An Analytical Performance Model of Disk Arrays under Synchronous I/O Workloads

Elizabeth Varki Arif Merchant Xiaozhou Qiu

Abstract

All server storage environments depend on disk arrays to satisfy their capacity, reliability, and availability requirements. In order to manage these storage systems efficiently, it is necessary to understand the behavior of disk arrays and predict their performance. We develop an analytical model that estimates mean performance measures of disk arrays under a synchronous I/O workload. Synchronous I/O requests are generated by jobs that each block while their request is serviced. Upon I/O service completion, a job may use other computer resources before issuing another I/O request. Our disk array model considers the effect of workload sequentiality, read-ahead caching, write-back caching, and other complex optimizations incorporated into most disk arrays. The model is validated against a mid-range disk-array for a variety of synthetic I/O workloads. The model is computationally simple and scales easily as the number of jobs issuing requests increases, making it potentially useful to performance engineers.

Index Terms: parallel input/output, RAID, performance evaluation, I/O performance, disk striping.

1 Introduction

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 requirements from their storage systems. To satisfy the capacity and availability requirements of these applications, storage systems typically contain large disk arrays that are capable of storing terabytes of data and have high availability. In order to perform capacity planning and predict the performance of enterprise storage systems, it is necessary to develop performance models of the disk arrays within these storage systems. The architecture of current disk arrays is complex. The mid-range and large disk arrays contain tens to hundreds of disks that can be configured using various RAID schemes. These disk arrays also have large caches for read-ahead and write-behind, and the array controllers are capable of performing optimizations such as adaptive prefetching, automatic detection of sequential I/O streams, and request coalescing [22]. In addition, the disks themselves have caches that implement smart prefetching schemes to improve performance. The optimizations have largely been developed independently which result in their combined effects being non-intuitive and difficult to understand [10], let alone model. As a result, most existing models of disk arrays do not consider the effect of the array cache and other optimizations (see Section 2), but these effects are critical to the performance of the array.

Performance models are built using either simulation techniques or analytic techniques. While both models can be used to predict the performance of disk arrays under various design trade-offs, analytic models are less expensive and orders of magnitude faster than simulation models. The speed of analytic models is the decisive factor for using these models within automatic storage management systems such as Minerva [1] that have to evaluate thousands of candidate data-to-device configurations. We develop an analytic performance model that computes the mean throughput and the mean response time of disk arrays under synchronous I/O workloads. Such workloads are composed of I/O requests generated by jobs that block until their request completes service. Upon unblocking, a job may spend time at other components (CPU, terminal) of the computer system before issuing another I/O request. In such systems, the I/O request rate to the storage system is not only dependent on the storage system performance but also on the performance of other system components. So, a performance model of storage devices under synchronous I/O workloads must model the effect of the interaction between different system components, which is difficult. Hence, almost all the disk array models in the literature [3, 4, 15, 16, 17, 21, 14, 13] address only asynchronous I/O workloads. Synchronous I/O workloads, however, make up a significant portion of the total I/O workload on computer systems (as much as $51-74\%$ in one study [19]). Therefore, it is important to analyze the performance of I/O systems under such workloads.

This paper computes disk array performance measures under synchronous I/O workloads for disks configured as a RAID 1/0 array (see Section 3 for a description). Our disk array model incorporates all the realistic aspects of disk arrays such as the effects and interactions of read-ahead and write-back caching, disk scheduling, request coalescing, and redundancy layout in the array. Our model also incorporates the effect of sequentiality, concurrency, and think time in the workload. Furthermore, our model uses a reasonably small and appropriate set of input parameters that can be measured for real workloads and disk arrays. Even though our disk-array model incorporates almost all the realistic aspects of disk arrays and their workloads, the model is simple and intuitive. The modeling technique is based on our extension of the mean value technique for fork-join networks [23]. The simplicity and computational efficiency of the model of a complex disk-array system is a key contribution of this paper. Another strength of this paper is that the predictions of our model are validated against measurements on a real array, the HP FC-60. The model predictions are, on average, accurate within 9% of the actual measurements from the array for a variety of synthetic workloads. A weakness of this paper is that our model has only been validated for single class workloads containing read-only and write-only requests. A model for multiple class workloads containing a mixture of read and write requests is presented without validation.

The remaining part of the paper is organized as follows: Section 2 summarizes existing disk-array models. Section 3 describes the architecture and configuration of disk arrays and lists the disk array and workload parameters relevant to performance modeling. Section 4 presents the fork-join model of disk arrays and our extension of the fork-join MVA technique. The read, write, and read-write models are described in Section 5. Section 6 presents the model validation results. The conclusions and potential future research are presented in Section 7.

2 Related Work

There are several published analytical models of disk arrays. All but one of these models analyse non-synchronous I/O workloads. The only model of disk arrays under synchronous I/O workloads is the model presented by Lee and Katz [12]. This early work models a very simple disk array with striping, but no redundancy. Features such as caching, controller overheads and disk scheduling are not considered, and the workload consists of randomly distributed I/O requests with no think time (i.e., terminal think time and CPU overhead are not modeled). Their model also treats reads and writes similarly and assumes that the mean service time of a request at a disk is known. All the other published models analyze disk arrays under asynchronous workloads. Bitton and Gray [2] present a model of seek time in shadowed disks, an early version of RAID1. Kim and Tantawi [11] analyze the positioning time delays in asynchronous disk interleaving, where sub-blocks of a data block are placed independently of one another. Analytical response time models of a disk array are presented by Chen and Towsley [3, 4]. Merchant and Yu [15, 16, 17] model normal and degraded mode response time for disk arrays using RAID1 and RAID5 layouts. Thomasian and Menon [21] analyze the performance of RAID5 with distributed sparing in an OLTP environment in normal, degraded and rebuild modes. Menon and Mattson [14, 13] include some simple cache effects in their response-time models of disk arrays using RAID5 layout. Disk drives are not directly modeled as a part of these models. Uysal, Alvarez, and Merchant [22] present a throughput model of disk arrays that is based on a hierarchical decomposition of the internal array architecture. Their model considers the effect of caching and is the only prior model validated against a real array.

Simulation models of disk systems include DiskSim [7] and Pantheon [25], which model disks, buses, adapters, controllers, and drivers. Only the disk component models in these have been validated against physical disks. The software RAID controller RaidFrame [6] can also be used for simulation of disk arrays.

The results of several of the prior analytical models have been compared against simple simulations, but neither the simulations nor the analytical results have been validated against real disk arrays, except as noted above. Comparisons with a simulator (see [9]) show that most of the models have substantial errors. It is likely that comparisons with a real array would reveal yet greater discrepancies. The analytical model presented in this paper is the first one to consider the synchronous nature of I/O workloads (including terminal think time) and also incorporate the effects of multiple behaviors of real arrays such as intelligent destaging (*i.e.*, flushing of dirty cache blocks to the disks), request coalescing into a single disk access, prefetching, and caching. It is one of the first analytical models validated against a commercial disk array.

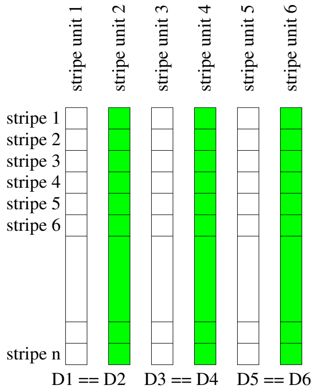

Figure 1: RAID 1/0 configuration

3 Disk Array Description

Disk arrays not only increase storage system speed by *striping* data across multiple disks in parallel but also increase storage system availability by using redundancy. An array of disks organized in this fashion is called a Redundant Array of Independent Disks (RAID) [18]. The RAID levels specify different techniques for distributing data and redundant information (parity) across the disks of an array. The RAID policy on a disk array is enforced by the array controller. Most disk array controllers have large caches that perform read-ahead and write-back (request completion is signaled as soon as the data are written to the non-volatile cache). In addition to these performance enhancing techniques, the controller uses other optimization techniques like automatic detection of sequential accesses, efficient demotion of cache dirty blocks, and request coalescing. The disks also have caches that implement optimization policies like read-ahead to improve performance.

Current disk arrays contain tens to hundreds of disks. To ease disk array management and improve efficiency, the disks within an array can be logically divided into independent, disjoint partitions of disks referred to as *logical units* (LUs). Each LU within an array can be configured at a different RAID level. We now describe RAID 1/0 which is the RAID level modeled in this paper. Next, we describe the HP FC-60 disk array, the array used to validate our model. Finally, the disk array and workload parameters of significance to the model are presented.

3.1 RAID 1/0 Overview

This paper models a disk array with its LUs configured as RAID 1/0 arrays operating in failure-free mode. A LU configured as a RAID 1/0 disk array consists of *LU_disks* disks. Each disk 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 LU 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 1). Each stripe consists of *LU_disks* stripe units, one from each disk of the array. 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 request data can be read from either disk. Write requests to the disk array are divided into subrequests that are at most equal to the size of a stripe unit. (If the starting location of the write falls between a stripe unit boundary then the size of the first subrequest will be less than a stripe unit; the size of the last subrequest may also be less than a stripe unit.) The subrequests are written stripe-wise (*i.e.*, row-wise). The first sub-request can be assigned to any disk (not necessarily disk-1) and following subrequests are written onto adjoining disks' stripe units that are in the same stripe as the first subrequest's stripe unit. Once the last stripe unit on a stripe is written, the next subrequest will be assigned to the first stripe-unit of the next stripe. Requests larger than a stripe will straddle multiple stripe units on a disk. In this case, multiple sub-requests to the same disk can be coalesced into a single sub-request. Thus, the requests to a RAID 1/0 disk array are striped across the disks of the array, combining the striping of RAID 0 and the mirroring of RAID 1.

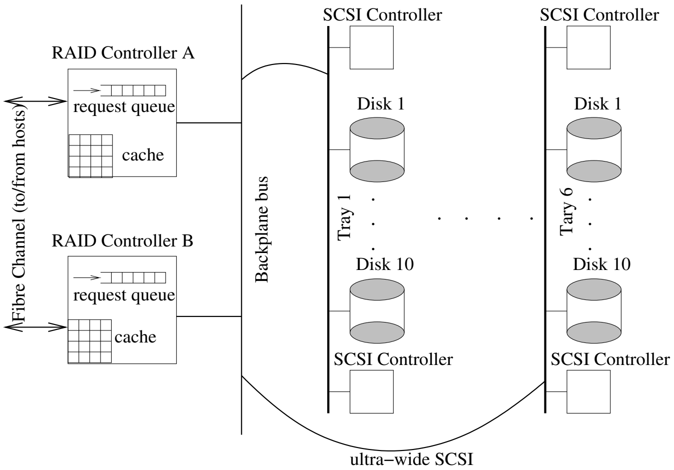

Figure 2: The HP FC-60 disk array

3.2 HP FC-60 Disk Array

The architecture of the Hewlett-Packard SureStore E disk array FC-60 [8], used to validate our disk array model, is described here. Figure 2 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). If the arrays are set up to mirror the dirty blocks, then the size of the write cache is limited to 128 MB. The backplane bus has 6 ultra-wide SCSI buses each connected to a SCSI controller. Each SCSI controller is connected to a tray. There can be up to 6 trays on the FC-60 and each tray has 2 SCSI controllers and up to 10 disks. Either one or both SCSI controllers on a tray can be in use depending upon a switch setting on the tray. If only one SCSI controller on a tray is in use, all the disks on the tray are connected to it. If both SCSI controllers on a tray are in use, odd numbered disks are connected to one controller and even numbered disks are connected to the other controller. Since there are only 6 ports on the backplane bus, only 6 SCSI controllers can be used at a time. In other words, the FC-60 can have up to 6 SCSI strings, each with one SCSI controller and either 5 disks or 10 disks. Thus, the FC-60 holds up to 60 disks - at 73 GB per disk drive, the total raw capacity is about 4.3 terabytes.

In the FC-60, a LU is formed by combining disks from different SCSI strings. So, a set containing disk i from each of the 6 trays can be configured as a LU. A typical configuration for the FC-60 is a fully configured array with 60 disks, 10 on each SCSI string, for a total of ten 6-disk LUs. Each array controller controls access to a disjoint set of LUs. However, if one array controller fails then the other array controller takes over the responsibilities of the failed controller.

3.3 Disk Array and Workload Parameters

The input parameters to the performance model are the array and workload parameters. Table 1 presents the array parameters of significance to the model. The FC-60 array used in our experiments has Cheetah ST173404LC disks, two controllers, and 256 MB of cache in each controller. All our experiments are run on one FC-60 LU containing 6 disks, one from each tray. The LU is configured at RAID 1/0 and uses a stripe unit size of 16 KB. The disk parameter values are obtained from the manufacturer's specifications or directly measured.

The workload parameters of significance to our model are presented in Table 2. These parameters sufficiently represent the steady state behavior of the workload and are input to our disk array model. The I/O workload is generated by a fixed number, M, of similar interactive jobs that each spends time at its terminal before issuing a request of size $request\_size$ to the disk array. A job is blocked while its I/O request is being serviced. On completion of its I/O request the job unblocks and spends some time, $terminal\_think\_time$, at its terminal before submitting another read or write I/O request. The fraction of read and write requests in the I/O workload is given by $read\_fraction$ and $write\_fraction$ ($= 1 - read\_fraction$). The sequentiality of a workload is an important parameter, and workloads typically contain a number of consecutive requests for sequential bytes (a *run*) followed by intervals of random (non-sequential) requests. This spatial locality of workloads is captured by the attribute run_count, the mean number

Parameter	Description	Value/Units
$cache\_size$	size of the cache at each controller	256 MB
$bus\_transfer\_rate$	mean cache transfer rate	86 MB/sec
$read\_ahead\_size$	mean number of additional bytes read from disk	≥ 0 bytes
LU_disks	number of disks in a logical unit	6 disks
$stripe\_unit\_size$	size of a stripe unit	16 KB
$stripe\_size$	number of bytes in a stripe (logical row)	48 KB
$disk\_type$	type of disks in the FC-60 disk array	Cheetah73
$disk\_capacity$	total formatted capacity of a disk	73.4 GB
$disk\_avg\_read\_position\_time$	mean disk read positioning time	9.72 ms
$disk\_avg\_write\_position\_time$	mean disk write positioning time	10.2 ms
$disk\_seq\_position\_time$	mean disk position time when workload is highly sequential	2 ms
$disk\_high\_traffic\_position\_time$	mean disk position time when disk queue is long	6.2 ms
$disk\_transfer\_rate$	mean disk transfer rate	33 MB/sec
$standard\_deviation\_position\_time$	standard deviation of disk positioning time	3 ms
N	maximum dirty data blocks that can be held in cache	> 0
$write\_cache\_low\_water\_mark$	maximum dirty blocks that can be held in cache without triggering disk writes	> 0
K	$N - write\_cache\_low\_water\_mark$	≈ 5 blocks

Table 1: Disk array parameters

Parameter	Description	Value/Units
M	multiprogramming level (i.e., number of jobs issuing I/O requests)	≥ 1
$terminal\_think\_time$	mean time spent by a job at its terminal	≥ 0 sec
$request\_size$	mean request size	4 KB to 256 KB
run_count	mean number of requests made to contiguous addresses	1-64 requests
$random\_count$	mean number of random requests between each run	≥ 0 requests
$read\_fraction$	fraction of read requests in the workload	0-1
$re\_reference\_distance$	amount of data accessed between consecutive accesses to the same block	bytes

Table 2: Workload parameters

of sequential requests in a run, and $random\_count$, the mean number of random requests between two runs. The temporal locality of workloads is captured by the attribute $re\_reference\_distance$, the number of bytes accessed between two accesses to the same block. The $re\_reference\_distance$ attribute is represented as a histogram of re-reference distances.

4 Queueing Model

Our performance technique is based on queuing analysis, so we describe the queuing model of the disk array system. The computer system is modeled by a closed queuing network linking the different system components. The network is closed because jobs that generate only synchronous I/O requests typically cycle between CPU execution and I/O wait states [7]. Figure 3 shows a closed network that models a computer system with M interactive jobs that cycle between the M terminals and the disk-array. A terminal's service time, known as its $terminal\_think\_time$, is the time that its "owner" job waits for user input during each cycle. The terminal think time also includes CPU processing time since this component is not explicitly modeled in the figure. The disk array is modeled by three components, the cache, the controller, and the disks. Even though the array cache is part of the array controller, it is modeled as a separate component since I/O requests that can be serviced by the cache need not be submitted to the disks. Components like

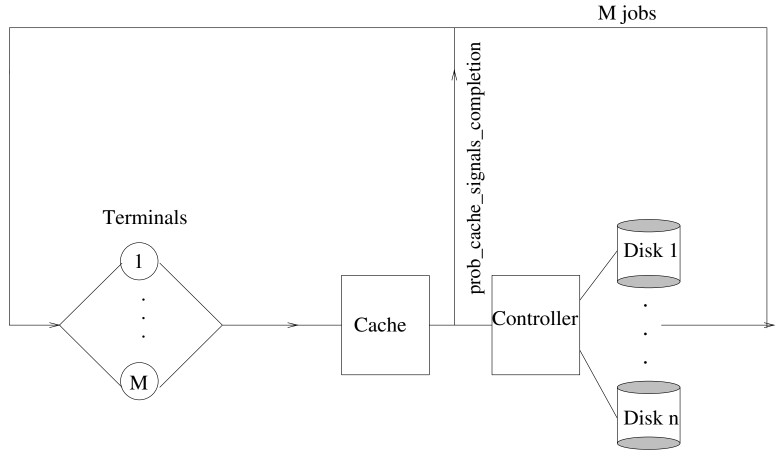

Figure 3: System model under a synchronous I/O workload

the (SCSI) disk controllers and the interconnects are not modeled explicitly since their service times are much smaller than the disk service times and do not have a significant impact on the disk-array performance. A job, on completing its terminal think time, submits a request to the array cache. If the request can be serviced by the cache, then the cache signals service completion without forwarding the request to the controller. If the request cannot be serviced by the cache, then the cache forwards the request to the controller who signals service completion once all subrequests of this request complete service at the disks. A job, on receiving its request's service completion signal, unblocks and spends time at its terminal before issuing another request to the disk array.

The modeled components of the computer system are the terminals, the array cache, the array controller, and the disks. The terminals are modeled by a delay server (not a queuing server) since each interactive job "owns" a terminal. The cache is modeled by a single queuing server. The controller is modeled by a *fork-join* queue. Each disk is modeled by a queuing server of this fork-join queue. A request to the controller is forked (divided) into subrequests that are assigned to the separate disks, and the request is joined (completes service) when all its subrequests complete service. A queuing network containing fork-join queues, as shown in Figure 3, is known as a fork-join network. We now describes the performance technique used to analyze fork-join networks that model disk arrays under synchronous workloads.

4.1 Multiple Class MVA Technique for Fork-Join Networks

The fork-join MVA technique in [23] assumes that a request (job) submitted to a fork-join queue divides into N subrequests that are assigned to each of the N queueing servers of this fork-join queue. For disk arrays, this assumption is unrealistic since requests may not access data from all the disks. In addition, the fork-join MVA technique of [23] assumes a single job class. Here, we extend the fork-join MVA technique to compute performance measures of fork-join networks modeling disk arrays under multiple job classes.

Consider a network with K' service centers and C job classes with workload intensity denoted by $\vec{M} = (M_1, M_2, \cdots, M_C)$, where M_c is the number of class c jobs. A service center can be a single queue, a fork-join queue, or a delay center. For each center k, let $V_{c,k}$ represent the probability that a c class job will visit center k during each cycle. If $S_{c,k}$ is the mean service time of a class c job at center k, the service demand of the class c job at center k, $D_{c,k}$, equals $V_{c,k} * S_{c,k}$. The MVA technique computes the mean performance measures (throughput, response time, queue length, utilization) at: (a) each service center for each job class ($X_{c,k}$, $R_{c,k}$, $Q_{c,k}$, $U_{c,k}$), (b) each center over all the classes (X_k, R_k, Q_k, U_k), and (c) the overall network for each class (X_c, R_c, Q_c, U_c). To refer to a performance parameter at a specific multiprogramming level $\vec{M}$, the term $(\vec{M})$ is attached to the parameter. The multiple class MVA technique is based on three equations:

- For each class c, the response time at each service center k is written as the sum of the service time and wait time for a class c job at the center:

$$R_{c,k}(\vec{M}) \begin{cases} = D_{c,k} & \text{delay centers} \\ = D_{c,k}[1 + A_{c,k}(\vec{M})] & \text{single queuing centers} \\ \approx D_{c,k} + d_{c,k} * A_{c,k}(\vec{M}) & \text{fork-join queuing centers [24]} \end{cases} \quad (1)$$

where $A_{c,k}(\vec{M})$ is the mean number of jobs seen at center k when a class c job arrives at the center. The key to the MVA technique is the computation of $A_{c,k}(\vec{M})$. For fork-join networks under multiple class workloads, we conjecture that

$$A_{c,k}(\vec{M}) \approx Q_k(\vec{M} - \vec{1_c}),$$

where $\vec{M} - \vec{1_c}$ is population $\vec{M}$ with one class c job removed. The above equations then equate the the mean arrival queue length at a center to the mean queue length at the center when the arriving job is removed from the network.

- For each class c, the system throughput is derived using Little's law to the entire network:

$$X_c(\vec{M}) \approx \frac{M_c}{terminal\_think\_time_c + \sum_{k=1}^{K'} R_{c,k}(\vec{M})} \quad (2)$$

- For each class c, the queue length at each service center is derived using Little's law to each center:

$$Q_{c,k}(\vec{M}) \approx X_c(\vec{M}) * R_{c,k}(\vec{M}) \quad (3)$$

The total queue length at center k for the C classes is given by:

$$Q_k(\vec{M}) \approx \sum_{c=1}^{C} Q_{c,k}(\vec{M})$$

The MVA technique iteratively solves Equations 1, 2, and 3 by varying the multiprogramming level of the network from $\vec{0}$ to $(M_1, M_2, \cdots, M_C)$ for each feasible population of the M_c (*i.e.*, each M_c varies from 0 to M_c). The iteration starts by setting $Q_k(\vec{0}) = 0$ since the queue length at each center is zero when there are no jobs in the network. Next, the mean performance measures of the network with one class c ($c = 1, \cdots, C$) job is computed by setting $A_{c,k}(\vec{1_c}) = Q_k(\vec{0})$. (Here, $\vec{1_c}$ represents a population with one class c job and zero jobs from other job classes.) Successive applications of the equations compute mean performance measures of the network for increasing multiprogramming levels. Note that in general the solution for each population $\vec{m}$ requires inputs from each $\vec{m} - \vec{1_c}$ ($c = 1, 2, \cdots, C$) population.

5 Disk Array Performance Model

The disk array performance modeling technique computes:

1. mean service demands at the cache, controller, and disks, and the probability that a request is serviced by the cache. The computation of these service demands incorporates the effects and interactions of a) caching, disk scheduling, request coalescing, and redundancy layout of the disk array; and b) sequentiality, concurrency, and multiple streams in the workload.

2. mean throughput (*i.e.,* the mean number of requests serviced per time unit) mean response time, mean queue length, and mean utilization of the disk array. The performance techniques used are a combination of the $M/M/1/K$ technique [5] and our extension of the fork-join MVA technique.

This section describes both the computation of the service demands at a disk array and the techniques that use these service demands to compute performance measures of a disk array. The performance technique is determined by the request type (read/write) and request size of the disk array workload since read requests, large write requests (greater than two stripes), and small write requests are treated differently by the disk array. For each of these workload types, we first describe the performance technique and then explain how the service demands for the technique are computed. The modeling technique for the read/large write/small write workload is explained after presenting the techniques for read-only, large write-only, and small write-only workloads. For notational simplicity and clarity, we assume that the read-only, write-only, and small write-only workloads are single class workloads (i.e., all jobs are identical), and that the read/large write/small write workload consists of three job classes, namely the read class, the large write class, and the small write class.

5.1 Read Workloads

All read requests are first submitted to the array cache. A request whose data are found in the cache (cache hit) is serviced by the cache, so the request is not submitted to the disks. A request whose data are not found in the cache (cache miss) is forwarded to the controller which then submits read subrequests to the appropriate disks. Let $hit\_probability$ refer to the probability that a read request is serviced entirely from the cache. The value of $hit\_probability$ depends on the cache size, the cache policies, and the workload characteristics. The computation of $hit\_probability$ is presented later in this section.

The performance model of a disk array under a synchronous read workload is derived from the MVA technique for fork-join networks. The network contains M identical interactive jobs. Each job spends $terminal\_think\_time$ time units at its terminal before issuing a request to the disk-array. The network contains $K' = 2$ service centers - service center 1 is the cache which is modeled by a single queue and service center 2 is the controller which is modeled by a fork-join queue. The disks represent service centers of this fork-join queue. A read request to the disk array is first submitted to the cache. With probability $hit\_probability$, the request's data are found in the cache and with probability $miss\_probability$ $(= 1 - hit\_probability)$ the request's data have to be read from the disks. Referring to Figure 3, $prob\_cache\_signals\_completion = hit\_probability$ for read requests. The pseudo-code of the performance technique for disk arrays under synchronous read workloads is presented in Figure 4. The input parameters to the technique are:

$$S_1 = cache\_service\_time \quad V_1 = hit\_probability(M) \quad D_1 = V_1 * S_1$$
$$S_2 = controller\_read\_service\_time(M) \quad V_2 = miss\_probability(M) \quad D_2 = V_2 * S_2$$
$$s_2 = disk\_read\_service\_time(M) \quad v_2 = disk\_read\_subrequest\_rate * miss\_probability(M) \quad d_2 = v_2 * s_2$$

We now describe the computation of $cache\_service\_time$, $controller\_read\_service\_time$, $hit\_probability$ ($miss\_probability$), $disk\_read\_service\_time$, and $disk\_read\_subrequest\_rate$. The effects of request coalescing, array redundancy, workload sequentiality, workload streams, disk scheduling, and read-ahead caching at the array cache and disk caches are incorporated into these computations.

5.1.1 Cache Hit Probability

The cache stores data that is accessed frequently (temporal locality) or that might be accessed in the near future (spatial locality). A read request causes data to be read into the cache from the disk. Most disk-array caches support *read-ahead* - data following the requested data are also read into the cache. A cache *hit* occurs if the requested read data are found in the cache. In this case the service time of the read request is equal to the cache service time. A cache *miss* (or *fault*) occurs if the requested data are not in the cache and have to be read from the disks into the cache. In this case the service time of the request is equal to the sum of cache service time and the maximum of the disk service times.

The cache hit probability approximation given here is an extension of cache probability approximation given in [20] and [22]. 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's data had been referenced in the past and the data are still in the cache. Assume that the read-ahead data set and the re-reference data set are disjoint. The hit probability is then given by

$$hit\_probability = read\_ahead\_hit\_probability + re\_reference\_hit\_probability.$$

We now describe how the $read\_ahead\_hit\_probability$ and the $re\_reference\_hit\_probability$ are computed. Assume that read-ahead requests are only performed on cache misses. This is a reasonable assumption since disks are

```
Q_1(0) = 0    Q_2(0) = 0
for m = 1 to M  do
begin
        R_1(m) = D_1 * [1 + Q_1(m−1)]
        R_2(m) = D_2 + d_2 * Q_2(m−1)

                                          m
        X_array(m) =    ---------------------------------------------
                          terminal_think_time + R_1(m) + R_2(m)

        Q_1(m) = X_array(m) * R_1(m)
        Q_2(m) = X_array(m) * R_2(m)
 end
 print "Disk Array Throughput ="  X_array(M)
 print "Disk Array Response Time ="  R_1(M) + R_2(M)
 print "Disk Array Queue Length ="  Q_1(M) + Q_2(M)
```

Figure 4: Technique for computing performance measures of a disk array under a synchronous read (large write) workload

typically not accessed on a cache hit. Let $read\_ahead\_size$ represent the amount of additional data bytes that are read into the cache from the disks when a cache miss occurs. The size of a read request submitted to the controller is given by:

$$controller\_read\_request\_size = request\_size + read\_ahead\_size$$

The mean number of read requests that can be serviced from the cache by one access to the controller can be estimated by

$$num\_requests\_serviced\_by\_one\_controller\_access = 1 + \left\lceil locality\_fraction \times \frac{controller\_read\_request\_size \; - \; request\_size}{request\_size} \right\rceil$$

where $locality\_fraction$, the fraction of sequential requests in a workload, is computed from the workload parameters run_count (the number of sequential requests in a run) and $random\_count$ (the number of non-sequential requests between two runs).

$$locality\_fraction = \begin{cases} 0 & \text{when } run\_count = 1 \\ \frac{run\_count}{run\_count + random\_count} & \text{when } run\_count > 1. \end{cases}$$

Since read-ahead is only performed on cache misses, there will be typically one cache miss for every $num\_requests\_\ serviced\_by\_one\_controller\_access$. Then the read-ahead miss probability can be approximated by

$$read\_ahead\_miss\_probability \approx 1/num\_requests\_serviced\_by\_one\_controller\_access$$

The read-ahead hit probability is given by

$$read\_ahead\_hit\_probability = 1 - read\_ahead\_miss\_probability.$$

If a data block has been referenced in the past, then there is a non-zero probability that the data block is still in the cache. This $re\_reference\_hit\_probability$ is approximated from the probability that the number of bytes accessed by the I/O workload between two accesses to the same block is less than the cache size. The $re\_reference\_hit\_probability$ is given as part of the I/O workload description and is a function of the read-cache size.

5.1.2 Cache Service Time

Let $cache\_service\_time$ represent the mean time to service a request from the cache. The cache service time is equal to the sum of cache search time, cache read or write overhead, and bus transfer time. Since cache search time and cache read/write overhead are very small compared to the bus transfer time, these values can be included in the bus transfer time. The bus transfer time is the time taken to transmit request data between the array cache and the CPU and is dependent on the data size and the speed of the interconnect.

$$cache\_service\_time = request\_size / bus\_transfer\_rate$$

5.1.3 Controller Read Service Time

The controller service time is equal to the time it takes to read a request's data from the disks. The controller receives I/O *requests* (of size $controller\_read\_request\_size = request\_size + read\_ahead\_size$) from the cache and outputs I/O *subrequests* to the disks. The controller signals request service completion only after all the subrequests of a request are serviced by the disks. That is, the controller service time is equal to the sum of the maximum of all the subrequests' service time and the cache service time. The service time of a subrequest is equal to the disk service time which is the sum of positioning time and transfer time. Of these, transfer time is a constant across disks whereas positioning time varies across disks. Kim and Tantawi [11] show that positioning time can be approximated by a normal distribution, and the mean of the maximum of x positioning times is given by $position\_time + standard\_deviation\_position\_time * \sqrt{2\log(x)}$ Thus,

$$controller\_read\_service\_time \approx disk\_read\_service\_time + standard\_deviation\_position\_time * \sqrt{2\log(num\_subrequests\_per\_controller\_read\_request)} + cache\_service\_time$$

At higher multiprogramming levels, the effect of parallelism at the disks increases. Since the MVA technique computes the longest queue length at the disks (and hence, the longest wait time), the controller service time can now be written as

$$controller\_read\_service\_time \approx disk\_read\_service\_time + cache\_service\_time.$$

The number of subrequests per read request, $num\_subrequests\_per\_controller\_read\_request$, depends on the request size, the stripe unit size, the RAID configuration, and the amount of access coalescing. In RAID 1/0 each disk has a mirror copy, so a read subrequest can be submitted to either one disk or divided between a disk and its mirror. In addition, to increase efficiency, subrequests that access contiguous data can be coalesced into a single subrequest by the controller. Based on the request size, the computation of $num\_subrequests\_per\_controller\_read\_request$ is divided into the following three cases:

Large reads: These refer to read requests of size $\geq LU\_disks * stripe\_unit\_size$. The number of subrequests per request will be greater than *LU_disks*. Since each datum is duplicated on two disks, several contiguous subrequests to the same disk can be distributed between the disk and its mirror. Experimental results indicate that the FC-60 distributes subrequests across all the disks when the request size is greater than $LU\_disks * stripe\_unit\_size$. In addition, the FC-60 performs access level coalescing of large requests that access two or more stripe units on the same disk. The controller combines all such accesses to a disk into a single subrequest. Thus,

$$num\_subrequests\_per\_controller\_read\_request = LU\_disks$$

Medium reads: These refer to read requests of size $> LU\_disks/2 * stripe\_unit\_size$ and $< LU\_disk * stripe\_unit\_size$. In this case, experimental results indicate that the FC-60 does not distribute subrequests between a disk and its mirror. Thus,

$$num\_subrequests\_per\_controller\_read\_request = LU\_disks/2$$

Small reads: These refer to read requests of size $\leq LU\_disks/2 * stripe\_unit\_size$. The number of subrequests per request is less than or equal to $LU\_disks/2$ and the maximum amount of data read from each disk is equal to the size of one stripe unit.

$$num\_subrequests\_per\_controller\_read\_request = \left\lceil \frac{controller\_read\_request\_size}{stripe\_unit\_size} \right\rceil$$

5.1.4 Disk Read Subrequest Rate

The disk read subrequest rate refers to the number of subrequests per request submitted to a disk. Since the FC-60 performs access level coalescing of subrequests of requests that access multiple stripe units on a disk, the disk read subrequest rate is always less than or equal to 1.

$$disk\_read\_subrequest\_rate = num\_subrequests\_per\_controller\_read\_request / LU\_disks$$

5.1.5 Disk Read Service Time

The disk read service time depends on the read subrequest size, the disk cache hit probability, and the speed of the disks. The disk subrequest size is given by

$$read\_subrequest\_size = controller\_read\_request\_size / num\_subrequests\_per\_controller\_read\_request$$

The disks have caches that typically implement smart prefetching based on detection of the workload sequentiality and the utilization of the disks. The computation of disk cache hit probability is similar to the computation of array cache hit probability. In addition, disks implement smart scheduling policies that reduce the mean disk positioning time (= sum of seek time and rotational latency). The reduction of disk positioning time is dependent on the sequentiality of the disk workload and the length of the disk queues. The sequentiality of the disk workload depends on the sequentiality of the I/O workload submitted to the disk array (represented by $locality\_fraction$), the number of workload streams submitted to a disk array (represented by M), and the probability that a disk receives a subrequest from an arriving request (represented by $disk\_subrequest\_rate$). We approximate the disk workload sequentiality as follows:

$$disk\_sequential\_fraction(M) \approx \begin{cases} locality\_fraction & \text{when } M = 1 \\ \frac{disk\_sequential\_fraction(M-1)}{1+(M-1)*(disk\_subrequest\_rate/2)} & \text{when } M > 1. \end{cases}$$

The computation of $locality\_fraction$ of the disk array workload is presented in the earlier subsection on array cache hit probability. The mean disk positioning time is given by:

$$\begin{aligned} disk\_read\_position\_time = {} & disk\_sequential\_fraction * disk\_seq\_position\_time + \\ & (1 - disk\_sequential\_fraction) * disk\_avg\_read\_position\_time \end{aligned}$$

The mean disk service time is equal to the sum of mean disk positioning time and mean disk transfer time. The disk service time parameter values are given as part of the device specification (and also validated against disk measurements).

$$disk\_transfer\_time = read\_subrequest\_size / disk\_transfer\_rate$$

$$disk\_read\_service\_time = disk\_read\_position\_time + disk\_transfer\_time$$

5.2 Large Write Workloads

We have determined experimentally that the FC-60 handles large write requests of size greater than two stripes differently from small write requests (of size at most equal to two stripes). Large write requests are written directly to the disks bypassing the cache. Small write requests are first written to the cache and completion is signaled as soon as the write-to-cache is completed. Referring to the queuing model of Figure 3, $prob\_cache\_signals\_completion = 0$ for large write requests, whereas $prob\_cache\_signals\_completion = 1$ for small write requests.

The performance algorithm for large write requests, is similar to the performance algorithm for read requests and is presented in Figure 4. The input parameters to the technique are:

$$\begin{aligned} & S_1 = cache\_service\_time \quad V_1 = 0 \quad D_1 = V_1 * S_1 \\ & S_2 = controller\_write\_service\_time(M) \quad V_2 = 1 \quad D_2 = V_2 * S_2 \\ & s_2 = disk\_write\_service\_time(M) \quad v_2 = 1 \quad d_2 = v_2 * s_2 \end{aligned}$$

For these large requests (larger than two full stripes) that are written directly to the disks, the computations of $controller\_write\_service\_time$ and $disk\_write\_service\_time$ are similar to the computation of these measures for read requests.

$$num\_subrequests\_per\_controller\_write\_request = LU\_disks$$

Due to mirroring, all write subrequests are written across two disks, which gives

$$write\_subrequest\_size = \frac{2\ *\ controller\_write\_request\_size}{LU\_disks}$$

5.3 Small Write Workloads

For small write requests, $prob\_cache\_signals\_completion = 1$ since all write requests are first written to the cache. The cache can hold at most N dirty data blocks. If the cache contains less than N dirty data blocks, then the write-to-cache happens immediately. If the cache contains N dirty data blocks, then the write-to-cache occurs only after enough dirty data are written to disk. It is hard to model the behavior of small write requests since the service time of a request at the cache depends on the amount of dirty data in the cache. As long as the cache contains less than N dirty data blocks, the write-to-cache service time of a request is equal to the $cache\_service\_time$. Once the cache contains N dirty data blocks, a request's service time is equal to the time it takes to write enough dirty data blocks to disks. The amount of dirty data in the cache at a point in time is determined by several factors such as the value of N, the workload characteristics (like request size and request arrival rate), and the destaging (*i.e.*, writing dirty data to disk) policies used in the FC-60. Experimental data indicates that the cache in the FC-60 writes dirty data blocks to the disks if there are more than $write\_cache\_low\_water\_mark$ dirty blocks, but not otherwise. Thus, $write\_cache\_low\_water\_mark$ is the maximum dirty blocks that can be held in cache without triggering disk writes.

It is difficult to use the MVA technique to model the behavior of the disk array under a workload of small write requests, since the service time of the cache varies depending on the amount of dirty data in the cache. If we assume that the cache is infinitely large (*i.e.,* $N \rightarrow \infty$), then all write requests are written to cache immediately and the service time of the cache equals $cache\_service\_time$. In this case, the MVA technique can be used to compute the throughput of the disk array. Now consider the case of a finite cache (*i.e.*, $0 < N < \infty$). Here, the cache is modeled as a simple birth-death process [5]. The corresponding Markov chain has states $\{0, 1, ..., N\}$ where state i corresponds to i dirty blocks, and N is the total size of the write cache. The "death" rate in state i is zero for $0 \leq i \leq write\_cache\_low\_water\_mark$, since the cache does not write dirty blocks to disk. In states i where $write\_cache\_low\_water\_mark < i \leq N$, the death rate is μ, which is the rate at which blocks can be written out to the disks. The "birth" rate, λ, in state $i < N$ can be approximated by the maximum throughput of the unbounded cache disk array system for a given synchronous write workload. The rate λ is computed using the MVA technique (see Figure 5). For state N, the cache is full, and hence the birth rate is zero. Based on this, the steady state probability distribution of dirty blocks in the cache is

$$\begin{aligned} P_i &= \frac{(1-\rho) * \rho^{(i\ -\ write\_cache\_low\_water\_mark)}}{(1-\rho^K)} \quad \text{for } write\_cache\_low\_water\_mark \leq i \leq N \\ &= 0 \quad \text{otherwise} \end{aligned}$$

where $K = N - write\_cache\_low\_water\_mark$ and $\rho = \lambda/\mu$. Note that this model is similar to a M/M/1/K queuing model. The rate λ is the maximum throughput of an infinitely large cache disk array system for a given synchronous write workload. The state N represents the state when the cache is full. So, $\lambda * (1\ -\ P_N)$ represents the throughput of a disk array with a bounded cache under a synchronous write workload.

$$X\_array = \lambda * (1 - P_N).$$

The "death" rate μ is the rate at which blocks are written to the disks and is calculated as the maximum rate at which data can be written out from the cache to the disks, assuming that a stripe unit is the maximum amount of data that can be written to a disk at a time. The effects of workload sequentiality and access level coalescing of large requests that straddle multiple stripe units on the same disk are indirectly incorporated into the death rate by appropriately adjusting the disk positioning time, as shown below. Since all subrequests to a disk must also be written to the disk's copy,

$$\mu = LU\_disks/(2\ *\ disk\_write\_service\_time)$$

.

The complete algorithm for performance prediction of disk arrays with write-back caching is presented in Figure 5. Since the disk array service rate is computed for subrequests of size at most equal to the stripe unit size, the disk array arrival rate of requests is also expressed in similar terms. So, for requests larger than a stripe unit, the cache service

```
S_1  = cache_service_time  V_1  = 1  D_1  = V_1  * S_1

                                    request_size
M = M * maximum( 1,  ------------------ )
                                  stripe_unit_size

Q_1(0) = 0
for m = 1 to M do
begin
      R_1(m) = D_1 * [1 + Q_1(m−1)]
                                           m
      X_max(m) = -------------------------------------
                          terminal_think_time + R_1(m)

      Q_1(m) = X(m) * R_1(m)
end
lambda = X_max(M)
                 LU_disks
mu = ---------------------------------
          2 * disk_write_service_time

            lambda
rho =  ----------
              mu
              1 − rho
P_0 = ----------------
            1 − rho^[K+1]

P_N = P_0 * rho^K
X_array = lambda * (1 − P_N) * maximum(1, request_size / stripe_unit_size)
print "Disk Array Throughput =" X_array
```

Figure 5: Technique for computing performance measures of a disk array with write-back caching, under a synchronous write workload

time is computed assuming that the request size is reduced to the stripe unit size. Correspondingly, the value of M (the number of jobs generating requests) is increased by the factor $request\_size/stripe\_unit\_size$. The size of a subrequest to a disk is

$$write\_subrequest\_size = \min(stripe\_unit\_size, request\_size)$$

The controller starts writing cache data to disks once the cache reaches its write cache low water mark. At this point, there are several outstanding requests to be written out per disk, so the controller can use efficient disk scheduling mechanisms that reduce the mean seek time. Hence, even for random workloads, the mean disk positioning time for such writes is lower than the average disk positioning time. For sequential workloads, the mean disk positioning time can be lowered further. We model the mean disk service time as follows:

$$disk\_write\_position\_time \approx \frac{disk\_high\_traffic\_seek\_time}{model\_run\_count} + \frac{disk\_rotation\_time}{2}$$

where

$$model\_run\_count(M) = \begin{cases} run\_count & \text{when } M = 1 \\ \min(run\_count, stripe\_unit\_size/write\_subrequest\_size) & \text{when } M > 1. \end{cases}$$

The value of *disk_high_traffic_seek_time* is obtained by measuring the average positioning time for the disk with a large number of simultaneous random requests.

5.4 Read + Large Write + Small Write Workloads

We now consider workloads that contain a mix of read, large write, and small write requests. The workload intensity is denoted by $\vec{M} = (M_r, M_{lw}, M_{sw})$, where M_r, M_{lw}, and M_{sw} are the read, large write, and small write population sizes. The performance of the disk array under read and large write workloads is computed using the MVA technique for closed networks while the performance of the disk array under small write workloads is computed using the $M/M/1/K$ technique for open networks with a maximum queue size. We now explain the performance technique for disk arrays under a mixed workload containing reads, large writes, and small writes.

1. First compute the performance measures of reads and large writes by assuming that all small write requests are written directly to disks. The performance of reads and large writes are not affected (or minimally affected) by this assumption since small writes are eventually written to the disks in a write-back caching scheme. The cache and/or disk service times of reads, large writes, and small writes are computed as shown in Sections 5.1, 5.2, and 5.3. Compute performance measures of the read and large-write job classes for this disk array with three closed job classes using the multiple class MVA technique.

2. Next compute the performance measures of small writes for the write-back cache disk array using the $M/M/1/K$ queuing technique by eliminating the read and large writes classes. In order to eliminate these classes, the birth rate λ and the death rate μ of the single class $M/M/1/K$ queue must reflect the effect of reads and large writes on the performance of small writes. The rate λ in state $i < N$ is approximated by $X_{sw}^{\infty cache}$, the maximum throughput of small writes for an unbounded write-cache disk array under a read/large write/small write class workload. Compute $X_{sw}^{\infty cache}$ using the multiple class MVA technique with three job classes.

$$\lambda = X_{sw}^{\infty cache}(\vec{M})$$

The rate μ for the $M/M/1/K$ queue is the rate at which blocks are written from the cache to the disks when the disk array is under a read/large write/small write class workload. This rate can be approximated by X_{sw}^{0cache}, the throughput of small writes for a write-through cache disk array (i.e., all small writes are written directly to the disks as in Step 1) when the submitted workload consists of reads, large writes, and small writes. Compute X_{sw}^{0cache} using the multiple class MVA technique with three job classes.

$$\mu = X_{sw}^{0cache}(\vec{M})$$

Note that the performance technique for workloads containing read/large writes, read/small writes, and large writes/small writes are special cases of the above technique.

6 Empirical Validation

The accuracy of the disk array model is tested by comparing the model's throughput results against measured throughput values from the FC-60 array for several synthetic workloads. The disk array throughput is the performance parameter used to validate our model. The response time and queue length can be calculated from the throughput using Little's Law. We use a FC-60 array with Cheetah ST173404LC disks, two controllers, and 256 MB of cache in each controller. All our experiments are run on one FC-60 LU containing 6 disks. Each disk in the LU is on a separate SCSI bus. The LU is configured at RAID 1/0 and uses a stripe unit size of 16 KB. Two FibreChannel links are used to connect the FC-60 array to a Brocade Silkworm 2800 switch. Table 1 (in Section 3) lists the disk array parameters relevant to the model.

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. We use 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 number of jobs generating requests range from 1 to 12 and the terminal think time ranges from 0 ms to 300 ms. The workloads are generated using a synthetic load generator and we collect a trace of all I/O activity at the device driver level on the host. The trace contains I/O submission and completion times, the logical addresses and sizes of requests, and all other relevant workload information. A trace analysis system is used to compute the throughput and other useful statistics from the workload.

Figures 6, 7, 8, and 9, and 10 present the model throughput and the actual (experimental) throughput for read and write workloads for a range of request sizes, multiprogramming levels, think times, and sequentiality degrees. The throughput predictions for 300 ms think time match experimental results almost exactly (within 1%) and are not plotted. Table 3 shows the average percentage errors in the model predictions for each of the experiment sets.

Workload	Think time(ms)	Average error(%)
Random read (run count = 1)	0	8.34
	10	6.72
	30	5.30
	100	2.66
	300	0.56
Random write (run count = 1)	0	6.15
	10	9.21
	30	7.21
	100	3.69
	300	1.37
Sequential read (run count = 64)	0	17.42
	10	13.98
	30	7.62
	100	4.14
	300	0.71
Sequential write (run count = 64)	0	12.26
	10	12.31
	30	10.12
	100	3.66
	300	1.62
Read (run count=16)	0	17.50
Write (run count=16)	0	10.10

Table 3: Average prediction error in the model for each experiment

Across all workloads and experiment sets, the accuracy of the model is within 9.1% on average. For random reads, the model is accurate within 4.7%, on average, and for random writes, the model is accurate within 5.5%, on average. For sequential workloads, the model predictions are, on average, within 8.8% and 8.0% for read and write workloads, respectively. The accuracy of the models improves as the think time increases.

The model matches the measured values fairly well except for sequential workloads, particularly reads at multi-programming level 1. The errors for sequential workloads indicate that the effect of the caching policies, the disk scheduling policies, and the other optimizations have not been accurately incorporated into cache and disk service time computations. These errors are reasonable [9], considering that we do not know the proprietary details of the caching policies, access coalescing, and disk scheduling on the FC-60, and it is difficult to deduce these details. The disk array throughput is the performance parameter used to validate our model. The response time and queue length can be calculated from the throughput using Little's Law. The errors in the response time prediction would be close to the errors in the throughput prediction when there is no think time.

Table 4 compares the average, maximum, and minimum percentage errors in our model predictions against the Lee and Katz model [12] predictions. The Lee/Katz model is the only prior model of disk arrays under synchronous I/O workloads. Their model does not consider redundancy, so we extended their disk service time model to include redundancy. Since the Lee/Katz model only considers random workloads with no think time, the table lists errors for this workload type. Workloads with think time or sequential request streams cannot be compared. For random read workloads, our model prediction errors are similar to the Lee/Katz model prediction errors. For large request sizes ($> 48K$), the average error in the Lee/Katz model is more than the error in our model. In particular, for a request size of 256K, the Lee/Katz model has an average error of 26.93% while our model has an average error of 14.28% (and a maximum error of 18.90%). In the case of random write workloads, our model predictions outperform the Lee/Katz model predictions by a large margin. This is reasonable since the Lee/Katz model treats reads and writes similarly and does not consider the effects of write-back caching.

Request Type	Request Size	Model	Average(%)	Max(%)	Min(%)
Random Read (No Think Time)	4K	Lee/Katz	7.83	13.23	1.07
		Ours	7.85	13.25	1.04
	8K	Lee/Katz	7.82	13.19	1.70
		Ours	7.85	13.24	1.64
	16K	Lee/Katz	7.15	12.25	1.20
		Ours	7.21	12.35	1.31
	32K	Lee/Katz	9.49	24.81	1.19
		Ours	8.85	14.30	0.01
	48K	Lee/Katz	11.71	35.13	1.07
		Ours	12.05	18.33	4.61
	64K	Lee/Katz	12.84	45.34	0.85
		Ours	9.71	18.55	3.23
	128K	Lee/Katz	10.74	58.89	0.42
		Ours	8.43	17.06	2.64
	256K	Lee/Katz	26.93	78.54	15.70
		Ours	14.28	18.91	0.38
Random Write (No Think Time)	4K	Lee/Katz	55.48	77.81	47.06
		Ours	2.85	4.62	0.38
	8K	Lee/Katz	52.07	75.83	42.82
		Ours	3.98	10.74	1.54
	16K	Lee/Katz	49.41	73.42	40.12
		Ours	7.38	18.79	3.57
	32K	Lee/Katz	37.83	50.91	34.62
		Ours	3.74	10.74	1.54
	48K	Lee/Katz	32.44	34.05	25.77
		Ours	2.28	10.56	0.14
	64K	Lee/Katz	10.21	27.10	1.24
		Ours	14.13	43.10	1.53
	128K	Lee/Katz	28.04	82.62	6.04
		Ours	8.32	18.20	0.51
	256K	Lee/Katz	36.42	117.31	14.82
		Ours	6.54	17.75	0.60

Table 4: Prediction errors of our model compared against prediction errors of the Lee/Katz model

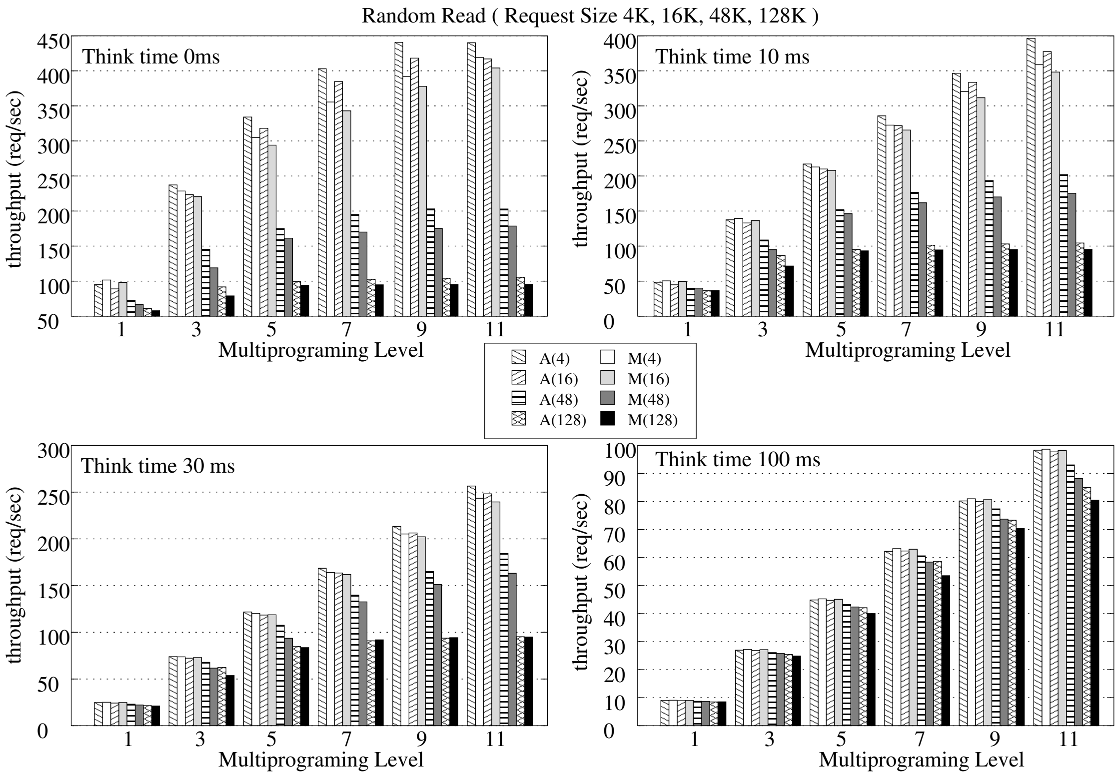

Figure 6: Model M($request\_size$) versus actual A($request\_size$) throughput results for random read workloads

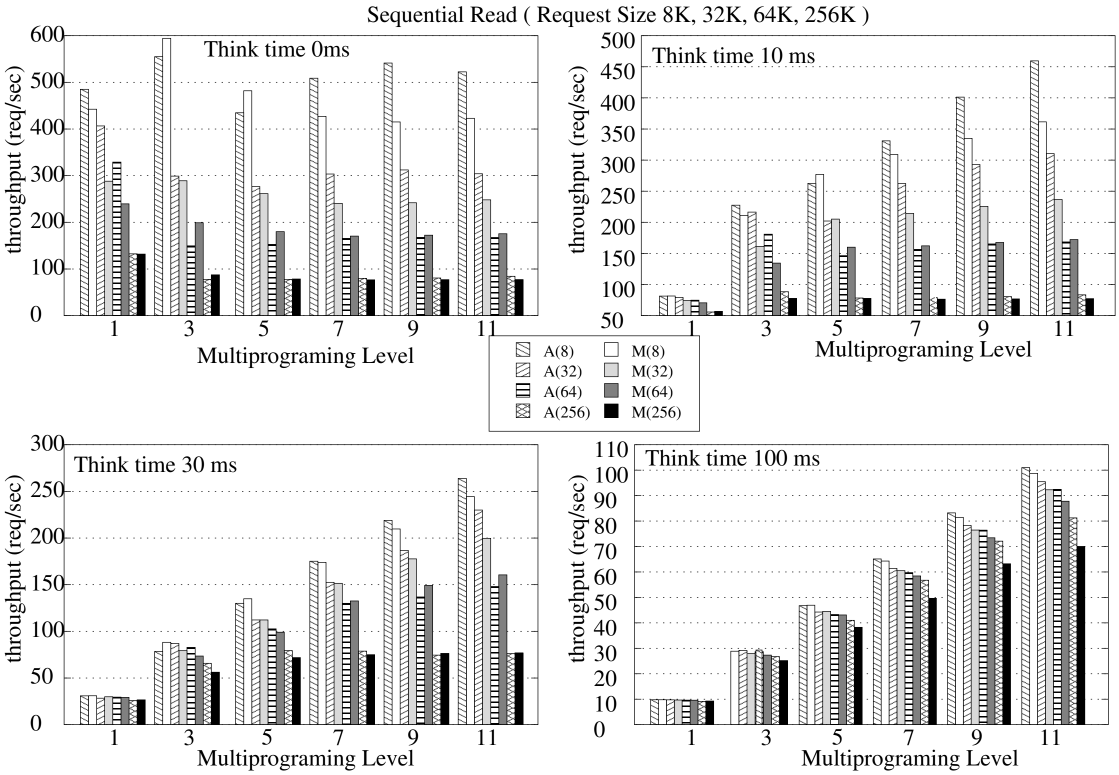

Figure 7: Model M($request\_size$) versus actual A($request\_size$) throughput results for sequential read workloads

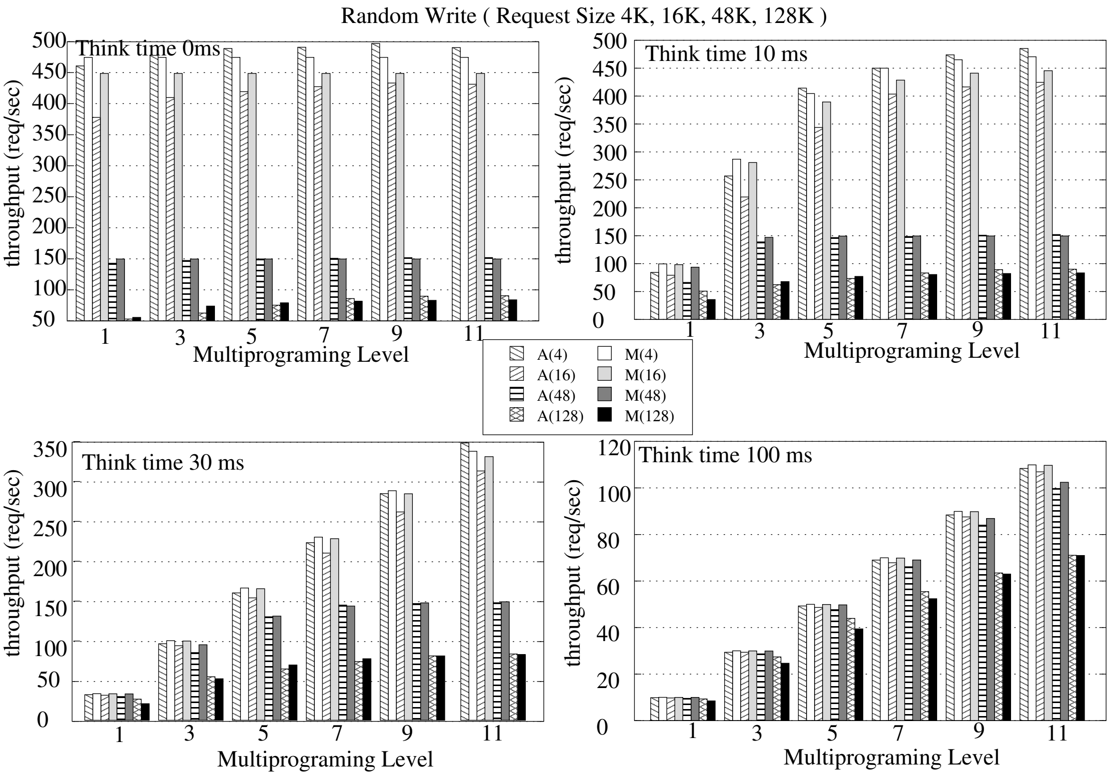

Figure 8: Model M($request\_size$) versus actual A($request\_size$) throughput results for random write workloads

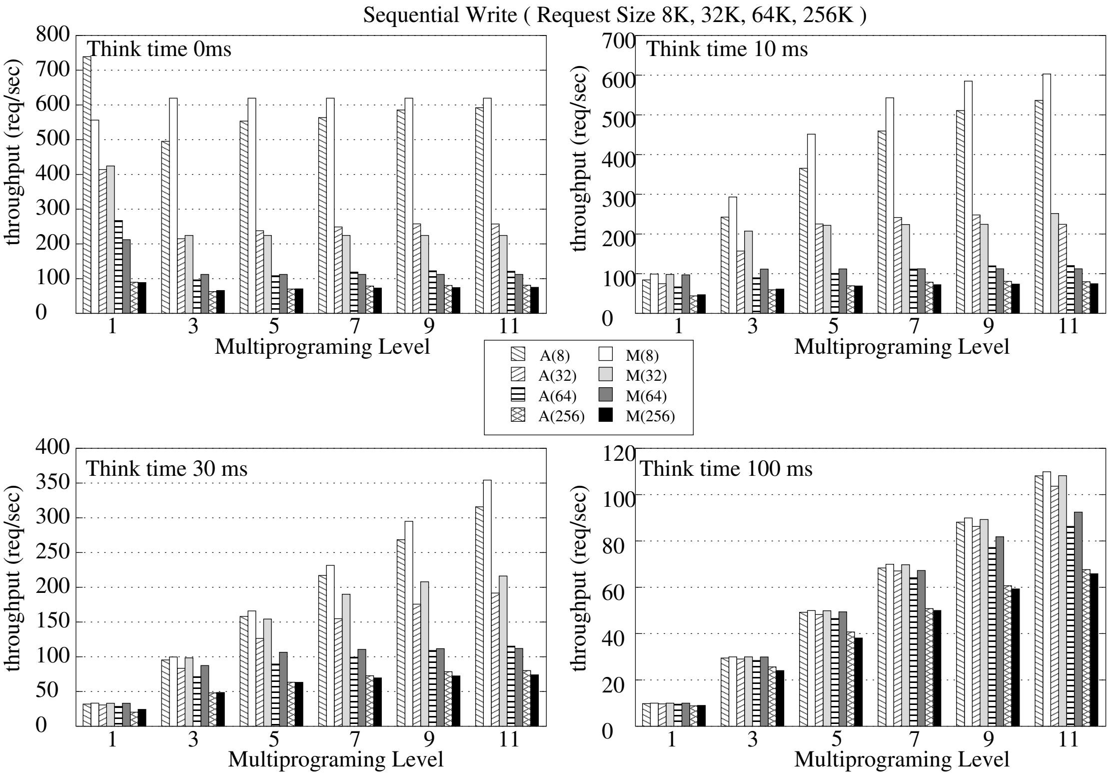

Figure 9: Model M($request\_size$) versus actual A($request\_size$) throughput results for sequential write workloads

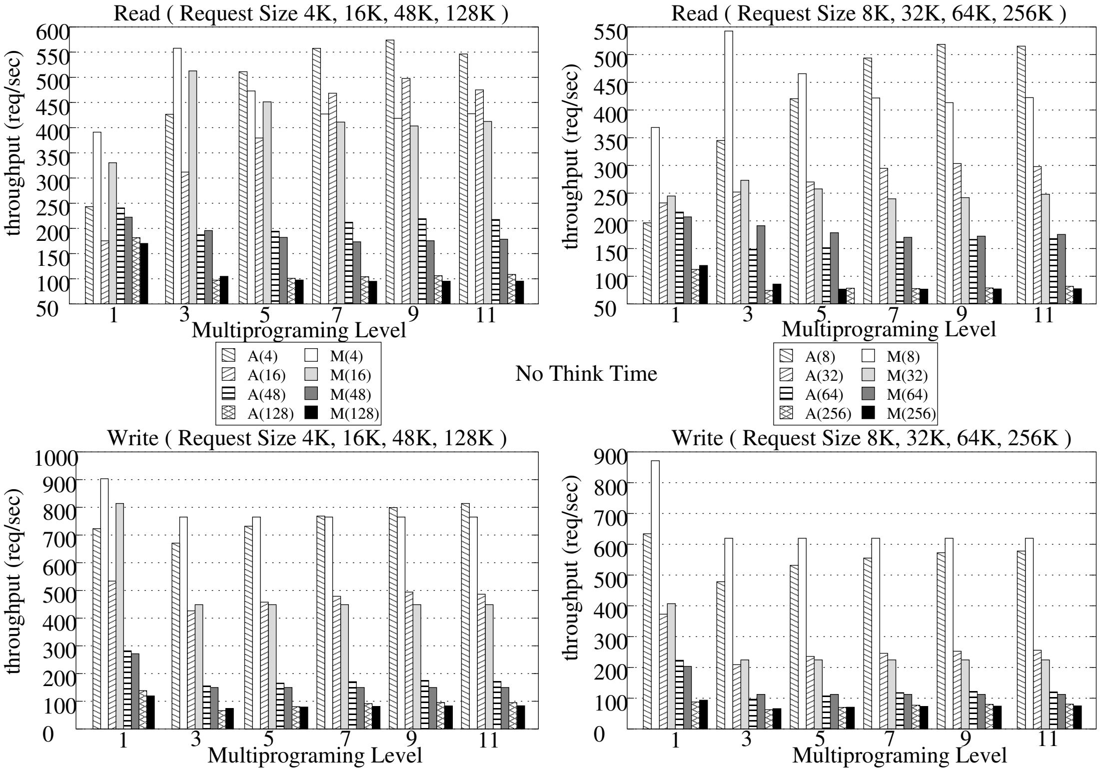

Figure 10: Model M($request\_size$) versus actual A($request\_size$) throughput results for read-only and write-only workloads with run count = 16

7 Conclusions

We have developed an analytical performance model of disk arrays under synchronous I/O workloads, which make up the majority of the total I/O workload on computer systems. This is the first such published model (that we know) which incorporates the effects of write-back caching, controller overhead and disk queueing in the disk array, as well as the effects of multiprogramming level, think-time, and sequentiality in the workload. It is only the second to validate the predictions against a commercial disk array, the HP FC-60. Another contribution of this work is the simplicity and computational efficiency of our model of realistic disk arrays. The performance of the disk array is computed using a combination of the $M/M/1/K$ queueing technique and our extension to the MVA technique for fork-join networks

The model predictions for read-only and write-only workloads are validated against measurements from a mid-size disk array for a variety of synthetic workloads. The model estimates are found to be, on average, within 9.1% of the actual throughput values, though there are outlier points where the error margins are much higher (23.0%). These errors are reasonable for validations against real disk arrays [22] since the details of caching, destaging policies, and optimizations are not revealed by the disk array's manufacturers. The errors in our model are substantially smaller than those for the only other published disk array model with synchronous workloads [12].

The paper studies a RAID 1/0 layout; however, it can be extended to model other layouts by replacing the disk-system service time sub-model. In future work, we plan to validate our read-write model against synthetic and real workloads, improve our models of caching and sequential read/write behavior, extend the model to include workloads containing mixtures of synchronous and asynchronous workloads, and model other RAID configurations.

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):483–518, November 2001.

[2] 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.

[3] 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.

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

[5] R.B. Cooper. *Introduction to Queueing Theory*. Mercury Press/Fairchild Publications, MD, 1990.

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

[7] 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.

[8] Hewlett-Packard Company. *HP SureStore E Disk Array FC60 User's guide*, December 2000. Pub. No. A5277-90001.

[9] 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.

[10] K. Keeton, G. Alvarez, 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 (held in conjunction with HPCA-7)*. IEEE, January 2001.

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

[12] 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.

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

[14] 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.

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

[16] 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.

[17] 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.

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

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

[20] E. Shriver, A. Merchant, and J. Wilkes. An analytic behavior model for disk drives with readahead caches and request reordering. In *Proc. SIGMETRICS*, pages 182–91, June 1998.

[21] A. Thomasian and J. Menon. RAID5 performance with distributed sparing. *IEEE Transactions on Parallel and Distributed Systems*, 8(6):640–57, June 1997.

[22] M. Uysal, G. Alvarez, and A. Merchant. A modular, analytical model for modern disk arrays. In *MASCOTS*, pages 183–193, August 2001.

[23] E. Varki. Mean value technique for closed fork-join networks. In *Proc. SIGMETRICS*, pages 103–112, May 1999.

[24] E. Varki. Response time analysis of parallel computer and storage systems. *IEEE Transactions on Parallel and Distributed Systems*, 12(11):1146–61, November 2001.

[25] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. In *Proc 15th ACM Symposium on Operating Systems Principles (SOSP),* Copper Mountain Resort, CO, volume 29:5, pages 96–108, December 1995.