riedel, Eds., USENIX, pp. 313–328.

[50] WANG, X., LI, Y., LUO, Y., HU, X., BROCK, J., DING, C., AND WANG, Z. Optimal footprint symbiosis in shared cache. In *CCGRID* (2015), IEEE Computer Society, pp. 412–422.

[51] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *OSDI* (2006), USENIX Association, pp. 307–320.

[52] 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.

[53] WEIL, S. A., LEUNG, A. W., BRANDT, S. A., AND MALTZAHN, C. RADOS: a scalable, reliable storage service for petabyte-scale storage clusters. In *PDSW* (2007), ACM Press, pp. 35–44.

[54] WILKES, J., GOLDING, R. A., STAELIN, C., AND SULLIVAN, T. The HP AutoRAID hierarchical storage system. *ACM Trans. Comput. Syst. 14*, 1 (1996), 108–136.

[55] WIRES, J., INGRAM, S., DRUDI, Z., HARVEY, N. J. A., AND WARFIELD, A. Characterizing storage workloads with counter stacks. In *OSDI* (2014), J. Flinn and H. Levy, Eds., USENIX Association, pp. 335–349.

[56] WONG, T. M., AND WILKES, J. My cache or yours? making storage more exclusive. In *USENIX Annual Technical Conference, General Track* (2002), C. S. Ellis, Ed., USENIX, pp. 161–175.

[57] ZHENG, W., AND ZHANG, G. Fastscale: Accelerate RAID scaling by minimizing data migration. In *FAST* (2011), USENIX, pp. 149–161.