An integrated performance model of disk arrays[*]

Elizabeth Varki
Department of Computer Science
University of New Hampshire
varki@cs.unh.edu

Arif Merchant
Hewlett Packard Laboratories
Palo Alto, CA
arif@hpl.hp.com

Jianzhang Xu
Department of Computer Science
University of New Hampshire
jx@cs.unh.edu

Xiaozhou Qiu[†]
Falconstor Software
Melville, NY
Leo.Qiu@falconstor.com

Abstract

All enterprise storage systems depend on disk arrays to satisfy their capacity, reliability, and availability requirements. Performance models of disk arrays are useful in understanding the behavior of these storage systems and predicting their performance. We extend prior disk array modeling work by developing an analytical disk array model that incorporates the effects of workload sequentiality, read-ahead caching, write-back caching, and other complex optimizations incorporated into most disk arrays. The model is computationally simple and scales easily, making it potentially useful to performance engineers.

1 Introduction

Enterprise storage systems represent a growing market. In recent years there has been an explosion of applications that use enormous amounts of data and have high Quality of Service (QoS) requirements from their storage systems. To satisfy the QoS requirements of these applications, enterprise storage systems typically contain large disk arrays, commonly referred to as RAID (Redundant Array of Independent Disks). Disk arrays increase storage system speed by *striping* data across multiple disks and increase storage system availability by using redundancy. In order to perform capacity planning and predict the performance of enterprise storage systems, it is necessary to develop performance models of disk arrays. The architecture of current disk arrays, however, is complex. The mid-range and large disk arrays contain tens to hundreds of disks and have large caches for read-ahead and write-behind. In addition, disk arrays have sophisticated array controllers that are capable of performing optimizations such as adaptive prefetching based on automatic detection of sequential I/O streams [**?**]. The disks also have caches that implement smart prefetching schemes to improve performance. A reasonably accurate performance model of disk arrays must incorporate the effects of all these features.

Several papers have analyzed the performance of disk arrays using analytical and simulation models. These prior papers can be classified into three groups. The first group of papers analyze the multiple disks in the array but ignore the array cache. Kim and Tantawi [**?**] present an analytic method for approximating the disk service time of requests striped across n disks. In this early work, redundancy and queueing are not considered. Chen and Towsley [**?**, **?**] incorporate both redundancy and queueing in their performance model of RAID 5 disk arrays in normal mode. Merchant and Yu [**?**, **?**, **?**] analyze RAID 5 disk arrays in both normal and recovery modes. Thomasian and Menon [**?**, **?**] present performance models of RAID 5 in normal, degraded, and rebuild modes. Further, their model uses sequential access probability to capture workload spatial locality. Kuratti and Sanders [**?**] also incorporate workload spatial locality by assuming that request size distributions are quasi-geometric in their RAID 5 model for transaction processing workloads. All the above models assume that the disk array is servicing asynchronous I/O workloads. (Asynchronous workloads are composed of I/O requests generated by jobs that can have more than one I/O request outstanding at a time.) Lee and Katz [**?**] present an analytic performance model of disk arrays under synchronous I/O workloads. Their model assumes that disk service time is known and their model does not incorporate the effects of redundancy or CPU delay. DiskSim [**?**], RaidFrame [**?**], and Pantheon [**?**] are some simulation models of disk ar-

[*] This work was supported in part by Hewlett Packard and by NSF under Career Award CCR-0093111 and Information Technology Research grant JIS-0082399.

[†] This work was done while Xiaozhou Qiu was a student in UNH.

rays. Only the disk components of these models have been validated against real disk arrays.

The second group of prior papers analyze read-ahead or write-behind by disk array caches. These papers typically assume that disk service time is a known input parameter. Menon and Mattson [**?**, **?**] model explicit read-ahead and write-back by the array cache of a RAID 5 disk array. They assume that the cache write hit probability is 1 (*i.e.*, the write-back cache is never full). The remaining papers in this group present new read-ahead and write-behind policies for disk array caches. Varma and Jacobson [**?**] present a write-back caching algorithm that varies the rate of writes to disks based on the utilization of the write cache. They assume that writes which arrive when the cache is full are written directly to disks. Mishra and Mohapatra [**?**] present a write-back caching algorithm that stores both data and parity in the write-cache. They assume that writes which arrive when the cache is full have to wait until enough write data are written to disks. All the above papers assume that cache read hit rate is known. Wong and Wilkes [**?**] present an array cache replacement policy that improves the array cache read hit rate by ensuring that a data block is cached at either a client or the disk array, but not both. None of the papers on the array cache analyze the impact of caching on disk service time.

The third group of papers analyze the effects of both multiple disks and array caches on the performance of storage systems. There is only one prior paper in this category. Uysal, Alvarez, and Merchant [**?**] present a throughput model of RAID 1/0 disk arrays under asynchronous I/O workloads. They consider read-ahead and write-back caching and its impact on disk disk service time.

In this paper, we integrate and extend the current body of work on disk arrays by developing a throughput and response time model of disk arrays under synchronous I/O workloads. Synchronous workloads are composed of I/O requests generated by jobs that have at most one outstanding I/O request at a time. Such workloads make up a significant portion of the total I/O workload on computer systems (as much as $51-74\%$ in one study [**?**]). Our model incorporates almost all the realistic features of current disk arrays such as the effects of intelligent destaging (*i.e.*, flushing of dirty cache blocks to the disks), coalescing of multiple subrequests into a single disk access, redundancy layout in the disk array, and prefetching. The effects of sequentiality, concurrency, and the synchronous nature of I/O workloads is also incorporated in the model. Since none of the current disk array simulation models incorporate all these features, we validate our model against measurements from a real disk array using synthetic workloads.

The remaining part of the paper is organized as follows: Section 2 presents the disk array model and the techniques used to evaluate the performance of the model. Section 3 describes the architecture of the real disk array used in this study. Section 4 presents the issues involved in computing input parameter values to the disk array model. Section 5 presents the model validation results. The conclusions are presented in Section 6.

Parameter	Description
M	number of workload streams
$\mathsf{CPU\_delay}$	mean time spent by a job at its CPU
$\mathsf{request\_size}$	mean request size
$\mathsf{run\_count}$	mean # of sequential requests per run [1]
$\mathsf{random\_count}$	mean # of random requests between 2 runs

Table 1. Workload characterization

[1] A run in a workload stream is a string of sequential requests to contiguous bytes.

2 Disk Array Performance Model

We use queueing network models to analyze a disk array's performance since components common to all disk arrays can be explicitly modeled as service centers of the queueing network, while the complexities and details of specific (proprietary) disk array architectures can be captured in the model's input values. For example the disks can be explicitly modeled as queueing service centers, while caching at the disks can be implicitly modeled by incorporating disk caching effects in the disk service time value.

This paper analyzes disk arrays under synchronous I/O workloads. Jobs generating synchronous workload streams typically cycle between the disk array and the CPU/terminal. The workload parameters of significance to our model are presented in Table 1.

Below, we present an analysis of how read and write requests are treated by disk arrays. Based on this analysis, we develop a queueing network model and then present the technique to evaluate the model. Optimizing the mean performance measures is sufficient for conventional applications [**?**], so our performance technique computes mean performance measures such as the disk array's throughput, the rate at which I/O requests are serviced by the disk array, response time, the mean time spent by a request at the disk array, waiting for and receiving service, and queue length, the mean number of I/O requests at the disk array. Table 2 presents our performance technique outputs for a disk array model with M workload streams.

2.1 Read Requests

Read requests are first submitted to the disk array cache. With probability $\mathsf{cache\_hit\_probability}$ a read request's data are found in the cache, and the disk array

Performance Measures	Description
array_response_time[M]	mean response time
array_throughput[M]	mean throughput
array_queue[M]	mean queue length

Table 2. Performance technique outputs

controller signals service completion. With probability cache_miss_probability a request is forwarded to the disks, and the disk array controller signals service completion after all subrequests of the request complete service. From a performance perspective, the key components of a disk array are the array cache and the LUNdisks unit (*i.e.*, the group of disks in the array). The cache is modeled by a single queueing server with mean service time cache_service_time. The LUNdisks unit is modeled by a parallel queueing system since a request submitted to the disks is divided into subrequests assigned to some (or all) of the disks, and the request completes service only after all its subrequests complete. The parameter disk_access_probability represents the probability that a subrequest accesses a disk in the array. The parameter disk_service_time represents the mean time a disk takes to service a subrequest. The parameter parallel_overhead represents the additional time (over disk_service_time) taken to execute all the subrequests of a request.

Figure 1 presents the queueing network model of the disk array system with read workloads. Since the focus of this study is disk array performance modeling, for simplification, we assume that there is no queueing at the CPU and model the CPU as a delay server with mean service time CPU_delay. The MVA technique for parallel queueing networks [**?**] is the standard technique used to evaluate mean performance measures of closed parallel queueing networks. Appendix A presents the parallel MVA technique using the disk array input parameters presented above. An explanation of the parallel MVA technique is beyond the scope of this paper and an interested reader is referred to [**?**].

2.2 Write Requests

Write requests are first written to the disk array cache and completion is signaled as soon as the write-to-cache is completed. The “dirty” data in the cache are eventually written to the disks. The write-back caching policy determines when dirty data are written from the array cache to the disks. The write-back caching policy is typically determined by two key parameters, namely, the low_water and max_dirty. The parameter low_water determines the maximum number of dirty blocks that can be held in the cache without triggering disk writes. Therefore, in steady

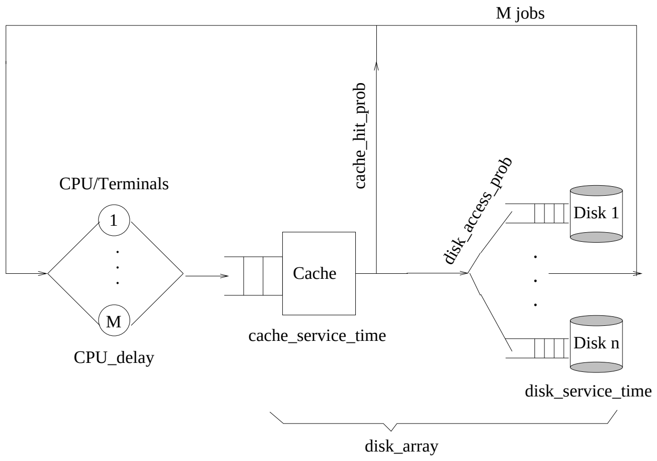

Figure 1. Queueing network model of disk arrays with read workloads

state, there is zero probability that the cache has less than low_water dirty blocks. The parameter max_dirty determines the maximum number of dirty data blocks that can be held in the array cache. Write requests that arrive when the cache is full must wait until enough dirty data are written out to the disks. Thus, there is zero probability that the cache has more than max_dirty blocks. Let i represent the number of dirty blocks in the cache. In steady state, the cache can have $\mathsf{low\_water} \leq i \leq \mathsf{max\_dirty}$ dirty blocks.

Let $\lambda[\mathsf{M}]$ represent the rate at which dirty blocks arrive at the cache and let $\mu[\mathsf{M}]$ represent the rate at which dirty blocks are written out to disks, when there are M workload streams accessing the disk array. In steady state, the cache can be modeled as a Markov birth-death M/M/1/K process [**?**], where $\mathsf{K} = (\mathsf{max\_dirty} - \mathsf{low\_water} + 1)$. The corresponding Markov chain shown in Figure 2, has states $\{\mathsf{low\_water}, ..., \mathsf{max\_dirty}\}$. Appendix B presents the technique used to compute the performance of disk arrays under writes.

Note that the M/M/1/K model assumes that the arrival distribution of dirty blocks to the cache and the service distribution of dirty blocks written to disks are exponential. The advantage of assuming exponential distribution is the efficiency and simplicity of the corresponding performance technique. The exponential assumption is not strictly satisfied, but the errors resulting from violation of this assumption are, at worst, comparable to inaccuracies resulting from other sources during modeling (e.g., errors in measurement data).

The input parameters for the read model and write model are summarized in Table 3. The values of the input parameters depend on the characteristics of the particular disk array being analyzed, so the next section presents the characteris-

Parameters	Description
	Read Model
disk_service_time[M]	mean disk service time
parallel_overhead[M]	sibling subrequest overhead
disk_access_probability[M]	prob. of accessing a disk
cache_service_time[M]	mean array cache service time
cache_hit_probability[M]	array cache hit probability
	Write Model
λ[M]	cache's data arrival rate
μ[M]	data rate from cache to disks

Table 3. Input parameters of the disk array model

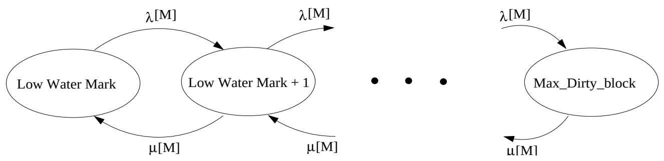

Figure 2. Markov chain representation of disk arrays with write workloads

tics of the disk array used in our study.

3 Disk Array Description

The Hewlett-Packard SureStore E FC-60 disk array [**?**] is used in our validation study since this mid-size storage system implements all of the features found in typical disk arrays. Figure 3 shows a representation of the FC-60 disk array. The FC-60 has 2 array controllers that are both connected to a single backplane bus. Each controller has 256 MB of battery backed cache memory (NVRAM). The backplane bus has 6 ultra-wide SCSI buses each connected to a SCSI controller. There can be up to 6 trays on the FC-60 and each tray has 2 SCSI controllers and up to 10 disks. Since there are only 6 ports on the backplane bus, only 6 SCSI controllers can be used at a time, with each SCSI controller handling 5 disks or 10 disks. Typically, the disks of an array are partitioned into independent, disjoint units called LUNs (logical units). In the FC-60, a LUN is formed by combining disks from different SCSI controllers. A typical configuration for the FC-60 is a fully configured array with 60 disks, 10 to each SCSI controller, for a total of ten 6-disk LUNs. Each array controller controls access to a disjoint set of LUNs. However, if one array controller fails then the other array controller takes over the responsibilities of the failed controller.

The RAID 1/0 configuration is a popular method of disk striping, so we model the FC-60 with its LUNs configured as RAID 1/0 arrays operating in failure-free mode. Each disk in the LUN is logically partitioned into equal sized blocks referred to as *stripe units*. The stripe unit size (stripe_unit_size), a multiple of the disk sector size, is set at the time of LUN configuration. The set of stripe units at the same physical location on each disk is referred to as a *stripe*. Thus, a RAID 1/0 disk array is logically partitioned into rows of stripes and columns of stripe units (Figure 4). To protect against failure of any one disk, RAID 1/0 uses mirrored redundancy with each disk having a mirrored pair. All write data have to be written both to the disk and its mirrored pair, and read data can be read from either disk.

All our experiments are run on one FC-60 LUN containing 6 disks, one from each tray. Table 4 presents the disk array configuration parameters of significance to the model. These parameter values are obtained from the manufacturer's specifications [**?**] or directly measured.

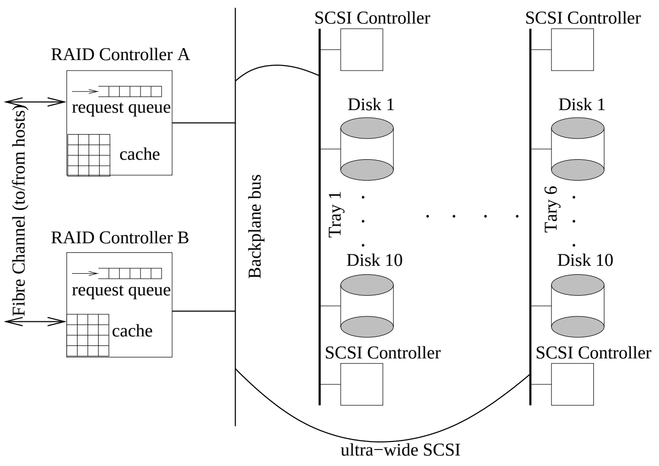

Figure 3. The HP FC-60 disk array

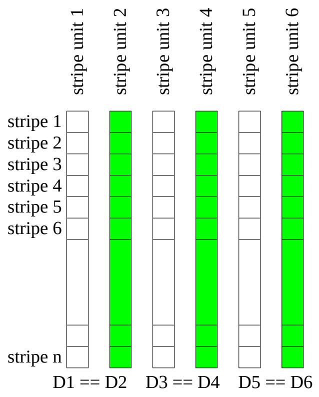

Figure 4. RAID **1/0 configuration**

Parameter	Description	Value/Units
cache_size	size of the cache at each controller	256 MB
cache_transfer_rate	mean cache transfer rate	62 MB/sec
read_ahead_size	mean number of additional bytes read from disk	≥ 0 bytes
stripe_unit_size	size of a stripe unit	16 KB
stripe_width	number of stripe units in a stripe (logical row)	6
disk_type	type of disks in the FC-60 disk array	Cheetah73
disk_capacity	total formatted capacity of a disk	73.4 GB
disk_read_seek_time	mean disk read seek time	6.05 ms
disk_write_seek_time	mean disk write seek time	6.55 ms
disk_revolutions	revolutions per minute	10000 rpm
disk_transfer_rate	mean disk transfer rate	31 MB/sec

Table 4. Disk array characterization

4 Disk Array Specific Input Parameters

Section 2 presents the disk array input parameters of significance to the performance techniques. Here, we present the computation of each of the input parameters to the performance techniques presented in Appendix A and Appendix B. For notational convenience, we drop [M] when referring to a parameter.

4.1 Disk Service Time

The disk service time is the sum of disk positioning time (= disk seek + rotate time) and disk transfer time. It is important to get accurate estimates of disk service time since the disks are the most heavily utilized components of a disk array. Hence, small errors in estimating disk service time will affect the disk array model's performance predictions more than large errors in estimating other parameter values. However, it is difficult to get accurate estimates of disk service time since the positioning time depends on several disk features such as the disk specifications, the disk caching policy, and the disk scheduling policy, and on several workload features such as the CPU delay, the number of workload streams, and the per-stream sequentiality. A disk service time computation technique must incorporate the effects of most (if not all) of these features in-order to be reasonably accurate. [**?**, **?**, **?**] present disk service time computation techniques that incorporate the effects of some of these features.

For the greatest accuracy, we compute disk positioning time by analyzing disk measurement data. This measurement data incorporates the effects of all the disk and workload features mentioned above. For random and sequential workloads, Figure 5 plots disk positioning time measurement data as a function of disk queue length. For random workloads the positioning time is given by

$$\mathsf{disk\_position\_time} = a + \frac{b}{\sqrt{1 + \mathsf{disk\_queue}}} \qquad (1)$$

By minimizing the root-mean square errors between the measured values and those given by the equation, it is determined that for the Cheetah 73 disk, the constants are $a = 3.53$ ms and $b = 8.81$ ms. For sequential workloads, the mean positioning time is given by

$$\mathsf{disk\_position\_time} = \begin{cases} Equation\ 1 & \mathsf{disk\_queue} > 3 \\ c + d * \mathsf{disk\_queue} & \mathsf{disk\_queue} \leq 3 \end{cases}$$

For Cheetah 73, $c = -2.73$ and $d = 3.68$.

For random workloads, the mean positioning time decreases as the disk queue length increases due to the effect of disk scheduling policies such as SCAN or CSCAN. When there is a single stream of sequential requests at the disk, the positioning time is close to zero, since the disk head moves little and read-ahead to the disk cache minimizes the effects of rotational latency [**?**]. As the number of sequential streams increases, the positioning time increases due to interference between streams. Disk measurements indicate that the sequentiality of the workload has minimal effect on disk service time for $\mathsf{disk\_queue} > 3$.

Thus, given a particular disk and its workload, the disk service time is a function of the disk queue length. However, the disk queue length is an output of a disk array (or disk) performance model. To address this circular relationship between disk service time and disk queue length, disk service time computation techniques typically assume that the disk queue length is a known input parameter. In our modeling study, disk queue length is a known input parameter only when $\mathsf{CPU\_delay} = 0$ in which case $\mathsf{disk\_queue} = \mathsf{M} * \mathsf{disk\_access\_probability}$. For workloads with $\mathsf{CPU\_delay} > 0$, a reasonably accurate approximate

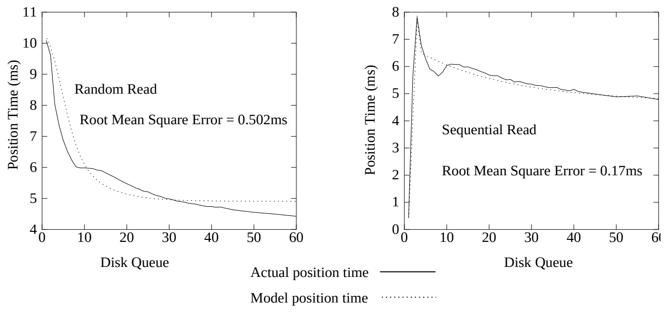

Figure 5. Disk mean read positioning time for random and sequential workloads

value of disk queue length must be computed and validated. Our computation of the approximate disk queue length is presented below.

Assume that $\mathsf{cache\_miss\_probability} = 1$. A job in the system is either at a disk or at a terminal/cache. If there are M jobs, then in the worst case, all the jobs are at the disks. Since the probability of a job being at any particular disk in the array is $\mathsf{disk\_access\_probability}$, $\mathsf{disk\_queue} = \mathsf{M} * \mathsf{disk\_access\_probability}$. Hence, $\mathsf{disk\_position\_time}$ (and subsequently $\mathsf{disk\_service\_time}$) for this disk queue length can be computed using one of the two $\mathsf{disk\_position\_time}$ equations given above. Then the disk response time for this worst case is given by $(\mathsf{disk\_service\_time} * \mathsf{M} * \mathsf{disk\_access\_probability})$. A job spends $\mathsf{CPU\_delay}$ at the CPU and in the worst case spends $(\mathsf{M} * \mathsf{cache\_service\_time})$ at the cache. Using Little's Law, the throughput of the disk array is given by $\mathsf{M}/(\mathsf{CPU\_delay} + (\mathsf{M} * \mathsf{cache\_service\_time}) + (\mathsf{disk\_service\_time} * \mathsf{M} * \mathsf{disk\_access\_probability}))$. Using Little's Law again, the approximate queue length at a disk is given by

$$\mathsf{disk\_queue} \approx (\mathsf{M} * (\mathsf{disk\_service\_time} * \mathsf{M} * \mathsf{disk\_access\_probability}))/(\mathsf{CPU\_delay} + (\mathsf{M} * \mathsf{cache\_service\_time}) + (\mathsf{disk\_service\_time} * \mathsf{M} * \mathsf{disk\_access\_probability}))$$

4.2 Parallelism Overhead

Parallelism overhead refers to the additional time (over $\mathsf{disk\_service\_time}$) taken to execute all the subrequests of a request. Since all sibling subrequests are the same size, the transfer time of each subrequest is the same ($\mathsf{disk\_transfer\_time} = \mathsf{subrequest\_size}/\mathsf{disk\_transfer\_rate}$), but the positioning time of each subrequest may be different. Let $\mathsf{max\_position\_time}$ represent the mean of the maximum positioning time from among $\mathsf{num\_subrequests\_per\_request}$ subrequests.

$$\mathsf{parallel\_overhead} = \mathsf{max\_position\_time} - \mathsf{disk\_position\_time}$$

In order to compute $\mathsf{max\_position\_time}$, the distribution of positioning time must be known. One could use the standard positioning time distribution [?] or bounded distributions such as the beta distribution [?] to model the distribution of positioning time. In [?], we compute parallel overhead by assuming that the positioning time distribution is modeled by the beta distribution.

4.3 Disk Access Probability

The $\mathsf{disk\_access\_probability}$ represents the probability that a request accesses data from a disk in the array. If it is assumed that requests access data uniformly from all the disks, then
$\mathsf{disk\_access\_probability} = \mathsf{cache\_miss\_probability} * \mathsf{num\_subrequests\_per\_request}/\mathsf{stripe\_width}$
The value of $\mathsf{num\_subrequests\_per\_request}$ is typically computed as a function of the stripe unit size and stripe width. For example, $\mathsf{num\_subrequests\_per\_request} = 1$ if the request size is smaller than the stripe unit size. However, the value of $\mathsf{num\_subrequests\_per\_request}$ depends on two other factors, namely, (a) the redundancy based load distribution policy that determines whether subrequest data are partly read from a disk and partly from its mirror or whether data are read entirely off one disk and not from its mirror, and (b) the access coalescing policy that determines whether large requests that straddle multiple stripe units on a disk are coalesced into a single subrequest. For example, consider large request sizes $\geq \mathsf{stripe\_width} * \mathsf{stripe\_unit\_size}$. Assume that access coalescing occurs for such large requests. Then, depending on the load distribution policy, $\mathsf{num\_subrequests\_per\_request} = \mathsf{stripe\_width}$ if request data are distributed between a disk and its mirror, else $\mathsf{num\_subrequests\_per\_request} = \mathsf{stripe\_width}/2$. Thus, in order to compute $\mathsf{num\_subrequests\_per\_request}$, the access coalescing policy and the load distribution policy for the disk array must be known in addition to the stripe unit size and the stripe width. The details of the FC-60 load distribution and access coalescing policy are outlined in [?].

4.4 Cache Parameters

A cache hit on a read request occurs if (a) this request is part of a sequential stream of requests submitted to the disk array and was read into the cache as part of read-ahead data, or (b) this request had been referenced in the past and the request's data are still in the cache. The random variables representing read-ahead and re-reference hits are independent since the probability that a request's data results in a read-ahead hit is not related to whether this request's data results in a re-reference hit. The cache hit probability is then given by

$\mathsf{cache\_hit\_probability} = 1 - (\mathsf{read\_ahead\_miss\_probability} * \mathsf{re\_reference\_miss\_probability})$

Techniques for computing the re-reference probability are presented in several papers [**?**, **?**, **?**]. The read-ahead probability must incorporate the effect of explicit read-ahead (*i.e.*, every read access from the disks results in an additional system-defined number of bytes being read into the cache) and also the effect of adaptive prefetching based on detection of I/O sequentiality. A technique for computing the explicit read-ahead hit rate is given in [**?**]. Here, we do not incorporate the effects of adaptive prefetching.

The cache service time is the rate at which requests are transferred from the array cache to the main system and is equal to $\mathsf{request\_size}/\mathsf{cache\_transfer\_rate}$.

4.5 Write Model Input Parameters

The parameter μ represents the rate at which dirty blocks are written out to the disks from the array cache. Since all data written to a disk must also be written to the disk's copy,

$$\mu = \frac{\mathsf{stripe\_width}}{2 \, * \, \mathsf{disk\_service\_time}}$$

We now outline our computation of λ, the rate at which dirty blocks arrive at the disk array. The disk array under write workloads is modeled by a $\mathsf{M/M/1/K}$ queue. An implication of this model is that requests that arrive when the cache is full (*i.e.*, $i = \mathsf{max\_dirty}$) are "lost." In reality, however, requests that arrive when a cache is full are not "lost" since these requests block until enough dirty data blocks are written to the disks. This behavior of disk arrays is implicitly captured in our model by setting $\lambda = \mathsf{max\_array\_throughput}$, where $\mathsf{max\_array\_throughput}$ is the maximum rate at which dirty blocks can be written to the disk array system for a given I/O workload. The maximum throughput of the disk array can be reached if all write blocks are written to the cache immediately, that is, if $\mathsf{max\_dirty} \to \infty$. In this case, the disk array is modeled by the cache alone and the the MVA technique can be used to compute $\mathsf{max\_array\_throughput}$.

5 Empirical Validation

A HP 9000-N4000 server with eight 440 MHz PA-RISC 8500 processors and 16 GB of main memory, running the HP-UX 11.00 operating system, is used to generated the workloads and access the FC-60 array. The workloads are generated using a synthetic load generator. The inputs to the load generator are:

- request size: 4K, 8K, 16K, 32K, 48K, 64K, 128K, 256K,
- request type: read-only, write-only,
- run count: 1 for random requests, 64 for sequential requests,
- random count: 0 (*i.e.*, a workload stream consists of runs of length $\mathsf{run\_count}$),
- CPU delay: exponentially distributed with mean 0 ms (no delay), 10 ms, 30 ms, 100 ms,
- multiprogramming level: 1 to 12.

A trace of all I/O activity at the device driver level is collected. We ran each experiment until the 95% confidence interval for each metric was less than 4% of the point value.

Figures 6 and 7 plot the model's mean response times and the actual response times for read and write workloads for varying request sizes and multiprogramming levels, and CPU delay of 0 ms and 10 ms. The model's performance predictions are a good match for the experimental performance measures. The model is better for random I/Os than for sequential I/Os, and better for reads than for writes. The average errors range from 3 ms (for random I/Os) to 5 ms (sequential I/Os).

We now analyze the effect of I/O workload sequentiality on the performance of disk arrays under read workloads. Our graphs indicate that the sequentiality of workloads has little impact on performance as the disk queue lengths and the request sizes increase. In fact, the performance under sequential workloads is better than the performance under random workloads only when $\mathsf{disk\_queue} \leq 3$. Also, for sequential workloads, when comparing the actual system's performance against the model's predictions, the performance of the actual system is better than the model only when $\mathsf{disk\_queue} \leq 3$. Since our model does not capture the effects of adaptive prefetching, this similarity between the actual system's performance and the model's predictions indicates that for the workloads tested, the adaptive prefetching policies of FC-60 are effective only at small disk queue lengths. When considering the disk array under write workloads, the performance of the disk array under sequential and random writes is quite similar This indicates that the sequentiality of the write workloads has little impact on performance as disk queue lengths and request sizes increase.

6 Conclusions

The paper presents and validates a disk array performance model that incorporates the effects of array caching, multiple disks, and sophisticated array controllers. The model is validated against a specific disk array (the FC-60

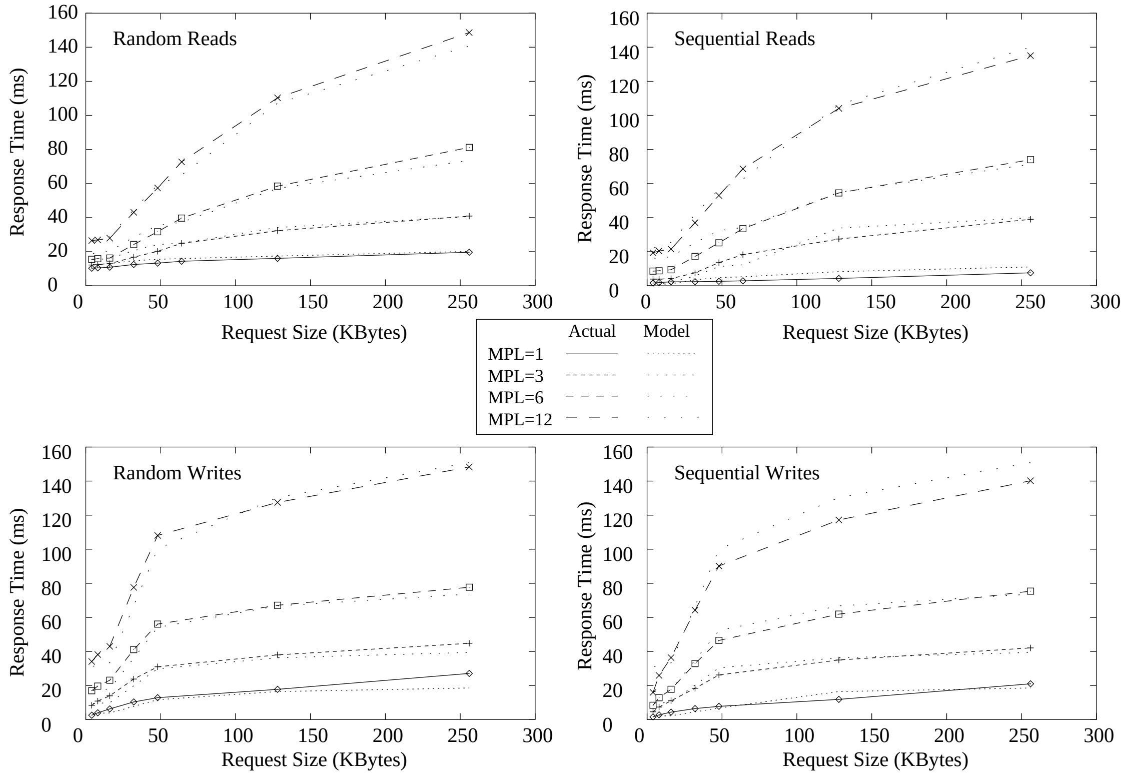

Figure 6. Model versus actual response times for $\text{CPU\_delay} = 0$ ms

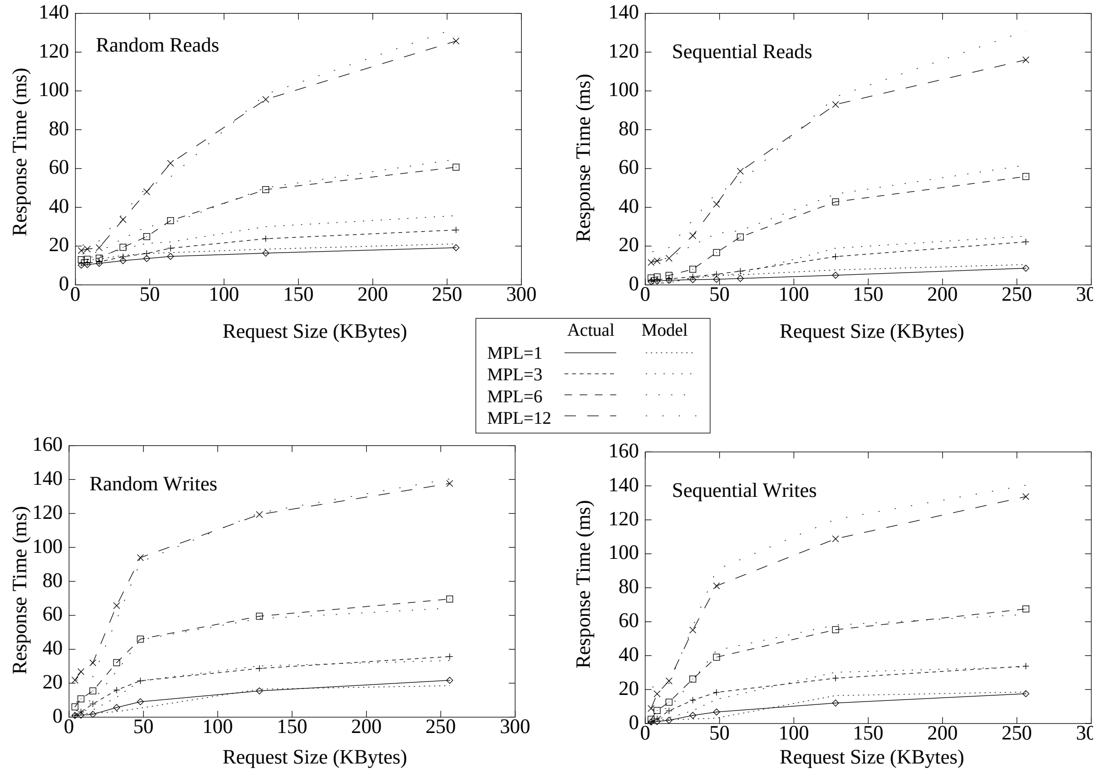

Figure 7. Model versus actual response times for $\text{CPU\_delay} = 10$ ms

array). However, the model can be used to compute performance measures of other disk arrays since the modeling approach is general in that features common to all disk arrays are explicitly modeled while the specifics of a particular proprietary system are implicitly modeled by incorporating their effects in the input parameter values. For example, multiple disks (common to all disk arrays) are explicitly modeled as service centers while the effect of caching at the disks, which varies depending on the caching policy of the specific disk array, is implicitly captured by reducing the disk service times appropriately.

An advantage of our integrated disk array performance model is that it is possible to evaluate the combined effects of array caching and multiple disks on the overall performance of the disk array. The model also helps identify features of a disk array that could be improved. For example, our study indicates that the array caching policies could be improved since the sequentiality of workload streams has little impact on the performance of FC-60. Our analytical model is suitable for use in capacity planning and in storage management systems, where it is necessary to estimate whether a given array can meet the performance requirements of a given set of workloads. Such systems typically use optimization techniques which require repeated evaluations of different storage configurations [**?**, **?**], and therefore require fast, yet reasonably accurate, performance predictions.

Acknowledgments

We thank John Wilkes, Eitan Bachmat, Mustafa Uysal, Manish Madhukar, Sajeev Varki, and several anonymous reviewers for their helpful comments.

A Appendix: Performance Technique for Reads

We show how the parallel MVA technique is used to compute the mean response time, throughput, and queue length of a disk array under read workloads. For multiprogramming level m varying from 1 to M, iteratively compute

1. $\mathsf{array\_response\_time}[m] = \mathsf{cache\_response\_time}[m] + \mathsf{LUN\_response\_time}[m]$

   where

   $\mathsf{cache\_response\_time}[m] = \mathsf{cache\_service\_time}[\mathsf{M}] * (1 + \mathsf{cache\_queue}[m-1])$

   $\mathsf{LUN\_response\_time}[m] \approx \mathsf{cache\_miss\_probability}[\mathsf{M}] * (\mathsf{parallel\_overhead}[\mathsf{M}] + \mathsf{disk\_service\_time}[\mathsf{M}]) + (\mathsf{disk\_access\_probability}[\mathsf{M}] * \mathsf{disk\_service\_time}[\mathsf{M}] * \mathsf{LUN\_queue}[m-1])$

   The parameter $\mathsf{cache\_queue}$ represents the mean queue length at the array cache. The parameter $\mathsf{LUN\_queue}$ represents the mean queue length at the LUNdisks. For $m = 1$, $\mathsf{cache\_queue}[m-1] = \mathsf{cache\_queue}[0] = 0$ and $\mathsf{LUN\_queue}[m-1] = \mathsf{LUN\_queue}[0] = 0$. For $m > 1$, the computation of $\mathsf{cache\_queue}$ and $\mathsf{LUN\_queue}$ is presented in Step 3 of this algorithm.

2. $\mathsf{array\_throughput}[m] = m/(\mathsf{CPU\_delay} + \mathsf{array\_response\_time}[m])$

3. The queue lengths at the LUNdisks and the cache are computed using Little's Law,

   $\mathsf{LUN\_queue}[m] = \mathsf{LUN\_response\_time}[m] * \mathsf{array\_throughput}[m]$

   $\mathsf{cache\_queue}[m] = \mathsf{cache\_response\_time}[m] * \mathsf{array\_throughput}[m]$

   The disk array queue length represents the total number of outstanding requests at the array cache and LUNdisks.

   $\mathsf{array\_queue}[m] = \mathsf{LUN\_queue}[m] + \mathsf{cache\_queue}[m]$

B Appendix: Performance Technique for Writes

The disk array under write workloads is modeled using the M/M/1/K queue, and here we show how the disk array's mean performance measures are computed. Let $\rho = \lambda[\mathsf{M}]/\mu[\mathsf{M}]$ represent the utilization of the M/M/1/K queue. Then, the steady state probability distribution of dirty blocks in the cache is [**?**]

$$P_i = \begin{cases} \frac{(1-\rho)*\rho^{(i\ -\ \mathsf{low\_water})}}{(1-\rho^{\mathsf{K}+1})} & \text{for } \mathsf{low\_water} \leq i \leq \mathsf{max\_dirty} \\ 0 & \text{otherwise} \end{cases}$$

Each job submits write requests that are equivalent to one or more dirty blocks. Let $\mathsf{number\_of\_dirty\_blocks}$ represent the number of dirty blocks per I/O request.

$$\mathsf{number\_of\_dirty\_blocks} = \left\lceil \frac{\mathsf{request\_size}}{\mathsf{stripe\_unit\_size}} \right\rceil$$

The mean throughput (*i.e.*, requests serviced per unit time) of the disk array is computed from the $\mathsf{M/M/1/K}$ queueing model

$$\mathsf{array\_throughput}[\mathsf{M}] = \frac{\lambda[\mathsf{M}] * (1 - P_{\mathsf{max\_dirty}})}{\mathsf{number\_of\_dirty\_blocks}}$$

The mean response time is computed using Little's Law on the entire system.

$$\mathsf{array\_response\_time}[\mathsf{M}] = \frac{\mathsf{M}}{\mathsf{array\_throughput}[\mathsf{M}]} - \mathsf{CPU\_delay}$$

The disk array queue length is computed using Little's Law on the disk array.

$$\mathsf{array\_queue}[\mathsf{M}] = \mathsf{array\_throughput}[\mathsf{M}] * \mathsf{array\_response\_time}[\mathsf{M}]$$

References