Published in the Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS-2001), August 15-18, 2001, Cincinnati, Ohio, USA

A Modular, Analytical Throughput Model for Modern Disk Arrays

Mustafa Uysal Guillermo A. Alvarez Arif Merchant
Hewlett-Packard Laboratories
1501 Page Mill Road, Palo Alto, CA 94304
{uysal,galvarez,arif}@hpl.hp.com

Abstract

Enterprise storage systems depend on disk arrays for their capacity and availability needs. To design and maintain storage systems that efficiently satisfy evolving requirements, it is critical to be able to evaluate configuration alternatives without having to physically implement them. In this paper, we describe an analytical model to predict disk array throughput, based on a hierarchical decomposition of the internal array architecture. We validate the model against a state-of-the-art disk array for a variety of synthetic workloads and array configurations. To our knowledge, no previously published analytical model has either incorporated the combined effects of the complex optimizations present in modern disk arrays, or been validated against a real, commercial array. Our results are quite encouraging for an analytical model: predictions are accurate in most cases within 32% of the observed array performance (15% on the average) for our set of experiments.

1 Introduction

In today's networked world, an increasing number of companies depend on their business-critical data being continuously available for access. Given the sharp decreasing trend [10] in per-byte equipment costs for data stored online (*i.e.*, in low-latency media such as hard disks, as opposed to tape or optical libraries), the amount of online data has been recently doubling in size every six to twelve months [21]. Enterprise systems store data in large disk arrays [9] to satisfy their considerable requirements for capacity (tens to hundreds of terabytes) and high availability (achieved by a combination of redundant storage, support for hot-swapping, and transparent fail-over capabilities).

The internal architectures of commercial disk arrays have become considerably more complex than the early academic prototypes [5] built a decade ago. Array internals now include hardware and firmware support for a variety of optimizations such as adaptive prefetching policies, automatic detection of sequential streams of accesses, efficient demotion policies of dirty blocks from cache, and request coalescing. The presence of very large caches for read-ahead and write-behind (up to 32 GB in the current generation of arrays [12]) and of sophisticated interconnects within each array potentially allow these optimizations to be more aggressive. In addition, the commodity disks where data are ultimately stored typically implement smart prefetching and caching policies (e.g. stream detection) on their own. The common goal is to reduce the likelihood of having to wait for an access that goes all the way down to the physical media. However, optimizations have been largely developed independently of one another; their combined effects often defy intuition, and can be very difficult to understand [2].

There is a need for models that can predict the performance of a given array, configured in a given way, and subject to a given workload. Predictive models give their users the ability to explore the consequences of multiple design points and architectural tradeoffs, without having to build physical prototypes. In particular, analytical models are less costly to develop, allow users to concentrate on the relevant parts of the system being modeled, and are orders of magnitude faster to generate predictions than simulations. The latter advantage is essential for an important application: to guide the decisions of constrained optimization engines. The Minerva system [1] for automatic system design and configuration needs to predict the performance of tens of millions of candidate system configurations before settling on a final solution. Previous attempts at building monolithic models of modern arrays have been unsuccessful because of the tradeoff between the degree of realism of the model on the one hand, and the complexity of the formal manipulations needed to build and evaluate it on the other.

The main contribution of this paper is an analytical model that can predict the *throughput* (*i.e.*, the number of I/Os serviced per time unit) a disk array will attain for a given workload. We instantiated the general model to the particular case of RAID1/0 storage (mirrored striping) in a state-of-the-art disk array. We also validated the model's

predictions against measurements taken on the real array; to the best of our knowledge, such a stringent level of validation has never been reported before in the literature.

Another contribution is a novel method for building analytical disk array models, based on hierarchical decompositions of the internal device architecture. Array models are built as sets of interconnected *component models* that represent array components such as disks, busses, and controllers. This approach has several advantages. Individual components are easier to model than the whole array. Because of this modularity, we can reuse component models previously developed for other arrays, or concentrate our efforts where improvement is needed without changing other parts of the model. Among the many possible decompositions, the ones that closely mirror the physical device structure are best: the modeler can then use his domain-specific knowledge about the way each component operates.

We believe that our methodology is general enough to model the complex internal interactions that are typical in modern storage devices, and the (often not evident) ways in which workload characteristics influence device performance. Results from validation experiments show that our model generates reasonably accurate predictions: our model was accurate within a 32% of the actual measurements from the array (15% on the average) for a variety of synthetic workloads and array configurations; although there were three corner cases where our models did poorly with about 40% relative error. This level of accuracy is adequate for many purposes. We have found that, as long as a model is validated with a wide enough sample of synthetic workloads, average error is a much better predictor of the model's usefulness than maximum error. Our models are very good at considering the correlation and contention effects of multiple workloads being executed in parallel; for instance, Minerva [1], which designs and provisions storage systems based on models with 20% error, produces systems with less than 2% deviation from specified requirements.

The remaining part of this paper is organized as follows. Section 2 compares our work with pre-existing approaches. We describe our style of workload specifications in Section 3, and the array model in Section 4. Section 5 discusses the result of our model validation experiments. We draw our conclusions in Section 6.

2 Related work

There is a large body of work on analytical models of disk arrays. Bitton and Gray [3] present a model of seek time in shadowed disks, an early version of RAID1. Kim and Tantawi [14] analyze the positioning time delays in asynchronous disk interleaving, a variety of disk striping where sub-blocks of a data block are placed independently of one another. Chen and Towsley, in several papers [4, 6], present an analytical queueing model of the response time in a disk array. Lee and Katz [15] present a model of disk striping under a simple synchronous workload. This model treats reads and writes similarly and assumes that the mean service time of a block request at a disk is known. Merchant and Yu [18, 19, 20] present several queueing models of response times for disk arrays using RAID1 and RAID5 layouts, both for normal and degraded modes; Thomasian and Menon [23] analyze the performance of RAID5 with distributed sparing in normal mode, degraded mode, and rebuild mode in an OLTP environment. Menon and Mattson [17, 16] present response-time models of disk arrays using RAID5 layout; unlike most other models, these include some cache effects, albeit very simple ones. Disk drives are not directly modeled as a part of these models.

While several of the papers above compare their results against simulation results, none of the simulations used are validated against real arrays, nor are any of the analytical results compared directly against real hardware. Many make simplifying assumptions, such as exponentially distributed disk service times, and most do not model the effects of either the disk or the array cache. None of the models above take into account the effects of the queue sizes on the disk service times.

As a consequence, these pre-existing models typically mis-predict by a considerable margin, even when validated against a simulator [13]; errors are likely to be still more significant when validated against a commercial array, that implements many optimizations not contemplated by the simulator. The analytical model presented in this paper is the first one to incorporate the effects of multiple behaviors of real arrays such as intelligent destaging (*i.e.*, flushing of dirty cache blocks to the disks), client accesses coalesced into a single disk access, prefetching, and caching.

Several simulation models of disks and disk arrays exist. DiskSim [8] and Pantheon [24] are two simulation environments for a disk sub-system, including disks, buses, adapters, controllers and drivers. Both include disk modules which have been carefully validated against real disks, but the rest of the components have not. RaidFrame [7] is a software RAID controller which can also be used as a stand-alone discrete-event simulator for disk arrays.

Our disk model is derived from Shriver *et al.* [22], as is the general idea of decomposing a storage system into components which transform the I/O stream passing through them.

3 Workload specifications

This section describes our approach to capture the performance characteristics of the workload presented to the

storage device. Since I/O workloads could be arbitrarily complex, this is a difficult problem. Our approach to this problem is to capture a small set of important workload characteristics that sufficiently describes the steady state behavior of the workload. Table 1 lists the workload attributes that our throughput model uses to predict the performance of a disk array.

Our workload specifications consist of collections of objects that contain a set of *(attribute, value)* pairs as well as sub-objects. Values may be either simple numerical values or distributions, defined by their type (for example Poisson, normal), and their mean and variance. In some cases, we specify separate values for read and write accesses.

Our workload specifications capture the characteristics of the access patterns being executed on the array. In the limit, a full trace of every single I/O contains all the information about a workload; but such a representation is excessively large and too low-level for useful manipulation. In our workload specifications, *store* objects have attributes that describe a chunk of contiguous data, such as size. *Stream* objects capture the dynamic aspects of a workload, including temporal requirements and behaviors. The stream's requests are described by the request_rate and request_size attributes. The run_count attribute models sequential locality by counting the mean number of consecutive I/O requests issued to consecutive addresses. All three attributes are specified separately for reads and writes.

Additionally, each stream has attributes whose values represent the average number of requests of each type that have been initiated but not yet completed. The queue_length attribute represents the average number of outstanding requests waiting to be serviced at any point in time. We capture the temporal locality of a stream by the re_reference_distance attribute. This attribute represents the number of bytes accessed by the stream between two accesses to the same block (we used a block size of 1KB for this paper), represented as a histogram of re-reference distances.

4 The model

This section starts by introducing the overall structure of the array model, and its correspondence with the real array. We then describe the three component models in top-down order: cache, array controller, and individual disk.

4.1 Overall structure

We modeled the Hewlett-Packard SureStore E Disk Array FC60 [11]. Figure 1(A) shows a schematic diagram of the FC60, omitting the components that are irrelevant for performance modeling purposes (*e.g.* redundant power supplies, fans, battery backup unit). The array may have up to 60 disks, allocated to up to six trays. To survive a controller failure, the FC60 can have two controllers, installed in the same controller enclosure. Each controller has a Fibre Channel connection to a storage area network, to which the client hosts are also connected. Each controller may have up to 512 MB of battery-backed cache (NVRAM). Dirty blocks are mirrored in both controller caches, to prevent data loss if a controller fails. The enclosure contains a backplane bus that connects the controllers to the trays, 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.

We modeled a RAID1/0 logical unit (LU, the independent, disjoint partition into which data is stored in an array) in the FC60, operating in failure-free mode, under a workload in which each run of I/Os on consecutive addresses contains only reads or only writes. In a RAID1/0 LU, the client's data is striped over two or more disks in fixed-size blocks; disks in the LU are arranged into mirrored pairs (in which each disk is an exact copy of the other element in its pair) for fault tolerance. In our experience, RAID1/0 is the most widely used RAID level in enterprise systems: the relatively high level of availability it provides is more important for system administrators than the expense of keeping two copies of all the data (especially since disk capacity is becoming abundant and cheap).

To guide our modeling decisions, we did not rely on any proprietary information about the performance of individual array components, the nature and scope of the policies implemented by their firmware, or the results of any performance measurement directly or indirectly performed inside the array. This fact significantly complicates building a model; second-guessing all this information from a working prototype is a very difficult task even for the (much simpler) case of an older single hard disk [25].

In our compositional style of modeling, each component model above the bottom of the hierarchy transforms an input workload specification into one or more output specifications that, in turn, become inputs for models at lower levels in the hierarchy. Component models communicate with one another in a single, well-defined way: by passing specifications as parameters. Outputs are computed by applying a set of transformations to the workload attributes in the input. Each component model recomputes the values of the attributes that are relevant to it, and relays the rest of the input specification without changes. For example, a write-back cache typically absorbs many requests, and communicates with the lower levels of the memory hierarchy only when there is a miss and when some dirty lines are evicted. Therefore, the output of the cache model will have much smaller read and write rates than its input (if the workload exhibits good locality), and the access sizes in its output may always be a multiple of the fixed line size. Attribute values that are irrelevant for one component model may be

Attributes	Description	Units
request_rate	mean rate at which requests arrive at the device	requests/sec
request_size	mean length of a request	bytes
run_count	number of requests made to contiguous addresses	requests
queue_length	mean size of the device queue	requests
re_reference_distance	amount of data accessed between consecutive accesses to the same block	bytes

Table 1. Workload attributes used by the throughput models in the paper.

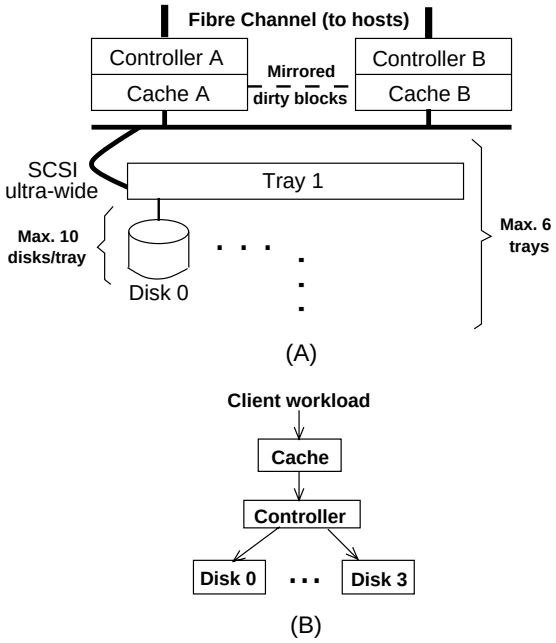


Figure 1. (A) Components in the data path of the real FC60. (B) Throughput model for a 4-disk RAID1/0 LU in an FC60.

essential for another. Models at the bottom of the hierarchy just compute performance predictions.

Figure 1(B) shows the structure of the FC60 model, where we concentrate on the components that have been observed to become bottlenecks in our extensive measurements. The cache component is the first one to process the workload; it absorbs some of the accesses (hits) and relays a workload specification representing the misses and demotions down to the controller. The controller translates LU-level accesses to accesses on the individual disks that comprise the RAID1/0 LU being modeled. Finally, the disk models compute their own throughput predictions without any further propagation. If the offered load in the input workload is within the capabilities of all array components, the predicted throughput from the composite model is equal to the total input request rate; otherwise, it is the minimum throughput for which at least one of the components (the bottleneck) is saturated. In the latter case, no more additional throughput can be gained by offering more load.

It is worth noting that our modeling approach is general enough to accommodate arrays substantially more complex than the FC60. For example, it would be equally possible to model an XP512 array [12], in which multiple controllers connected to a crossbar backplane share the same cache banks.

4.2 Cache model

Throughput limits: The cache model imposes no direct throughput limits; the combined bandwidth and throughput limitations of the controller and array cache are captured in the controller model.

Transformations: The array cache model receives as input a list of I/O request streams and their characteristics. It outputs a list of streams with characteristics modified to reflect a reductions in request rate due to cache hits.

In steady state, both read and write requests cause corresponding disk reads or writes only when there is a cache miss. Read requests which are found in the array cache are served from there and, in steady state, writes are only propagated to the disk because of a cache miss, which causes a cache block to be demoted to disk. Suppose the total cache size is $\mathsf{total\_cache\_size}$, there are n streams S_1, S_2, ..., S_n in the input to the array cache model and denote the corresponding output streams from the model as S_1', S_2', ..., S_n'. To determine $\mathsf{request\_rate}(S_i')$, we make the approximation that the cache is divided into n parts of size $\mathsf{cache\_size}(S_1)$, $\mathsf{cache\_size}(S_2)$, ..., $\mathsf{cache\_size}(S_n)$ devoted to the n streams, where

$$\mathsf{cache\_size}(S_i) = \mathsf{total\_cache\_size} \cdot \left(\frac{\mathsf{request\_rate}(S_i)}{\sum_{j=1}^{n} \mathsf{request\_rate}(S_j)} \right) \quad (1)$$

We then approximate the probability that a request is a hit in the cache by the probability that the number of bytes accessed by the stream between two accesses to the same

block is less than $\mathsf{cache\_size}(S_i)$; we call this the "re-reference distance" whose distribution is supplied as a part of the workload specification. Thus,

$$\mathsf{request\_rate}(S'_i) = \mathsf{request\_rate}(S_i) \cdot \mathsf{Prob}[\mathsf{re\_reference\_distance}(S_i) > \mathsf{cache\_size}(S_i)] \quad (2)$$

4.3 Controller model

Throughput limits: The controller model limits the total rate of requests (I/Os per second) and the bandwidth requirement (bytes/sec) of the request streams impinging upon it. The model implements the inequalities

$$\sum_{i=1}^{n} \mathsf{request\_rate}(S_i) \leq \mathsf{max\_controller\_throughput} \quad (3)$$

$$\sum_{i=1}^{n} \mathsf{request\_rate}(S_i) \cdot \mathsf{E}[\mathsf{request\_size}(S_i)] \leq \mathsf{max\_controller\_bandwidth} \quad (4)$$

The values of $\mathsf{max\_controller\_throughput}$ and $\mathsf{max\_controller\_bandwidth}$ are constants measured from the device and supplied as a part of the device description. The controller is saturated when the left hand side of the inequalities approaches one.

Transformations: The controller model receives a list of input streams and their characteristics and outputs a corresponding list of streams for every disk. We describe below how the attributes of the output streams are derived.

In a real array, the controller translates accesses made on each LU into accesses on the individual disks. Requests which are cache hits are served from the cache. Reads which are cache misses are served from the disks, and usually also inserted into the cache. Writes which are cache misses cause demotions from the array cache, leading to disk writes. Access to data which is distributed over multiple disks cause I/O requests at each of the disks. The precise translation performed by the controller depends on the RAID level of the LU (always RAID1/0 in this paper), on other LU parameters such as stripe unit size, and on the failure state of the array (always failure-free in this paper).

A real controller performs this translation for each I/O request; the model of the controller, however, need only perform the translations at a statistical level. In other words, we only need to generate the workload attributes of the resulting I/O request streams going to the disks.

Consider an LU with disks D_1, D_2, ..., $D_{\mathsf{LU\_disks}}$. For each input stream S_i impinging on this LU, the model first generates S'_i by applying the cache model; S'_i corresponds to the stream of read cache misses and write cache demotions. Then it generates streams S_{ij} of the requests which go to disk D_j, for $j = 1, 2, \ldots, \mathsf{LU\_disks}$. We show below how the model computes the workload characteristics of S_{ij}.

In the RAID1/0 layout we are modeling, the data on the LU are divided into *stripes*. Each stripe is laid out across disks D_1, D_3, ..., $D_{\mathsf{LU\_disks}-1}$, with an identical copy on disks D_2, D_4, ..., $D_{\mathsf{LU\_disks}}$. The portion of a stripe on one disk is a *stripe unit*. Reads can be directed at either copy of the data, and writes must go to both.

Requests that cross stripe unit boundaries must be split into requests for the corresponding disks. The data are separately read into the controller's buffer, combined and returned to the host. On the other hand, there are cases when several requests which access contiguous data can be *coalesced* into a single request which can be more efficiently serviced at the disk level. These coalesced requests can be read into the controller's buffer and split into the requested component pieces and returned to the host. There are several kinds of coalescing, and which ones to implement is a policy choice that the array designer makes.

- **Access level coalescing:** A single, large request of size greater than a stripe accesses multiple contiguous stripe units on some or all disks. If an array controller presents these requests to the disk as a single combined request, it implements access-level coalescing.

- **Run level coalescing:** A sequential run of requests, when mapped to the corresponding disk blocks, may result in accesses to several consecutive blocks on the same disk. If the controller combines these into a single disk request, it implements run level coalescing. The FC-60 does not appear to implement run-level coalescing.

- **Stripe unit level coalescing:** This is a restricted form of run level coalescing, where several requests from a sequential run fall into the same stripe unit on a disk. If the controller combines these into a single disk request, it implements stripe unit level coalescing.

In most cases, it is difficult to determine definitively which kind of coalescing the controller implements. We constructed models based on the above three methods of coalescing for each kind of access. We then measured the performance of the array for a small number of test workloads and picked the coalescing model which matched most closely.

Reads: The FC60 controller reads only whole stripe units from a disk [11]; thus, if the read request is smaller than a stripe unit, the entire stripe unit is read into the array cache. If the read request straddles multiple stripe units, all the stripe units involved are read, resulting in an overhead of one stripe unit per read request (half a stripe unit before the

data requested and half a stripe unit after). Our experiments indicate that the FC-60 does not implement any coalescing for reads.

As we have argued above, each read is a stripe unit, and the mean number of stripe units read per read request is

$$\mathsf{disk\_accesses\_per\_read}(S_i) = 1 + \mathsf{read\_request\_size}(S_i)/\mathsf{stripe\_unit\_size} \quad (5)$$

Assuming that the disk read requests are uniformly distributed over all the disks,

$$\mathsf{read\_request\_size}(S_{ij}) = \mathsf{stripe\_unit\_size} \quad (6)$$

$$\mathsf{read\_request\_rate}(S_{ij}) = \frac{\mathsf{disk\_accesses\_per\_read}(S_i) \cdot \mathsf{read\_request\_rate}(S_i)}{\mathsf{LU\_disks}} \quad (7)$$

$$\mathsf{read\_run\_count}(S_{ij}) = \frac{\mathsf{disk\_accesses\_per\_read}(S_i) \cdot \mathsf{run\_count}(S_i)}{\mathsf{LU\_disks}} \quad (8)$$

Writes: Writes to disk in the FC-60 are caused by demotions from the cache. We lack a detailed model of how this modifies the write stream to the disk, but we assume that the cache acts mainly as a staging area for the writes, and at steady state, the write stream of demotions from the cache looks similar to the external write stream except as noted. We say that a write access is *large* if it modifies more than a stripe's worth of data, and is *small* otherwise. From our synthetic experiments, we concluded that the FC-60 implements two kinds of write coalescing: access-level coalescing for large writes, and stripe-unit level coalescing for the subset of the small writes that fit inside a single stripe unit. The two cases are therefore handled separately.

Large writes: Assuming that the data written are distributed uniformly over the disks and keeping in mind that each datum is written on two disks, we have

$$\mathsf{write\_request\_size}(S_{ij}) = 2 \cdot \mathsf{write\_request\_size}(S_i)/\mathsf{LU\_disks} \quad (9)$$

Since each write touches all disks in the LU, and the components of the write on each disk are coalesced into a single disk write request, we have

$$\mathsf{disk\_accesses\_per\_write}(S_{ij}) = 1 \quad (10)$$

$$\mathsf{write\_request\_rate}(S_{ij}) = \mathsf{write\_request\_rate}(S_i) \quad (11)$$

$$\mathsf{write\_run\_count}(S_{ij}) = \mathsf{write\_run\_count}(S_i) \quad (12)$$

The above equations are used only when $\mathsf{write\_request\_size}(S_i) \geq \mathsf{LU\_disks} \cdot \mathsf{stripe\_unit\_size}/2$.

Small writes: Write requests from a sequential run which fall in the same stripe unit on a disk are coalesced into a single disk write request. However, requests from a run which do not fall into the same stripe unit are not necessarily demoted together. This has the combined effect of increasing the request size but decreasing the run count of the stream to the disk.

$$\mathsf{write\_request\_size}(S_{ij}) = \min(\mathsf{stripe\_unit\_size}, \mathsf{run\_count}(S_i) \cdot \mathsf{write\_request\_size}(S_i)) \quad (13)$$

The average number of disk writes per write request in S_i is

$$\mathsf{disk\_accesses\_per\_write}(S_i) = 2 \cdot \mathsf{write\_request\_size}(S_i)/\mathsf{write\_request\_size}(S_{ij}) \quad (14)$$

These accesses touch $\mathsf{LU\_disks}$ disks; therefore,

$$\mathsf{write\_request\_rate}(S_{ij}) = \frac{2 \cdot \mathsf{write\_request\_size}(S_i) \cdot \mathsf{write\_request\_rate}(S_i)}{\mathsf{write\_request\_size}(S_{ij}) \cdot \mathsf{LU\_disks}} \quad (15)$$

Finally, because stripe units from a run of small writes are often not demoted together, we have

$$\mathsf{write\_run\_count}(S_{ij}) = 1 \quad (16)$$

The above equations are used only when $\mathsf{write\_request\_size}(S_i) < \mathsf{LU\_disks} \cdot \mathsf{stripe\_unit\_size}/2$.

Queue length: Stream S_i has $\mathsf{queue\_length}(S_i)$ requests outstanding on the average; we assume that they are divided between reads and writes in proportion to the read and write rates. Let $\mathsf{read\_accesses}(S_i) = \mathsf{read\_request\_rate}(S_i) \cdot \mathsf{disk\_accesses\_per\_read}(S_i)$ and $\mathsf{write\_accesses}(S_i) = \mathsf{write\_request\_rate}(S_i) \cdot \mathsf{disk\_accesses\_per\_write}(S_i)$. Since there are $\mathsf{LU\_disks}$ disks in the LU, the mean number of disk requests outstanding at disk D_j from stream S_i is

$$\mathsf{queue\_length}(S_{ij}) = \mathsf{queue\_length}(S_i) \cdot \left(\frac{\mathsf{read\_accesses}(S_i) + \mathsf{write\_accesses}(S_i)}{\mathsf{LU\_disks} \cdot \mathsf{request\_rate}(S_i)}\right) \quad (17)$$

4.4 Disk model

Throughput limits: The disk model for a disk D_j implements the throughput inequality

$$\sum_{i=1}^{n} \mathsf{read\_utilization}(S_{ij}) + \mathsf{write\_utilization}(S_{ij}) < 1 \quad (18)$$

The disk is saturated when the left hand side of the inequality approaches one. The utilizations are computed as

$$\mathsf{read\_utilization}(S_{ij}) = \mathsf{read\_request\_rate}(S_{ij}) \cdot \mathsf{disk\_read\_service\_time}(S_{ij}) \quad (19)$$

$$\mathsf{write\_utilization}(S_{ij}) = \mathsf{write\_request\_rate}(S_{ij}) \cdot \mathsf{disk\_write\_service\_time}(S_{ij}) \quad (20)$$

The mean disk read service time depends upon whether the datum is found in the disk cache. Since the service time

of a read datum found in the cache is very small compared to accessing the magnetic medium, we approximate it as zero. For data not found in the disk cache, the service time is the sum of positioning time and transfer time. Thus, the mean disk read service time is

$$\mathsf{disk\_read\_service\_time}(S_{ij}) = (1 - \mathsf{disk\_cache\_hit\_prob}) \cdot \left(\mathsf{disk\_read\_pos\_time}(S_{ij}) + \frac{\mathsf{read\_request\_size}(S_{ij})}{\mathsf{disk\_transfer\_rate}}\right) \quad (21)$$

The disk positioning time is estimated by

$$\mathsf{disk\_read\_pos\_time}(S_{ij}) = \frac{\mathsf{mean\_read\_disk\_seek\_time}}{\sum_{k=1}^{n} \mathsf{queue\_length}(S_{kj})} + \frac{\mathsf{disk\_rotation\_time}}{2} \quad (22)$$

The mean_read_disk_seek_time and disk_rotation_time are device parameters obtained through measurement.

Cache hits for read typically arise due to *read-ahead*. After servicing a read, the disk controller continues to transfer data into the on-board disk cache in anticipation of more sequential reads in the future. This improves the performance of sequential workloads as there are no positioning delays for the requests served from the disk cache. Disk caches are *segmented*, each segment consisting of a fixed amount of memory. Disks usually access no more than a segment worth of data for read-ahead. If a request from a different stream arrives during the read-ahead operation, the disk stops read-ahead and switches to serving the new request. We estimate the amount of read-ahead for a stream when requests from multiple sequential streams are queued together as:

$$\mathsf{read\_ahead\_amount}(S_{ij}) = \mathsf{disk\_cache\_segment\_size} \cdot \left(\frac{\mathsf{read\_request\_rate}(S_{ij}) \cdot \mathsf{queue\_length}(S_{ij})}{\sum_{k=1}^{n} \mathsf{read\_request\_rate}(S_{kj}) \cdot \mathsf{queue\_length}(S_{kj})}\right) \quad (23)$$

From this, the cache hit probability can be estimated as

$$\mathsf{disk\_cache\_hit\_prob} = \mathsf{read\_request\_size}(S_{ij}) / \min(\mathsf{read\_ahead\_amount}(S_{ij}), \mathsf{read\_run\_count}(S_{ij}) \cdot \mathsf{read\_request\_size}(S_{ij})) \quad (24)$$

Finally, disk caches are write-through only since they do not have a battery backup to save data in case of a power loss. While only the first request in a sequential write run experiences a seek time, unlike reads, all operations continue to experience rotational delays. This is because by the time the next request in a sequential run arrives at the disk, the disk platter has already rotated from its position. Our model estimates the average positioning delay incurred for the write requests in a sequential run as:

$$\mathsf{disk\_write\_pos\_time}(S_{ij}) = \frac{\mathsf{disk\_rotation\_time}}{2} + \frac{\mathsf{mean\_write\_disk\_seek\_time}}{\mathsf{write\_run\_count}(S_{ij}) \cdot \sum_{k=1}^{n} \mathsf{queue\_length}(S_{kj})} \quad (25)$$

Now the write service time can be computed as the sum of positioning and transfer times:

$$\mathsf{disk\_write\_service\_time}(S_{ij}) = \mathsf{disk\_write\_pos\_time}(S_{ij}) + \frac{\mathsf{write\_request\_size}(S_{ij})}{\mathsf{disk\_transfer\_rate}} \quad (26)$$

5 Empirical validation

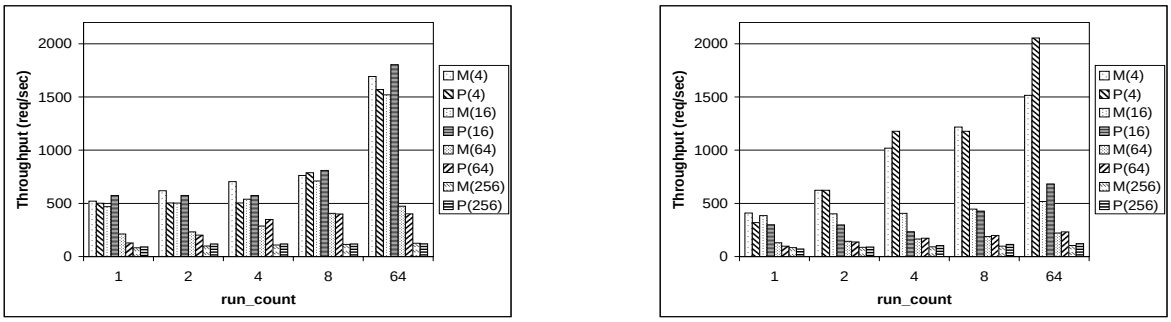
We validated the accuracy of our analytical model by comparing the predicted maximum throughput (I/Os per second) against measured values for a variety of workloads running against an FC-60. The FC-60 array used in our experiments has 30 disks (Seagate ST118202LC disks, each 18 GB in size), two controllers and 256 MB of cache in each controller. We configured two RAID1/0 LUs in our FC-60 consisting of four and six disks, respectively. We used disks from separate back-end SCSI buses for each LU. We used a stripe unit size of 16 KB and a cache page size of 4 KB in all of our experiments. We connected the FC-60 array to a Brocade Silkworm 2800 switch via two FibreChannel links. We used an HP 9000-N4000 server with eight 440 MHz PA-RISC 8500 processors and 16 GB of main memory to generate workloads and to access the FC-60 array. The host was running HP-UX 11.0 as its operating system. Table 2 contains the parameters for our model of the FC60.

In our validation experiments, we used synthetic workloads with request sizes ranging from 4 KB to 256 KB and the degree of sequentiality (run_count) ranging from 1 (random) to 64 (highly sequential). The load offered to the array attempted to keep the array busy by keeping up to 64 outstanding requests at every time. The workloads are generated using a synthetic load generator. For each workload execution, we collected a trace of all I/O activity on the host at the device driver level. The trace contained information about I/O submission and completion times, logical address and size of I/Os, and the length of the queue when each request completes, among others. We later analyzed the trace to compute the throughput and other useful statistics.

Figures 2 and 4 present the measured throughput from the array and the predicted throughput from the model for read and write workloads for a variety of request sizes and degree of sequentiality. Our results indicate that our model has on the average an accuracy of about 15% for both read and write workloads across all of the configurations tested (Figures 3 and 5 contain the relative error plots). This is an encouraging result for predicting the throughput of a modern disk array. Although the maximum relative error was higher, up to 42% for the four- and six-disk RAID1/0 configurations, most of the the model predictions were within 30% of the measured throughput from the FC60. Our results show that the model is conservative, for it does not over-predict the throughput by more than 22% with the ex-

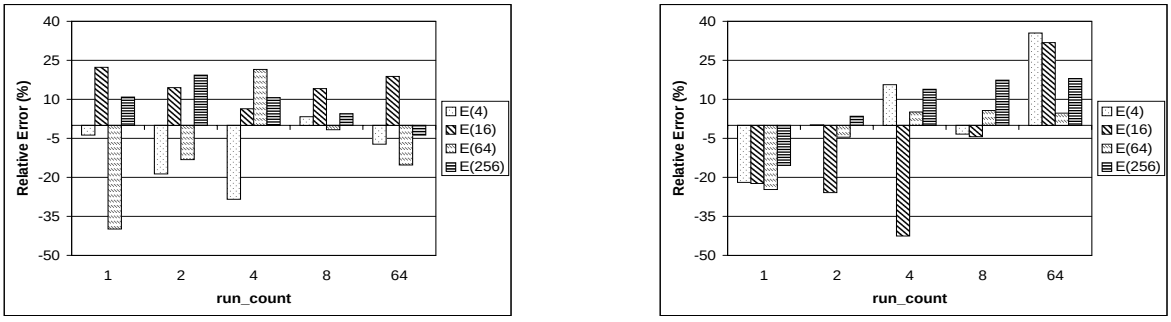
Parameter	Description	Value
total_cache_size	size of the cache at each controller	256 MB
max_controller_throughput	max number of I/Os the controller can process in a second	11,338 I/Os/sec
max_controller_bandwidth	maximum controller bandwidth in bytes/sec	84 MB/s
stripe_unit_size	amount of data controller writes to a single drive before switching to the next drive in the same LU	16 KB
LU_disks	number of disks in a LU	4, 6 disks
disk_type	type of disks in our FC-60 array	Seagate ST118202LC
disk_rotation_time	time for the disk platter to make one full rotation	6e-3 sec
disk_cache_segment_size	disk cache segment size	64 KB
disk_read_pos_time	mean read positioning time at disk	6.37e-3 sec
disk_write_pos_time	mean write positioning time at disk	7.00e-2 sec
disk_transfer_rate	mean read and write transfer rate, respectively	1.80e+7, 1.65e+7 byte/sec

Table 2. Model parameters for the FC-60 Array.



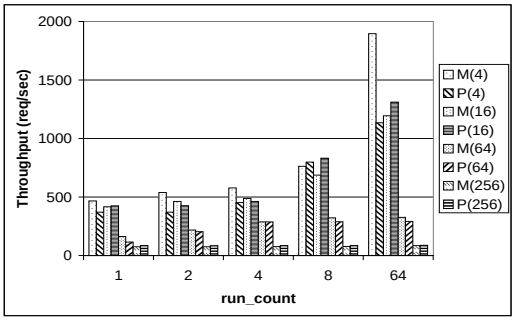
(a) reads (b) writes

Figure 2. Results from validating the array model for the 6-disk RAID1/0 configuration on the FC60 array. In the legend, M(X) denotes the measured throughput from the array for request size of X KB; and P(X) denotes the predicted throughput from the model for request size of X KB.

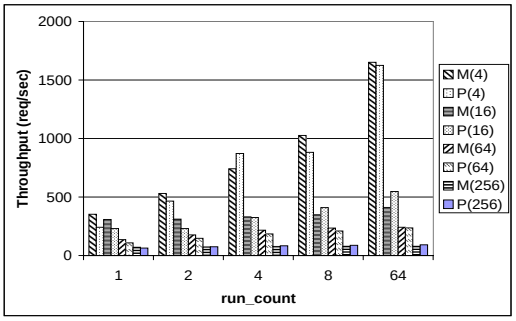


(a) reads (b) writes

Figure 3. Relative prediction error in the model for the 6-disk RAID1/0 configuration. In the legend, E(X) denotes the relative throughput prediction error of the array model for request size of X KB.

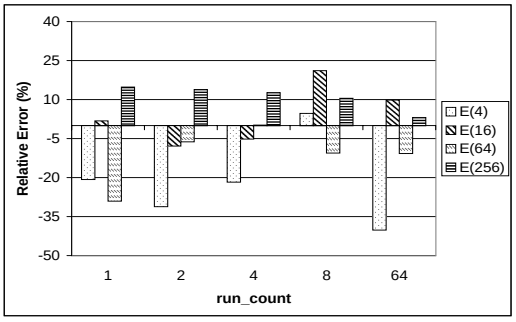


(a) reads

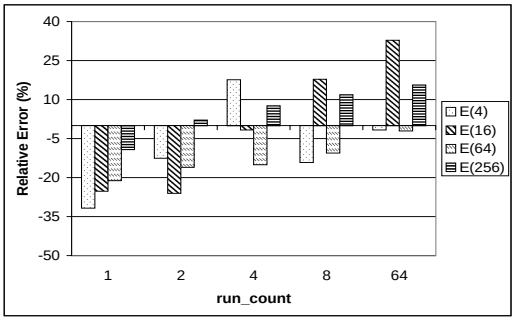


(b) writes

Figure 4. Results from validating the RAID1/0 throughput model for the 4-disk RAID1/0 configuration. In the legend, M(X) denotes the measured throughput from the array for request size of X KB; and P(X) denotes the predicted throughput from the model for request size of X KB.



(a) reads



(b) writes

Figure 5. Relative prediction error in the model for the 4-disk RAID1/0 configuration. In the legend, E(X) denotes the relative throughput prediction error of the array model for request size of X KB.

ception of two cases for the configurations tested. We note that our model becomes increasingly more optimistic as the sequentiality in our write-only workloads increases. We believe that this is due to our lack of a sufficiently detailed model of write destaging from the cache. Our model assumes that the request stream to the disks is similar to the input write request stream, however, due to demotions, the write stream to the disks might be less sequential than our model assumes.

6 Conclusions

We have presented an analytical throughput model of RAID1/0 LUs in a mid-range disk array, and compared the model's predictions with measurements taken on different configurations of the real array. To our knowledge, this is the first analytical model validated against a state-of-the-art disk array in the published literature. Our model is built around a hierarchical decomposition of the relevant parts of the array's internal architecture. The accuracy figures for the model are quite encouraging: although there were corner cases where our model did poorly with about 40% error, the relative errors in all other predictions were 15% on the average and 32% at the maximum. For perspective, this level of accuracy has been shown to be perfectly accurate to automatically design and provision storage systems [1].

One intent of developing this model was to learn techniques which could be applied to other arrays and other RAID levels. We found that there are a myriad factors which affect the performance of a disk array, all of which must be incorporated into an accurate model. In particular, it is necessary to model the effects of sequentiality in

the workload, and how it interacts with the RAID level. Also, it is critical to model the number of requests outstanding at the disk, and its impact on disk service times. Other factors include the effects of controller cache and of read-ahead in the disk. We found that a hierarchical decomposition of the model made the incremental inclusion of additional factors significantly easier. Our current version models many optimizations that are common in modern disk arrays: coalesced read accesses on back-end disks, reading whole stripe units into the cache, destaging of dirty blocks based on spatial locality, write coalescing, and array- and disk-level caching.

Acknowledgements: We thank John Wilkes and Doug Obal for their comments on an earlier version of this 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. Technical Report HPL-2001-139, Hewlett-Packard Labs, June 2001. To appear in ACM Transactions on Computer Systems.

[2] G. A. Alvarez, K. Keeton, E. Riedel, and M. Uysal. Characterizing I/O-intensive workload sequentiality on modern disk arrays. In *4th Workshop on Computer Architecture Evaluation using Commercial Workloads*, January 2001.

[3] D. Bitton and J. Gray. Disk shadowing. In *Proc. of 14th Int'l. Conf. on Very Large Data Bases (VLDB)*, pages 331–8, August 1988.

[4] J.B. Chen and B.N. Bershad. The impact of operating system structure on memory system performance. In *Proc. of 14th ACM Symp. on Operating Systems Principles (SOSP)*, pages 120–33, December 1993.

[5] P.M. Chen, G.A. Gibson, R.H. Katz, and D.A. Patterson. An evaluation of redundant arrays of disks using an Amdahl 5890. In *Proc. of ACM Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 74–85, May 1990.

[6] S. Chen and D. Towsley. A performance evaluation of RAID architectures. *IEEE Transactions on Computers*, 45(10):1116–30, October 1996.

[7] W. V. Courtright II. *A transactional approach to redundant disk array implementation*. PhD thesis, Department of Electrical Engineering and Computer Science, Carnegie-Mellon University, May 1997.

[8] G. R. Ganger and Y. N. Patt. Using system-level models to evaluate I/O subsystem designs. *IEEE Transactions on Computers*, 47(6):667–78, June 1998.

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

[10] E.G. Grochowski and R.F. Hoyt. Future trends in hard disk drives. *IEEE Transactions on Magnetics*, 32(3):1850–4, May 1996.

[11] Hewlett-Packard Company. *HP SureStore E Disk Array FC60 User's guide*. Pub. No. A5277-90001.

[12] Hewlett-Packard Company. *HP SureStore E Disk Array XP512 Owner's guide*. Pub. No. A5951-90900.

[13] M. Holland and G. Gibson. Parity declustering for continuous operation in redundant disk arrays. In *Proc. of 5th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 23–35, October 1992.

[14] M.Y. Kim and A.N. Tantawi. Asynchronous disk interleaving: approximating access delays. *IEEE Transactions on Computers*, 40(7):801–10, July 1991.

[15] E.K. Lee and R.H. Katz. An analytic performance model of disk arrays. In *Proc. of ACM Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 98–109, May 1993.

[16] J. Menon. Special issue on disk arrays – introduction. *Distributed and Parallel Databases*, 2(3):241, July 1994.

[17] J. Menon and J. Kasson. Methods for improved update performance of disk arrays. In *Proc. of 25th Int'l. Conf. on System Sciences*, volume 1, pages 74–83, January 1992.

[18] A. Merchant and P. S. Yu. An analytical model of reconstruction time in mirrored disks. *Performance Evaluation*, 20(1–3):115–29, May 1994.

[19] A. Merchant and P.S. Yu. Analytic modeling and comparisons of striping strategies for replicated disk arrays. *IEEE Transactions on Computers*, 44(3):419–33, March 1995.

[20] A. Merchant and P.S. Yu. Analytic modeling of clustered RAID with mapping based on nearly random permutation. *IEEE Transactions on Computers*, 45(3):367–73, March 1996.

[21] G. Papadapoulos. Moore's Law ain't good enough. Keynote speech at Hot Chips X, August 1998.

[22] E. Shriver, A. Merchant, and J. Wilkes. An analytical behavior model for disk drives with readahead caches and request reordering. In *Proc. of Int'l. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 182–91, June 1998.

[23] A. Thomasian and J. Menon. Raid5 performance with distributed sparing. *IEEE Transactions on Parallel and Distributed Systems*, 8(6):640–57, June 1997.

[24] J. Wilkes. The Pantheon storage-system simulator. Technical Report HPL-SSP-95-14, HP Laboratories, December 1995.

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