USENIX Association

Proceedings of the FAST 2002 Conference on File and Storage Technologies

Monterey, California, USA
January 28-30, 2002

Conference on File and Storage Technologies (FAST'02), pp. 189–201,
28–30 January 2002, Monterey, CA. (USENIX, Berkeley, CA.)

Selecting RAID levels for disk arrays

Eric Anderson, Ram Swaminathan, Alistair Veitch, Guillermo A. Alvarez and John Wilkes
Storage and Content Distribution Department,
Hewlett-Packard Laboratories, 1501 Page Mill Road, Palo Alto, CA 95014
{anderse,swaram,aveitch,galvarez,wilkes}@hpl.hp.com

Abstract

Disk arrays have a myriad of configuration parameters that interact in counter-intuitive ways, and those interactions can have significant impacts on cost, performance, and reliability. Even after values for these parameters have been chosen, there are exponentially-many ways to map data onto the disk arrays' logical units. Meanwhile, the importance of correct choices is increasing: storage systems represent an growing fraction of total system cost, they need to respond more rapidly to changing needs, and there is less and less tolerance for mistakes. We believe that automatic design and configuration of storage systems is the only viable solution to these issues. To that end, we present a comparative study of a range of techniques for programmatically choosing the RAID levels to use in a disk array.

Our simplest approaches are modeled on existing, manual rules of thumb: they "tag" data with a RAID level before determining the configuration of the array to which it is assigned. Our best approach simultaneously determines the RAID levels for the data, the array configuration, and the layout of data on that array. It operates as an optimization process with the twin goals of minimizing array cost while ensuring that storage workload performance requirements will be met. This approach produces robust solutions with an average cost/performance 14–17% better than the best results for the tagging schemes, and up to 150–200% better than their worst solutions.

We believe that this is the first presentation and systematic analysis of a variety of novel, fully-automatic RAID-level selection techniques.

1 Introduction

Disk arrays are an integral part of high-performance storage systems, and their importance and scale are growing as continuous access to information becomes critical to the day-to-day operation of modern business.

Before a disk array can be used to store data, values for many configuration parameters must be specified: achieving the right balance between cost, availability, and application performance needs depends on many correct decisions. Unfortunately, the tradeoffs between the choices are surprisingly complicated. We focus here on just one of these choices: which RAID level, or data-redundancy scheme, to use.

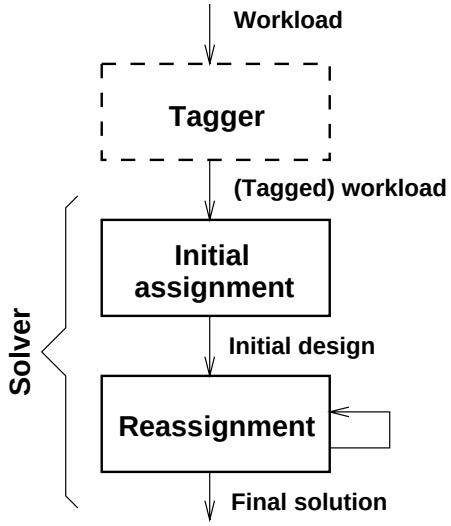


Figure 1: The decision flow in making RAID level selections, and mapping stores to devices. If a tagger is present, it irrevocably assigns a RAID level to each store before the solver is run; otherwise, the solver assigns RAID levels as it makes data layout decisions. Some variants of the solver allow revisiting this decision in a final reassignment pass; others do not.

The two most common redundancy schemes are RAID 1/0 (striped mirroring), where every byte of data is kept on two separate disk drives, and striped for greater I/O parallelism, and RAID 5 [20], where a single parity block protects the data in a stripe from disk drive failures. RAID 1/0 provides greater read performance and failure tolerance—but requires almost twice as many disk drives to do so. Much prior work has studied the properties of different RAID levels (e.g., [2, 6, 20, 11]).

Disk arrays organize their data storage into *Logical Units*, or LUs, which appear as linear block spaces to their clients. A small disk array, with a few disks, might support up to 8 LUs; a large one, with hundreds of disk drives, can support thousands. Each LU typically has a given RAID level—a redundancy mapping onto one or more underlying physical disk drives. This decision is made at LU-creation time, and is typically irrevocable: once the LUhas been formatted, changing its RAID level requires copying all the data onto a new LU.

Following previous work [7, 25], we describe the workloads to be run on a storage system as sets of *stores* and *streams*. A store is a logically contiguous array of bytes, such as a file system or a database table, with a size typically measured in gigabytes; a stream is a set of access patterns on a store, described by attributes such as request rate, request size, inter-stream phasing information, and sequentiality. A RAID level must be decided for each store in the workload; if there are k RAID levels to choose from and m stores in the workload, then there are k^m feasible configurations. Since $k \geq 2$ and m is usually over a hundred, this search space is too large to explore exhaustively by hand.

Host-based logical volume managers (LVMs) complicate matters by allowing multiple stores to be mapped onto a single LU, effectively blending multiple workloads together.

There is no single best choice of RAID level: the right choice for a given store is a function of the access patterns on the store (e.g., reads versus writes; small versus large; sequential versus random), the disk array's characteristics (including optimizations such as write buffer merging [23], segmented caching [26], and parity logging [21]), and the effects of other workloads and stores assigned to the same array [18, 27].

In the presence of these complexities, system administrators are faced with the tasks of (1) selecting the type and number of arrays; (2) selecting the size and RAID level for each LU in each disk array; and (3) placing stores on the resulting LUs. The administrators' goals are operational in nature, such as minimum cost, or maximum reliability for a given cost—while satisfying the performance requirements of client applications. This is clearly a very difficult task, so manual approaches apply rules of thumb and gross over-provisioning to simplify the problem (e.g., "stripe each database table over as many RAID 1/0 LUs as you can"). Unfortunately, this paper shows that the resulting configurations can cost as much as a factor of two to three more than necessary. This matters when the cost of a large storage system can easily be measured in millions of dollars and represents more than half the total system hardware cost. Perhaps even more important is the uncertainty that surrounds a manually-designed system: (how well) will it meet its performance and availability goals?

We believe that automatic methods for storage system design [1, 5, 7, 4] can overcome these limitations, because they can consider a wider range of workload interactions, and explore a great deal more of the search space than any manual method. To do so, these automatic methods need to be able to make RAID-level selection decisions, so the question arises: what is the best way to do this selection? This paper introduces a variety of approaches for answering this question.

The rest of the paper is organized as follows. In Section 2 we describe the architecture of our RAID level selection infrastructure. We introduce the schemes that operate on a per-store basis in Section 3, and in Section 4 we present a family of methods that simultaneously account for prior data placement and RAID level selection decisions. In Section 5, we compare all the schemes by doing experiments with synthetic and realistic workloads. We conclude in Sections 6 and 7 with a review of related work, and a summary of our results and possible further research.

2 Automatic selection of RAID levels

Our approach to automating storage system design relies on a *solver*: a tool that takes as input (1) a workload description and (2) information about the target disk array types and their configuration choices. The solver's output is a design for a storage system capable of supporting that workload.

In the results reported in this paper, we use our third-generation solver, *Ergastulum* [5] (prior solver generations were called Forum [7] and Minerva [1]). Our solvers are constraint-based optimization systems that use analytical and interpolation-based *performance models* [3, 7, 18, 23] to determine whether performance constraints are being met by a tentative design. Although such models are less accurate than trace-driven simulations, they are much faster, so the solver can rapidly evaluate many potential configurations.

As illustrated in Figure 1, the solver designs configurations for one or more disk arrays that will satisfy a given workload. This includes determining the array type and count, selecting the configuration for each LU in each array, and assigning the stores onto the LUs. In general, solvers rely on heuristics to search for the solution that minimizes some user-specified goal or objective. All experiments in this paper have been run with the objective of minimizing the hardware cost of the system being designed, while satisfying the workload's performance re-

quirements.

During the optional *tagging* phase, the solver examines each store, and tags it with a RAID level based on the attributes of the store and associated streams.

During the *initial assignment* phase, Ergastulum explores the array design search space by first randomizing the order of the stores, and then running a best-fit search algorithm [10, 15, 16] that assigns one store at a time into a tentative array design. Given two possible assignments of a store onto different LUs, the solver uses an externally-selected *goal function* to choose the "best" assignment. While searching for the best placement of a store, the solver will try to assign it onto the existing LUs, to purchase additional LUs on existing arrays, and to purchase additional arrays. A goal function that favors lower-cost solutions will bias the solver towards using existing LUs where it can.

At each assignment, the solver uses its performance models to perform *constraint checks.* These checks ensure that the result is a feasible, valid solution that can accommodate the capacity and performance requirements of the workload.

The *reassignment* phase of the solver algorithm attempts to improve on the solution found in the initial phase. The solver randomly selects a complete LU from the existing set, removes all the stores from it, and *reassigns* them, just as in the first phase. It repeats this process until every single LUs has been reassigned a few times (a configurable parameter that we set to 3). The reassignment phase is designed to help the solver avoid local minima in the optimization search space. This phase produces a near-optimal assignment of stores to LUs. For more details on the optimality of the assignments and on the operation of the solver, we refer the interested reader to [5].

2.1 Approaches to RAID level selection

We explore two main approaches to selecting a RAID level:

1. *Tagging* approaches: These approaches perform a pre-processing step to tag stores with RAID levels before the solver is invoked. Once tagged with a RAID level, a store cannot change its tag, and it must be assigned to an LU of that type. Tagging decisions consider each store and its streams in isolation. We consider two types of taggers: *rule-based*, which examine the size and type of I/Os; and *model-based*, which use performance models to make their decisions. The former tend to have many *ad hoc* parameter settings; the latter have fewer, but also need performance-related data for a particular disk array type. In some cases we use the same performance models as we later apply in the solver.

2. *Solver-based*, or *integrated*, approaches: These omit the tagging step, and defer the choice of RAID level until data-placement decisions are made by the solver. This allows the RAID level decision to take into account interactions with the other stores and streams that have already been assigned.

   We explored two variants of this approach: a *partially adaptive* one, in which the RAID level of an LU is chosen when the first store is assigned to it, and cannot subsequently be changed; and a *fully adaptive* variant, in which any assignment pass can revisit the RAID level decision for an LU at any time during its best-fit search. In both cases, the reassignment pass can still change the bindings of stores to LUs, and even move a store to an LU of a different RAID level.

   Neither variant requires any *ad hoc* constants, and both can dynamically select RAID levels. The fully adaptive approach has greater solver complexity and longer running times, but results in an exploration of a larger fraction of the array design search space.

Table 1 contrasts the four families of RAID level selection methods we studied.

We now turn to a detailed description of these approaches.

3 Tagging schemes

Tagging is the process of determining, for each store in isolation, the appropriate RAID level for it. The solver must later assign that store to an LU with the required RAID level. The tagger operates exactly once on each store in the input workload description, and its decisions are final. We followed this approach in previous work [1] because the decomposition into two separate stages is natural, is easy to understand, and limits the search space that must be explored when designing the rest of the storage system.

We explore two types of taggers: one type based on rules of thumb and the other based on performance models.

3.1 Rule-based taggers

These taggers make their decisions using rules based on the size and type of I/Os performed by the streams. This is the approach implied by the original RAID paper [20], which stated, for example, that RAID 5 is bad for "small"

Approach	*Goal functions*	*Solver*	*Summary*
Rule-based tagging	no change	simple	many constants, variable results
Model-based tagging	no change	simple	fewer constants, variable results
Partially-adaptive solver	special for initial assignment	simple	good results, limited flexibility
Fully-adaptive solver	no change	complex	good results, flexible but slower

Table 1: The four families of RAID-level selection methods studied in this paper. The two tagging families use either rule-based or model-based taggers. The model-based taggers use parameters appropriate for the array being configured. The fully adaptive family uses a substantially more complex solver than the other families. The *Goal functions* column indicates whether the same goal functions are used in both solver phases: initial assignment and reassignment. The *Summary* column provides an evaluation of their relative strengths and weaknesses.

writes, but good for "big" sequential writes. This approach leads to a large collection of device-specific constants, such as the number of seeks per second a device can perform, and device-specific thresholds, such as where exactly to draw the line between a "mostly-read" and a "mostly-write" workload. These thresholds could, in principle, be workload-independent, but in practice, we found it necessary to tune them experimentally to our test workloads and arrays, which means that there is no guarantee they will work as well on any other problem.

The rules we explored were the following. The first three taggers help provide a measure of the cost of the *laissez-faire* approaches. The remaining ones attempt to specify concrete values for the rules of thumb proposed in [20].

1. *random:* pick a RAID level at random.
2. *allR10:* tag all stores RAID 1/0.
3. *allR5:* tag all stores RAID 5.
4. *R5BigWrite:* tag a store RAID 1/0 unless it has "mostly" writes (the threshold we used was at least 2/3 of the I/Os), and the writes are also "big" (greater than 200 KB, after merging sequential I/O requests together).
5. *R5BigWriteOnly:* tag a store RAID 1/0 unless it has "big" writes, as defined above.
6. *R10SmallWrite:* tag a store RAID 5 unless it has "mostly" writes and the writes are "small" (i.e., not "big").
7. *R10SmallWriteAggressive:* as R10SmallWrite, but with the threshold for number of writes set to 1/10 of the I/Os rather than 2/3.

In practice, we found these rules needed to be augmented with an additional rule to determine if a store was *capacity-bound* (i.e., if space, rather than performance, was likely to be the bottleneck resource). A capacity-bound store was always tagged as RAID 5. This rule required additional constants, with units of bytes-per-second/GB and seeks-per-second/GB; these values had to be computed independently for each array. (Also, it is unclear what to do if an array can support different disk types with different capacity/performance ratios.)

We also evaluated each of these taggers without the capacity-bound rule. These variations are shown in the graphs in Section 5 by appending *Simple* to each of the tagger names.

3.2 Model-based taggers

The second type of tagging methods we studied used array-type-specific performance models to estimate the effect of assigning a store to an LU, and made a selection based on that result.

The first set of this type use simple performance models that predict the number of back-end I/Os per second (*IOPS*) that will result from the store being tagged at each available RAID level, and then pick the RAID level that minimizes that number. This removes some *ad hoc* thresholds such as the size of a "big" write, but still requires array-specific constants to compute the IOPS estimates. These taggers still need the addition of the capacity-bound rule to get decent results. The IOPS-based taggers we study are:

8. *IOPS:* tag a store RAID 1/0 if the estimated IOPS would be smaller on a RAID 1/0 than on a RAID 5 LU. Otherwise tag it as RAID 5.
9. *IOPS-disk:* as *IOPS* except the IOPS estimates are divided by the number of disks in the LU, resulting in a per-disk IOPS measure, rather than a per-LU measure. The intent is to reflect the potentially different number of disks in RAID 1/0 and RAID 5 LUs.
10. *IOPS-capacity:* as *IOPS* except the IOPS estimates are multiplied by the ratio of raw (unprotected) capacity divided by effective capacity. This measure factors in the extra capacity cost associated with RAID 1/0.

The second set of model-based taggers use the same performance models that needed to be constructed and calibrated for the solver anyway, and does not depend on any *ad hoc* constants. These taggers use the models to compute, for each available RAID level, the percentage changes in the LU's utilization and capacity that will result from choosing that level, under the simplifying assumption that the LU is dedicated solely to the store being tagged. We then form a 2-dimensional vector from these two results, and then pick the RAID level that minimizes:

11. *PerfVectLength:* the length (L_2 norm) of the vector;
12. *PerfVectAvg:* the average magnitude (L_1 norm) of the components;
13. *PerfVectMax:* the maximum component (L_∞ norm);
14. *UtilizationOnly:* just the utilization component, ignoring capacity.

4 Solver-based schemes

When we first tried using the solver to make all RAID-level decisions, we discovered it worked poorly for two related reasons:

1. The solver's goal functions were cost-based, and using an existing LU is always cheaper than allocating a new one.
2. The solver chooses a RAID level for a new LU when it places the first store onto it – and a 2-disk RAID 1/0 LU is always cheaper than a 3- or more-disk RAID 5 LU. As a result, the solver would choose a RAID 1/0 LU, fill it up, and then repeat this process, even though the resulting system would cost more because of the additional disk space needed for redundancy in RAID 1/0. (Our tests on the FC-60 array (described in Section 5.2) did not have this discrepancy because we arranged for the RAID 1/0 and RAID 5 LUs to contain six disks each, to take best advantage of the array's internal bus structure.)

We explored two options for addressing these difficulties. First, we used a number of different initial goal functions that ignored cost, in the hope that this would give the reassignment phase a better starting point. Second, we extended the solver to allow it to change the RAID level of an LU even after stores had been assigned to it.

We refer to the first option as *partially-adaptive*, because it can change the RAID level associated with an individual store—but it still fixes an LU's RAID level when the first store is assigned to it. Adding another goal function to the solver proved easy, so we tried several in a search for one that worked well. We refer to the second option as *fully-adaptive* because the RAID level of the store and the LUs can be changed at almost any time. It is more flexible than the partially-adaptive one, but required more extensive modifications to the solver's search algorithm.

4.1 Partially-adaptive schemes

The partially-adaptive approach works around the problem of the solver always choosing the cheaper, RAID 1/0 LUs, by ignoring cost considerations in the initial selection – thereby avoiding local cost-derived minima – and reintroducing cost in the reassignment stage. By allowing more LUs with more-costly RAID levels, the reassignment phase would have a larger search space to work within, thereby producing a better overall result.

Even in this scheme, the solver still needs to decide whether a newly-created LU should be labeled as RAID 5 or RAID 1/0 during the initial assignment pass. It does this by means of a *goal function*. The goal function can take as input the performance, capacity, and utilization metrics for all the array components that would be involved in processing accesses to the store being placed into the new LU. We devised a large number of possible initial goal functions, based on the combinations of these metrics that seemed reasonable. While it is possible that there are other, better initial goal functions, we believe we have good coverage of the possibilities. Here is the set we explored:

1. *allR10:* always use RAID 1/0.
2. *allR5:* always use RAID 5.
3. *AvgOfCapUtil:* minimize the average of capacities and utilizations of all the disks (the L_1 norm).
4. *LengthOfCapUtil:* minimize the sum of the squares of capacities and utilizations (the L_2 norm) of all the disks.
5. *MaxOfCapUtil:* minimize the maximum of capacities and utilizations of all the disks (the L_∞ norm).
6. *MinAvgUtil:* minimize the average utilizations of all the array components (disks, controllers and internal buses).
7. *MaxAvgUtil:* maximize the average utilizations of all the array components (disks, controllers and internal buses).

8. *MinAvgΔUtil:* minimize the arithmetic mean of the change in utilizations of all the array components (disks, controllers and internal buses).

9. *MinAvgΔUtilPerRAIDdisk:* as with scheme (8), but first divide the result by the number of *physical* disks used in the LU.

10. *MinAvgΔUtilPerDATAdisk:* as with scheme (8), but first divide the result by the number of *data* disks used in the LU.

11. *MinAvgΔUtilTimesRAIDdisks:* as with scheme (8), but first multiply the result by the number of *physical* disks used in the LU.

12. *MinAvgΔUtilTimesDATAdisks:* as with scheme (8), but first multiply the result by the number of *data* disks used in the LU.

The intent of the various disk-scaling schemes (9–12) was to explore ways of incorporating the size of an LU into the goal function.

Goal functions for the reassignment phase make minimal system cost the primary decision metric, while selecting the right kind of RAID level is used as a tie-breaker. As a result, there are fewer interesting choices of goal function during this phase, and we used just two:

1. *PriceThenMinAvgUtil:* lowest cost, ties resolved using scheme (6).

2. *PriceThenMaxAvgUtil:* lowest cost, ties resolved using scheme (7).

During our evaluation, we tested each of the reassignment goal functions in combination with all the initial-assignment goal functions listed above.

4.2 Fully-adaptive approach

As we evaluated the partly-adaptive approach, we found several drawbacks that led us to try the more flexible, fully-adaptive approach:

- After the goal functions had become cost-sensitive in the reassignment phase, new RAID 5 LUs would not be created. Solutions would suffer if there were too few RAID 5 LUs after initial assignment.
- It was not clear how well the approach would extend to more than two RAID levels.
- Although we were able to achieve good results with the partially-adaptive approach, the reasons for the results were not always obvious, hinting at a possible lack of robustness.

To address these concerns, we extended the search algorithm to let it dynamically switch the RAID level of a given LU. Every time the solver considers assigning a store to an LU (that may already have stores assigned to it), it evaluates whether the resulting LU would be better off with a RAID 1/0 or RAID 5 layout.

The primary cost of the fully-adaptive approach is that it requires more CPU time than the partially-adaptive approach, which did not revisit RAID-level selection decisions. In particular, the fully-adaptive approach roughly doubles the number of performance-model evaluations, which are relatively expensive operations. But fully-adaptive approach has several advantages: the solver is no longer biased towards a given RAID level, because it can identify the best choice at all stages of the assignment process. Adding more RAID levels to choose from is also possible, although the total computation time grows roughly linearly with the number of RAID levels. And there no longer is a need for a special goal function during the initial assignment phase.

Our experiments showed that, with two exceptions, the *PriceThenMinAvgUtil* and *PriceThenMaxAvgUtil* goal functions produced identical results for all the fully-adaptive schemes. Each was better for one particular workload; we selected *PriceThenMaxAvgUtil* for our experiments, as it resulted in the lowest average cost. We found that it was possible to improve the fully-adaptive results slightly (so that they always produced the lowest cost) by increasing the number of reassignment passes to 5, but we did not do so to keep the comparison with the partially-adaptive solver as fair as possible.

5 Evaluation

In this section, we present an experimental evaluation of the effectiveness of the RAID level selection schemes discussed above.

We took workload specifications from [1] and from traces of a validated TPC-D configuration. We used the Ergastulum solver to design storage systems to support these workloads, ensuring for each design that the performance and capacity needs of the workload would be met. To see if the results were array-specific, we constructed designs for two different disk array types.

The primary evaluation criterion for the RAID-level selection schemes was the cost of the generated configurations, because our performance models [3, 18] predicted that all the generated solutions would support the workload performance requirements. The secondary criterion was the CPU time taken by each approach.

We chose not to run the workloads on the target phys-

Workload	*Capacity*	*#stores*	*#streams*	*Access size*		*Run count*		*%reads*
filesystem	0.09 TB	140	140	20.0 KB	(±13.8)	2.6	(±1.3)	64.2%
scientific	0.19 TB	100	200	640.0 KB	(±385.0)	93.5	(±56.6)	20.0%
oltp	0.19 TB	194	182	2.0 KB	(±0.0)	1.0	(±0.0)	66.0%
fs-light	0.16 TB	170	170	14.8 KB	(±7.3)	2.1	(±0.7)	64.1%
tpcd30	0.05 TB	316	224	27.6 KB	(±19.3)	57.7	(±124.8)	98.0%
tpcd30-2x	0.10 TB	632	448	27.6 KB	(±19.3)	57.7	(±124.8)	98.0%
tpcd30-4x	0.20 TB	1264	896	27.6 KB	(±19.3)	57.7	(±124.8)	98.0%
tpcd300-1	1.95 TB	911	144	53.5 KB	(±12.8)	1.13	(±0.1)	98.3%
tpcd300-5	1.95 TB	935	374	49.1 KB	(±10.6)	1.23	(±1.9)	92.7%
tpcd300-7	1.95 TB	941	304	51.1 KB	(±10.7)	1.12	(±0.1)	95.0%
tpcd300-9	1.95 TB	933	399	49.8 KB	(±10.6)	1.20	(±1.9)	85.6%
tpcd300-10	1.95 TB	910	321	45.3 KB	(±12.3)	1.28	(±2.2)	80.3%

Table 2: Characteristics of workloads used in experiments. "Run count" is the mean 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. *tpcd* 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.

ical arrays because it was not feasible. First, we did not have access to the applications used for some of the workloads—just traces of them running. Second, there were too many of them. We evaluated over a thousand configurations for the results presented; many of the workloads run for hours. Third, some of the resulting configurations were too large for us to construct. Fortunately, previous work [1] with the performance models we use indicated that their performance predictions are sufficiently accurate to allow us to feel confident that our comparisons were fair, and that the configurations designed would indeed support the workloads.

5.1 Workloads

To evaluate the RAID-level selection schemes, we used a number of different workloads that represented both traces of real systems and models of a diverse set of applications: an active file system (*filesystem*), a scientific application (*scientific*), an on-line transaction processing benchmark (*oltp*), a lightly-loaded filesystem (*fs-light*), a 30 GB TPC-D decision-support benchmark, running three queries in parallel until all of them complete (*tpcd30*), the *tpcd30* workload duplicated (as if they were independent, but simultaneous runs) 2 and 4 times (*tpcd30-2x* and *tpcd30-4x*), and the most I/O-intensive queries (i.e., 1, 5, 7, 9 and 10) of the 300 GB TPC-D benchmark run one at a time on a validated configuration (*tpcd300-query-N*).

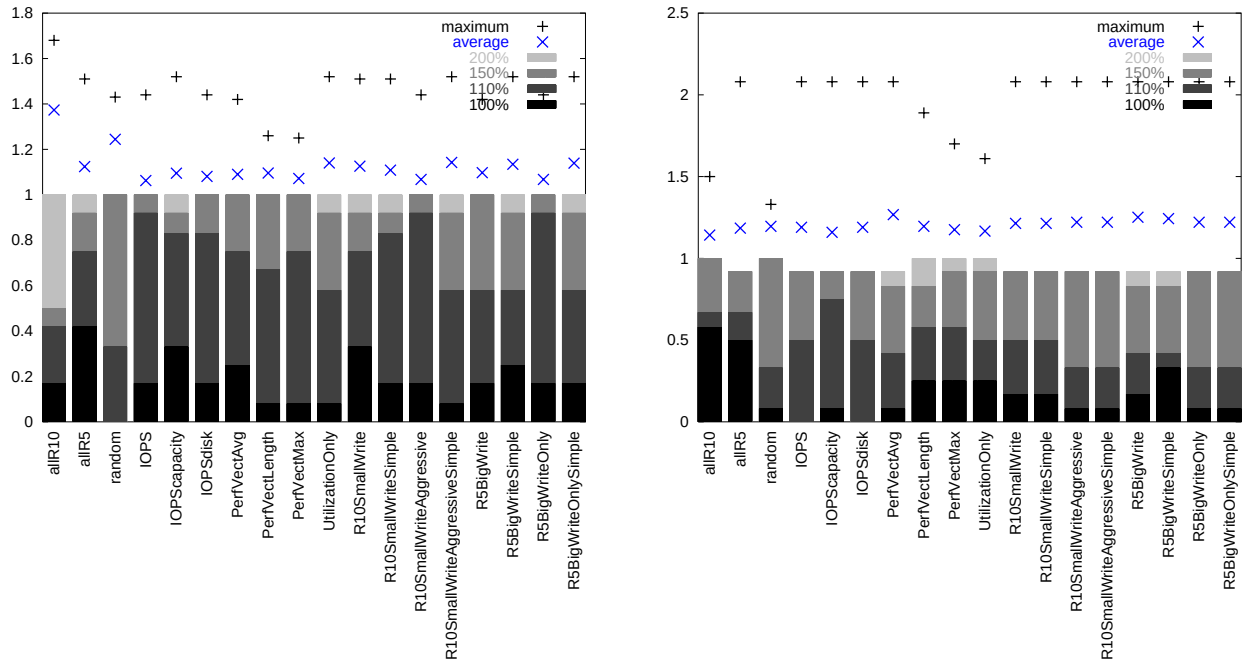
Table 2 summarizes their performance characteristics. Detailed information on the derivations of these workloads can be found in [1].

5.2 Disk arrays

We performed experiments using two of the arrays supported by our solver: the Hewlett-Packard SureStore Model 30/FC High Availability Disk Array (FC-30, [12]) and the Hewlett-Packard SureStore E Disk Array FC-60 (FC-60, [13]), as these are the ones for which we have calibrated models.

The FC-30 is characteristic of a low-end, stand-alone disk array of 3–4 years ago. An FC-30 has up to 30 disks of 4 GB each, two redundant controllers (to survive a controller failure) and 60 MB of battery-backed cache (NVRAM). Each of the two array controllers is connected to the client host(s) over a 1 Gb/s FibreChannel network. Our FC-30 performance models [18] have an average error of ±6% and a worst-case error of ±20% over a reasonable range of LU sizes.

The FC-60 is characteristic of modern mid-range arrays. An FC-60 array can have up to 60 disks, placed in up to six disk enclosures. Each of the two array controllers is connected to the client host(s) over a 1 Gb/s FibreChannel network. Each controller may have up to 512 MB of NVRAM. The controller enclosure contains a backplane bus that connects the controllers to the disk enclosures, via six 40 MB/s ultra-wide SCSI busses. Disks of up to 72 GB can be used, for a total unprotected capacity of 4.3 TB. Dirty blocks are mirrored in both controller caches, to prevent data loss if a controller fails. Our interpolation-based FC-60 performance models [3] have an average error of about 10% over a fairly wide range of configurations.



(a) **FC-30 array**

(b) **FC-60 array**

Figure 2: Tagger results for the FC-30 and FC-60 disk arrays. The results for each tagger are plotted within a single bar of the graph. Over all workloads, the bars show the proportion of time each tagger resulted in a final solution with the lowest cost (as measured over all varieties of RAID level selection), within 110% of the lowest cost, within 150% of the lowest cost and within 200% of the cost. The taller and darker the bar, the better the tagger. Above each bar, the points show the maximum (worst) and average results for the tagger, as a multiple of the best cost. The *allR10* and *allR5* taggers tag all stores as RAID 1/0 or RAID 5 respectively. The *random* tagger allocates stores randomly to either RAID level. The *IOPS* models are based on very simple array models. The *PerfVect...* and *UtilizationOnly* taggers are based on the complete analytical models as used by the solver. The remaining taggers are rule-based.

5.3 Comparisons

As described above, the primary criteria for comparison for all schemes is that of total system cost.

5.3.1 Tagger results

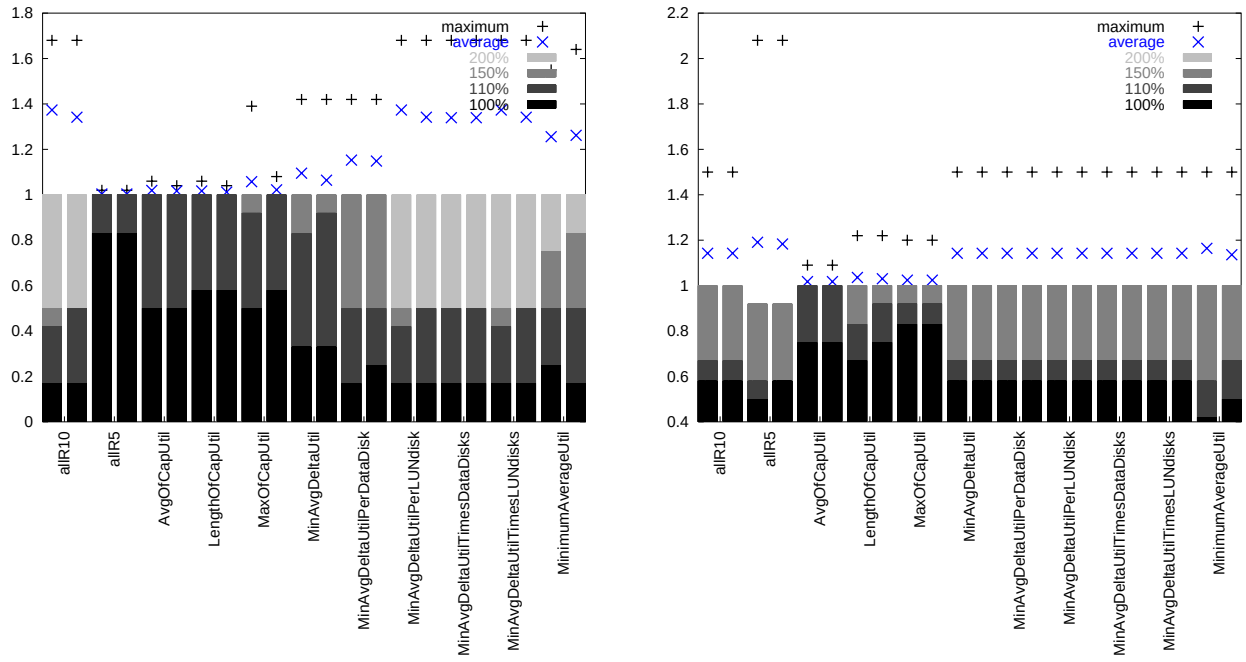
Figure 2 shows the results for each of the taggers for the FC-30 and FC-60 arrays. There are several observations and conclusions we can draw from these results.

First, there is no overall winner. Within each array type, it is difficult to determine what the optimal choice is. For instance, compare the *PerfVectMax* and *IOPS* taggers for the FC-30 array. *IOPS* has a better average result than *PerfVectMax*, but performs very badly on one workload (*filesystem*), whereas *PerfVectMax* is much better in the worst case. Depending on the user's expected range of workloads, either one may be the right choice.

When comparing results across array types, the situation is even less clear—the sets of best taggers for each array are completely disjoint. Hence, the optimal choice of RAID level varies widely from array to array, and no single set of rules seems to work well for all array types, even when a subset of all array-specific parameters (such as the test for capacity-boundedness) is used in addition.

Second, the results for the FC-60 are, in general, worse than for the FC-30. In large part, this is due to the relative size and costs of the arrays. Many of the workloads require a large number (more than 20) of the FC-30 arrays; less efficient solutions—even those that require a few more complete arrays—add only a small relative increment to the total price. Conversely, the same workloads for the FC-60 only require 2–3 arrays, and the relative cost impact of a solution requiring even a single extra array is considerable. Another reason for the increased FC-60 costs is that many of the taggers were hand-tuned for the FC-30 in an earlier series of experiments [1].

With a different array, which has very different perfor-



(a) **FC-30 array** (b) **FC-60 array**

Figure 3: Partially-adaptive results for the FC-30 and FC-60 disk arrays. There are two bars for each initial assignment goal function: the one on the left uses the *PriceThenMinAvgUtil* reassignment goal function, the one on the right *PriceThenMaxAvgUtil*.

mance characteristics, the decisions as to what constitutes a "large write" become invalid. For example, consider the stripe size setting for each of the arrays we used. The FC-60 uses a default of 16 KB, whereas the FC-30 uses 64 KB, which results in different performance for I/O sizes between these values.

Third, even taggers based completely on the solver models perform no better, and sometimes worse, than taggers based only on simple rules. This indicates that tagging solutions are too simplistic; it is necessary to take into account the interactions between different streams and stores mapped to the same LU or array when selecting RAID levels. This can be done through the use of adaptive algorithms, as shown in the following sections.

5.3.2 Partially-adaptive results

Figure 3 shows results for each of the partially-adaptive rules for the FC-30 and FC-60 arrays. Our results show that the partly adaptive solver does much better than the tagging approaches. In particular, minimizing the average capacity and utilization works well for both arrays and all the workloads.

From the data, it is clear that *allR5* is the best partially-adaptive rule for the FC-30 but not for the FC-60. However the rules based on norms (*AvgOfCapUtil*, *MaxOfCapUtil* and *LengthOfCapUtil*) seem to perform fairly well for both arrays—an improvement over the tagging schemes. The family of partially-adaptive rules based on change in utilization seems to perform reasonably for the FC-30, but poorly for the FC-60—with one exception, *MinAvgΔUtilTimesDataDisks*, that performed as well as the norm-rules.

5.3.3 Fully-adaptive results

Tables 3 and 4 show, for each workload, the best results achieved for each family of RAID level selection methods. As can be seen, the fully-adaptive approach finds the best solution in all but one case, indicating that this technique better searches the solution space than the partly adaptive and tagging techniques. Although the fully-adaptive approach needs more modifications to the solver, a single goal function performs nearly perfectly on both arrays, and it is more flexible.

	Taggers			*Partly adaptive*		*Fully*
Workload	*PerfVectMax*	*IOPSdisk*	*R10SmallWriteAggressive*	*AllR5*	*AvgOfCapUtil*	*adaptive*
filesystem	1%	1%	1%	0%	0%	0%
filesystem-lite	1%	1%	3%	0%	0%	0%
oltp	1%	1%	1%	0%	0%	0%
scientific	2%	2%	2%	0%	2%	0%
tpcd30-1x	7%	8%	8%	0%	0%	0%
tpcd30-2x	4%	2%	2%	2%	2%	0%
tpcd30-4x	1%	44%	44%	2%	2%	2%
tpcd300-query-1	0%	0%	0%	0%	0%	0%
tpcd300-query-5	18%	4%	4%	0%	0%	0%
tpcd300-query-7	25%	0%	0%	0%	4%	0%
tpcd300-query-9	22%	4%	8%	0%	4%	0%
tpcd300-query-10	4%	8%	8%	0%	0%	0%
average	**7.2%**	**6.3%**	**6.8%**	**0.3%**	**1.2%**	**0.12%**

Table 3: Cost overruns for the best solution for each workload and RAID selection method for the FC-30 array. Values are in percent above the best cost over all results for that array—that is, if the best possible result cost $100, and the given method resulted in a system costing $115, then the cost overrun is 15%. Increasing the number of reassignment passes to 5 results in the fully-adaptive scheme being best in all cases; we do not report those numbers to present a fair comparison with the other schemes.

	Taggers			*Partly adaptive*		*Fully*
Workload	*FC60UtilizationOnly*	*IOPScapacity*	*allR10*	*AvgOfCapUtil*	*MaxOfCapUtil*	*adaptive*
filesystem	0%	44%	0%	0%	0%	0%
filesystem-lite	24%	0%	24%	0%	0%	0%
oltp	0%	2%	0%	0%	0%	0%
scientific	0%	108%	0%	0%	0%	0%
tpcd30-1x	12%	12%	0%	0%	0%	0%
tpcd30-2x	9%	9%	0%	0%	0%	0%
tpcd30-4x	7%	7%	0%	7%	0%	0%
tpcd300-query-1	7%	2%	50%	5%	0%	0%
tpcd300-query-5	61%	2%	37%	0%	9%	0%
tpcd300-query-7	32%	1%	0%	0%	20%	0%
tpcd300-query-9	36%	2%	10%	0%	0%	0%
tpcd300-query-10	12%	2%	50%	9%	0%	0%
average	**16.7%**	**15.9%**	**14.3%**	**1.75%**	**2.4%**	**0%**

Table 4: Cost overruns for the best solution for each workload and RAID selection method for the FC-60 array. All values are percentages above the best cost seen across all the methods.

5.3.4 CPU time comparison

The advantage of better solutions does not come without a cost: Table 5 shows that the CPU time to calculate a solution increases for the more complex algorithms, because they explore a larger portion of the search space. In particular, tagging eliminates the need to search any solution that uses an LU with a different tag, and makes selection of a new LU's type trivial when it is created, whereas both of the adaptive algorithms have to perform a model evaluation and a search over all of the LU types.

The fully-adaptive algorithm searches all the possibilities that the partially-adaptive algorithm does, and also looks at the potential benefit of switching the LU type on each assignment. It takes considerably longer to run. Even so, this factor is insignificant when put into context: our solver has completely designed enterprise storage systems containing $2–$5 million of storage equipment in under an hour of CPU time. We believe that the advantages of the fully-adaptive solution will outweigh its computation costs in almost all cases.

5.3.5 Implementation complexity

A final tradeoff that might be considered is the implementation complexity. The modifications to implement partially-adaptive schemes on the original solver took a few hours of work. The fully-adaptive approach took a few weeks of work. Both figures are for a person thoroughly familiar with the solver code. However, the fully-adaptive approach clearly gives the best results, and is independent of the devices and workloads being used; the development investment is likely to pay off very quickly in any production environment.

6 Related work

The published literature does not seem to report on systematic, implementable criteria for automatic RAID level selection. In their original paper [20], Patterson, Gibson and Katz mention some selection criteria for RAID 1 through RAID 5, based on the sizes of read and write accesses. Their criteria are high-level rules of thumb that apply to extreme cases, e.g., "if a workload contains mostly small writes, use RAID 1/0 instead of RAID 5". No attempt is made to resolve contradictory recommendations from different rules, or to determine threshold values for essential definitions like "small write" or "write-mostly". Simulation-based studies [2, 14, 17] quantify the relative strengths of different RAID levels (including some not mentioned in this paper), but do not derive general guidelines for choosing a RAID level for given access patterns.

The HP AutoRAID disk array [24] side-steps the issue by dynamically, and transparently, migrating data blocks between RAID 1/0 and RAID 5 storage as a result of data access patterns. However, the AutoRAID technology is not yet widespread, and even its remapping algorithms are themselves based on simple rules of thumb that could perhaps be improved (e.g., "put as much recently written data in RAID 1/0 as possible").

In addition to RAID levels, storage systems have multiple other parameters that system administrators are expected to set. Prior studies examined how to choose the number of disks per LU [22], and the optimal stripe unit size for RAID 0 [9], RAID 5 [8], and other layouts [19]. The RAID Configuration Tool [27] allows system administrators to run simple, synthetic variations on a user-supplied I/O trace against a simulator, to help visualize the performance consequences of each parameter setting (including RAID levels). Although it assists humans in exploring the search space by hand, it does not automatically search the parameter space itself.

Apart from the HP AutoRAID, none of these systems provide much, if any, assistance with mixed workloads.

The work described here is part of a larger research program at HP Laboratories with the goal of automating the design, construction, and management of storage systems. In the scheme we have developed for this, we run our solver to develop a design for a storage system, then implement that design, monitor it under load, analyze the result, and then re-design the storage system if necessary, to meet changes in workload, available resources, or even simple mis-estimates of the original requirements [4]. Our goal is to do this with no manual intervention at all – we would like the storage system to be completely self-managing. An important part of the solution is the ability to design configurations and data layouts for disk arrays automatically, which is where the work described in this paper contributes.

7 Summary and conclusions

In this paper, we presented a variety of methods for selecting RAID levels, running the gamut from the ones that consider each store in isolation and make irrevocable decisions to the ones that consider all workload interactions and can undo any decision. We then evaluated all schemes for each family in isolation, and then compared the cost of solutions for the best representative from each family. A set of real workload descriptions and models of commercially-available disk arrays was used for the performance study. To the best of our knowledge, this is the first systematic, automatable attempt to select RAID

Workload	*Taggers*		*Partly adaptive*		*Fully adaptive*	
filesystem	92	(±14)	131	(±42)	273	(±53)
filesystem-lite	51	(±3)	85	(±33)	232	(±28)
oltp	212	(±29)	279	(±46)	669	(±155)
scientific	66	(±5)	116	(±49)	277	(±55)
tpcd30-1x	44	(±10)	85	(±23)	782	(±197)
tpcd30-2x	265	(±49)	393	(±92)	3980	(±1414)
tpcd30-4x	1098	(±159)	2041	(±739)	24011	(±7842)
tpcd300-query-1	689	(±44)	1751	(±2719)	1541	(±300)
tpcd300-query-5	1517	(±85)	2907	(±3593)	4572	(±1097)
tpcd300-query-7	1556	(±90)	2401	(±2126)	5836	(±1345)
tpcd300-query-9	1680	(±73)	2693	(±2362)	6647	(±2012)
tpcd300-query-10	1127	(±77)	2144	(±1746)	2852	(±563)
mean	**700**	**(±633)**	**1252**	**(±2016)**	**4306**	**(±6781)**

Table 5: Mean and (standard deviation) of the CPU time in seconds, for each workload and RAID selection method for the FC-60 array.

levels in the published literature.

The simpler tagging schemes are similar to accepted knowledge and to the back-of-the-envelope calculations that system designers currently rely upon. However, they are highly dependent on particular combinations of devices and workloads, and involve hand-picking the right values for many constants, so they are only suitable for limited combinations of workloads and devices. Furthermore, because they put restrictions on the choices the solver can make, they result in poorer solutions.

Integrating RAID level selection into the store-to-device assignment algorithm led to much better results, with the best results being obtained from allowing the solver to revise its RAID-level selection decision at any time.

We showed that the benefits of the fully-adaptive scheme outweigh its additional costs in terms of computation time and complexity. Analysis of the utilization data from the fully-adaptive solver solutions showed that some of the solutions it generated in our experiments were provably of the lowest possible cost (e.g., when the capacity of every disk, or the bandwidth of all but one array, were fully utilized).

For future work, we would like to explore the implications of providing reliability guarantees in addition to performance; we believe that the fully-adaptive schemes would be suitable for this, at the cost of increased running times. We would also like to automatically choose components of different cost for each individual LU within the arrays, e.g., decide between big/slow and small/fast disk drives according to the workload being mapped onto them; and to extend automatic decisions to additional parameters such as LU stripe size and disks used in an LU.

Acknowledgements: We thank Arif Merchant, Susan Spence and Mustafa Uysal for their comments on earlier drafts of the paper.

References

[1] 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*, 19(4), November 2001.

[2] G. A. Alvarez, W. Burkhard, L. Stockmeyer, and F. Cristian. Declustered disk array architectures with optimal and near-optimal parallelism. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, pages 109–20, June 1998.

[3] E. Anderson. Simple table-based modeling of storage devices. Technical report HPL–SSP–2001–4, Hewlett-Packard Laboratories, July 2001. http://www.hpl.hp.com/SSP/papers/.

[4] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: running circles around storage administration. In *File and Storage Technologies Conference (FAST)*, Monterey, 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, Hewlett-Packard Laboratories, June 2001. http://www.hpl.hp.com/SSP/papers/.

[6] M. Blaum, J. Brady, J. Bruck, and J. Menon. Evenodd – an efficient scheme for tolerating double-disk failures in RAID architectures. *IEEE Transactions on Computers*, 44(2):192–202, February 1995.

[7] E. Borowsky, R. Golding, A. Merchant, L. Schreier, E. Shriver, M. Spasojevic, and J. Wilkes. Using attribute-managed storage to achieve QoS. In *5th International Workshop on Quality of Service*, Columbia university, new York, NY, June 1997.

[8] P. Chen and E. Lee. Striping in a RAID level 5 disk array. In *International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS)*, pages 136–145, May 1995.

[9] P. Chen and D. Patterson. Maximizing performance in a striped disk array. In *International Symposium on Computer Architecture (ISCA)*, pages 322–331, May 1990.

[10] W. Fernandez de la Vega and G. Lueker. Bin packing can be solved within $1+\epsilon$ in linear time. *Combinatorica*, 1(4):349–355, 1981.

[11] L. Hellerstein, G. Gibson, R. Karp, R. Katz, and D. Patterson. Coding techniques for handling failures in large disk arrays. *Algorithmica*, 12(2/3):182–208, 1994.

[12] Hewlett-Packard Company. *HP SureStore E Model 30/FC High Availability Disk Array—User's Guide*, August 1998. Publication A3661–90001.

[13] Hewlett-Packard Company. *HP SureStore E Disk Array FC60—User's guide*, December 2000. Publication A5277–90001.

[14] M. Holland and G. A. Gibson. Parity declustering for continuous operation in redundant disk arrays. In *5th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, volume 20, pages 23–35, October 1992.

[15] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3(4):299–325, December 1974.

[16] C. Kenyon. Best-fit bin-packing with random order. In *Symposium on Discrete Algorithms*, pages 359–364, January 1996.

[17] E.K. Lee and R.H. Katz. Performance consequences of parity placement in disk arrays. In *4th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 190–199, Santa Clara, CA, April 1991.

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

[19] C.-I. Park and T.-Y. Choe. Striping in disk array RM2 enabling the tolerance of double disk failures. In *Supercomputing*, November 1996.

[20] 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.

[21] D. Stodolsky, M. Holland, W.V. Courtright II, and G. Gibson. Parity-logging disk arrays. *ACM Transactions on Computer Systems*, 12(3):206–35, August 1994.

[22] P. Triantafillou and C. Faloutsos. Overlay striping and optimal parallel I/O for modern applications. *Parallel Computing*, 24(1):21–43, January 1998.

[23] M. Uysal, G. A. Alvarez, and A. Merchant. A modular, analytical throughput model for modern disk arrays. In *International Symposium on Modeling, Analysis and Simulation on Computer and Telecommunications Systems (MASCOTS)*, August 2001.

[24] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 14(1):108–36, February 1996.

[25] John Wilkes. Traveling to Rome: QoS specifications for automated storage system management. In *Proceedings of the International Workshop on Quality of Service (IWQoS'2001)*, Karlsruhe, Germany, June 2001. Springer-Verlag.

[26] B. Worthington, G. Ganger, Y. Patt, and J. Wilkes. On-line extraction of SCSI disk drive parameters. In *International Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS)*, pages 146–56, May 1995.

[27] P. Zabback, J. Riegel, and J. Menon. The RAID configuration tool. Technical Report RJ 10055 (90552), IBM Research, November 1996.