Ergastulum: quickly finding near-optimal storage system designs

E. Anderson M. Kallahalla S. Spence R. Swaminathan Q. Wang
{anderse,maheshk,suspence,swaram,qwang}@hpl.hp.com.

Storage Systems Department
Hewlett Packard Laboratories
Palo Alto, CA 94304

Abstract

The cost of large storage systems is dominated by management costs. Typically, skilled administrators configure storage manually using rules of thumb. However, designing a storage system for a given workload is a difficult task, because there are millions of possible configurations and mappings of data, and because storage system behavior is complex. *Ergastulum* is a new storage system designer that can be used both to guide administrators in their design decisions and as part of an automatic storage system management tool like Hippodrome [4]. Ergastulum generalizes the best-fit bin packing heuristic with randomization and backtracking to efficiently search through the huge number of possible design choices. Design decisions are informed by device models that estimate storage system performance. We show that Ergastulum quickly generates near-optimal storage system designs. It is faster and generates better solutions than previous tools, and it is substantially faster than an integer programming implementation that generates optimal solutions for simplified device models. We conclude that Ergastulum is a comprehensive solution to the storage system design problem.

1 Introduction

Note: the tech-report version of this document is primarily intended to make the additional experiments that have been run available. It is not nearly as polished as the shorter paper version.

Previous studies [1, 20] have indicated that the cost of large storage systems, over the course of their lifetime, is dominated by management costs. Currently storage systems are managed by skilled and expensive administrators using rules of thumb. Unfortunately, because storage systems are complex and application workloads are complicated, the resulting systems are often over-provisioned, which makes them unnecessarily expensive, or under-provisioned, in which case they perform poorly. Given that enterprise class storage systems that cost $1,000,000 are not uncommon, over-provisioning by just a factor of two in an attempt to reduce the management costs can be extremely expensive.

This paper presents Ergastulum: a storage system design tool. Ergastulum takes a workload description, efficiently explores the huge number of possible storage system designs that could support the workload and chooses the one that best achieves some externally-specified goal, such as a minimal, balanced design. Administrators can use Ergastulum to determine if they have under or over-provisioned their storage system by running the tool on their existing workload, and comparing the resulting design to the one they are using. Even better, they can use Ergastulum as part of an automatic storage management system such as Hippodrome [4], thereby reducing storage management costs. Hippodrome iterates through a loop, first improving the storage system design using Ergastulum, second implementing the new design, and third analyzing the workload as it runs on the new storage system.

Ergastulum generates a *storage system design* based on an input workload, a set of allowed device types, and a goal. This design captures both a map-

ping of the workload's data onto a storage system and the configuration of that storage system, including the number of disk arrays used, the number and type of drives in those arrays and the RAID configuration of the drives.

Ergastulum uses analytic storage system performance models to evaluate whether a candidate design supports a given workload. Analytic models are better than trace-driven simulators because they are faster. Ergastulum treats the performance models as a black box, making frequent calls to them through a narrow interface. Ergastulum is currently used with several different performance models [30, 3, 21], depending on which device types are used.

The storage system design problem is complex. With capacity-only constraints and fixed-size disks, it is as hard as the NP-complete bin-packing problem [13]. For arbitrary goals or models only an exhaustive search would find an optimal design. Consider a goal function or model where exactly one random division of stores between two devices is optimal: finding that division may require searching all possible configurations.

One way to solve this design problem is to cast it as a integer-linear programming (IP) optimization problem. However, it is difficult to convert the complex device performance models into a form suitable for IP constraints. Furthermore, the techniques that are used to solve IP problems are very slow. However, as we will show in Section 4.2, this approach is useful for evaluating Ergastulum.

Another approach to this problem is to use genetic algorithms to try to "evolve" a good storage system design. This approach was attempted [33], but it was found to be substantially slower than previous approaches [2] and at best it generated the same quality solutions.

A third approach to this problem is to use bin-packing algorithms. Unfortunately, storage systems introduce constraints that existing algorithms don't handle. For instance, device utilization constraints are non-additive; i.e., the net utilization of a device by multiple parts of a workload is not the simple sum of the independent utilizations of each part. Existing bin packing algorithms [18, 25] require that the constraints are additive but, as we show, some of the best-fit packing algorithms [19] can be adapted to handle non-additive constraints.

The approach used in Ergastulum is a generalization of the best-fit bin-packing algorithm. First, we add randomization because it often helps in complex search problems. Second, we add re-assignment, where existing parts of the assignment are un-done and then re-tried, to help the design tool avoid local minima. Third, we can run re-assignment at different granularities to fine-tune the design.

Ergastulum is designed to find "good" (as opposed to optimal) solutions in a reasonable amount of time. As the problem is NP-hard, we cannot expect to find optimal solutions in all cases. However, we provide experimental evidence that the solutions generated by our design tool are near optimal.

The primary contribution of this paper is to describe a fast tool for finding near-optimal storage system designs. Section 2 summarizes the inputs and dependencies of a storage system design tool. Section 3 describes the architecture, the central data structure, and the search techniques used in Ergastulum. Section 4 experimentally evaluates Ergastulum in comparison to the Minerva design tool [2], in comparison to the optimal results generated by an integer programming design tool, and probabilistically by generating hundreds of thousands of random designs. Section 5 provides a summary of other related work, and finally, Section 6 concludes and discusses future directions.

2 Inputs to Ergastulum

The inputs to the Ergastulum design tool are a workload description, a set of potential devices, optional user-specified constraints, and an externally-specified goal. The design tool chooses the appropriate types of devices to use, the configuration of each device and the mapping of application data onto the devices, to best achieve the specified goal.

2.1 Workloads, devices and constraints

The workload description [34] captures capacity and performance requirements in terms of *stores* and *streams*. A *store* is a logically contiguous block of storage used to hold, for example, a file system, a database table-space, or application data. The size of

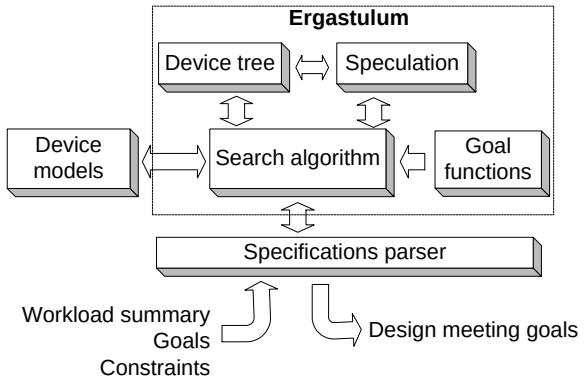


Figure 1: Ergastulum's architecture: The search algorithm uses the device tree to keep track of the current design, and the speculation engine to rapidly switch between alternative designs. Different goal functions are used in different phases of the search. Ergastulum takes its input (and generates its output) in the Rome specification language [34]. The device models are used to quantitatively evaluate the quality of the design.

the store is its most significant attribute. A *stream*, which captures the dynamic component of the workload, summarizes an I/O request pattern. Streams capture information about the I/O accesses to a single associated store, such as an average request rate, an average request size, and a summary of the sequentiality in block accesses. Further information on stream attributes can be found in [30, 3, 21].

Disks in an array are grouped together into *logical units* (LUs) using RAID [23]. Stores are packed into the LUs of configured disk arrays. The array types available to the design tool are specified as inputs, along with any restrictions on their configuration. Disk arrays inherently have certain constraints such as limits on the number of disks in a RAID group, the maximum number of disks in the device, and the maximum bandwidth of controllers. Ergastulum uses this information to generate a configuration, setting device parameters for each LU, such as RAID level, stripe size, and disks used, and global parameters such as cache page sizes.

The user may place additional constraints on valid designs. For example, they may restrict the valid device configurations, by limiting choices of RAID levels or disk types. They can specify a maximum total system cost, or a maximum utilization of any device. Furthermore, they can specify that certain stores be placed on particular devices, or on LUs with particular RAID levels.

The final input is a user-specified goal, which is used by the design tool to determine the "best" choice from a set of possible configurations. There are many possible goals for a storage system design such as minimal price, balanced load or low utilization. The goal is implemented as a function that chooses between two different configuration choices.

2.2 Device performance models

As mentioned above, Ergastulum needs device performance models. The device performance models take a possible configuration of a device and a mapping of stores onto the LUs of a device, and returns a prediction of the utilization of the components of the device. For example, if a single disk can handle 100 small random reads/second, and it has two uncorrelated streams on it each performing 25 reads/second, then the performance model would report that the disk is 50% utilized. Ergastulum uses three different performance models: monolithic equation-based models as described in [21], modular equation-based models as described in [30], and table-based models as described in [3], depending on which device types are used. All the models use summaries [34, 31] of the workload (I/O traces) to make their predictions. Since neither device performance modeling nor I/O summarization is the focus of this paper, we refer interested readers to the above references for more details.

3 Architecture of Ergastulum

Ergastulum consists of three main components: a data structure, called the *device tree*, that keeps track of the current design, previous designs and possible configuration changes; a search algorithm that uses various strategies to find a near-optimal design; and a state management component, called *speculation*, that allows Ergastulum to easily roll back to a previously generated design with low overhead. These components are illustrated in Figure 1.

3.1 Device tree

Our search heuristic uses the *device tree* as the central data structure. An example is shown in Figure 2. The device tree serves four different purposes.

First, the device tree represents the current storage system design. It records the configuration parameters for the different storage devices, the mapping of stores onto LUs and the current utilization of different components of each device. In Figure 2, the current design contains an existing, but expandable, XP256 disk array [16].

Second, the device tree represents the possible configuration changes that can be made to the design. It records where new arrays or components can be added to the configuration, for example, where a new RAID group can be added to an array. It also records where array configurations can be transformed into alternative configurations, for example changing a RAID-1/0 LU into a RAID-5 LU. In Figure 2, the device tree can potentially be changed to add a new FC-60 disk array [15] and some LUs.

Third, in cooperation with speculation, the device tree represents prior, saved designs. All necessary information is recorded to allow Ergastulum to quickly switch to a prior design.

Fourth, attributes attached to nodes in the device tree isolate the design tool from external modules. The attributes encapsulate the device models, and act as the interface to the goal functions. For instance, Figure 2 shows existing or potential attributes capturing aspects of device performance associated with an array. These attributes represent controller, backplane and SCSI bus utilization.

The device tree is a naturally expandable structure, highly suited to incrementally building a storage system design. Changes to the configuration represented by the device tree are performed by methods associated with each node of the device tree. Thus, the search algorithm is independent of the idiosyncrasies of the actual device configurations. Two types of functions may be associated with each node in the tree: *expansion* (add new tree nodes) and *transformation* (change existing tree nodes). Expansion functions increase the resources in the tree by adding a child node, for example by creating a new disk array or adding an LU (and its associated disks)

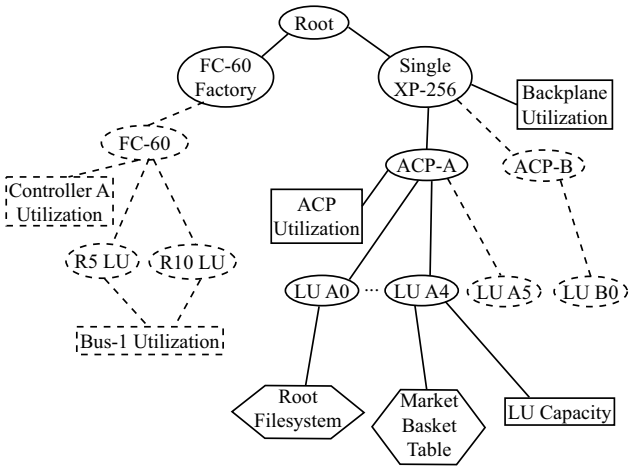


Figure 2: A sample device tree, part-way through an assignment. Devices are shown in ovals, attributes are in rectangles, and stores are in hexagons. Solid lines indicate existing nodes in the device tree, dashed ones indicate potential nodes that could be added. The FC-60 factory on the left can create multiple FC-60 disk arrays which in turn can create RAID-5 or RAID-1/0 LUs; FC-60 LUs share SCSI bus utilizations. Because the administrator specified that a single XP-256 disk array should be used, there is no factory for the XP. The XP-256 has a multi-level internal structure; up to 4 array control pairs (ACP) can be attached to the backplane and LUs are attached to an individual ACP. Many additional attributes and possible transformations are possible.

to a disk array. In Figure 2, an expansion function defined at the FC-60 factory node causes the addition of a new FC-60 array to the device tree. Transformation functions transform nodes in the tree by replacing one node with another that has different parameters; for example, by transforming a RAID-1/0 LU into a RAID-5 LU, or by changing the disk type in an LU. In Figure 2, a transformation function defined at the RAID-5 LU node of the FC-60 could replace that node with a RAID-1/0 LU node.

Nodes in the device tree have *attributes* which capture device properties such as the price of a particular node, the available and maximum capacity of an LU, or the utilization of the SCSI buses in a device. Some attributes (such as SCSI bus utilization) can be shared between multiple nodes in the device tree. Utilization attributes are determined by the device performance models. As stores are added and removed from the tree, and the device

configuration changes, the values of the attribute will change appropriately. Assigning a store to a particular leaf should only affect the attributes associated with nodes on the path from the root to that leaf. Constraints on the device configuration can be recorded as maximum values of attributes. The goal functions use the values stored in attributes to inform their choice between possible configurations.

3.2 Search heuristic

The search heuristic in Ergastulum is a generalized version of best-fit bin packing with randomization. Ergastulum uses multiple phases to optimize the design.

Initial assignment starts with empty devices. The list of stores is randomized, and each store is assigned into the device tree using a best-fit search of the tree. Because best-fit bin packing is a greedy algorithm, initial assignment often ends in a local minima. To escape from a local minima, a random subset of stores are removed from the device tree and re-assigned. Re-assignment in Ergastulum currently operates at two levels of granularity: on one or more LUs and on a single store.

The different assignment phases use slightly different goal functions because they serve different purposes. For example, LU-level reassignment tries to reduce the total number of LUs, by using the goal of a tight packing, but store-level reassignment tries to balance the load for use in a real system. Earlier work [6] showed that using a special goal function during initial assignment can also help the design tool avoid local minima.

The key operation is the process of adding a single store into the device tree. Ergastulum uses the same algorithm for adding a single store during both initial assignment and reassignment. The rest of the search algorithm assumes that adding a single store finds the best incremental design with the new store added into the tree.

3.2.1 Adding a store into the device tree

Given a store to assign, the design tool searches through all leaf nodes in the device tree looking for the "best" location for the new store. The "best" location is determined by a *path comparison function*, which is a simplification of the general goal function. Since the store will be assigned to a particular leaf, and the only attributes which change are along the path from the root to the leaf, it is sufficient and faster to choose between two different assignments by looking at only the attributes along the two paths. Ergastulum provides both paths to the path comparison function, and the path comparison function selects the "better" path.

The search through all of the leaf nodes is performed as a depth-first search of the device tree to minimize recalculation of attributes in interior nodes. At each node in the tree, we first try to assign the store to that node (a *simple* assignment). If a constraint at that node is violated, or the path comparison function indicates that this path is worse than the best path so far, the algorithm skips searching any children of the node. If the simple assignment succeeds, and the search has reached a leaf, then this path is saved as the "best" path so far, and the search then backtracks.

After trying the simple assignment at a node and its children, the search algorithm then iteratively tries the various expansion and transformation functions defined at that node. If a function succeeds, then a simple assignment is re-tried on this alternative state of the node. Otherwise, the search algorithm undoes any changes to the tree made by the function, and tries the next function. Once all the functions have been tried and the best resulting path has been saved, the depth-first search continues. The functions can therefore be viewed as "virtual" children in the depth first search. Logically, saving the best path, and undoing the changes made by the expansion and transformation functions is done by saving and restoring the entire tree. Section 3.3 describes how we use *speculation* to reduce the overhead of the save and restore operations.

Removing stores from the device tree so they can be assigned elsewhere is essentially the inverse of adding a store. First, the attributes are updated appropriately. Then a contraction function (roughly the inverse of the expansion functions) is run on the nodes that were updated to determine if any of the nodes in the tree can be removed. Removing nodes in the tree may further update the attributes.

3.2.2 Initial assignment phase

In the initial assignment phase, Ergastulum builds an initial configuration from an empty device tree, starting from a root node. Figure 3 shows the initial assignment phase.

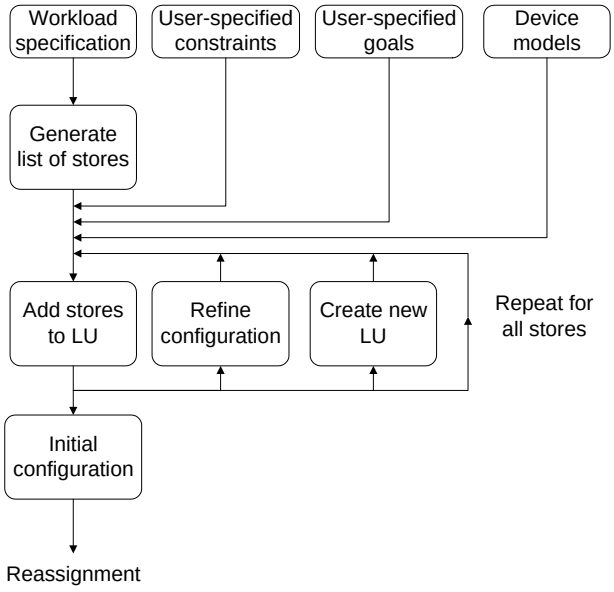


Figure 3: Initial assignment phase of Ergastulum

Initial assignment takes the list of available stores, randomizes it and iterates through the list, adding each store into the device tree. If a store fails to fit anywhere in the tree because of some constraint, Ergastulum reports that the store did not fit, and stops after initial assignment without generating a design. Otherwise, once all stores have successfully been assigned, the resulting device tree represents the *initial assignment*.

3.2.3 Reassignment phase

The initial assignment phase is followed by a number of reassignment phases. Each reassignment phase randomly removes some set of stores from the current assignment, and reassigns them into the tree. It then uses a tree comparison function (derived from the goal function) to select which design is better. For improved efficiency, and to avoid having to keep both trees directly available, the tree comparison function first summarizes a tree and later compares the two summaries.

The reassignment phases differ in the rule that they use to remove stores from the existing configuration. This is done to support the different goals of each reassignment stage. We currently use two rules for removing stores: remove all the stores from one or more randomly selected LUs and remove a single random store. In each case, after removing the stores, they are randomly sorted, and assigned back into the tree using the single-store assignment algorithm described earlier.

The LU-level reassignment phase attempts to reduce the number of LUs used in the design by removing all of the stores on one or more LUs, and trying to reassign them into a tighter packing. We optionally remove all the stores from multiple LUs because experimentation has shown that removing the stores on a single LU results in no changes, either because that LU only contains a small number of stores or because there is little space available on other LUs. The design tool supports options to specify both a minimum total number of stores to remove and a minimum number of LUs. LUs are selected randomly from the set of remaining, not-yet-reassigned LUs. Once all the LUs are reassigned, this phase may be repeated for a user-specified number of rounds.

The store level reassignment phase is designed to balance the load across the minimized devices. It works by removing a random store and reassigning it with the goal of balancing the load. Store level reassignment reassigns some fraction of the stores in the configuration.

3.3 State management with speculation

To support the rapid try/undo pattern exhibited by the design tool, we use an idea called *speculation* to avoid making complete copies of the device tree. Speculation only copies data in the tree just before it is written; before a node in the device tree modifies itself, duplicates are made of anything that might change. Before modification of a sub-tree, speculation only makes a copy of the nodes in that sub-tree, since changes in the sub-tree can't affect nodes outside of it.

Speculation uses multiple stacks of "in-flight" configurations to support swapping between widely different configurations. The search algorithm only

needs to restore or save from the top of any stack because of the structure imposed by the depth-first search. We support an arbitrary number of state stacks, as some expansion or transformation functions may want to try out a possible change, but the search algorithm itself only uses a current "trial" and "best" state.

The main advantage of speculation is that it eliminates the need to track how the design tool got to a particular configuration. This means that we only need to implement the "forward" modifications to the device tree, rather than having to record the operations that would undo a particular transition. Recording the full set of transitions would get prohibitively complex for handling substantially different designs. A second advantage of speculation is that it makes switching between different designs linear in the number of changed nodes, rather than the number of operations. As the number of nodes is usually much smaller than the number of operations, this is a substantial improvement.

4 Evaluation

We evaluate Ergastulum using two metrics. Our primary metric is the quality of the solution. For example, given a device model, a minimum cost goal, and an input workload, the best solution is the one with the lowest cost. Our second metric is the running time used by the various approaches, with a more qualitative goal of showing that the performance of the tool is "good enough" to be usable in practice.

We first compare Ergastulum with Minerva [2] to show that our new solution is better than previous work. We then attempt to evaluate how well Ergastulum does on an absolute scale. Our first comparison point uses integer programming (IP) to generate optimal solutions for small inputs and simplified device models. Our second comparison point uses random sampling to show that with realistic workload and device models Ergastulum generates near-optimal solutions, provided that we use reassignment to optimize the initial assignment.

4.1 Comparison with Minerva

The design of Ergastulum benefits from experience with an existing configuration design tool called Minerva [2]. Minerva sometimes generates poorer quality solutions, because it separates the steps of selecting RAID levels for stores, configuring devices and assigning stores onto devices. In addition, Minerva takes considerably longer to generate solutions, because it uses an expensive search heuristic, that prevents it from taking full advantage of incremental models,

Since Minerva only supports the the Hewlett-Packard SureStore Model 30/FC High Availability (FC-30) disk array [14], it is used in this comparison. Fully-configured, an FC-30 can support 8 LUs, using 30 4GB disks, two front-end controllers and 60 MB of NVRAM cache.

We perform the comparison using the workloads previously used to evaluate Minerva [2]. Most of the workloads are semi-synthetic; although they have been extrapolated from traces, phasing behavior is simplified. The *tpcd* workloads use the full analysis of a 10 GB TPC-D-like benchmark.

1. The *file-system* workload is a heterogeneous workload extrapolated from a work-group file server trace. It is characterized by moderate-sized requests (average 20KB) and essentially random accesses.
2. The *filesystem-cap* workload reduces the request rates and increases the store capacities to make the workload capacity-bound.
3. The *oltpSplit* specification is based on a trace of the TPC-C online transaction processing benchmark [29]. It has small random requests, and about 75% reads.
4. The *scientific3* workload is based on measurements of the NWchem application described in [27]. It has large sequential reads and writes.
5. *tpcd-2x* takes two copies of a 10 GB TPC-D-like analysis (to make it large enough to be interesting) and specifies that the two copies have uncorrelated overlaps. This workload has long, complex database queries, with interesting phasing behavior and a mix of both random and sequential accesses.

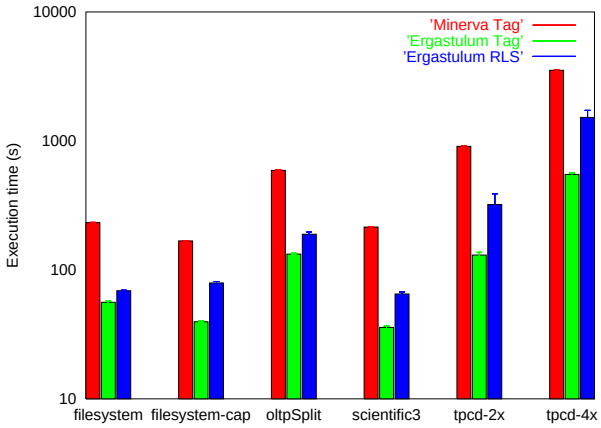


Figure 4: Comparison of execution times between Minerva and two variants of Ergastulum. Ergastulum-Tag is restricted to making the same RAID level choices as Minerva. Ergastulum takes only 15-25% of Minerva's execution time (note the log-scale y-axis).

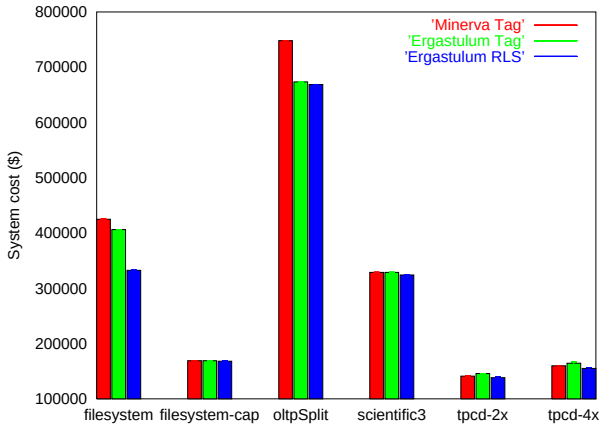


Figure 5: System cost comparison between Minerva and Ergastulum. When Ergastulum is restricted to using the same RAID levels as Minerva for each store, the results are comparable. Allowing Ergastulum the flexibility of selecting RAID levels itself makes Ergastulum-RLS as good or better than Minerva in all cases.

6. *tpcd-4x* has four copies of the TPC-D-like analysis, combined as for *tpcd-2x*.

Three results are generated for each workload by 1) Minerva, tagging stores with a RAID level before assignment, 2) Ergastulum, using the same separate tagging approach (Ergastulum-Tag); and 3) Ergastulum, using integrated RAID level selection (Ergastulum-RLS). The two different cases for Ergastulum enable us to compare the device configuration and assignment differences separately from the approach taken to RAID level selection (see [6] for more details on the latter). Results are based on ten iterations in each case. LUs are configured as 4-disk RAID-1/0 or 5-disk RAID-5. Both tools use the monolithic, equation-based device performance models [21] that were designed to accurately model the FC-30.

4.1.1 Execution time

Figure 4 shows that Ergastulum-Tag takes only 15-25% of Minerva's execution time to generate a configuration design (note the log-scale y-axis). Ergastulum-RLS takes longer than Ergastulum-Tag (but still less than Minerva) because integrated RAID level selection explores more alternatives. Ergastulum-RLS consistently comes up with the lowest cost solution.

For a typical execution of the *scientific3* workload, Minerva spends half of its time generating its first complete assignment and the other half running a single optimization pass. Conversely, Ergastulum spends only 6% of its time on generating a (somewhat worse) initial configuration, relying on randomized reassignment to improve the configuration. This difference is because Minerva uses a more complicated heuristic to try to determine the correct location for a store in a single pass.

4.1.2 System cost

Figure 5 compares the system costs for Ergastulum and Minerva. We can see that with the RAID levels fixed, Minerva and Ergastulum-Tag are quite comparable, each doing slightly better than the other in some cases. In particular, Minerva does better on the two *tpcd* workloads, while Ergastulum-Tag does better on the *filesystem* and *oltp* workloads. However, the added flexibility provided by Ergastulum's integrated RAID level selection allows Ergastulum-RLS to always generate solutions that are as good as Minerva, and often improve the solutions further.

The tools determine the RAID configuration for LUs in different ways. Minerva always explores the option of configuring arrays with LUs of mixed RAID levels initially. Ergastulum-Tag's choice of

RAID layout for its LUs is influenced by the order in which it initially assigns the stores. By default, it uses a *random-partition* ordering of stores: this results in all the RAID-5 stores being assigned first. Thus, Minerva tends to have more arrays containing LUs with different RAID layouts, while Ergastulum tends to have more arrays with all LUs using the same RAID layout. The *filesystem* and *oltpSplit* workloads demonstrate that this can result in Ergastulum using less disks than Minerva.

The tools also differ in their flexibility in changing device configuration and exploring store assignment alternatives. During one stage of execution, Minerva judges the number of LUs that it needs based on aggregates of store capacities and utilizations. During a separate stage of execution, Minerva's assignment algorithm then packs as many low cost stores as it can into the first arrays but has less options left when assigning the higher cost stores subsequently. Ergastulum-Tag incrementally adds LUs to the configuration as it needs them during the process of assigning stores. It explores more alternatives than Minerva, informed by the device performance models on each attempted store assignment, with greater flexibility since, unlike Minerva, it can change the device configuration dynamically. The result is that Ergastulum-Tag can pack stores more efficiently than Minerva. Thus, Ergastulum-Tag is able to pack the *filesystem* workload into LUs to consistently obtain around 90% utilization, whereas Minerva distributes the workload more, obtaining only about 60% utilization for three of the five arrays used.

Comparing Minerva with Ergastulum-RLS enables reveals the additional flexibility introduced into Ergastulum by RAID level selection. Minerva selects the RAID level to be used for the given stores, independently of their placement on devices: the store assignment stage must then accommodate these decisions. Ergastulum-RLS can make and revise RAID level decisions during assignment of the workload. Ergastulum-RLS's ability to swap LUs between RAID5 and RAID1/0 enables it to explore solutions that Minerva is incapable of considering. This is demonstrated in the lower-cost configurations produced for the *filesystem-cap* and *scientific3* workloads. In the previous solutions, the respectively capacity-bound and highly-sequential workloads were accommodated on all RAID-5 LUs. For *filesystem-cap*, Ergastulum-RLS identifies one LU that would be better as 4-disk RAID-1/0, enabling it to beat previous solutions by the cost of one less disk. For *scientific3*, it sets one of the LUs on each of four arrays to RAID-1/0, enabling it to beat previous solutions by the cost of four less disks.

While Minerva accommodates the higher-capacity stores of the *filesystem* and *oltpSplit* workloads on RAID-5 LUs, Ergastulum-RLS determines that it is able to pack all stores of these workloads on RAID-1/0 LUs. This enables it to use one less LU than previous solutions for *filesystem* and one less array for *oltpSplit*.

Minerva handles the overlapping streams of *tpcd-2x* and *tpcd-4x* slightly better than Ergastulum. However, Ergastulum-RLS is able to compensate sufficiently to still improve upon their Minerva-generated system-cost. Ergastulum-RLS's greater use of RAID-1/0 enables it to beat Minerva's solution cost for *tpcd-2x* and *tpcd-4x*. For the *tpcd-4x* workload, the use of RAID-1/0 saves seven disks across the configuration, reducing the system cost to below that for Minerva, despite the use of one more LU.

4.2 Comparison with IP design tool

One way to assess the quality of Ergastulum solutions is to use integer-linear programming (IP) [22] to generate optimal solutions. Integer programming uses branch and bound techniques to eventually find an optimal configuration. Unfortunately, IP cannot be used to solve the full storage system design problem because the device performance models can't be converted into integer-linear constraints. Therefore, we restrict our evaluation to a simpler utilization model which allows an IP formulation.

To simplify the IP formulation, we make the following assumptions: a) all disks have the same capacity, b) all I/Os are read-only, c) only one stream is associated with each store, and d) all LUs are RAID-1/0. We also use the simpler stream-interaction formula as described in [5].

The simplified interaction formula is as follows: Conceptually, when two streams access data (stores) from the same disk, the load that they present to the

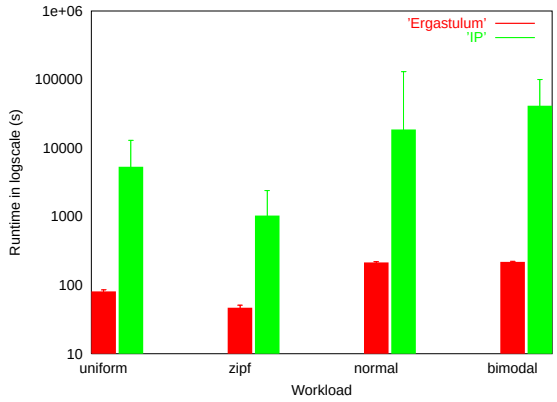


Figure 6: Comparison of the IP design tool and Ergastulum execution time. Ergastulum generated optimal solutions 80%–90% of the time and was never more than 2.5% from optimal. Ergastulum is about 100 times faster than the IP design tool.

disk is complicated to determine. For instance, if two streams access data from a disk at an average rate of 10MB/s, the net rate that the disk sees could be either 20MB/s if the streams are "on" at the same time, 10MB/s if only one of them is on at any time, or a different number if they overlap in activity. This notion of overlapping streams is characterized by the overlap fraction which was first used in Minerva [2]. In general, the expression used to calculate the net rate from the individual rates is quite complex [30].

The IP design tool supports both simple disks with capacity and utilization constraints, and a restricted model of the Hewlett-Packard SureStore E Disk Array FC-60 [15]. Fully-configured, an FC-60 contains 6 disk enclosures (or trays) containing a total of 60 9, 17, 36 or 73GB disks, one controller enclosure containing two controllers, each with 512 MB of NVRAM cache, and one 40MB/s Ultra SCSI connection between the controller enclosure and each of the six disk enclosures. LUs in an FC-60 can be created using RAID-1/0 and RAID-5 with arbitrary numbers of disks. The IP design tool uses only the 17GB disks, and the RAID-1/0 LU type.

4.2.1 Experiments

We carried out three different types of experiments in increasing order of complexity. They are:

1. *Disk with capacity only:* All disks have the same capacity, store assignments must not violate capacity constraints.
2. *Disk with capacity and utilization:* Extending 1 by adding utilization constraints to each disk.
3. *FC-60:* Using the simplified FC-60 model described above.

Workload	Store count	Store capacity	Random 4k reads/s
1	8	1/2 GB	100-130
2	100	1/4 GB	7.7-8.6
3	200	1/4 GB	3.8-4.3

Table 1: Synthetic workloads on FC-60 arrays. These workloads are similar to the synthetic workloads used in [4]

The capacity-only disk packing problem is a 1-dimensional bin-packing problem. We used 17 GB disks with four different store size distributions:

1. **uniform**: 500 stores with capacity uniformly distributed in (4MB, 4GB)
2. **zipf**: 500 Zipf distributed stores with Zipf parameter 0.8 and range (0.8GB, 17GB)
3. **normal**: 500 normally distributed stores with a mean of 17GB/10, and a standard deviation of 17GB/80.
4. **bimodal**: The union of two normal distributions: 200 stores obeying normal distribution $N_1(17\text{GB}/4, 17\text{GB}/80)$ and 300 stores obeying normal distribution $N_2(17\text{GB}/6, 17\text{GB}/80)$.

We ran each of the experiments 10 times to calculate a mean, lower, and upper bound. Ergastulum found the optimal solution in 80% - 90% of the cases, and otherwise was within 2.5% of the optimal design cost. Figure 6 shows (in log scale) that Ergastulum is usually about 100 times faster than the IP design tool.

We next hand-crafted a set of workloads for which the best-fit 1-dimensional bin packing algorithm is known to work poorly: a scalable workload with four sets of stores: $17\text{GB}/4 \pm i \times 256$, $i = 0, 1, \ldots, 2n-1$, and $17\text{GB}/6 \pm i \times 256$, $i = 0, 1, \ldots, 3n-1$. The only optimal solution is to assign each $\pm$ pair of stores on the same disk, which would lead to a solution using $2n$ disks. For $n = 10$ (20 disks optimally), the Ergastulum design tool finds a 21 disk solution in 3.98 seconds, the IP design tool finds the same solution in 9 seconds, and takes 67 hours to find

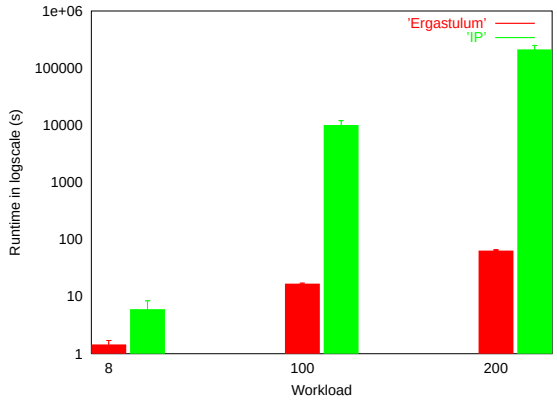


Figure 7: Execution time comparison for the three workloads shown in Table 1. Ergastulum generated optimal solutions for all experiments. The error bar (mean, lower and upper bounds) were calculated based on 4 samples.

the optimal solution. With $n = 50$, Ergastulum finds a 106 disk solution whereas the optimal solution is 100 disks. For $n = 100$, Ergastulum uses 211 disks whereas the optimal solution is 200 disks. Thus, we can see that Ergastulum's solution is approximately 5% worse than the optimal solution.

Using both capacity and utilization constraints on single disks, we created a workload of 100 stores, where the store capacity is uniformly distributed in (4MB, 10GB), and the stream for each store has a request rate uniformly distributed between 0 and 50 random 4K requests/second. A single disk can do about 100 random 4K requests/second. Ergastulum found a solution in 11 seconds, the IP design tool took 24 days to find the same solution.

Finally, we evaluate Ergastulum with a set of synthetic workloads on the simplified FC-60 array, as summarized in Table 1. Each experiment was run 4 times. For all these workloads, the Ergastulum design tool produced the optimal solution, as did the IP design tool. Figure 7 shows the running time of Ergastulum versus the IP design tool for the three workloads. For the largest problem, Ergastulum took 60 seconds and the IP design tool took 60 hours.

4.3 Solution quality for real models

Determining the quality of solution generated by a heuristic for an at-least NP-hard problem is very difficult. However, we can estimate the quality of solutions by sampling a large collection of random designs. With the appropriate design tool options, the final design cost is determined entirely by the store order. Therefore, we can explore the overall search space by generating random orders for the stores, and performing just initial assignment. By executing this process many times, we can limit the probability that we missed a better design. The specific requirements on the design tool can be found in [5]; in essence all configurations have to be reachable, and adding stores to a device should not decrease the utilization.

We observe that if we had an optimal configuration, we could order the stores by a prefix traversal of the final device tree, and the design tool would regenerate the same configuration. We require three additional properties to make that statement true. First, for set of stores, the device models should predict that all subsets have less utilization than the full set. Unfortunately our device models do not exhibit this property, but they are hopefully close enough. Second, the stores need to be in prefix order, in particular, it should not be possible to move a store to a leaf earlier in the traversal. It is trivial to transform an optimal solution into one obeying this property; if the stores are not in this order, the same configuration will not be re-generated, but the system cost may still be the same. Third, the design tool needs to not get "stuck" at any level of the device tree; the search algorithm needs to search through all possible values of each node at each level for each store. A simple way to violate this property would be to run the design tool with two raid levels, but with transformation functions disabled.

We performed the experiments using both FC-30 (see Section 4.1) and FC-60 (see Section 4.2) models, using nine different device options, as shown in Table 2. We used 20 different workloads, and ran about half a million individual experiments. Additional results are available [5]. We stopped running experiments with a given workload-option pair if we had run at least 500 experiments that included at least 100 experiments for each unique system cost found for that workload-option pair. The workloads used for these experiments are all from automatically generated traces of real systems.

- *tpcd-4x* From the Minerva experiments (see Section 4.1).

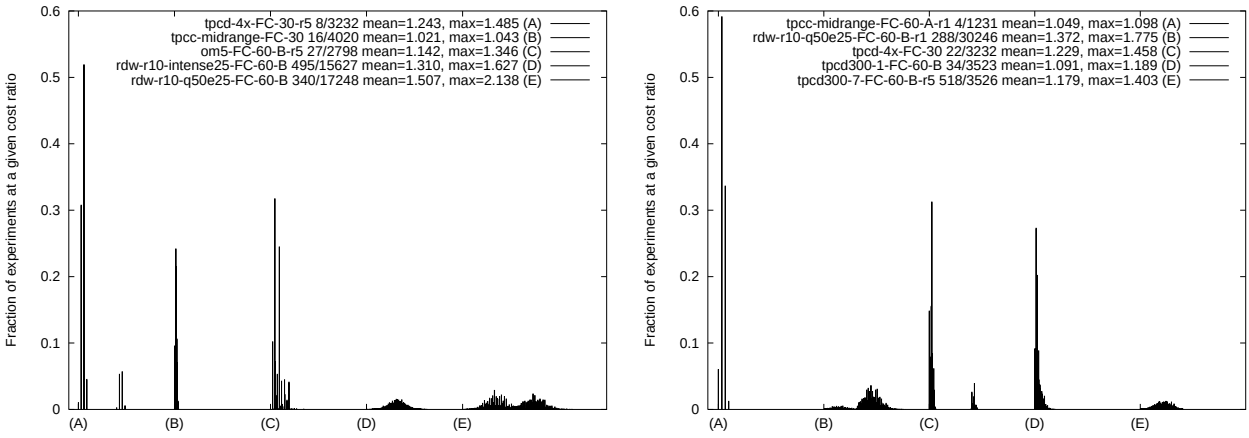


Figure 8: Subset of results of running thousands of random experiments with different workloads and device parameters. Designs with the same total system cost are considered the same. All costs are calculated relative to the smallest cost found in the random experiments, so the leftmost column in each group (at the center of the label) has the minimal cost. The number of different costs found, total number of runs, mean relative cost, and the maximum relative cost are shown in the captions.

- *tpcc-midrange* A midrange TPC-C configuration using one array and a two processor server. The database was performing about 16.5K tpmC.
- *om5* Trace from an OpenMail email server. This server had about 2000 active users.
- *rdw-*$\{$*intense25,q50e25*$\}$ Trace of a 1 TB retail data warehousing workload. The *intense25* variant has 25 intense (15+ minute by themselves) simultaneous queries. The *q50e25* variant has 50 quick (few seconds individually), and 25 each (tens of seconds individually) simultaneous queries.
- *tpcd300-*$\{$*1,7*$\}$ Trace of queries one and seven of a TPC-D configuration at the 300GB scale factor.

For the paper, we have selected 10 interesting and representative results. In all of the results, designs are grouped by system cost. We can learn a number of different things from these graphs. First, the storage system design problem is very complex; some of the experiments generate hundreds of distinct system costs. Some problems, such as *tpcd300-1-FC-60-B*, are easier as the results are heavily weighted toward the minimal cost configurations. Other problems, such as *rdw-r10-q50e25-FC-60-B*, are more difficult as the results are weighted toward more expensive configurations. Many of the runs generate normal or bi-modal distributions of system cost.

array	*LU type*	*disk size (GB)*	*name*
FC-30	RAID-1/0	4	-FC-30-r1
FC-30	RAID-5	4	-FC-30-r5
FC-30	RAID-1/0 + 5	4	-FC-30
FC-60	RAID-1/0	9	-FC-60-A-r1
FC-60	RAID-5	9	-FC-60-A-r5
FC-60	RAID-1/0 + 5	9	-FC-60-A
FC-60	RAID-1/0	9, 18, 36, 72	-FC-60-B-r1
FC-60	RAID-5	9, 18, 36, 72	-FC-60-B-r5
FC-60	RAID-1/0 + 5	9, 18, 36, 72	-FC-60-B

Table 2: Configurations explored in the initial assignment experiments.

Second, just randomly generating designs is not likely to find a near-optimal design. In many cases, the best result found occurs with a probability less than 1%. Fortunately, as we will see in the next section, reassignment gets to the lower-cost configurations fairly quickly.

Third, mistakes can be expensive. For the *rdw-r10-q50e25-FC-60-B* experiment, the ratio between the most expensive and the least expensive configuration found is about 2.1. For 46 of the 145 experiments, the ratio between the most expensive and the least expensive configuration is over 1.2.

4.3.1 All graphs

The following graphs, created using the same process described above include all of the interesting experiments that we have run with the design tool. We considered an single cost in an experiment interesting if it occurred at least 1% of the time, or had a cost less than the mean cost. We considered an experiment interesting if there were at least two interesting costs.

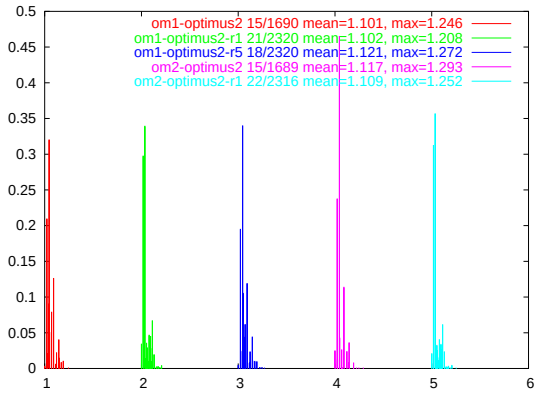


Figure 9: Results for all interesting experiments presented with the same structure as figure 8

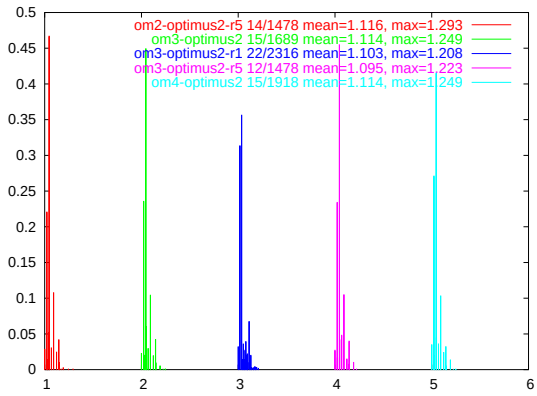


Figure 10: Results for all interesting experiments presented with the same structure as figure 8

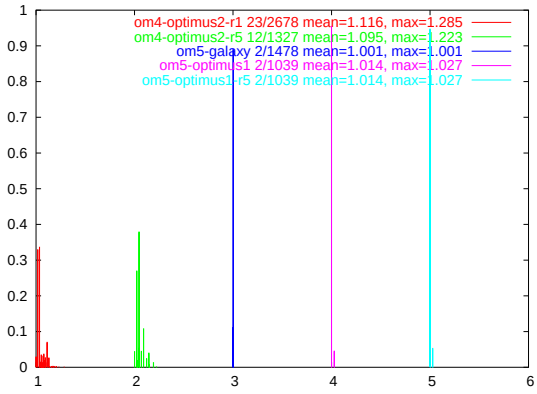


Figure 11: Results for all interesting experiments presented with the same structure as figure 8

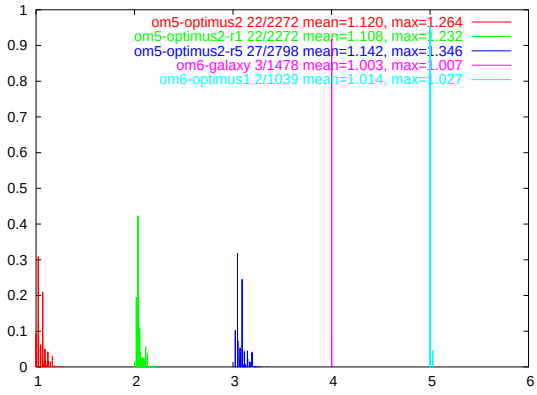


Figure 12: Results for all interesting experiments presented with the same structure as figure 8

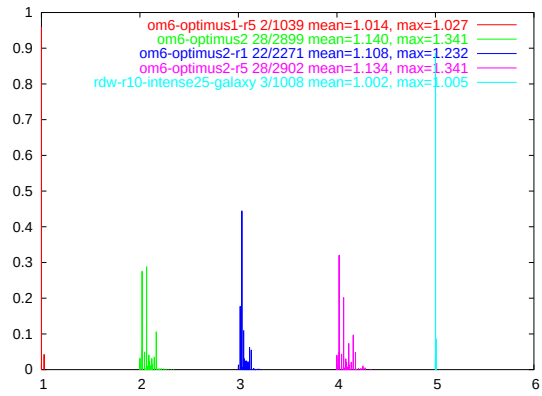


Figure 13: Results for all interesting experiments presented with the same structure as figure 8

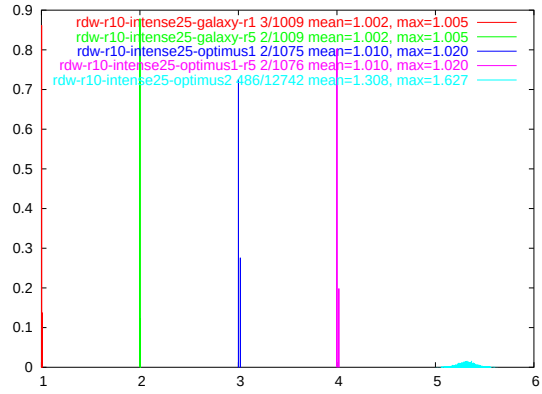


Figure 14: Results for all interesting experiments presented with the same structure as figure 8

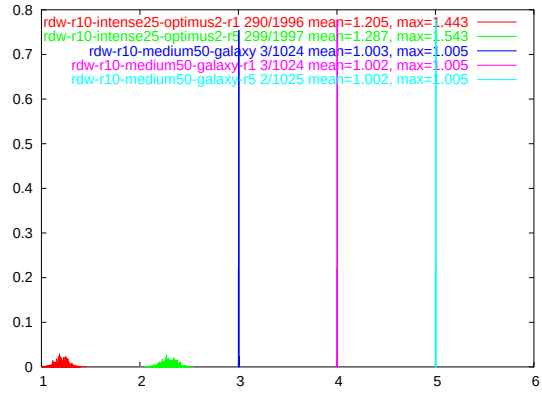


Figure 15: Results for all interesting experiments presented with the same structure as figure 8

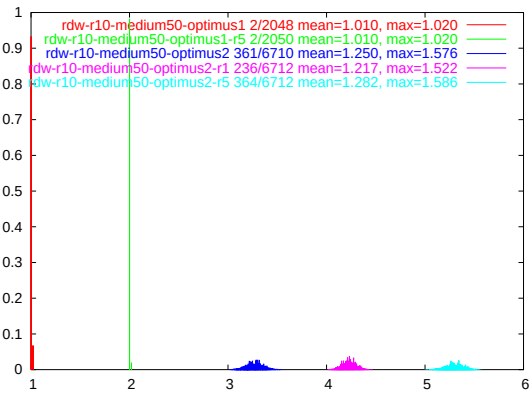


Figure 16: Results for all interesting experiments presented with the same structure as figure 8

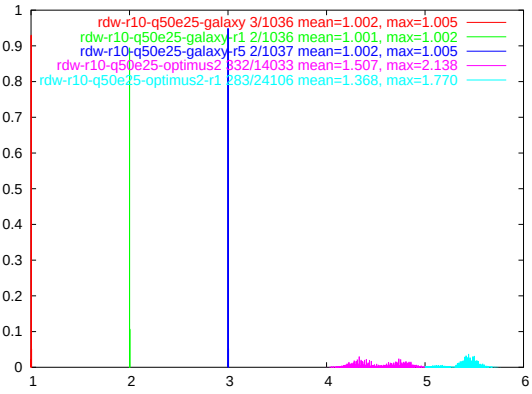


Figure 17: Results for all interesting experiments presented with the same structure as figure 8

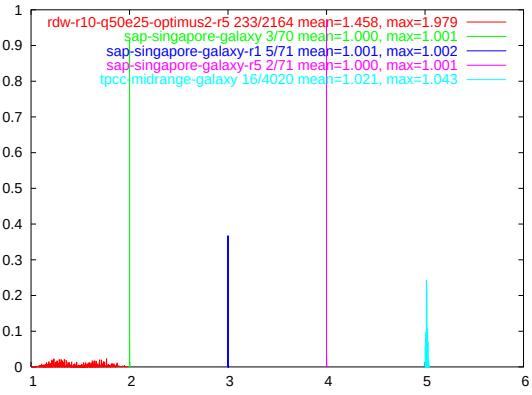


Figure 18: Results for all interesting experiments presented with the same structure as figure 8

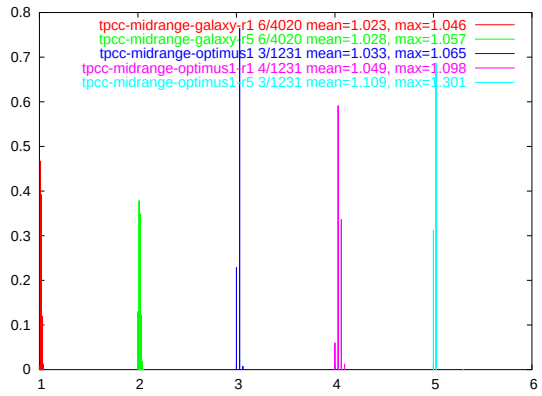


Figure 19: Results for all interesting experiments presented with the same structure as figure 8

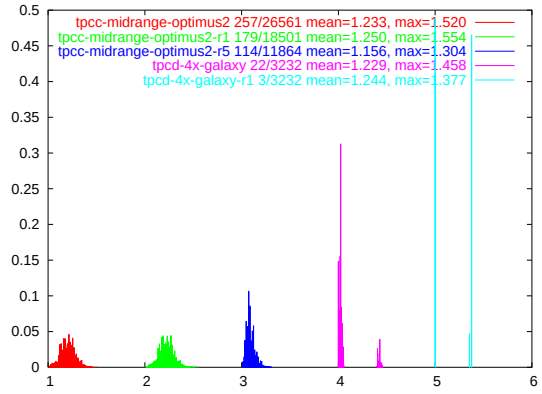


Figure 20: Results for all interesting experiments presented with the same structure as figure 8

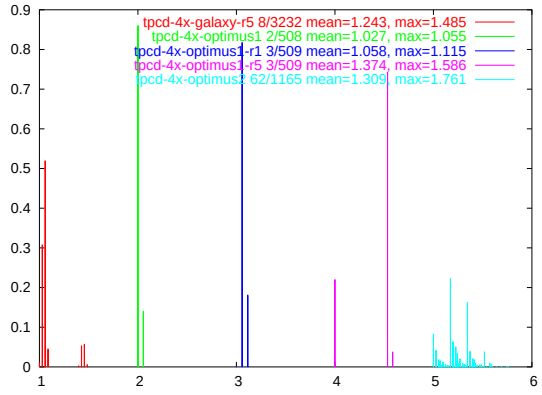


Figure 21: Results for all interesting experiments presented with the same structure as figure 8

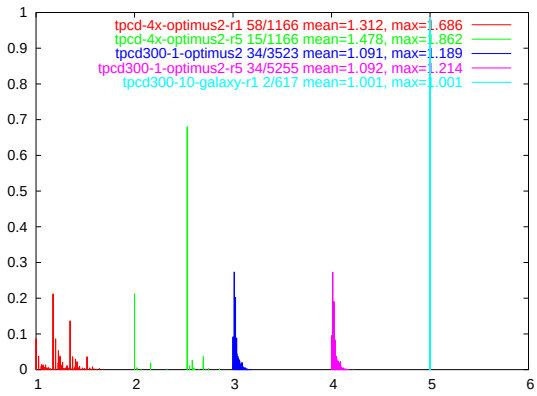


Figure 22: Results for all interesting experiments presented with the same structure as figure 8

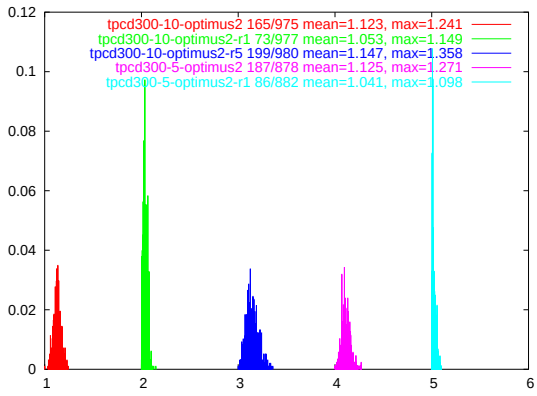


Figure 23: Results for all interesting experiments presented with the same structure as figure 8

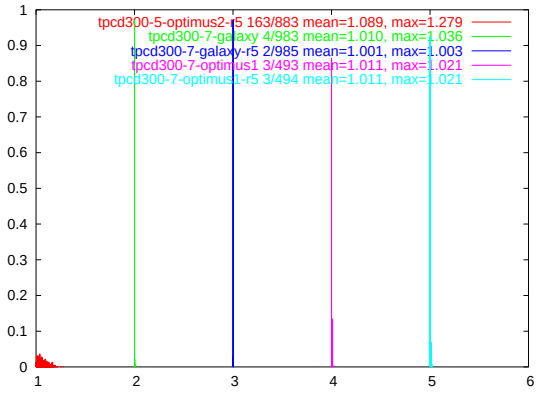


Figure 24: Results for all interesting experiments presented with the same structure as figure 8

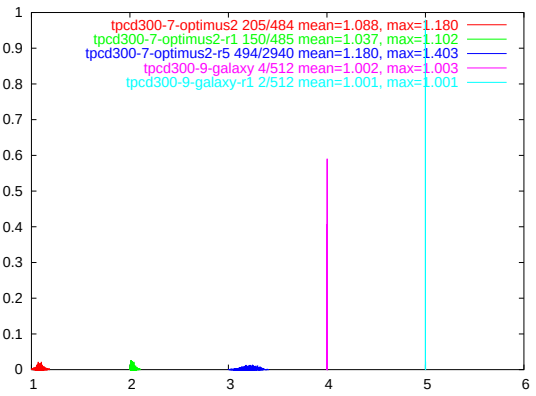


Figure 25: Results for all interesting experiments presented with the same structure as figure 8

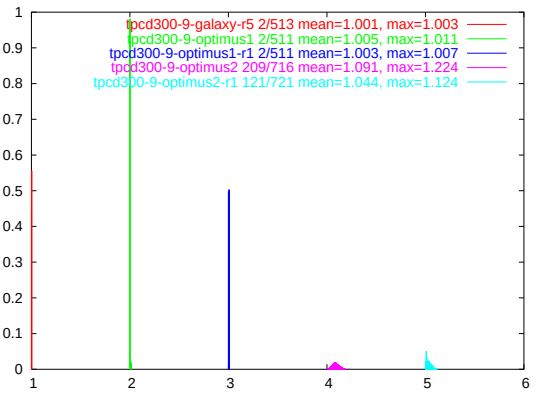


Figure 26: Results for all interesting experiments presented with the same structure as figure 8

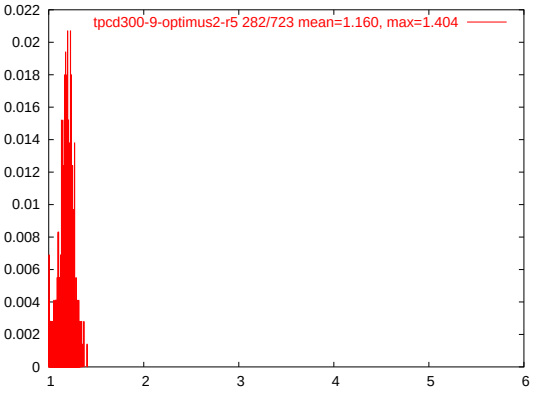


Figure 27: Results for all interesting experiments presented with the same structure as figure 8

4.4 Solution quality with reassignment

As demonstrated in the previous section, stopping after generating a random design is not likely to generate a good design. We now explore how well reassignment improves the quality of the initial assignment. We use two of the initial assignments for each distinct cost from the experiments described in Section 4.3.

We then ran 50 rounds of reassignment, using a few different reassignment functions, and a minimum number of LUs to reassign. All of the reassignment functions select the cheapest path. If the two designs are the same, then the *MaxAvgAll* function selects the path with the higher mean utilization of all attributes, and the *MinAvgAll* function selects the path with the lower mean utilization. Both of these functions consider paths that differ by less than 5% to be the same. *Delta* selects the path with the lower increase in utilization, and if the increase is the same to within 0.1%, then it uses the *MaxAvgAll* function. We have varied the minimum number of LUs to reassign between 1-5, so an experiment named *3xMaxAvgAll* means that three LUs were removed in each reassignment round, and the *MaxAvgAll* comparison function was used. We chose 50 rounds arbitrarily based on earlier experience that most of the benefit occurs in the first few rounds, but occasionally the design is improved in later rounds. As with the random store order experiments, we have run many more experiments than we have room to show here. Additional results are available in the technical report [5].

For each reassignment rule, and for each round, we plot the current min, mean, and max system cost derived from reassigning each of the initial assignments. We can learn many things the subset of the results shown in Figure 28. First, the very expensive designs are eliminated fairly quickly by almost all of the reassignment functions. Second, figure (a) shows that using *MaxAvgAll* rather than the *MinAvgAll* usually results in better final costs, although in the first few rounds, *MinAvgAll* does better; this result is why the *Delta* comparison uses *MaxAvgAll* rather than *MinAvgAll*. Third, figures (b) and (c) show that no single LU reassignment count is best. Fourth, the mean system cost continues to drop as the number of reassignment rounds increases. Fifth, figure (c) shows that reassignment may find a slightly better configuration than was ever discovered by running all of the random seeds.

MaxAvgAll tends to do better than *MinAvgAll* because with *MinAvgAll*, the design tool balances load across all the available devices, so it is less likely that a design will be generated which can remove an LU. Conversely, with *MaxAvgAll*, load is concentrated on a few of the devices, so one LU may eventually free up, allowing it to be removed.

4.4.1 All reassignment graphs

The following graphs, created using the same process described above include all of the reassignment experiments that we have run with the design tool. Because we present more reassignment functions than described above, we show both the statistics graphs with min, mean, and max all on the same graph, and the results with mean, max, and min split out. In some cases, the experiments are not complete; Currently the tpcd-4x-galaxy-r5, tpcc-midrange-galaxy, om5-optimus2-r5, tpcc-midrange-optimus1-r1, and tpcd-4x-galaxy runs are complete; the rest include a random sampling of starting price categories.

4.5 Scaling Ergastulum

The demand for greater automation of storage management grows as storage systems become increasingly large. Thus, a series of experiments have been done to evaluate how well Ergastulum scales as the workloads is has to assign get larger. Scaling has been done on the number of stores and the number of streams in the workloads. The effects on both runtime and memory usage are presented.

4.5.1 Workloads

These experiments have been done with a set of scalable workloads.

- *fixed stores-only* is a workload of stores, with no streams, where each store has a fixed capacity of 2GB.

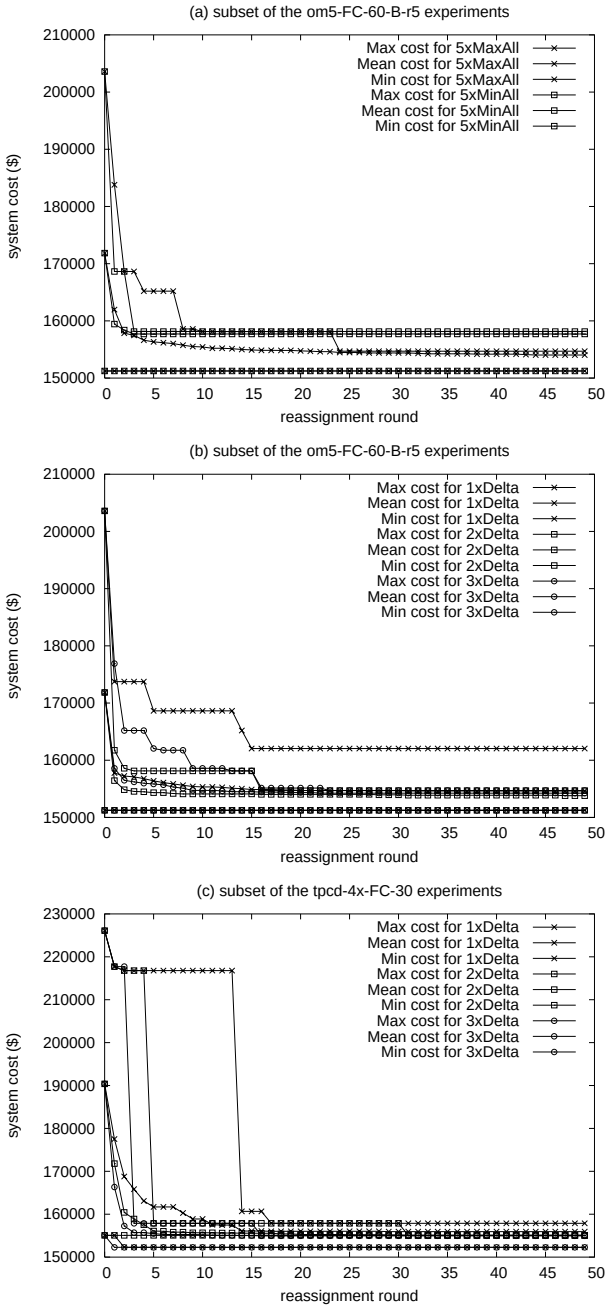


Figure 28: Reassignment round vs. max, mean, min system cost, starting with various initial assignments and using many different reassignment rules. All figures show that very bad configurations are usually eliminated quickly. Figure (a) shows that *MaxAvgAll* is better than *MinAvgAll*. Figures (b) and (c) show that the best reassignment rule is not clear; *2xDelta* is the best rule for (b), but *3xDelta* is the best rule for (c). Figure (c) shows that reassignment can find better configurations than initially discovered using random seeds. The max, mean and min for each experiment are plotted with the same symbol; the mean is always between the max and the min.

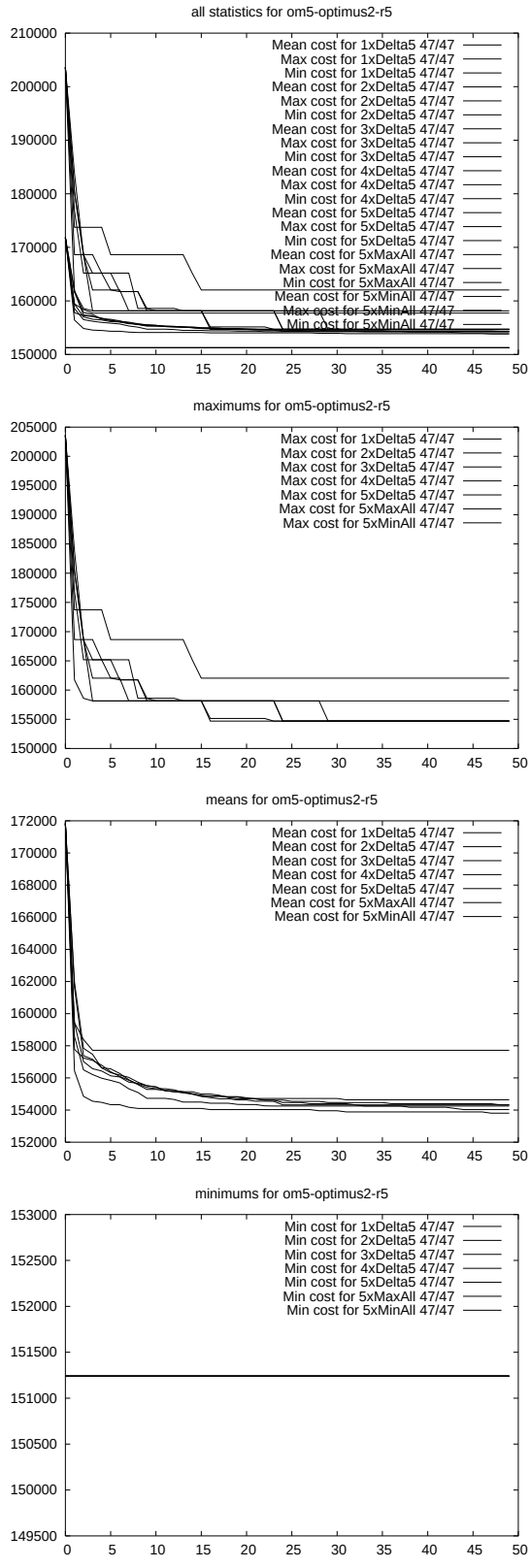
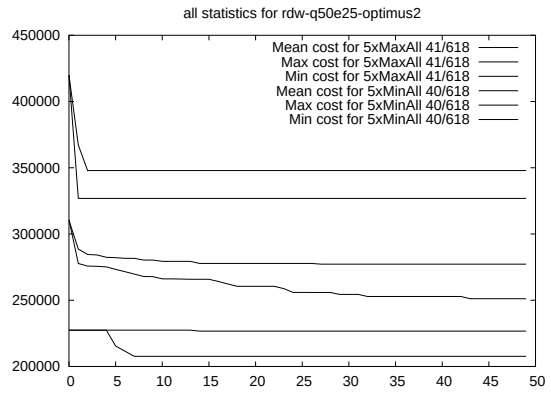
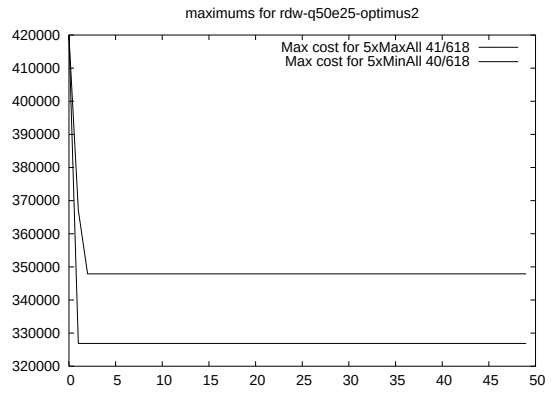
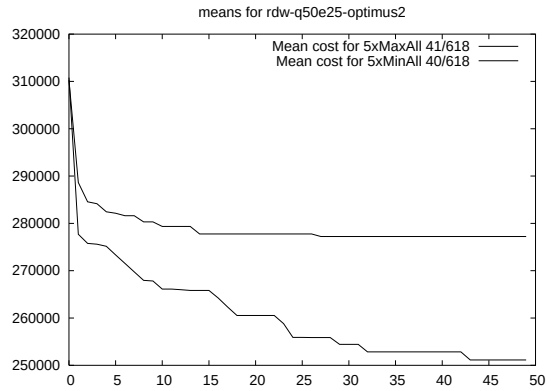


Figure 29: Reassignment graphs for om5-optimus2-r5







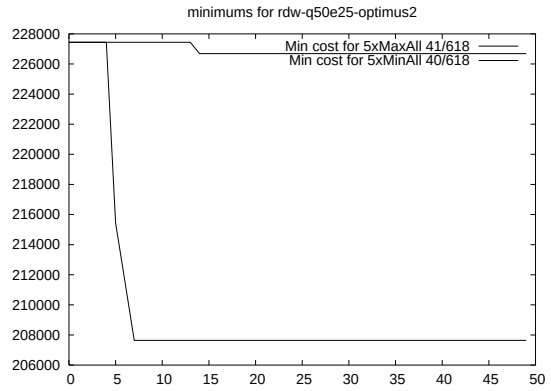
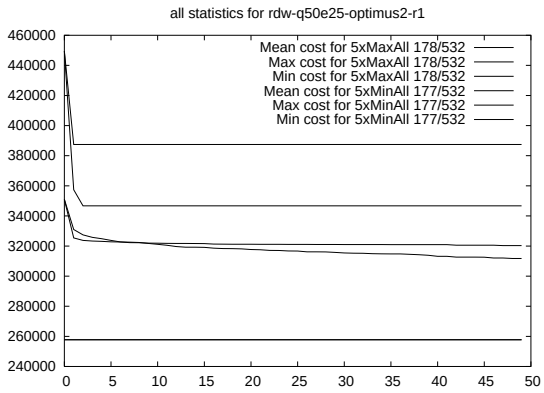
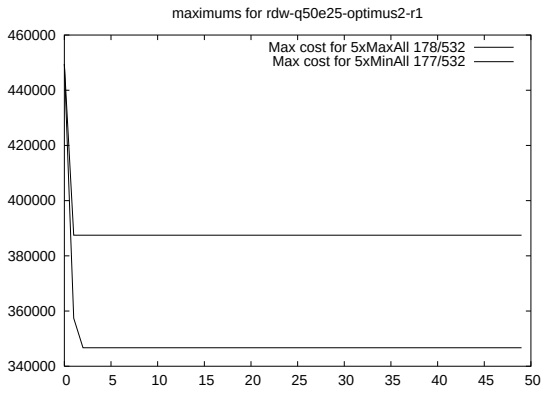
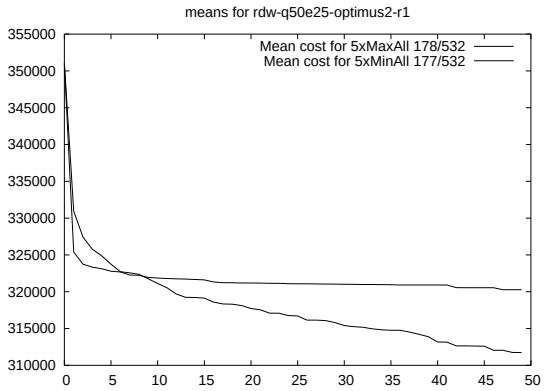


Figure 30: Reassignment graphs for rdw-r10-q50e25-optimus2







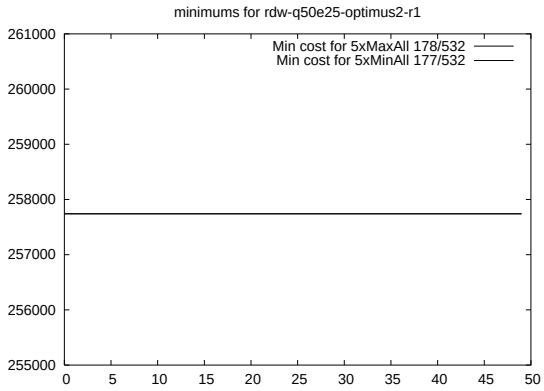
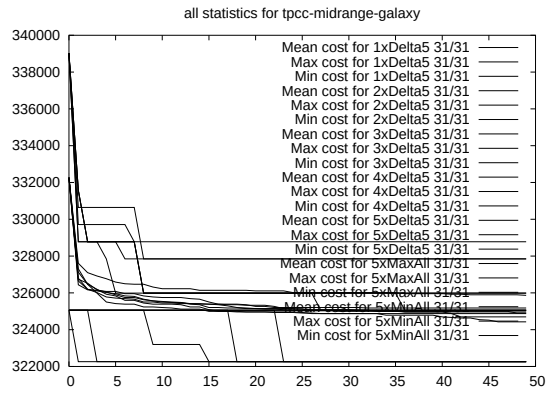
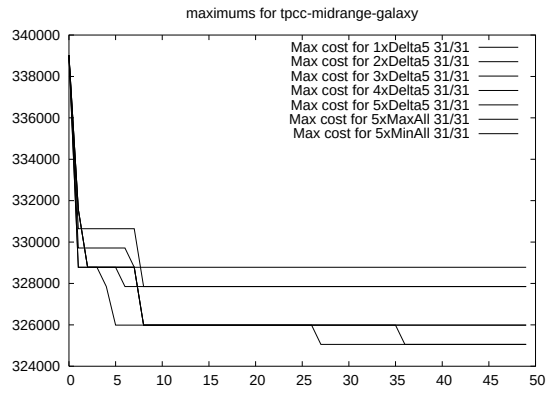
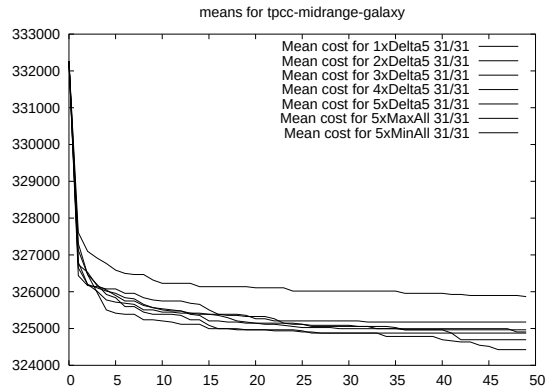


Figure 31: Reassignment graphs for rdw-r10-q50e25-optimus2-r1







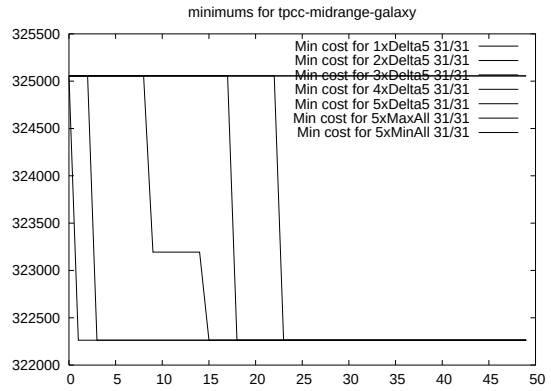
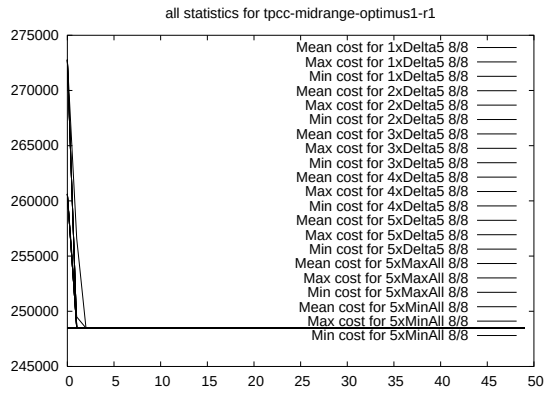
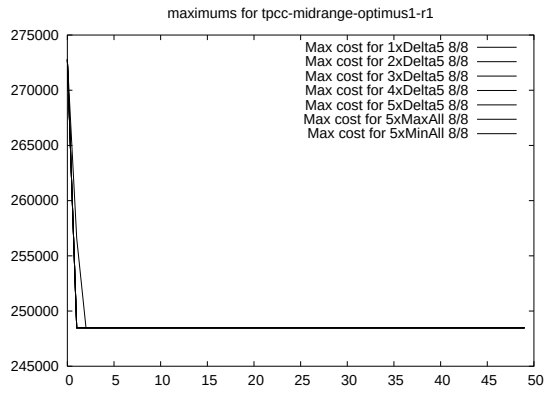
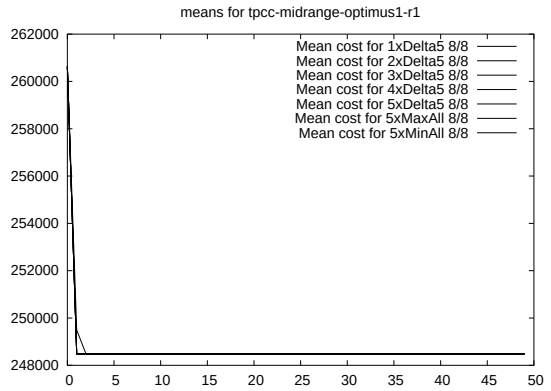


Figure 32: Reassignment graphs for tpcc-midrange-galaxy







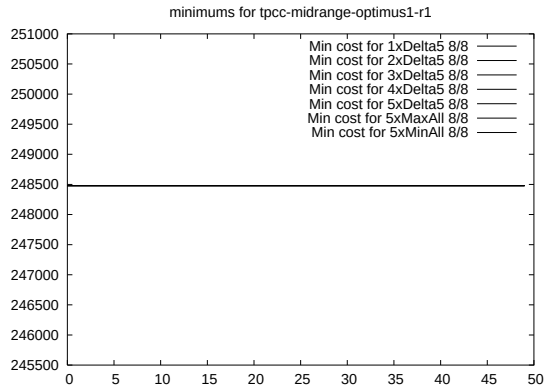
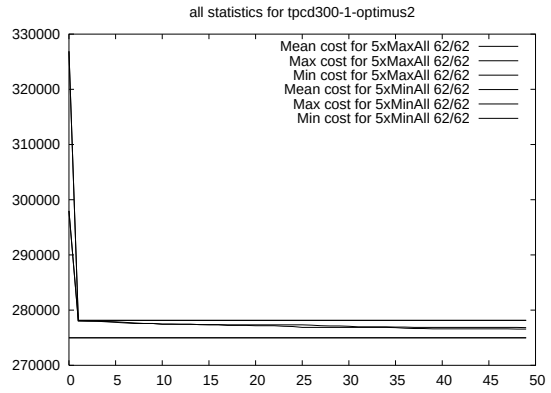
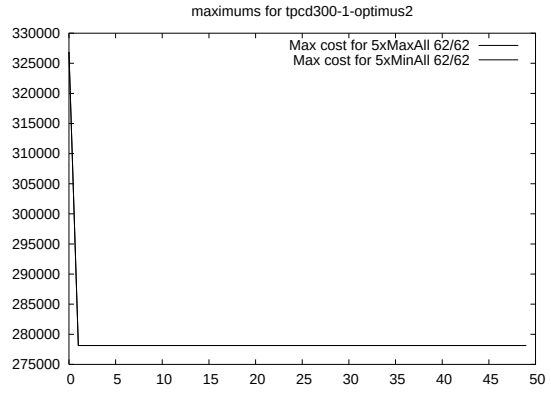
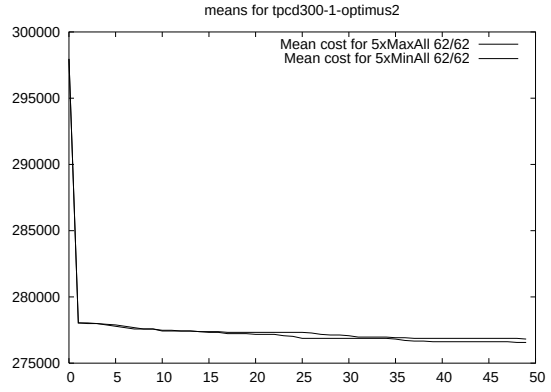


Figure 33: Reassignment graphs for tpcc-midrange-optimus1-r1







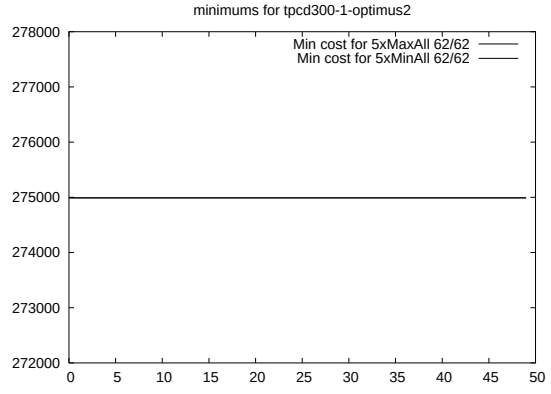
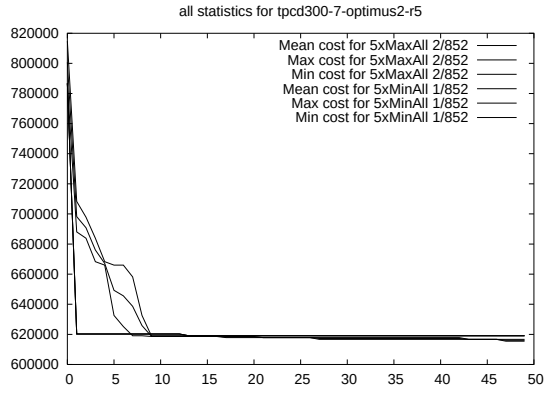
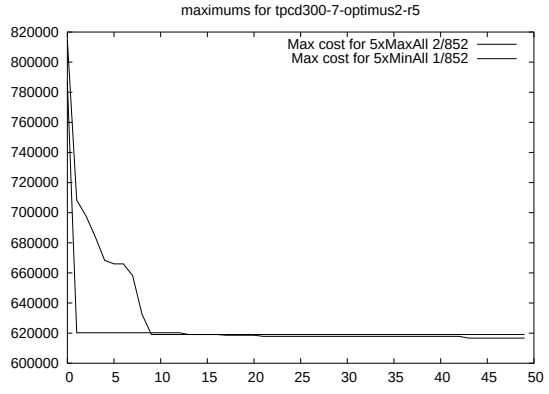
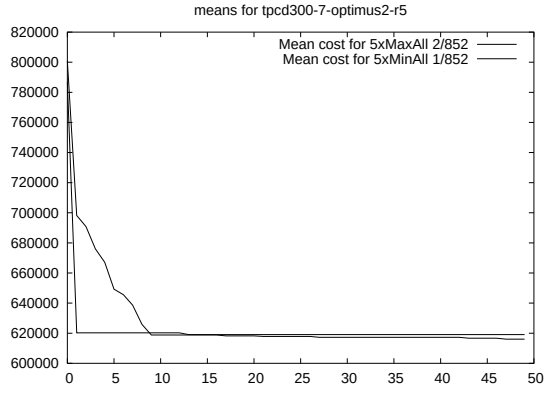


Figure 34: Reassignment graphs for tpcd300-1-optimus2







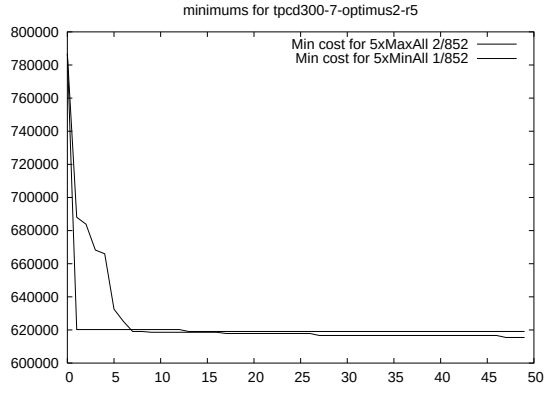
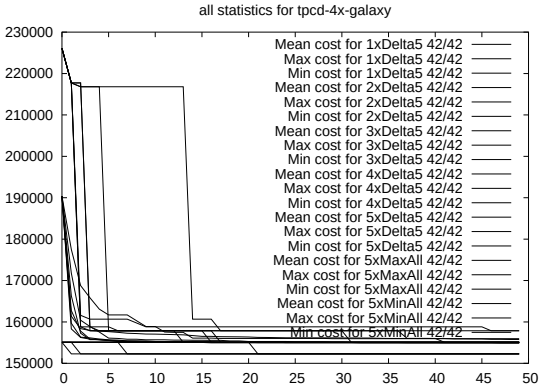
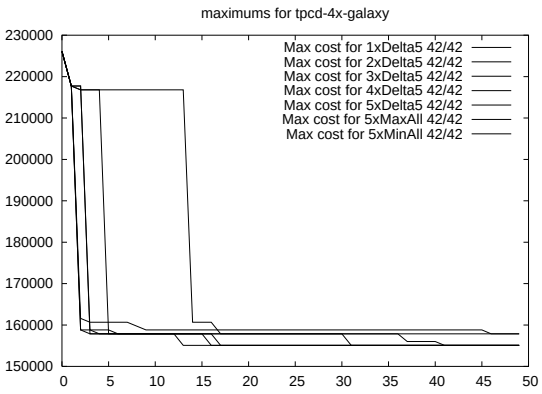
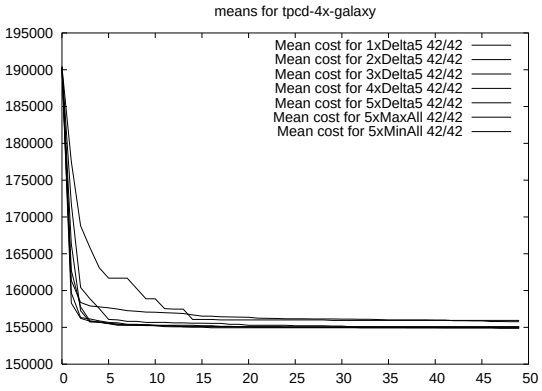


Figure 35: Reassignment graphs for tpcd300-7-optimus2-r5







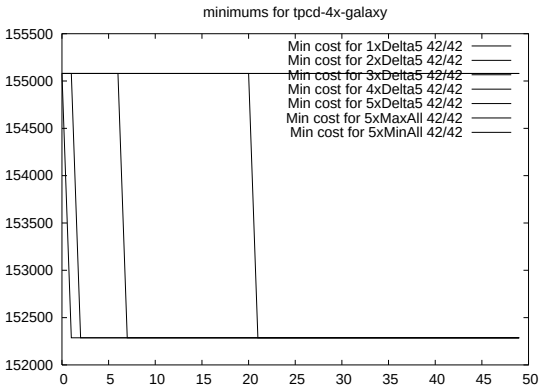
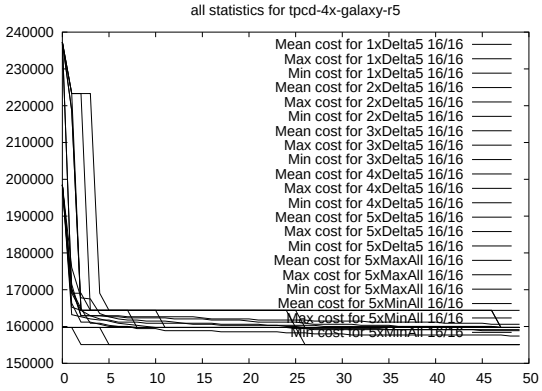
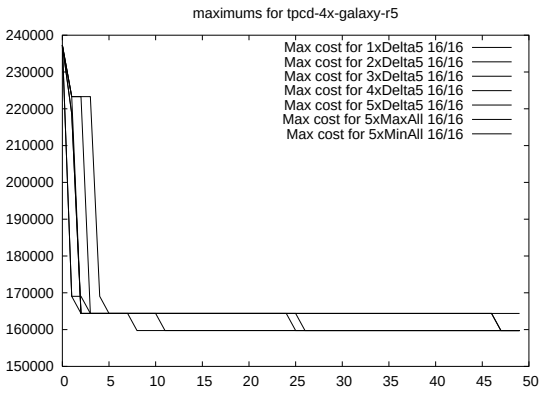
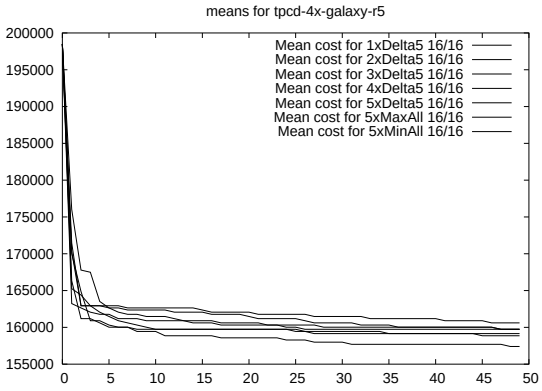


Figure 36: Reassignment graphs for tpcd-4x-galaxy







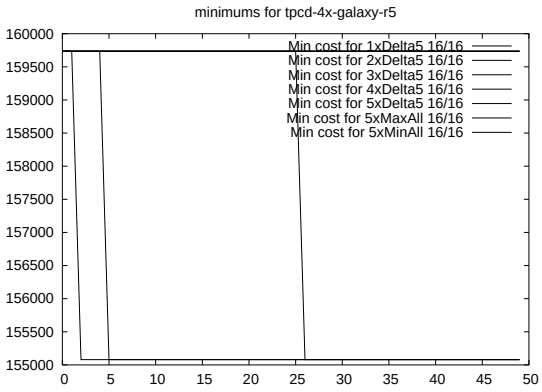


Figure 37: Reassignment graphs for tpcd-4x-galaxy-r5

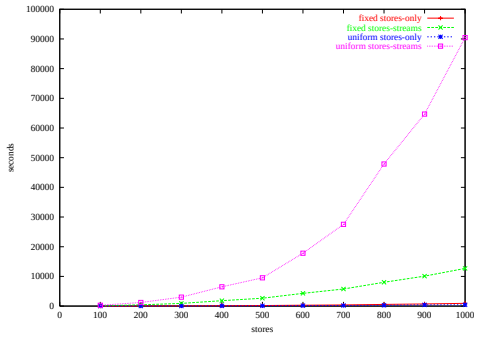


Figure 38: Execution time comparison

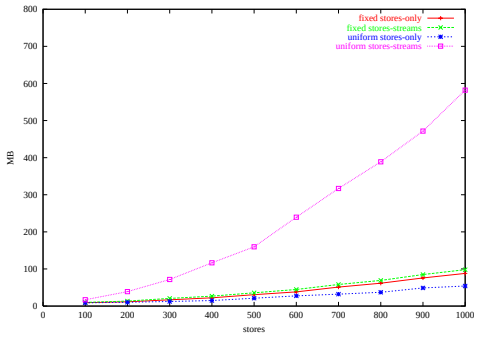


Figure 39: Memory usage comparison

- *fixed store-stream* is a workload of stores with streams, where each store has a fixed capacity of 2GB and one read and write stream is associated with each store. Request rate is fixed at 10 reads per second and 1 write per second. Request size is fixed at 20 KB for reads and 10 KB for writes. Streams are always on.

- *uniform stores-only* is a workload of stores, with no streams, where each store has a capacity taken from a uniform distribution between 0.25 GB and 2 GB.

- *uniform store-stream* is a workload of stores and streams, with one read and write stream associated with each store. Each store capacity is taken from a uniform distribution between 0.25 GB and 2 GB. Each request rate is taken from a uniform distribution of 0-40 reads per second and 0-20 writes per second. Each request size is taken from a uniform distribution of 0.5 MB for reads and 0.125 MB for writes.

4.6 Time scale

Ergastulum's runtime scales as the number of stores and streams is increased, as illustrated in Figure 38.

4.7 Memory scale

Ergastulum's memory usage scales as the number of stores and streams is increased, as illustrated in Figure 39.

5 Related work

Buzen [8] proposed a capacity planning tool in which workloads have types; possible types are time, sharing, transaction processing and batch processing. The requirements of each type are assumed to be the same. Devices are modeled by grouping devices into groups where only one of the groups can service a request at once. Ergastulum is a more versatile tool, free of simplifying assumptions such as all workloads have same requirements and devices have to be grouped.

Hill [17] developed and patented a scheme for the assignment of a workload to devices, based on attributes of I/O rate and capacity. It is not clear how well Hill's scheme would handle the more complex attributes needed for modeling storage systems.

Shriver [26] proposed a formalization of the storage design problem that models workload units with various attributes, objective functions for specifying goals and constraints on devices that can contain workload units. Forum [7] implements that formalization and handles storage design for independent disk systems as a multi-dimensional constrained bin-packing optimization problem using an adaptation of the complex Toyoda [28] bin-packing heuristic. Minerva [2] extends Forum to handle disk arrays. The comparison with Minerva, and hence a superset of Forum can be found in Section 4.1 where we show that Ergastulum is faster and generates as good or better designs.

Existing solutions to the file assignment problem [11, 35] use heuristic optimization models to assign files to disks to get improvements in I/O response times. The file allocation schemes described

in [12, 24] automatically determine an optimal stripe width for files, and stripe those files over a set of homogeneous disks. They then balance the load on those files based on a form of "hotspot" analysis, and swapping file blocks between "hot" and "cold" disks. Ergastulum can select the appropriate number of devices to use, supports RAID systems, and uses far more sophisticated performance models to predict the effect of system modifications.

The AutoAdmin index selection tool [10] can automatically "design" a suitable set of indexes, given an input workload of SQL queries. It has a component that intelligently searches the space of possible indexes, similar to Ergastulum, and an evaluation component (model, in Ergastulum terms) to determine the effectiveness of a particular selection based on the estimates from the query optimizer.

Muse [9] controls server allocation and energy-conscious, adaptive resource provisioning for Internet hosting centers. Unlike Ergastulum, it focuses on allocating computational resources. Its resource allocation framework is based on an economic model that factors in the trade-offs between the service quality and the cost.

The Appia [32] system automatically generates network fabric designs. It uses significantly simpler performance models, and substantially different heuristics because there is less natural structure present to guide the search. The use of randomized reassignment might be beneficial to Appia, as it has been shown to help Ergastulum substantially.

6 Conclusion

In this paper, we have presented Ergastulum, a new and efficient way of solving the storage configuration problem. We have highlighted interesting features of Ergastulum to show that it is highly flexible and can be used in different scenarios. We have also established that Ergastulum produces near-optimal solutions and is substantially faster than other approaches. In summary, we believe that Ergastulum is a novel way to quickly solve the storage configuration problem.

In the future we plan to extend Ergastulum to handle new disk arrays, and verify that the techniques we have described can correctly handle the additional complexity posed by those arrays. We are considering extending Ergastulum outside of the storage system domain; in particular, we believe that we could configure and map application host requirements onto appropriate hosts, so that Ergastulum could handle both the storage and host configuration in a data center. Finally we are examining extensions to Ergastulum that allow us to handle multiple data centers.

References

[1] N. Allen. Don't waste your storage dollars: what you need to know. Research note, Gartner Group, March 2001.

[2] G.A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. Becker-Szendy, R. Golding, A. Merchant, M. Spasojevic, A Veitch, and J. Wilkes. Minerva: an automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, November 2001.

[3] E. Anderson. Simple table-based modeling of storage devices. Technical note, HPL-SSP-2001-4, HP Labs, July 2001. http://www.hpl.hp.com/research/itc/scl/ssp/papers/.

[4] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: running rings around storage administration. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, January 2002.

[5] E. Anderson, M. Kallahalla, S. Spence, R. Swaminathan, and Q. Wang. Ergastulum: an approach to solving the workload and device configuration problem. Technical Report HPL–SSP–2001–5, HP Labs, July 2001. http://www.hpl.hp.com/SSP/papers/.

[6] E. Anderson, R. Swaminathan, A. Veitch, G. Alvarez, and J. Wilkes. Selecting RAID levels for disk arrays. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, Monterey, CA, January 2002.

[7] E. Borowsky, R. Golding, A. Merchant, L. Schrier, E. Shriver, M. Spasojevic, and J. Wilkes. Using attribute-managed storage to achieve QoS, 1997.

[8] J. P. Buzen, R. P. Goldberg, A. M. Langer, E. Lentz, H. S. Schwenk, D. A. Sheetz, and A. Shum. BEST/1—design of a tool for computer system capacity planning. In *AFIPS National Computer Conference (NCC), S. P. Ghosh and L. Y. Liu, Ed.*, pages 447–455, June 1978.

[9] J.S. Chase, D.C. Anderson, P.N. Thakar, A.M. Vahdat, and R.P. Doyle. Managing energy and server resources in hosting centers. In *18th ACM Symposium on Operating System Principles (SOSP'01)*, pages 103–116, Chateau Lake Louise, Banff, Canada, October 2001.

[10] S. Chaudhuri and V. Narasayya. AutoAdmin "What-if" index analysis utility. In *SIGMOD International Conference on Management of Data*, pages 367–378, June 1998.

[11] L. W. Dowdy and D. V. Foster. Comparative models of the file assignment problem. *ACM Computing Surveys*, 14(2):287–313, June 1982.

[12] P. Zabback G. Weikum and P. Scheuermann. Dynamic file allocation in disk arrays. In *SIGMOD Conference*, pages 406–415, 1991.

[13] M.R. Garey and D.S. Johnson. *Computers and Intractability*. W.H. Freeman and Company, 1979.

[14] Hewlett-Packard Company. *Model 30/FC High Availability Disk Array – User's Guide*, August 1998. Pub. No. A3661-90001.

[15] Hewlett-Packard Company. *HP SureStore E Disk Array FC60 - Advanced User's Guide*, December 2000.

[16] Hewlett-Packard Company. *HP Surestore Disk Array XP256 - Configuration Guide*, December 2001. Chapter 4.7.5 HP e3000 Business Servers Configuration Guide.

[17] R. A. Hill. System for managing data storage based on vector-summed size-frequency vectors for data sets, devices, and residual storage on devices, 1994.

[18] D.S. Johnson, A. Demers, J.D. Ullman, M.R. Garey, and R.L. Graham. Worst case bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3:299–325, 1974.

[19] C. Kenyon. Best-fit bin packing with random order. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms*, January 1997.

[20] E. Lamb. Hardware spending matters. *Red Herring*, pages 32–33, June 2001.

[21] A. Merchant and G. A. Alvarez. Disk array models in Minerva. Technical Report HPL-2001-118, Hewlett-Packard Labs, April 2001.

[22] G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley and Sons, 1988.

[23] D.A. Patterson, G.A. Gibson, and R.H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *SIGMOD international Conference on the Management of Data*, pages 109–116, Chicago, IL, 1988.

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

[25] P.W. Shor. The average-case analysis of some online algorithms for bin packing. *Combinatorica*, 6:179–200, 1986.

[26] E. Shriver. A formalization of the attribute mapping problem. Hp labs technical report HPL-SSP-95-10, HP Labs, July 1996.

[27] E. Smirni and D.A. Reed. Lessons from charaterizing the input/output behavior of parallel scientific applications. *Performance Evaluation*, 33(1):27–44, June 1998.

[28] Yoshiaki Toyoda. A simplified algorithm for obtaining approximate solutions to zero-one programming problems. *Management Science*, 21(12):1417–1427.

[29] Transaction Processing Performance Council. *TPC benchmark C, standard specification, revision 1.0*, August 1992.

[30] M. Uysal, G. A. Alvarez, and A. Merchant. A Modular, Analytical Throughput Model for Modern Disk Arrays. In *Proceedings of the 9th International Symposium on Modeling, Analysis and Simulation on Computer and Telecommunications Systems (MASCOTS 2001)*, Cincinnati, OH, August 2001.

[31] A. Veitch, K. Keeton, M. Spasojevic, and J. Wilkes. The Rubicon workload characterization tool. Technical report, HP Laboratories, April 2001. http://www.hpl.hp.com/SSP/.

[32] J. Ward, M. O'Sullivan, T. Shahoumian, and J. Wilkes. Appia: Automatic storage area network fabric design. In *Conference on File and Storage Technologies (FAST)*, Monterey, CA, January 2002.

[33] J. Wilkes, 1999. personal communication.

[34] J. Wilkes. Traveling to Rome: QoS specifications for automated storage system management. In *Proceedings of the International Workshop on Quality of Service.*, June 2001.

[35] J. Wolf. The placement optimization program: a practical solution to the disk file assignment problem. In *ACM SIGMETRICS Conference*, pages 1–10, Berkeley, CA, May 1989.