MINERVA: An Automated Resource Provisioning Tool for Large-Scale Storage Systems

Guillermo A. Alvarez, Elizabeth Borowsky, Susie Go, Theodore H. Romer,
Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic,
Alistair Veitch, John Wilkes
Computer Systems and Technology Laboratory
HP Laboratories Palo Alto
HPL-2001-139
June 11th , 2001*

E-mail: galvarez@hpl.hp.com

Enterprise-scale storage systems, which can contain hundreds of host computers and storage devices and up to tens of thousands of disks and logical volumes, are difficult to design. The volume of choices that need to be made is massive, and many choices have unforeseen interactions. Storage system design is tedious and complicated to dc by hand, usually leading to solutions that are grossly over-provisioned, substantially under-performing or, in the worst case, both.

To solve the configuration nightmare, we present MINERVA: a suite of tools for designing storage systems automatically. MINERVA uses declarative specifications of application requirements and device capabilities; constraint-based formulations of the various sub-problems; and optimization techniques to explore the search space of possible solutions.

This paper also explores and evaluates the design decisions that went into MINERVA, using specialized micro and macro-benchmarks. We show that MINERVA can successfully handle a workload with substantial complexity (a decision-support database benchmark). MINERVA created a 16-disk design in only a few minutes that achieved the same performance as a 30-disk system manually designed by human experts. Of equal importance, MINERVA was able to predict the r esulting system's performance before it was built.

MINERVA: an automated resource provisioning tool for large-scale storage systems

Guillermo A. Alvarez*, Elizabeth Borowsky, Susie Go, Theodore H. Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, Alistair Veitch, John Wilkes

Content indicators: D.4.2 storage management; G.1.6 optimization; D.4.8 performance.

Abstract

Enterprise-scale storage systems, which can contain hundreds of host computers and storage devices and up to tens of thousands of disks and logical volumes, are difficult to design. The volume of choices that need to be made is massive, and many choices have unforeseen interactions. Storage system design is tedious and complicated to do by hand, usually leading to solutions that are grossly over-provisioned, substantially under-performing or, in the worst case, both.

To solve the configuration nightmare, we present MINERVA: a suite of tools for designing storage systems automatically. MINERVA uses declarative specifications of application requirements and device capabilities; constraint-based formulations of the various sub-problems; and optimization techniques to explore the search space of possible solutions.

This paper also explores and evaluates the design decisions that went into MINERVA, using specialized micro- and macro-benchmarks. We show that MINERVA can successfully handle a workload with substantial complexity (a decision-support database benchmark). MINERVA created a 16-disk design in only a few minutes that achieved the same performance as a 30-disk system manually designed by human experts. Of equal importance, MINERVA was able to predict the resulting system's performance before it was built.

*Contact author, galvarez@hpl.hp.com. This work was done while all authors were working for Hewlett-Packard Laboratories, 1U-13, 1501 Page Mill Rd., Palo Alto, CA 94304, USA

1 Introduction

Enterprise-scale computer installations are extremely difficult to design and configure. They can contain tens to hundreds of host computers, connected by a Storage Area Network (SAN) such as FibreChannel [2] or Gigabit Ethernet [1] to tens to hundreds of storage devices, with up to tens of thousands of disks and logical volumes. Total capacities in the tens of terabytes are becoming common.

Disk arrays [18] use multiple, independent disks that store redundant copies of the customer information, in order to provide the levels of capacity, performance, and reliability required by mid-range and high-end computing systems. Commercial disk arrays export LUNs (Logical UNits), which are sets of disks bound together using a layout such as RAID 1/0 or RAID 5, and addressed as a single entity by client applications. The complexity of configuring the storage system is compounded as individual storage devices such as disk arrays usually have many parameter settings of their own.

Storage systems are traditionally designed by hand, which is tedious, slow, error-prone, and frequently results in solutions that perform poorly or are over-provisioned. Often, the only certain way of ensuring that a design meets its goals is to build and measure it. Besides from incurring prohibitive costs, this can take weeks or months.

Our solution to this problem is MINERVA, a suite of tools for the automated design of large-scale computer storage systems. Because the solution space is potentially huge, MINERVA handles the combinatorial complexity of the problem by dividing it into three stages: (1) choosing the right set of storage devices for a particular application workload, (2) choosing values for configuration parameters in the devices, and (3) mapping the user data onto the devices. Each of these is a large combinatorial problem in its own right. In particular, (3) can be shown to be at least as hard as bin-packing, which is NP-hard [12]; this is one reason why storage system design is so difficult.

MINERVA combines a number of different elements:

- Declarative descriptions of storage workloads and their requirements. These can be supplied by human users, extracted from a library of common cases and scaled to the problem at hand, or automatically derived from an existing, running system.

- A constraint-based representation of each design problem and a set of optimization strategies and heuristics to search for solutions, guided by predictions of the likely effects of each choice made.

- Fast, validated analytic performance models that estimate the effects of the interaction between workloads and storage devices. Although these models are vital to our work, and we spend much of our time developing them, they are not the focus of this paper, which is concerned with how those models are used in a complete design system.

MINERVA takes as input descriptions of the workload to be run on the system being designed, and of the capabilities of available storage devices. Its output is an *assignment*: a selection of storage devices, their configuration, and a placement of all the pieces of the workload onto those storage devices. A separate evaluation component is used to make predictions of the resulting system's performance, thereby reducing the need to build costly physical prototypes.

An automated system can explore the space of storage configurations and the complex interactions between parts of the workload and storage devices much more thoroughly than a person can. Therefore it can generally find a better solution to the storage design problem. However, a human can also take into account considerations which are not easily quantified, such as preference for a symmetrical solution or a particular hardware vendor. Occasionally, a human can use domain-specific knowledge to improve upon an automated solution. Thus, we do not attempt to remove people from the design loop, but to assist them by automating as much as is possible, and by allowing them to precisely quantify the consequences of design decisions.

The main contributions of this paper are in demonstrating that the MINERVA approach is sound, exploring several of the design issues involved, and showing that the MINERVA-generated designs meet their requirements and are as good or better than hand-generated ones.

The remainder of this paper is organized as follows. Section 2 describes the MINERVA system, including its role in the storage system lifecycle, detailed descriptions of each of the systems components, and the heuristics used in each of the optimization subproblems. Section 3 reports the results of a series of experiments, which demonstrate that our approach allocates resources correctly for a wide range of workloads. We survey some related areas in Section 4. We conclude in Section 5 with some thoughts on what we have learned and on where we intend to proceed with this work.

2 The MINERVA system

MINERVA is our system for rapidly designing a storage system that meets workload requirements and has near-minimum cost. The two key ideas behind MINERVA are our use of fast analytical device models to evaluate proposed designs, and heuristic search techniques. The heuristics ensure that we do not have to exhaustively search the (huge) design space, while the models allow

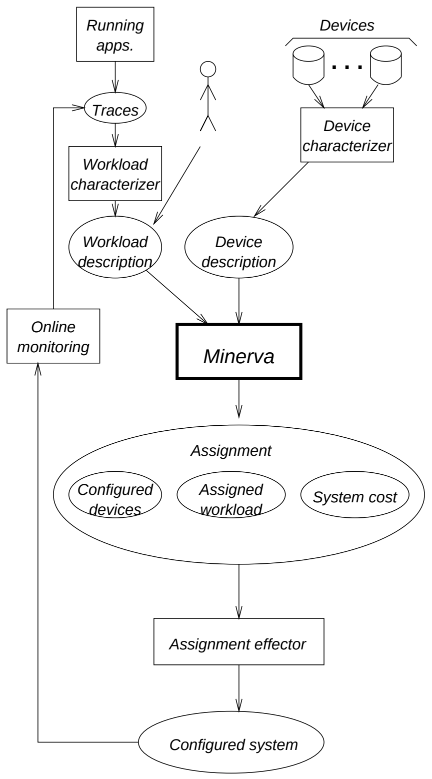


Figure 1: MINERVA*'s role in the storage system life cycle.* MINERVA *takes as input workload and device descriptions which describe the applications requirements and the capabilities of available devices. Based on these, it generates as output an assignment of the workload.*

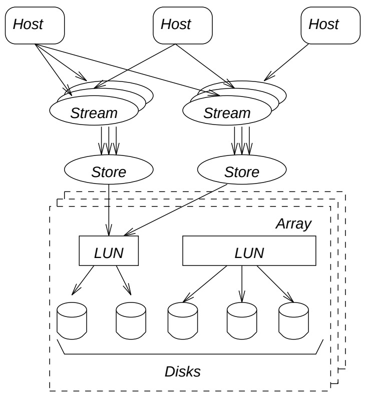


Figure 2: *Objects in the* MINERVA *storage management framework. Hosts generate workloads, which are characterized as a set of dynamic* streams *accessing static* stores. *One or more stores are mapped to each* LUN.

us to quickly evaluate each candidate solution chosen by the heuristics.

2.1 Storage system lifecycle

In order to explain the functions of MINERVA, it is important to understand the overall storage system lifecycle of which it is a major component, as this determines the input and output that the system must consume and produce. The system lifecycle is depicted in Figure 1. The major inputs to MINERVA are *workload descriptions* and *device descriptions*.

A workload description contains information about the data to be stored on the system and its access patterns. This information can be originated from several different sources, including system administrators or from measurements (I/O traces) taken on a running system. Our workload descriptions contain two types of objects: *stores* and *streams*. Stores are logically contiguous chunks of data such as a database table or a file system, with a stated capacity. Each store is accessed by zero or more streams; each stream is a sequence of accesses performed on the same store. Their relationships are depicted in Figure 2. Stream specifications describe both the access pattern and the performance requirements of applications. Hence, streams can be interpreted as a contract: if applications comply with the access pattern specifications (e.g. if they initiate no more than a given number of accesses per second), then the storage system must satisfy the associated performance requirements. Table 1 describes the attributes of a stream.

Device descriptions contain information about the devices that MINERVA can use when designing the storage system. They contain information such as the number and types of disks each array can support, any constraints on the LUN configurations available, and performance characteristics for each of the array components. Device descriptions can be built by using information from the device's manufacturer (when available), or by running de-

Attribute	Description	Units
requestrate	mean rate at which requests arrive at the device	requests/sec
requestsize	mean length of a request	bytes
runcount	mean number of requests made to contiguous addresses	requests
ontime	mean duration of the period when a stream is actively generating I/Os	sec
offtime	mean duration of the period when a stream is not active	sec
overlapfraction	fraction of the "on" period when two streams are active simultaneously; represents correlation between streams	none

Table 1: Workload characteristics used by MINERVA.

vice characterization tools on the real devices to compute the relevant attributes.

As output, MINERVA generates an *assignment* – a set of devices chosen from the input device descriptions and a mapping of the stores from the workload description onto those devices. The assignment should be a near-minimum cost configuration that supports the input workloads.

The disk array configuration and binding of stores to LUNs is made real by an *effector* tool with the assistance of the host logical volume manager, thereby automating a particularly error-prone operation that is usually done manually.

The complexity of the assignment problem is extremely high. Consider a workload consisting of m stores. Assuming that there is only one kind of array and we use n LUNs, it is easy to see that there are $2^n n^m$ possible configurations, since each of the LUNs could be configured as either RAID 1/0 or RAID 5, and each of the m stores can be assigned to any of the n LUNs. Since each workload could require an entire LUN, we may need as many as m LUNs, which gives us $\sum_{k=1}^{m} 2^k k^m = O((2m)^m)$ possible configurations to consider. For a modest workload consisting of 30 stores, an exhaustive search would consider as many as $3 \cdot 10^{53}$ possible configurations; if evaluating each configuration required $10\mu s$, finding the optimal solution by brute force would require 10^{41} years.

No real system is static: changes in workloads, device failures, and acquisition of new devices all cause systems to evolve. MINERVA can be used to handle this, too, as Figure 1 shows. The new workload requirements can be synthesized from measurements taken on the running system, and be fed as input to MINERVA, along with the existing storage system design. The output from MINERVA now represents changes to the configuration of the system, rather than a complete new design.

2.2 Architectural overview

Figure 3 shows the control flow between each of the high-level components making up the MINERVA system. The storage design problem is split into three main subproblems: array allocation, array configuration, and store assignment. The approach of partitioning the problem was selected to make the overall search feasible, by reducing the number of possible choices. Each step makes progressively more fine-grained decisions, using successively more detailed models. By relying on the coarse-grained results from the previous component, each stage is able to cut down its search space to manageable proportions. In addition, step-specific heuristics can be used to reduce the solution space further again.

The goal of the array allocation step is to select a set of configured arrays that satisfy the resource requirements of the workload. This step is further divided into two components, the *tagger*, which assigns a preferred RAID level to a part of the workload, and the *allocator*, which determines how many arrays are needed.

Array configuration is handled by the *array designer*, which configures a single array at a time. This stage em-

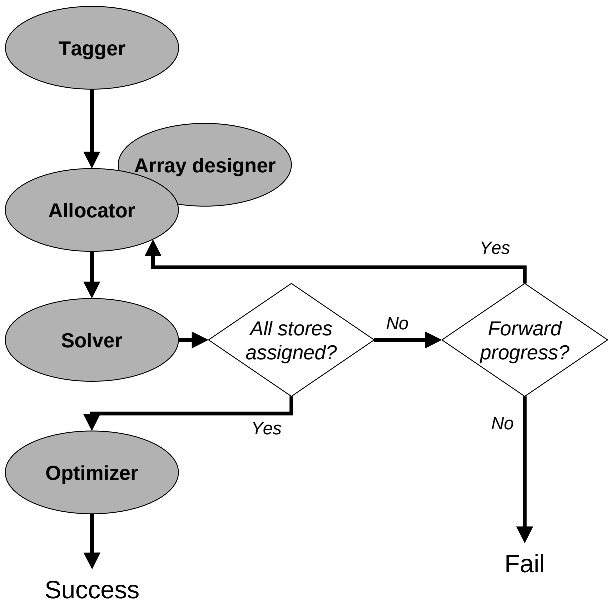


Figure 3: MINERVA *control flow. The array designer is called as a subroutine by the allocator.*

bodies the device-specific knowledge needed to configure an individual array type, whereas the allocator uses more generic device models.

The store assignment problem is handled by the *solver*, which assigns stores to LUNs generated by the array designer. Since the allocator has no information about which LUN will eventually be used for a given store, its performance predictions are less accurate than the solver's, and it may under- or over-provision the devices it offers to the solver, resulting in suboptimal solutions. Over-provisioning results in too many storage resources being used, while under-provisioning restricts the solver to only consider a subset of the stores and arrays on each iteration. To address the under-provisioning issue, we re-run the allocator and solver on any unassigned stores, and repeat this loop as long as there is forward progress. Over-provisioning is solved through the introduction of a final optimization pass. The *optimizer* prunes out unused resources and performs a re-assignment that attempts to balance the load across the remaining devices.

Figure 4 depicts the intermediate results generated by MINERVA on a sample workload. We show a simple case to clarify the role of each component; interesting, realistic workloads have hundreds or thousands of stores. In the figure, MINERVA is run on a combination of stores from two different applications. (This distinction is irrelevant to our tool, as stores and streams are uniformly treated according to their specifications, regardless of which application they correspond to.) These stores are presented to the tagger, which will annotate each one as requiring either RAID 1/0 or RAID 5 storage. The allocator and array designer then make an initial pass at the number and type of arrays required to support the workload, and allocate two arrays – one of which is dedicated exclusively to RAID 1/0 storage, the other with LUNs of both storage classes. The solver assigns stores from the original workload onto the LUNs, producing a workload assignment that packs the storage as tightly as possible (minimizing cost), while meeting the workloads performance requirements. In the example shown, the solver determines that the stores can all be packed onto a single array. The optimizer examines the solvers solution, and removes the unnecessary LUNs and devices, producing a final assignment.

The *evaluator* is a tool we have developed separately. It can be used to verify the correctness of the final MINERVA solution, by applying the analytical device models. Since it runs only once, it computes more sophisticated performance metrics than is feasible in the solver. Since we have largely kept its development separate from the solver, it also provides an independent check on our implementation of the complex analytical models. It can also be used as a stand-alone tool for assessing the performance of a proposed storage system design, whether it is generated by hand, by MINERVA, or by some other tool.

We will now describe each of the MINERVA components in more detail.

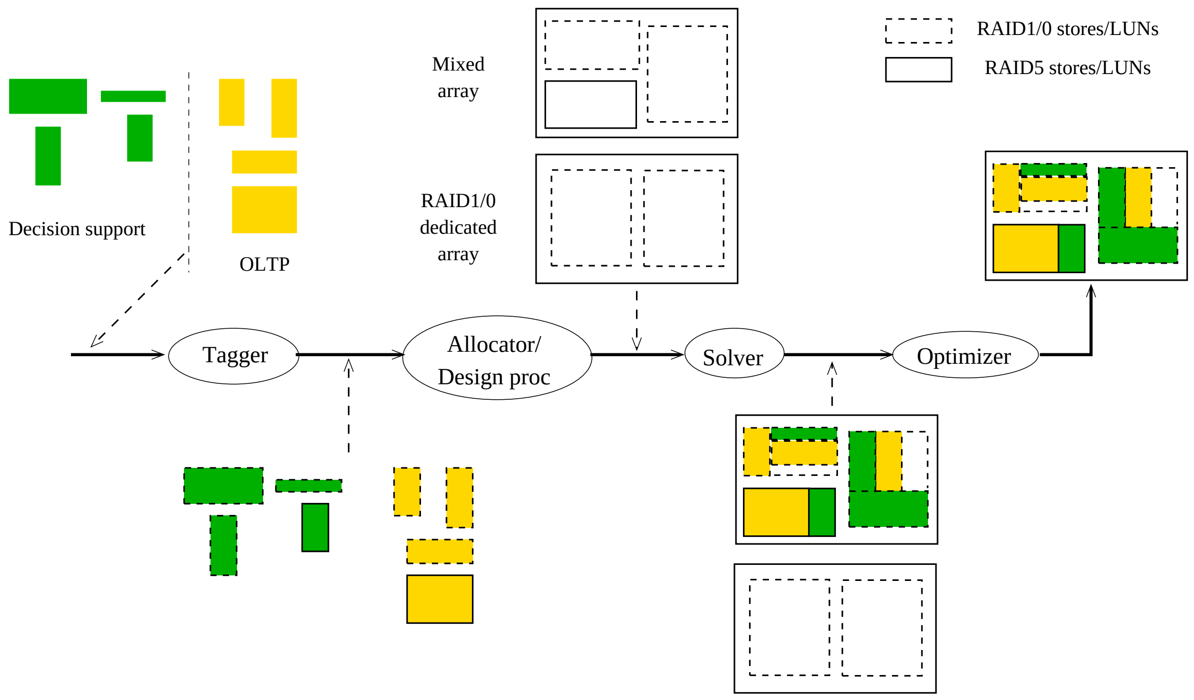


Figure 4: MINERVA *running on a sample combination of on-line transaction processing and decision support workloads. This example requires a single iteration of the outer loop, as all stores are assigned in the first pass. The dedicated* RAID *1/0 array originally configured by the allocator turns out to be unnecessary; it is deleted in the final optimization pass.*

2.3 Analytical device models

Our optimization algorithms rely on analytical models to determine if the proposed solution is feasible, i.e. if it satisfies all workload requirements and does not overload any device.

Each stream in the workload is modeled as an ON-OFF Markov-modulated Poisson process of requests. We take into account the request rate, the request size, what fraction of requests are reads versus writes, how sequential the requests are, and the correlations (phasing) between the ON periods of different streams.

The array model consists of a model of the array controller, a model of the bus connecting the controller and the disks, and a model of the disks. These sub-models compute the load imposed by the workload on each component and verify that the maximum utilization of each is less than one. Some of these sub-models are quite complex, since they must take into account the effects of caching, sequentiality, read-ahead, the location of the data accessed relative to the stripe unit and stripe boundaries, and of requests from different streams that arrive interleaved in time to the array.

When the inner details of a disk array are not available (for example, when there are proprietary design details), we use a “best guess” generic disk array model with parameters that are measured or estimated. We also allow for *calibration factors*, which correct the performance estimates based on measurements of micro-benchmarks running against the real device. Our array throughput models are fully described in [16].

We will use the *Hewlett-Packard SureStore Model 30/FC High Availability* (FC-30) disk array [7] as a case study to validate our models. Details of this disk array, and of the methods used to measure its performance, are given in Section 3.1; we show that the model achieves a maximum error of 20% in the throughput predictions.

2.4 The tagger

The goal of the tagger is to choose a storage class that will support each store's access pattern efficiently; this helps the allocator apply device models to the workload. The tagger considers two redundant storage classes, RAID 5 and RAID 1/0 (striped mirroring) [18]; we have developed and validated analytical array performance models for both storage classes. For simplicity, we use the term RAID 1/0 to include the case of a 2-disk RAID 1 LUN.

Since it does not have information about which stores are going to end up being mapped to the same LUN, the tagger has to rely on (sometimes very) approximated values to make decisions. A set of simple rules is evaluated for one store at a time, in a predetermined order (first rule that fires is the defining one) to determine which RAID level will be most efficient for the store. For clarity, we describe the case in which each store has a single stream; the extension to multiple streams is straightforward, keeping in mind that overlap fractions are ignored by the tagger. We use subscripts to distinguish between the read and write values of attributes, whenever appropriate.

The first rules select those stores which are likely to be capacity-bound, and tags them as RAID 5. These rules calculate the total bandwidth generated by the workload, per GB of storage:

$$bw = (requestrate_r \cdot requestsize_r + 2 \cdot requestrate_w \cdot requestsize_w)/storesize$$

and the approximate seeks per second per GB of storage

$$seeks = \frac{\frac{requestrate_r}{runcount} + 2 \cdot \frac{requestrate_w}{runcount}}{storesize}$$

(where we approximate each client write as resulting in two disk-level writes in the array back-end). If both of these values are sufficiently small (the exact values are device-specific) then the store is determined to be capacity-bound. Device-specific values are inevitable, for the capabilities of different arrays (e.g. the maximum number of seeks that their disks can make per second) are also different.

If the stores are not capacity bound, the following rules estimate the average number of I/O operations per second (IOPS) that will be generated by the stream for each RAID level being considered:

$$\begin{aligned}
runsize &= runcount \cdot (requestsize_r \cdot readfrac \\
&\quad + requestsize_w \cdot (1 - readfrac)) \\
IOPS_r &= \frac{requestrate_r}{runcount} \cdot \left(\frac{runsize}{susize} + 1\right) \\
IOPSR10_w &= \frac{requestrate_w}{runcount} \cdot \left(\frac{runsize}{susize} + 1\right) \\
IOPSR5_w &= \frac{requestrate_w}{runcount} \cdot ndisks
\end{aligned}$$

where $readfrac$ is the fraction of all requests that are reads in the stream, $susize$ is the size of each stripe unit in bytes, and $ndisks$ is the number of disks in the LUN. The second term in the second and third formulas denotes the average number of stripe units involved in processing one run of client requests; for RAID 5, we assume that each write run involves all the disks in the LUN. The tagger then selects the RAID level that will result in the smallest number of per-disk IOPS as the store tag. In Figure 4, six stores were tagged as RAID 1/0, and the two remaining ones were tagged as RAID 5.

2.5 The allocator and array designer

The goal of the allocator is to select a reasonable, rather than optimal, set of configured arrays that satisfy the resource requirements of the workload. Within the allocator, the search problem is again partitioned, by using a hierarchy of three progressively more refined searches, shown in Figure 5. At the highest (and coarsest) level the allocator considers only the type and number of arrays. The second level of the allocator explores the space of

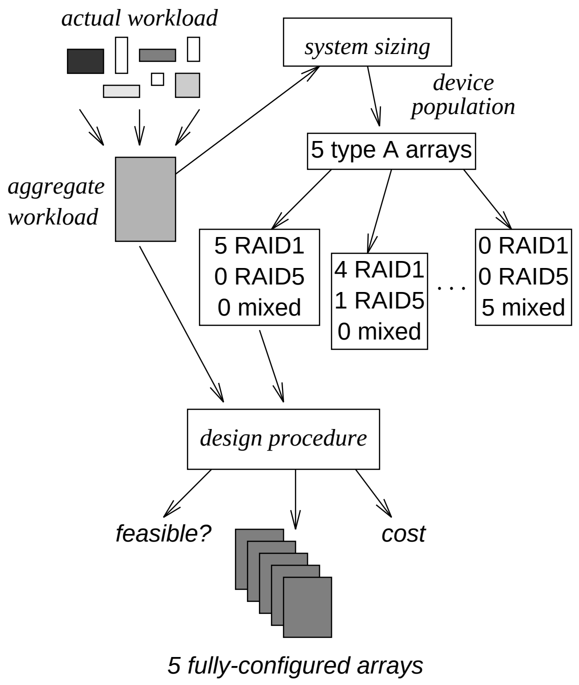


Figure 5: *How the allocator evaluates a population of arrays: using an aggregation of the workloads, the allocator tries all possible combinations of array configurations to find the one that is feasible with lowest cost, if any.*

array configurations, given a fixed type and number of arrays. In the lowest level, the array designer decides how to best divide each single array into LUNs of different sizes and RAID levels to support a subset of the workload.

2.5.1 Allocator models

Because stores have not yet been assigned to LUNs, the allocator can only apply analytical device models to the aggregate workload (formed by adding together attributes for each stream), partitioned by storage class. The allocator uses simplified models that assume the workload to be infinitely and uniformly divisible. Because these simplified models ignore phasing, the allocator can overprovision. It may also rarely underprovision, since in practice the workload is divided into finite-size pieces.

To handle stores and streams with large resource requirements, MINERVA uses a *rillifier* that automatically divides stores into *shards* (pieces of stores) and streams into *rills* (pieces of streams). The size of the shards and rills is selected such that their resource requirements can be met by a single LUN of the target disk array. The current MINERVA implementation of the rillifier divides stores with capacity greater than 2 GB into smaller shards. We chose the 2 GB threshold so that all shards would fit comfortably into LUNs built from the 4 GB disks in the FC-30 array. Only one of our input workloads had a single stream with a request rate large enough to exceed the bandwidth and phased utilization constraints of an entire FC-30 array, so we split it manually into two shards and two rills.

2.5.2 Allocator search

At the highest level of the search, the allocator uses a branch-and-bound strategy to choose how many instances of each type of array to provision. The lower bound reflects the aggregate capacity and bandwidth requirements of the workload, and the resources available on the candidate array. The algorithm searches in order of increasing cost, terminating at the first configuration that supports the aggregate workload. For example, if the workload requires 300 GB of usable RAID 1/0 capacity, five FC-30 arrays each filled with thirty 4 GB disks would suffice. In each step in the branch-and-bound search, the allocator considers the configuration with the lowest cost. If the allocator can configure the arrays represented by this node to support the workload, the algorithm is complete. Otherwise, a new candidate is added to the population by simply incrementing the number of arrays. The search is bounded by the trivial upper bound that the number of arrays should be no greater than the number of stores.

The second level of searching tests whether n arrays can support the workload by searching the space of possible configurations for n arrays of the given type as illustrated in Figure 5. It begins by generating array configurations that are mixed, i.e. each array contains both

RAID 1/0 and RAID 5 LUNs. To prune the search, it will use the same configuration for every mixed array. For example, if a particular type of device can support both RAID 1/0 and RAID 5 storage, the n devices will be partitioned into a set of identically configured devices that supports a mixture of the two storage classes. If those configured devices fail to satisfy the workload requirements, the allocator iteratively converts one array to dedicated (containing only RAID 1/0 or RAID 5 storage classes), and tests whether the modified set of arrays can support the workload. The *branch and bound—bias dedicated* variant of the allocator performs this search in reverse order, beginning with all arrays dedicated and converting them to mixed one at a time.

The allocator uses the array designer to "build" the devices in each of these partitions, determining the exact configuration and parameter settings for a particular class of arrays. It is possible that the array designer will fail, in which case the allocator continues to search the configuration space.

In the example of Figure 4, the allocator generated and configured two arrays: a mixed one, and a RAID 1/0-dedicated one. The tagged workload was bandwidth-constrained, and the aggregate models' calculations determined that a second array would be needed to provide the aggregate bandwidth.

2.5.3 The array designer

At the lowest level, the allocator uses the array designer to determine the exact LUN sizes and other parameter settings for a single array. The array designer embodies the array-specific rules for creating LUNs built using the disks available in the array, such as balancing LUN placement across back-end buses, while meeting array-specific constraints such as the maximum number of disks per LUN and the maximum number of LUNs per array.

For a dedicated array, the array designer always uses all the disks the array can hold. It begins with a simple configuration, dividing the disks into LUNs of equal size (except for one final LUN, which might be smaller). For example, if a particular array allows LUNs with stripe widths 3, 4, or 9, and there are 30 disks available, the array designer will build a population of candidate configurations which has ten 3-disk LUNs, seven 4-disk LUNs, and three 9-disk LUNs plus a 3-disk LUN. In addition to these three simple configurations, the array designer adds in one more candidate to the population by generating a greedy configuration, where LUN sizes are determined by the capacity and bandwidth requirements of the workload. Of these four possible configurations, only those that satisfy the requirements of the workload are considered further. If there is more than one possible configuration, the array designer will choose the configuration with the most uniform LUN stripe width, because uniform LUN sizes tend to allow the solver and optimizer to tightly pack LUNs onto the arrays. If multiple configurations have the same uniformity, the array designer will select the stripe width closest to a preferred stripe width that is chosen to maximize performance.

For arrays with mixed storage classes, the array designer considers the storage classes one at a time. Only as many disks will be bound into LUNs as are needed for the workload at hand; this leaves unused disks for other storage classes on the same array. A population of candidate configurations is then built by considering simple designs which satisfy the aggregate capacity and bandwidth requirements of the workload and then progressing to more complex designs. First, a set of simple configurations of only one LUN size is generated which satisfies the resource requirements. To this set of candidate configurations, a more complex set is generated by considering configurations with LUNs of two or more different stripe widths.

This process is repeated for every storage class required for the array, generating a collection of candidate configurations for each class. A final list of the cross product of each of these sets is created and pared down to contain only the legal combinations based on the array-specific constraints. As in the dedicated array case, the array designer choses the configuration which has the most uniform LUN stripe widths, and in case of a tie the smallest deviation from the preferred stripe width.

If the array designer finds that the workload it was given greatly under-utilizes a single array, it will still configure that array with all disks configured into LUNs, to enable the solver to use the extra capacity. The optimizer will remove unnecessary resources later.

Once the configuration has been selected, the final assignment of target disks to LUNs is done in a round-robin fashion across busses and the complete array design is returned to the allocator. It is possible that the array designer will not find any design. This can occur in cases where the allocator requests requirements which simply cannot be met, or where the resource requirements are very high, since the design procedure does not perform an exhaustive search of the space of all possible configurations. The array designer will generally only fail to generate a configured array in the extremely rare case where a feasible solution is an array with LUNs of more than four different stripe widths per storage class. In that case, the allocator will retry using a slightly different configuration; in our experience, the second try was always successful.

2.6 The solver

Once a set of configured arrays has been selected, the solver assigns stores to LUNs. We have traditionally [4] treated this as a multidimensional constrained bin-packing optimization problem. The constraints correspond to the performance capabilities of the arrays, and the goal is to produce an assignment that minimizes some objective function. We evaluate the constraints using analytical device models, testing whether a given assignment can support the requirements of the stores and streams. For example, the combined size of all stores assigned to a LUN must not exceed the capacity it provides; nor can the phased utilization of a LUN exceed 100%. Currently we model four constraints: LUN capacity, LUN phased utilization, array bus bandwidth, and array controller utilization. Each store must be assigned to a LUN matching its tag.

In our example in Figure 4, the solver was able to fit all stores into a single mixed array. The aggregate bandwidth requirements estimated *a priori* by the allocator turned out to be too pessimistic, because they ignored the effect of phasing among streams (or, equivalently, they assumed that every stream was ON all the time).

We have investigated and extended several optimization heuristics. The first solver we consider is *simple random*. It considers 50 random orderings of stores and LUNs, and uses a first-fit strategy to assign stores to LUNs, choosing the assignment that minimizes the objective function.

The remaining two solvers, *Toyoda* and *ToyodaWeighted*, use best-fit heuristics. We repeatedly consider all possible assignments of unassigned stores to LUNs and make the assignment which is the best fit. Whether a store fits in a LUNis determined by applying the constraints. The best assignment is determined by a heuristic function, called a *gradient*, which combines the objective function that users care about (e.g., system cost in the current MINERVA implementation) with a measure of how economically the store utilizes unused resources.

The solver *Toyoda* is our mapping of Toyoda's method [22] to the assignment problem. This solver computes a gradient for each (store, LUN) pair that serves as an estimate of the cost/benefit tradeoff of assigning the store to the LUN. Figure 6 illustrates this for two stores, in a simple 2-dimensional example. The axes on the graph correspond to two different resources, such as capacity and phased utilization. The vector L indicates how much of each resource has already been used on this LUN, and the vectors S_1 and S_2 indicate how much of each resource would be consumed if store1 or store2 were assigned to this LUN. For each candidate store, the "penalty" (shown

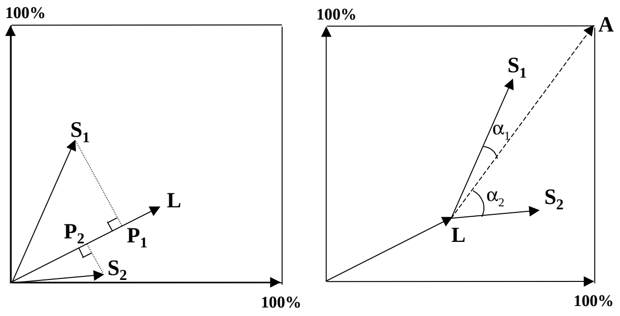


Figure 6: *How the* Toyoda *(left) and* ToyodaWeighted *(right) solvers select stores. Each axis corresponds to a resource provided by the* LUN. *The vector L describes the resources that have already been used, and the vectors S_1 and S_2 describe the resources that would be consumed by two different stores if they were assigned to the* LUN. *The* Toyoda *algorithm (left) will assign the store whose vector projected onto L is shortest—in this example, S_2. For the* ToyodaWeighted *algorithm (right), the vector LA describes the resources available on the* LUN, *and the algorithm will assign the store whose vector's direction is closest to the direction of LA—in this example, S_1.*

as P_1 and P_2 in the figure) of assigning that store to the current LUN is the projection of the vector representing the resource usage for that store onto the vector representing the resources already consumed. The final gradient is computed as $(1/penalty - lun\_cost)$, favoring assignments to LUNs that are already in use or that have low cost. *Toyoda* iteratively assigns the store with the highest gradient, and recomputes the gradients until no more stores can be assigned. In practice this means that LUNs are likely to be filled in order of increasing cost, and that within each LUN stores will be selected to minimize resource contention.

The third solver we consider in this paper, *ToyodaWeighted*, uses the same approach as *Toyoda*, but computes gradients differently, as illustrated in the right half of Figure 6. The vector LA represents how much of each resource is available given the set of stores currently assigned to the LUN. The gradient for a store is computed as the cosine of the angle between the resource usage vector of the store and LA, the vector representing the resources remaining on the LUN. This heuristic therefore favors assigning stores that use the resources remaining on the LUN in a balanced way. The *ToyodaWeighted* solver also supports arbitrary objective functions: the final gradient is computed as $objective\_value \cdot \cos(\alpha)$. For the MINERVA goal of a minimum cost assignment, we use $objective\_value = max\_lun\_cost - lun\_cost$, where *max_lun_cost* is the maximum possible cost over all LUNs in the system being designed.

2.7 The optimizer

Once MINERVA successfully produces a set of arrays that supports the input workload, we perform a final global optimization pass. The *optimizer* first re-runs the solver against the entire input workload and the set of arrays produced in the main loop of MINERVA, to see if the number of arrays can be reduced. This is clearly the case for the example in Figure 4, as the RAID 1/0-dedicated array is not needed at all. The array is deleted from the final MINERVA output.

Once cost has been minimized, MINERVA then runs the *ToyodaWeighted* solver to generate the final configuration, but this time with a special objective function that strives to balance load across the LUNs: $objective\_value = 1 - lun\_utilization$, where *lun_utilization* is the current utilization of the LUN. Thus the algorithm will favor assigning stores to underutilized LUNs. If the solver succeeds in finding a solution, the result is the final output of MINERVA. Otherwise, we fall back on the minimum cost solution found earlier.

We also consider two variants on the baseline optimizer, each using a different algorithm for the final load balancing pass. The first, *simple random*, is a version of the randomized first-fit solver: it generates 50 trial solutions, and selects the one with the lowest variance in LUN utilization. The second, *simple balance*, assigns stores to LUNs in round-robin, first-fit order, subject only to capacity and utilization constraints.

2.8 The clusterer

Initial tests with early MINERVA prototypes showed that the solver and optimizer did not scale well up to workloads with several hundred stores (especially if many of them were RAID 5). Every time a store placement decision is considered, the solver/optimizer must recompute complex estimations of LUN utilization in the presence of inter-stream correlations. The high cost of doing this resulted in unacceptably long running times – in the order of several days.

To solve this problem we started aggregating multiple input stores together into *clusters* and having the solver map whole clusters to LUNs. We examined many possible clustering heuristics, such as limiting the size of each cluster (built by a first-fit traversal on a sorted list of input stores), or the number of stores per cluster, or the total bandwidth requirements of all the streams accessing a cluster. Clustering stores based purely on their size (capacity) often resulted on clusters having a large number of streams on them, which in turn implied modest speedup gains as the running times of the LUN utilization calculations grow quickly with this parameter. The empirical rule of thumb derived from our experiments was to have at most 20 streams per cluster. On the other side, clustering solely based on an aggregate bandwidth limit resulted in significantly more expensive solutions, as many of our test workloads were bandwidth-bound to begin with (we believe that our clustering test suite is an unbiased, although possibly simplistic, representation of the workloads MINERVA will handle in real situations). The best policy for cluster construction was a hybrid bandwidth/capacity limit: we add stores to a cluster until the cluster reaches 10MB/s bandwidth, or 2GB size. By cutting down on the number of objects the solver must deal with, aggregation leads to dramatic performance improvements. This optimization typically resulted in a speedup of 15x-20x for the longer runs.

The downside is that the quality of the final bin packing may suffer, because we are artificially constraining the original input problem: stores in the same cluster will necessarily be mapped to the same LUN. This may restrict the number of packings the solver is allowed to consider. However, the hybrid bandwidth/capacity clustering policy caused just a slight degradation in our experiments: the quality (cost) of the final solution became about 3% higher for our TPC-D case study.

Given the considerable performance improvement and the negligible degradation in the quality of the solution, we permanently incorporated this optimization into MINERVA. All results in this paper were computed on a prototype where a clusterer module pre-processes the input workload before the tagger step, and a declusterer module regenerates the original set of stores at the end. For simplicity, these steps are not shown in Figure 4.

3 Evaluation

In the previous section, we described the MINERVA components. In this one, we evaluate MINERVA's accuracy, performance, design, and usefulness. We first validate the performance predictions of our analytical models, and then use those models to explore MINERVA's sensitivity to workload changes and the effect of alternate designs for MINERVA's components. We conclude the section by measuring the performance of a live storage system designed by MINERVA to support a decision-support database benchmark.

3.1 Device model validation

In order to validate MINERVA's device model predictions for a real disk array, we used the HP SureStore Model 30/FC High Availability (FC-30) disk array [7] as a case study.

An FC-30 supports two front-end controllers and up to 30 disks, which can be configured into up to 8 LUNS.

Each LUN uses a RAID 1 (mirroring), RAID 1/0 (striped mirroring), or RAID 5 (striping with left-symmetric rotated parity) layout. For convenience, we will also use the term RAID 1/0 for 2-disk RAID 1 LUNs. Our FC-30 has 30 disks of 4 GB each, two controllers and 60 MB of cache RAM, which was split 20:40 between read-cache and write cache; sniffing (background scanning of the disks for errors) was disabled. Both ports of the FC-30 array were connected by a single 1 Gb/s FibreChannel arbitrated loop [2] to a four-processor HP9000 K410 server with 1 GB of main memory running HP-UX 11.0. We used a synthetic workload generator generating many concurrent requests to simulate the effect of multiple applications running on the server. Accurate I/O timings were obtained from the HP-UX in-kernel tracing facility [20, 14].

We started by calibrating a generic array model using read-only or write-only access patterns with uniformly-dispersed ("random") requests of various sizes. We then validated the models by measuring the maximum request rates supported by the FC-30 for a richer set of micro-benchmark workloads, and comparing the measurements against the models' predictions. The workloads used in the validation experiment varied three parameters: request size, read fraction and run count. They consist of a baseline workload (32KB requests, 50% reads and, run count = 1), plus a series of workloads in which one of the parameters was varied at a time. In addition, we applied the workloads to several different LUNs: 2-, 4-, and 8-disk RAID 1/0 LUNs, and 4- and 8-disk RAID 5 LUNs. The validation tests consisted of the cross-product of all workloads and all LUNs listed in Table 2—i.e. each workload was run against each LUN. The streams in this experiment access the LUN synchronously, issuing a new request as soon as the previous one has completed.

Figure 7 shows a subset of the results from the validation studies (in particular, for run count = 1). The FC-30 serviced between 70 and 360 requests/s during these experiments. On average, the models are moderately accu-

Storage device	30-disk FC-30
Arrival time process	AFAP (as fast as possible)
Number of streams	At least 4 per disk
Number of LUNs	1
Disks / LUN	2 (for RAID 1)
	4, 8 (for RAID 1/0)
	4, 8 (for RAID 5)
Stripe unit size	64KB
Request size	8KB, 32KB, 64KB
Run count	1, 4, 8, 32
Read fraction	0, 1/4, 1/2, 3/4, 1

Table 2: *Workloads used to validate the analytic LUN models. Baseline values are underlined. The number of streams per disk was selected beforehand to nearly maximize throughput, and kept constant through all experiments.*

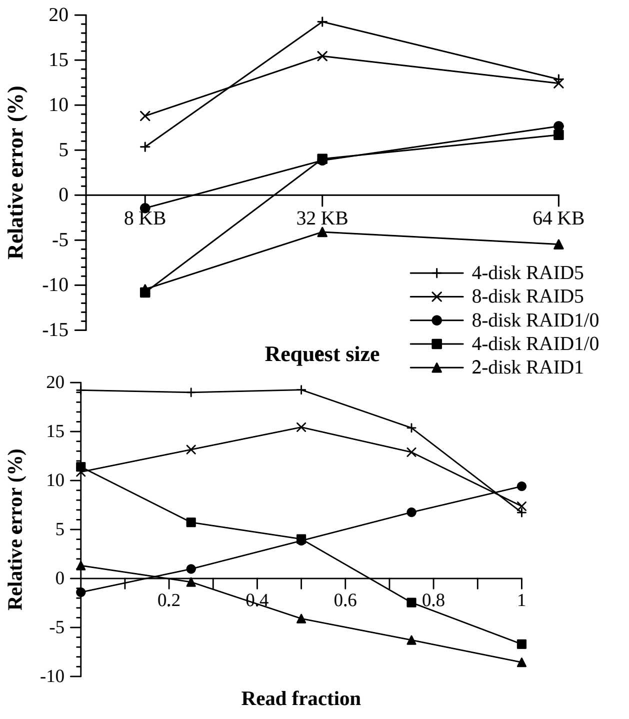


Figure 7: *Relative error in the model's throughput predictions, computed as (measured–predicted)/measured, when request sizes and read fractions deviate from the baseline values. Points above the x axis represent pessimistic model predictions.*

	Large, seq. (RAID 5)	Small, random (RAID 1/0)
number of stores	125	125
store size (GB)	2	2
number of streams	125	125
requestSize (KB)	32	8
requestRate (IO/s)	16	16
read fraction (%)	25	25
runCount	10	1
onTime (s)	4	4
offTime (s)	2	2

Table 3: *Characteristics of the synthetic baseline workload used in the scaling tests for* MINERVA. *The workload consists of 250 stores, each with one stream; 125 stores/streams of each variety.*

rate but slightly pessimistic, with a mean error of $+5.4\%$, and a range of $[-11\%, +19\%]$.

3.2 Safety and sensitivity

We used the validated models to examine the effect of scaling various workload parameters on the designs produced by MINERVA. We will show that for a wide range of input workloads MINERVA consistently designs storage systems that support the workload while using resources efficiently. We will also show that the resources provided by MINERVA change smoothly as the input workload requirements are varied.

All of the scaling experiments start with the same synthetic baseline workload, and then vary one characteristic at a time. We chose the parameters of the baseline workload, shown in Table 3, to fulfill three criteria. First, that there be a mix of stores better suited to RAID 5 (large writes) and RAID 1/0 (small writes). Second, that the resulting system requires a nontrivial number of arrays (much larger than 1), thereby demonstrating how MINERVA can configure an interesting system. Third, that each store be roughly balanced in its appetite for capacity and disk bandwidth.

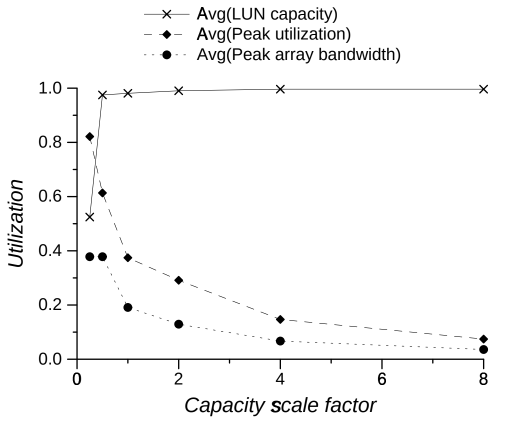
Another goal of this set of experiments is to show that the capacity and performance requirements of every store assigned by MINERVA are always satisfied. This safety property is equivalent to stating that all assigned stores pass the evaluator test, in which analytical device models are used to check MINERVA's output. Since the experiments described in this section require much more than one disk array, and therefore can not be physically implemented in our lab (described at the beginning of Section 3.1), we rely on our earlier analysis of the evaluator's accuracy to consider its predictions indicative of the performance we would observe in a real system.

3.2.1 Scaling store size and bandwidth

The first series of experiments scales the size of the stores up and down from the baseline values. Unsurprisingly, for small store sizes the workload is bandwidth-limited, while for large store sizes the workload is capacity-limited, as illustrated in the upper graph in Figure 8. The lower graph in the figure shows how MINERVA addresses the increasing capacity requirements, by creating systems whose total size scales roughly linearly with the store size, and that are mostly or completely made up of RAID 5 LUNs.

We scaled the bandwidth through request rate scaling. At low bandwidths, the storage is capacity-bound, so RAID 5 is used exclusively. As the bandwidth increases, the split into RAID 5 and RAID 1/0 disks occurs, with the numbers increasing linearly to meet the increased demands. Figure 9 illustrates this, showing how the systems designed by MINERVA adapt as the request rate in the baseline workload is scaled up and down.

The low bandwidth utilization at large bandwidth is due to fragmentation. As bandwidth requirements increase, fewer and fewer stores get mapped to each LUN (and to each array) as the bandwidth becomes an increasingly scarce resource. The number of disk slots used per array remains fairly high throughout the experiments; this is because disk bandwidth is a bottleneck as well, and therefore MINERVA needs many disks that are only partially filled with data.



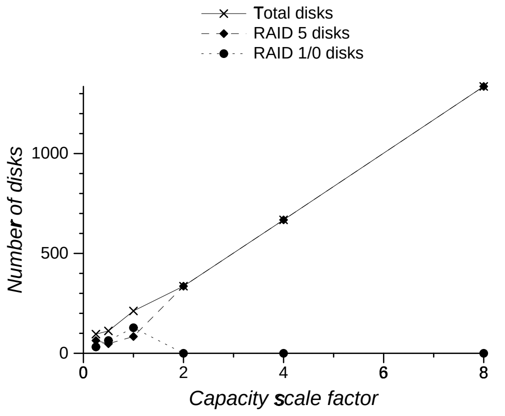
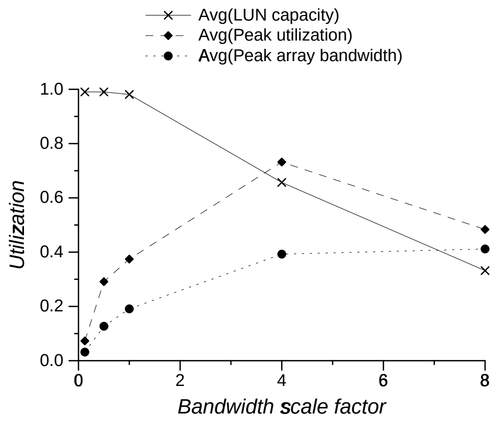


Figure 8: *Store size scaling tests: The upper graph shows the average fraction of capacity used over all* LUN*s, the average of peak utilization over all* LUN*s, and the peak array bandwidth utilization over all purchased arrays. The lower graph shows the total number of disks used in all arrays, and how those disks were partitioned among disks in* RAID *5* LUN*s and disks in* RAID *1/0* LUN*s. Each FC-30 array can hold 30 disks. The x axis is the multiplication factor applied to the baseline quantities in order to derive each workload used in the experiments.*



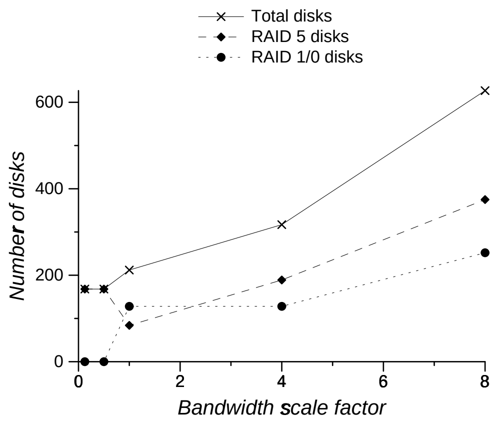
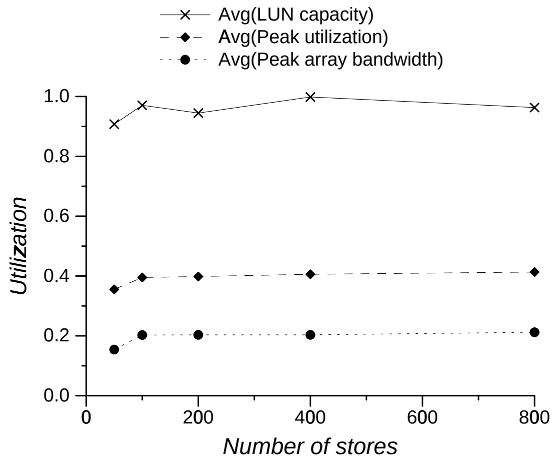


Figure 9: *Array utilization and makeup when scaling bandwidth.*

3.2.2 Scaling the number of stores

The next series of experiments scales the number of the stores up and down from the baseline number of 250. As expected, the number of arrays scales linearly with the number of stores, and the configuration of those arrays also scales linearly, as seen in the lower graph in Figure 10. The upper graph in the figure shows that the utilization of the array resources is consistent except when



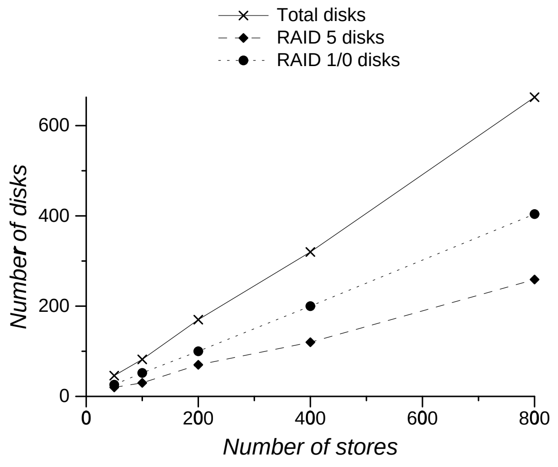


Figure 10: *Array utilization and makeup when scaling the number of stores.*

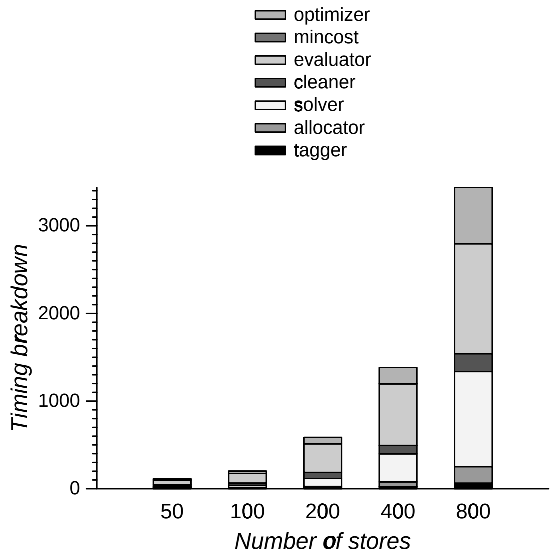


Figure 11: *Running times (in seconds) for the individual* MINERVA *components when scaling the number of stores.*

there are a very small number of stores. In this case there are simply not enough stores to fully exploit the arrays.

The number of stores is also the primary factor in MINERVA's run time. As can be seen from Figure 11, the running time of MINERVA's Toyoda solver algorithms grow quadratically with the total number of stores.

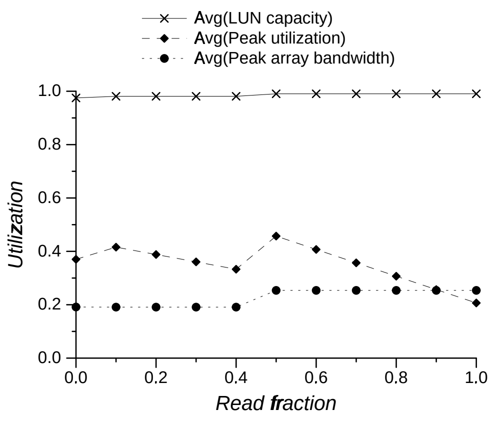
3.2.3 Varying the read/write mix

This series of experiments scales the percentage of reads up and down from the baseline value of 25%. Figure 12 depicts output configurations for workloads ranging from all writes (0%) to all reads (100%).

The lower graph in that figure shows that as the read fraction increases, the storage requirements shift from RAID 1/0 to RAID 5. This is because the tagger rules determine that, with a smaller proportion of writes, larger RAID 5 LUNs, which use a smaller fraction of the disk space for storing redundancy, provide more capacity per unit cost, and so are the more cost-effective choice for this workload.

3.2.4 Exploring workload variability

All the workloads discussed so far in this section have been homogeneous: attribute values were exactly the same from one stream/store to another. We now explore the impact of more realistic, heterogeneous workloads. The value for each attribute for each type of stream and



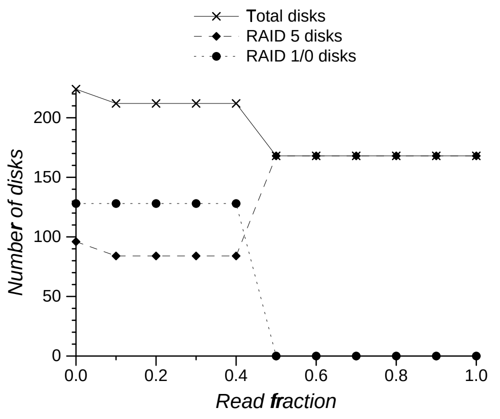


Figure 12: *Array utilization and makeup when changing the read fraction.*

store is drawn at random from a log-normal distribution. The mean value of the distributions are the baseline value for the attributes, from Table 3. The standard deviation of the distributions are the scale factor times the baseline value for that attribute; the scale factor ranges from 0 to 1. The mean value of the distribution does not change as the variance increases with the scale factor. The preceding experiments in this section had always used a standard deviation of zero. Since the values are drawn at random for this section, the experiment is repeated 5 times, and the results averaged.

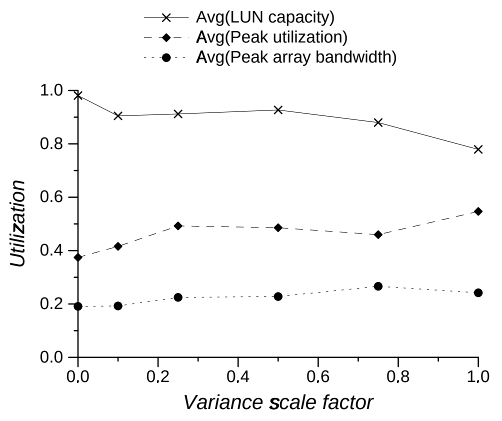
As the variability increases, the capacity utilization drops, while the number of stores assigned to RAID 5 LUNs increases, as seen in Figure 13. This has two causes. First, the base RAID 1/0 streams are relatively close to triggering the tagger rule that detects capacity bound workloads. By either decreasing the request rate, or increasing the capacity, this rule is triggered, and the store is tagged as RAID 5 (see Figures 8 and 9). Second, when the store sizes are uniform, it is always possible to pack the 2 GB stores efficiently onto LUNs built out of 4 GB disks. But as the standard deviation of the store sizes increase, such an optimal packing becomes less likely, decreasing the capacity utilization.

3.2.5 Summary of safety and sensitivity experiments

The experiments in this section confirm that MINERVA designs systems that meet workload requirements, provide resources that match the workload needs, and use RAID 5 or RAID 1/0 storage appropriately. MINERVA's solutions adapt smoothly to increasing workload requirements and to variations in other characteristics of the workload. Except for some edge cases when fragmentation makes some inefficiency in resource usage inevitable, MINERVA generally produces solutions that meet workload requirements without significant overprovisioning.

3.3 Evaluating the MINERVA design choices

There are many components in MINERVA (see Figure 3) with multiple possible designs for each component. To evaluate the effect of our design choices, we considered alternative designs for each of the components. We started with the baseline set of MINERVA components and then varied each of them in isolation. We compared the cost of the resulting system designs for a set of five workloads, using published list prices. Table 4 shows the alternate components that were evaluated for this study.



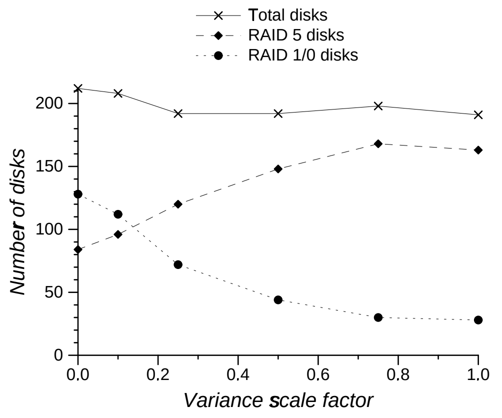


Figure 13: *Array utilization and makeup when scaling the variance of all workload attributes.*

Component	Selections
Tagger	IOPSdisk, allR10, allR5, random
Allocator	Branch and bound, branch and bound bias dedicated, all dedicated
Solver	Toyoda, Toyoda weighted, random
Optimizer	Toyoda weighted, simple random, simple balance

Table 4: *The alternate choices for* MINERVA *components used to evaluate the design. The baseline components are underlined.*

3.3.1 Workloads

The workloads used in these experiments are based on a combination of traces and models of a diverse set of applications: an active filesystem (*filesystem*), a scientific application (*scientific*), an on-line transaction processing benchmark (*oltp*), a parallelized decision-support benchmark (*dss-p*), and a lightly-loaded filesystem (*fs-light*). Table 5 summarizes the performance characteristics of the workloads.

The *filesystem* workload is an extrapolation of a trace of our local file server, which was used to create the baseline attributes for a file system stream and store. We then added three scaling parameters which enabled creation of a heterogeneous set of streams and stores to represent a wide variety of file system usage. The first scaling parameter was the number of users (default 1); a larger number of users translates into larger request rates, and longer on times. The second parameter was the percentage of large files in the filesystem (default 15%); a greater percentage of large files will cause larger request sizes and larger run counts in the workload. Finally, the third parameter was simply the size of the store (default 1GB). We created a heterogeneous set of 140 streams and stores, with store sizes ranging from .25 to 1.2 GB, the number of users ranging from 4 to 100, and the percentage of large files ranging from 5% to 35%. We took this as representative of a heterogeneous file system workload. The resulting workload included moderate size requests (20 KB on average), and little sequentiality. The *fs-light* workload represents a larger filesystem being accessed less intensively; it was created in a similar way.

The *oltp* and *dss* workloads are both taken from traces of database benchmarks. For *oltp*, we modeled the TPC-C benchmark [8] traces analytically to create a workload generator. We then generated a TPC-C workload corresponding to roughly 30 users and 400 transactions per minute for use in our tests. TPC-C is an application with a high rate of random, read-mostly accesses. Workload *dss*,

Workload	Capacity (GB)	#stores	#streams	Req. size (KB)		Run count		Reads (%)
filesystem	85.7	140	140	20.0	(13.8)	2.6	(1.3)	64.2
scientific	186.3	100	200	640.0	(385.0)	93.5	(56.6)	20.0
oltp	192.5	194	182	2.0	(0.0)	1.0	(0.0)	66.0
dss-p	49.6	316	224	27.6	(19.3)	57.7	(124.8)	98.0
fs-light	156.6	170	170	14.8	(7.3)	2.1	(0.7)	64.1

Table 5: *Characteristics of workloads used in the evaluation of* MINERVA*'s baseline components. "Run count" is the average number of consecutive sequential accesses made by a stream. Thus workloads with low run counts (*filesystem, oltp, fs-light*) have essentially random accesses, while workloads with high run counts (*scientific*) have sequential accesses.* dss-p *has both streams with random and sequential accesses. The access size and run count columns list the mean and (standard deviation) for these values across all streams in the workload.*

representing decision support systems, was taken from traces of our TPC-D-inspired benchmark, described in more detail in Section 3.4. This benchmark is characterized by long, complex database queries with interesting phasing behavior. Some accesses are extremely sequential, and some quite random. To obtain a *dss* workload of large enough scale for these experiments, we took two such traces and combined them to make a single large workload.

3.3.2 The tagger

We compared the rule-based tagger presented in Section 2.4 with three naive taggers: one that labeled all stores as RAID 5, one that labeled all stores as RAID 1/0, and one that labeled stores at random. Table 6 shows how the various taggers classified the workloads.

The different rules which the various taggers used sometimes gave very different storage layouts for the various workloads. For example, the *scientific* workload (which has many large sequential writes) is best suited for RAID 5 storage, while the *filesystem* and *oltp* workloads (which have many small random writes) are better served with a RAID 1/0 layout. For the *dss-p* workload all of the taggers resulted in configurations with nearly the same cost. This is because for this read-dominated workload RAID 5 LUNs and RAID 1/0 LUNs can provide roughly the same effective throughput per disk, so the choice between the two is arbitrary. In practice RAID 1/0 might be preferred for its superior performance in degraded mode, although MINERVA does not currently take this into account explicitly.

Table 6 shows that in all cases, the rule-based tagger does as well or better than any of the naive taggers. The naive taggers do almost as well as the rule-based tagger on some workloads, although each performs poorly on at least one, generating configurations which are more costly than taggers which adapt to the different requirements.

3.3.3 The allocator

We evaluated the two allocators presented in Section 2.5: *branch-and-bound* first attempts to use arrays with mixed storage classes, while *branch-and-bound-bias-dedicated* prefers using arrays dedicated to one RAID level, but will use mixed arrays if they provide a better fit to the workload. We also consider a naive allocator (*All dedicated*) that produces a set of dedicated RAID 5 arrays and RAID 1/0 arrays, but no mixed arrays. We had also considered naive allocators *AllR5* and *AllR10* that ignored the information provided by the taggers, but since their effect is similar to that of using naive taggers, we will not discuss them further. Table 7 summarizes the five allocators.

Figure 14 shows the system cost for each of the allocators on different workloads. The figure shows that the two versions of the *branch and bound* allocators consistently produce the lowest cost solution; there is little or no difference between them. The naive *all-dedicated* allocator produces a higher system cost solution in several cases.

Tagger	filesystem			scientific			oltp			dss-p			fs-light		
	R5	R1/0	$	R5	R1/0	$	R5	R1/0	$	R5	R1/0	$	R5	R1/0	$
IOPSdisk	60	80	422	100	0	310	42	194	748	316	0	140	170	0	170
All R5	140	0	576	100	0	310	194	0	1101	316	0	140	170	0	170
All R10	0	140	488	0	100	340	0	194	746	0	316	148	0	170	255
Random	69	71	544	45	55	380	100	94	990	168	148	148	87	83	244

Table 6: *Comparison of storage classifications and final system cost (in thousands of dollars) by the various taggers on several realistic workloads.*

Allocator	Description
Branch and bound	Consider mixed arrays first, then dedicated arrays
Branch and bound—bias dedicated	Consider dedicated arrays first, then mixed arrays
All dedicated	Design only dedicated arrays

Table 7: *Alternative allocator designs.*

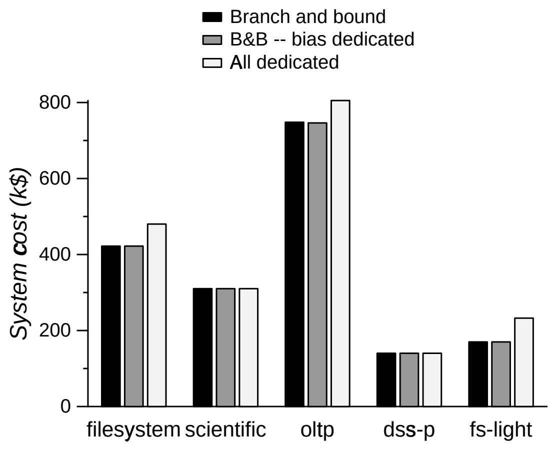


Figure 14: *Comparison of system cost for various allocators on several realistic workloads.*

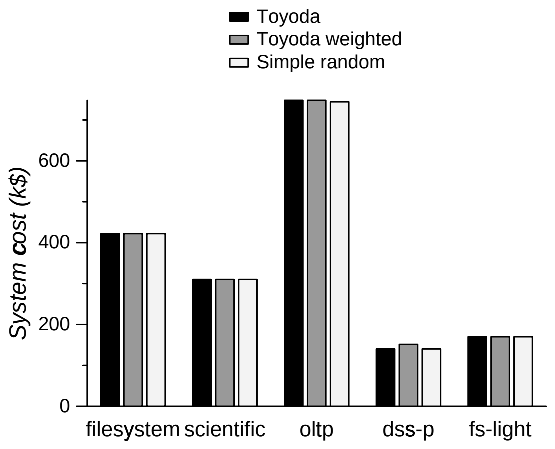


Figure 15: *Comparison of system cost for various solvers on several realistic workloads.*

3.3.4 The solver

We evaluated the three solvers presented in Section 2.6. Figure 15 shows the system cost for each of the solvers on different workloads. There is very little, if any, difference for each of the workloads presented. After clustering, aggregations of multiple input stores end up being very similar in size. Only in the (less frequent cases) in which bandwidth requirements limit cluster size there is room for different bin-packers to make slightly different decisions.

3.3.5 The optimizer

Figure 16 shows the maximum over all LUNs of peak utilization for the three optimization algorithms presented in Section 2.7. For a fixed number of LUNs, the lower the maximum peak utilization, the better the load balance.

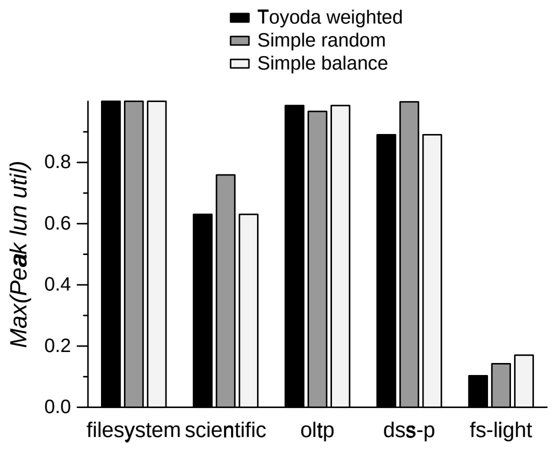


Figure 16: *Comparison of maximum* LUN *utilization for various optimizers on several realistic workloads. Lower bars reflect a better load balance.*

For most of the workloads, *Toyoda weighted* produced the most balanced configurations according to this metric; in one case, the difference between the three optimizer algorithms is small. *Simple random* is sometimes poorly suited to optimizing load balance, because it fills each LUN before moving onto the next. As a result this algorithm does not achieve as good a load balance for some workloads.

3.3.6 Summary of component design

We found that using MINERVA's baseline components resulted in system designs with low cost. In the case of the *fs-light* workload, the capacity of every disk in the system generated was fully utilized, resulting in a system that is provably of the lowest possible cost. The *dss-p* solution required only 28 disks (with a possible minimum of 24), spread across two arrays. Two arrays were necessary as the aggregate workload bandwidth required was higher than a single array controller could sustain. We believe that these results demonstrate that MINERVA can generate low cost designs that are guaranteed to meet the workload requirements.

The experiments presented in this section cover only a small portion of the space of possible workloads and possible component designs. In the future we plan to consider both a broader set of workloads and more sophisticated components. In particular, as workloads become much larger, and the assignment space becomes larger, we may need better algorithms that search more selectively.

3.4 Whole-system validation

The previous two sets of experiments have used microbenchmarks, evaluated using analytical performance models. While this kind of evaluation is helpful for showing how a system responds to particular kinds of input or changes, it does not guarantee that the system works as a whole. We address whole-system validation by configuring a system based on measurements from a real application running on a hand-configured system, then re-running the application to ensure that performance is as expected. Section 3.4.1 describes the application we selected for this validation; Section 3.4.2 shows how a human expert configured the initial system; Section 3.4.3 then describes how MINERVA improved on that configuration.

The experiment tested three hypotheses:

1. The resources used in the MINERVA-generated configuration will be no more costly than that of the hand-configured system.

2. Application performance on the generated configuration will be comparable to the hand-configured system.

3. The configuration tools will find a solution in a short time, with minimal human effort.

The steps of the experiment are shown in Figure 17. Since we wanted to measure a real system, we were restricted to the physical resources available to us, which meant the

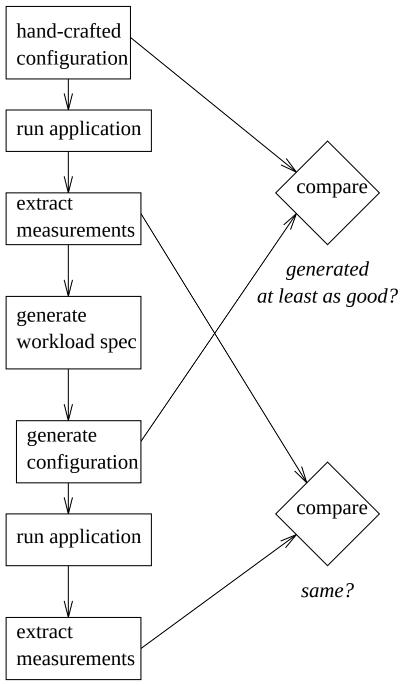


Figure 17: *Steps in the end-to-end validation test.*

single FC-30 array described in Section 3.1. This means that MINERVA cannot reduce the number of disk arrays; it can only improve the performance of the application, or reduce the number of disk drives used within the single array.

3.4.1 The application

We developed *dss*, a decision-support benchmark based loosely on TPC-D [9], as a suitably realistic application. TPC-D is a well-known benchmark that simulates a wholesale supplier database with long-running, complex queries on large tables, and exhibits interesting phasing and concurrency effects. In addition, it has the advantage of repeatability and is an open specification. This benchmark simulates a set of quasi-ad-hoc decision support queries against a large database. The TPC-D benchmark consists of 17 different queries on the data, which model various business questions that could be asked of the data, along with a stream of two different kinds of update transactions.

For our work, we omitted the update functions and picked a subset comprising 12 of the 17 TPC-D queries (we included only those queries that completed in less than one hour.) We believe that these queries are a sufficient representation of a complex decision support application. To support this, some preliminary performance predictions indicate that our *dss* workload is well-balanced in that it appears capacity-bound on small, fast disks, and bandwidth-bound on large, slow ones.

The twelve queries were divided into three parallel execution queues, with four queries in each queue (queries 2, 3, 13, 14 in queue 1; 4, 8, 15, 17 in queue 2; and 6, 11, 12, 16 in queue 3), so that expected execution time was balanced across the queues. This also replicates real-world parallel query execution better than the regular TPC-D benchmark allows.

The measurements were taken on the same server and disk array described in Section 3.1. The benchmark was run on Oracle version 8.0.5. We used the HP-UX internal trace facility to collect traces of system behavior in order to obtain detailed workload characterizations. The benchmark had 158 stores with a total capacity of 25 GB, and 112 streams with significant correlations and anti-correlations. The stores are mapped to LUNs using LVM, HP-UX's logical volume manager.

The presented results are the averages of five runs; we saw standard deviations on individual query execution times ranging from 0.4% to 12% (for the shorter queries that took less than 3 minutes to complete), although the typical standard deviation was less than 4%. The metrics for each run were individual query times and their arithmetic mean.

3.4.2 The initial configuration

The initial manual configuration was developed with the advice of experts in Hewlett-Packard's system bench-

Configuration	RAID 1/0 (4 disk)	RAID 1 (2 disk)	RAID 5 (4 disk)
Human experts	7	1	–
MINERVA baseline	–	–	4
MINERVA all RAID 1/0	4	–	–

Table 8: LUN *types and number of* LUNs *for the hand-configured, baseline* MINERVA *and* RAID *1/0 only* MINERVA *configurations*

marking team that produces the audited TPC-D benchmarks. The configuration is similar to the configurations used for official TPC-D benchmark reports, but adjusted for the size of the database and the available storage devices.

The hand-configured system tries to maximize potential parallelism by striping data as widely as possible across all the disks in the array. It uses a single FC-30 disk array, divided into 7 LUNs of 4 disks each, all configured as RAID 1/0, with the remaining two disks configured as an eighth RAID 1 LUN. It further tries to exploit parallelism by striping data as widely as possible; hence, LUNs 1 through 6 all contain part of most of the database tables. Small tables were collected onto LUN 8, and LUN 7 was used as temporary table space. The capacity utilization of about 42% is not untypical for decision-support databases.

Table 8 shows the types of LUNs for this configuration, and includes the MINERVA-generated configurations for comparison (these are described in detail in the following section).

3.4.3 Automatically-generated assignments

To generate a workload for MINERVA, we collected traces when the queries were run on the hand-generated configuration, and calculated various attributes using our workload characterization tool.

This workload information, plus device-specific information on the FC-30 array, was run through MINERVA twice. The first run used the baseline components of MINERVA, as described in Section 3.3; MINERVA designed a system using RAID 5 storage only. For comparison, a second configuration was generated with the restriction that it generate an all RAID 1/0 array.

In both cases MINERVA produced optimized configurations that significantly reduce the number of disks needed: the baseline configuration uses four RAID 5 LUNs of four disks each, and the all RAID 1/0 configuration also uses four LUNs of four disks each. A detailed examination of the assignment shows that MINERVA stripes table types across multiple LUNs, although not as aggressively as the the human experts' layout. This is because MINERVA takes into account the workload requirements of the stream accessing each table when assigning tables to LUNs. Many of the tables have only a small number of accesses, and MINERVA tends to put a larger proportion of these tables on a single LUN.

MINERVA produced these solutions in 10 to 13 minutes each.

3.4.4 Evaluation

To evaluate the quality of the assignment which was produced by MINERVA, we consider both query execution times and the per-stream averages for request rate and response time.

The graph in Figure 18 shows individual query execution times for all three configurations as well as the corresponding arithmetic mean values. Compared to the human-generated configuration, the mean query time increased by 1.9% for the baseline MINERVA configuration, and decreased (improved) by 2.5% for the all RAID 1/0 configuration.

We looked at changes in the requestRate, requestSize and runCount attributes because these attributes can significantly influence predictions for individual device utilizations and, as a result, the assignment decisions.

As expected, there were no noticeable differences in values of the requestSize and runCount attribute for these two configurations. Values of requestRate attributes were

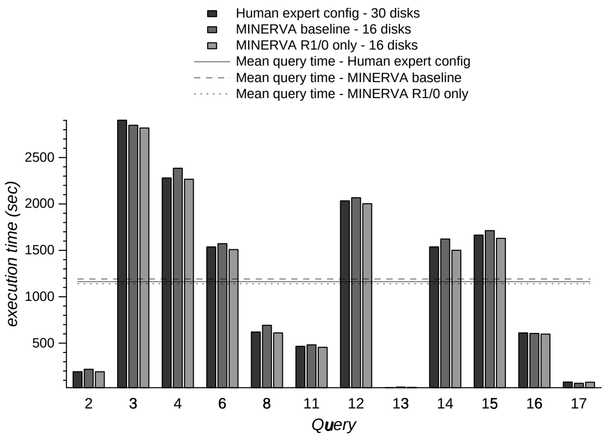


Figure 18: *Comparison of query execution times for TPC-D on a hand-generated array configuration (30 disks),* MINERVA*-generated array configuration (16 disks using* RAID *5), and restricted to* RAID *1/0 (also using 16 disks). Horizontal lines show the arithmetic mean of the execution time over all queries.*

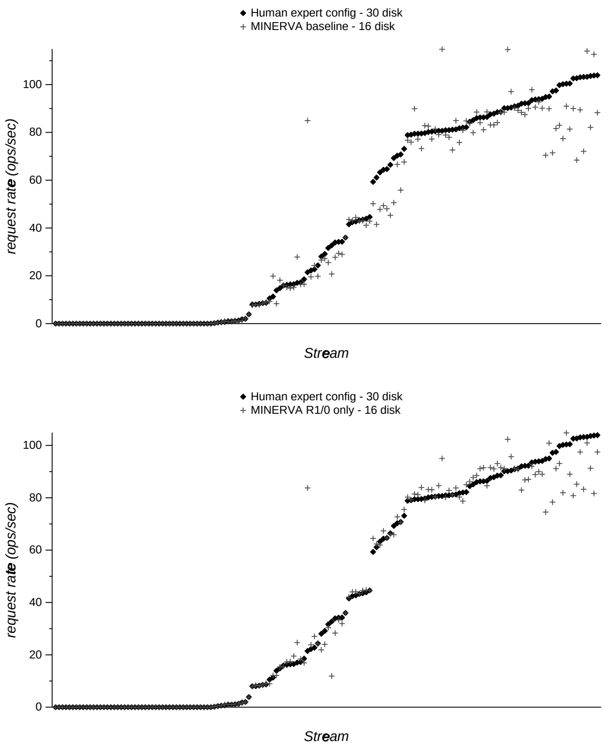


Figure 19: *Comparison of various stream requestRate attribute values for TPC-D between a hand-generated array configuration and* MINERVA*-generated array configurations (baseline using* RAID *5, and* RAID *1/0 only). Streams are presented in order of increasing attribute value in the original configuration.*

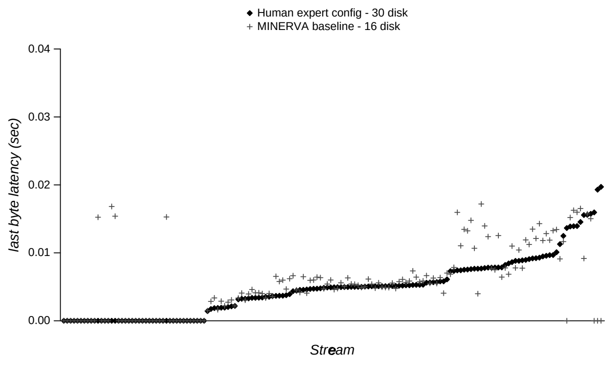
significantly different only in few cases (which can probably be attributed to normal variance in application behavior), as shown by the graphs in Figure 19.

Likewise, there is a visible discrepancy for a few streams in the response times (Figure 20). This discrepancy may explain the slight slowdown of the benchmark under the MINERVA configurations, especially for the mixed case.

The results show that MINERVA was able to reduce resource consumption (number of disks used, and therefore system cost), while preserving the performance of the human-generated configuration. A detailed examination of the results and workloads show that it uses two main characteristics of the workload to achieve this. First, it tends to combine stores with different characteristics (capacity-bound versus bandwidth-bound) that are accessed simultaneously on the same LUN. Second, it uses the knowledge of phasing behavior and anti-correlations between streams to colocate stores that don't interfere with each other, while separating stores whose access patterns are correlated onto separate LUNs. This improves load balance, not just on average, but also during the various phases.

A human expert with access to the same workload characterization data that MINERVA uses would probably design a similarly efficient system. Even for this small benchmark example (158 stores), however, the combinatorial complexity is likely to be daunting for a human; for larger systems the task becomes even more difficult.

The experimental results demonstrate that MINERVA is capable of handling complex, real-life workloads and generating reasonable layouts in a completely autonomous fashion in a short time. It produced a configuration that used significantly fewer devices, without resulting in significant application-level performance degradation. Although the response times per stream varied from the orig-



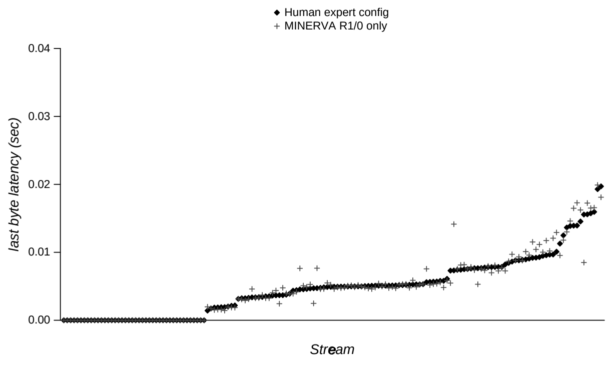


Figure 20: *Comparison of device response times for TPC-D between a hand-generated array configuration and* MINERVA*-generated array configurations (baseline using* RAID *5, and* RAID *1/0 only).*

inal configuration in a few cases, the execution times for each of the twelve queries were comparable.

4 Related work

Gelb [13] inspired much of our work, by suggesting that the logical view of data be separated from physical device characteristics to simplify the use and administration of storage. To the best of our knowledge, no existing tool will automatically design and configure storage systems. Commercial products for storage management such as IBM Tivoli, HP SureStore, Compaq SANworks, CA Unicenter and HighGround Storage Resource Manager, provide some degree of abstraction for the storage devices and data, as well as some management tools. However, these products just allow system administrators to implement their device configuration and data placement decisions more easily; a human is still required to make most decisions.

Previous research has applied optimization techniques to the *file assignment problem*. Files, typically characterized by the probability that they will be accessed from particular nodes in a network, are mapped to storage devices with the aim of optimizing some objective such as communication overhead or reliability [3, 10, 19]. Our work is most closely related to the variant where files characterized by size and access rate are placed on a given set of storage devices, with the aim of minimizing access latency or maximizing throughput [24]. However, these approaches do not design the system that will efficiently provide the needed quality of service; they make use of a given system so that an objective function is maximized, regardless of application requirements. We also allow a richer characterization of application behavior than has been considered in the past, including measures of locality, ON-OFF phasing and correlations between workloads. In addition, MINERVA also handles disk arrays, not just simple disk drives, and optimizes their parameter settings as well as the data placement.

MINERVA is the first published approach which recasts the resource provisioning problem in a constrained optimization framework. Because of that, we have evaluated diverse standard optimization techniques to determine how suitable they were for this particular problem. Our problem can be viewed as a variant of a multi-constraint knapsack or of a multidimensional bin-packing. As such, we have adapted several standard heuristics to fit our specific problem. For example, we use traditional bin-packing heuristics [6] such as randomized and greedy versions of a basic first-fit algorithm. We have also adapted the gradient metric of [22] to work with our non-linear constraints. Other common heuristic approaches include simulated annealing [11], relaxation approaches [23, 19] and genetic algorithms [5]. We have tried genetic algorithms for our problem, but found that they did no better than the simpler and faster heuristics presented in this paper, because of the high cost of evaluating each trial solution.

We use analytical models of storage devices to predict the throughput that a storage device can deliver as a function of the workload being imposed on it. There has been considerable work in analytical performance modeling of disk arrays (e.g. [15, 21, 17]), but the majority of these studies are restricted to far simpler workloads (such as Poisson request arrivals) than those we support. We found that such simplistic workload models resulted on high prediction errors for our real-life workloads, hence the need to capture more aspects of workloads and devices.

5 Conclusions

Configuring and planning large storage systems by hand is a lengthy process, prone to errors and guesswork, and there is no good way to evaluate the quality of the results short of building the system—which is expensive, slow, and not always possible. MINERVA addresses all of these issues: it designs storage systems that are as good as or better than the ones produced by people; it does so in minutes rather than days; and it accompanies them with predictions of the resulting system's performance. This makes MINERVA capable of providing substantial support for the resource-provisioning step of storage system design.

Our evaluation of MINERVA showed that:

- it can generate good (and sometimes, provably optimal) configurations over a wide range of synthetic workloads, as well as a complex, multi-phase real-world benchmark;
- the set of attributes used to describe storage devices and workloads seems to capture what is needed to make good configuration decisions;
- the sequencing of tasks (choosing the storage devices, configuring them, and placing data onto them) is a viable way of breaking the circular dependencies between the assignment and configuration subtasks;
- the choice of algorithms in each of the components was important, and the choices used in MINERVA were appropriate in most areas (save that the *simple random* heuristic was slightly better than *Toyoda* for the solver step).

The experimental results with the *dss* benchmark demonstrated that MINERVA is capable of handling realistic workloads. For this test case, MINERVA produced two configurations in a relatively short amount of time that had performance comparable to the hand-generated one, and were much cheaper. This was done without oversimplifying the problem or its solution: for example, all three configurations spread data base tables over multiple LUNs. The MINERVA solution also accurately predicted the performance of the resulting storage system.

In our future work, we plan to try backtracking solvers; to generalize the current tagger and allocator heuristics to support a wider range of arrays; to extend our device models to increase their accuracy, broaden their scope, and improve their efficiency; and to extend the workload requirements to include reliability. Another possible improvement is to make MINERVA handle other optimization objectives than minimizing the purchase cost of the system (e.g. "minimize yearly management costs"). This extension impacts almost all of the current MINERVA components; in the general case, the objective would be another input supplied by the system administrator to our tool. We believe that our formulation as a constrained optimization problem is powerful enough to handle all these extensions without fundamental changes. We are also engaged in gathering a library of real workloads with heterogeneity, phasing, and complicated access patterns to drive MINERVA to its limits.

Although progress is needed before human-staffed capacity planning centers become obsolete, MINERVA generates good storage system designs with much less expense and time than is the case today. Our results indicate that MINERVA is an important step towards the ultimate goal of developing a fully-automated capacity planning system. Of equal importance, MINERVA was able to successfully predict the resulting system's performance before it was built. We conclude that automatic storage system configuration is both feasible and promising.

Acknowledgments

We are grateful to Eric Anderson, who wrote the set of scripts for the TPC-D experiments and provided valuable feedback. Thanks are also due to Kimberly Keeton, Mustafa Uysal, Michael Hobbs, Ram Swaminathan and the anonymous reviewers for their feedback on earlier drafts of this paper.

References

[1] 3Com Corporation. *Gigabit ethernet comes of age*, June 1996. Technology white paper.

[2] ANSI. *Fibre Channel Arbitrated Loop*, April 1996. Standard X3.272-1996.

[3] B. Awerbuch, Y. Bartal, and A. Fiat. Competitive distributed file allocation. In *Proc. 25th ACM Symposium on Theory of Computing (STOC)*, pages 164–73, May 1993.

[4] E. Borowsky, R. Golding, A. Merchant, L. Schreier, E. Shriver, M. Spasojevic, and J. Wilkes. Using attribute-managed storage to achieve QoS. In *Proc. 5th Intl. Workshop on Quality of Service*, June 1997.

[5] P. Chu and J. Beasley. A genetic algorithm for the generalized assignment problem. *Computers and Operations Research*, 24(1):17–23, 1997.

[6] E. Coffman, M. Garey, and D. Johnson. Approximation algorithms for bin-packing: An updated survey. In G. Ausiello, M. Lucertini, and P. Serafini, editors, *Algorithm Design for Computer System Design*, pages 49–106. Springer-Verlag, 1984.

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

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

[9] Transaction Processing Performance Council. *TPC benchmark D, standard specification, revision 1.2*, November 1996.

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

[11] A. Drexl. A simulated annealing approach to the multiconstraint zero-one knapsack problem. *Computing*, 40(1):1–8, 1988.

[12] M. Garey and D. Johnson. *Computers and intractability: a guide to the theory of NP-completeness.* W.H. Freeman, San Francisco, 1979.

[13] J. P. Gelb. System managed storage. *IBM Systems Journal*, 28(1):77–103, 1989.

[14] G. H. Kuenning. Kitrace—precise interactive measurement of operating systems kernels. *Software—Practice and Experience*, 25(1):1–21, January 1995.

[15] J. Menon and D. Mattson. Performance of disk arrays in transaction processing environments. In *Proc. 12th ICDCS*, pages 302–309, June 1992.

[16] A. Merchant and G. Alvarez. Disk array models in Minerva. Technical Report HPL-2001-118, Hewlett-Packard Laboratories, April 2001. http://www.hpl.hp.com/techreports.

[17] A. Merchant and P. S. Yu. Analytic modeling of clustered RAID with mapping based on nearly random permutation. *IEEE Trans. Computers*, 45(3):367–373, 1996.

[18] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. ACM SIGMOD*, pages 109–116, 1988.

[19] K. R. Pattipati and J. L. Wolf. A file assignment problem model for extended local area network environments. In *Proc. 10th ICDCS*, pages 554–61, May 1990.

[20] C. Ruemmler and J. Wilkes. Unix disk access patterns. In *Proc. Winter USENIX*, pages 405–420, January 1993.

[21] A. Thomasian and J. Menon. Performance analysis of RAID5 disk arrays with a vacationing server model for rebuild mode operation. In *Proc. 10th Intl. Conf. Data Eng.*, pages 111–119, February 1994.

[22] Y. Toyoda. A simplified algorithm for obtaining approximate solutions to zero-one programming problems. *Management Science*, 21(12):1417–27, August 1975.

[23] M. Trick. A linear relaxation heuristic for the generalized assignment problem. *Naval Research Logistics*, 39:137–51, 1992.

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