RADIR: Lock-free and Wait-free Bandwidth Allocation Models for Solid State Drives

Pooja Aggarwal†, Giridhar Yasa‡, and Smruti R. Sarangi†
†*Department of Computer Science & Engineering, Indian Institute of Technology, New Delhi, India*
‡ *Netapp India Pvt. Ltd, EGL Software Park, Domlur, Bangalore, India*
e-mail: † *{pooja.aggarwal,srsarangi}@cse.iitd.ac.in,* ‡ *giridhar.yasa@netapp.com*

***Abstract*—Novel applications such as micro-blogging and algorithmic trading typically place a very high load on the underlying storage system. They are characterized by a stream of very short requests, and thus they require a very high I/O throughput. The traditional solution for supporting such applications is to use an array of hard disks. With the advent of solid state drives (SSDs), storage vendors are increasingly preferring them because their I/O throughput can scale up to a million IOPS (I/O operations per second). In this paper, we design a family of algorithms, RADIR, to schedule requests for such systems. Our algorithms are lock-free/wait-free, linearizable, and take the characteristics of requests into account such as the deadlines, request sizes, dependences, and the amount of available redundancy in RAID configurations. We perform simulations with workloads derived from traces provided by Microsoft and demonstrate a scheduling throughput of 900K IOPS on a 64 thread Intel server. Our algorithms are 2-3 orders of magnitude faster than the versions that use locks. We show detailed results for the effect of deadlines, request sizes, and the effect of RAID levels on the quality of the schedule.**

***Keywords*-Flash bandwidth, resource allocation model, lock-free, wait-free, I/O scheduling**

I. Introduction

At the moment, there are two disruptive changes happening in the world of storage systems and software. The first is that there is an increasing shift towards SSDs (solid state drives) that are significantly faster than conventional hard drives. The second is that a new class of applications such as micro-blogging and algorithmic trading have emerged, which have very small requests yet require very high throughput. In other words, very soon most storage systems will have extremely high throughput storage devices, and applications to utilize them. Hence, it is necessary to work on vital pieces of the storage platform today such that we can support the software and hardware of tomorrow. In response to these requirements, researchers have proposed a set of platforms that are explicitly tailored for such applications, which are characterized by a fast stream of small requests.

Some of the notable platforms in this space are Sparrow [1], Dremel [2], Spark [3], and Impala [4]. Additionally, research by Ousterhout et. al. [1, 5] has indicated that by dividing a large Facebook job into many small fragments, it is possible to improve the response time by 5.2X using the Spark [3] framework. This gives us one more additional incentive to consider the problem of designing fast storage platforms for supporting such novel workloads with very short jobs on ultra-high throughput SSDs. In specific, we focus on designing a high throughput scheduler for such platforms in this paper.

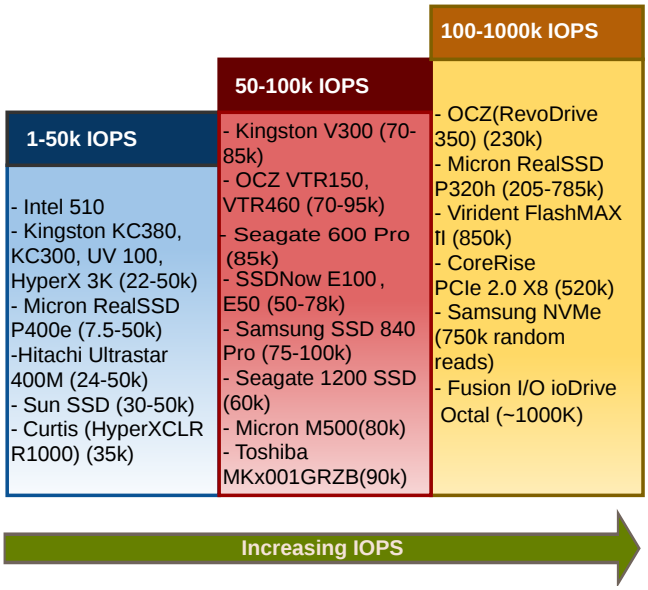

Figure 1: I/O throughput of solid state drives

Till now, the scheduler was not a bottleneck because the system as a whole was not constrained by the scheduling throughput or latency. The I/O throughput was a bigger bottleneck. Modern hard disks can at the most service 10-20k IOPS (I/O operations per second) for 100% sequential reads and it drops down to 100 IOPS if there are random reads and writes. Conventional lock based schedulers suffice in this case. In fact, we shall show in Section V that conventional schedulers can scale till 200k IOPS. However, enterprise class SSDs and rack mount SSDs (with DRAM based caches) are an order of magnitude faster than hard drives. Some of the fastest SSD drives being designed today such as Fusion I/O's ioDrive Octal can support up to a million IOPS for small requests. Figure 1 summarizes the I/O throughput of different devices. We can observe a

spectrum of devices that have throughputs between 500k IOPS to 1 million IOPS. Supporting these devices is well beyond the means of conventional scheduling techniques (refer to Section V), and thus a more scalable solution is required.

First, let us try to understand why conventional solutions fail to scale beyond 200k IOPS in our testbed. Most conventional solutions typically export a $schedule$ function that is used by multiple threads to schedule their requests for a set of storage drives. There is some state associated with each device that records the time intervals in the future that it is already reserved for. The scheduler needs to find a free interval, and schedule the current request. We term this state as the *reservation record*. In current approaches the code to modify the reservation record of a storage drive is encapsulated in a critical section. This is because the data structure to store the reservation record has hitherto between sequential. Now, the main sources of delay are the time it takes to acquire the lock, and the time it takes to modify the reservation record. Even with highly scalable locks such as the MCS or Fast-Path lock [6], the critical path of the process of acquiring the lock scales superlinearly [6]. Secondly, the reservation record cannot be modified concurrently, and thus it is a sequential bottleneck.

A. Parallel Slot Schedulers

To genuinely satisfy our requirements of having a scheduler that schedules roughly a million requests per second, it is necessary to remove the lock, and parallelize the process of updating the reservation record. There is some related work in this area [7], which proposes to use slot schedulers that allow multiple threads to schedule requests in parallel. A slot scheduler divides time into discrete quanta called *slots* and reserves a certain a number of slots for each request. Having the notion of slots helps convey an implicit notion of timing across the threads, and helps threads effectively synchronize with each other. However, slot schedulers such as the non-blocking scheduler proposed by Aggarwal et. al. [7] do not take the characteristics of requests into account such as their duration, dependencies, deadlines, and amount of available redundancy. This is because any single thread has limited visibility of all the tasks in the system, and because of the rapid entry and exit of tasks, the system is very dynamic and rapidly changing. It is difficult to optimize the schedule in this scenario.

In this paper, we try to balance both the goals (time and quality of the schedule) by designing a parallel non-blocking slot scheduler, which explicitly takes the nature of tasks into account. We extend the parallel non-blocking slot scheduling algorithms developed by Aggarwal et. al. [7] and evaluate their feasibility for SSDs with different RAID configurations.

A slot scheduler divides time into discrete quanta called *slots*. We represent the time associated with a single port

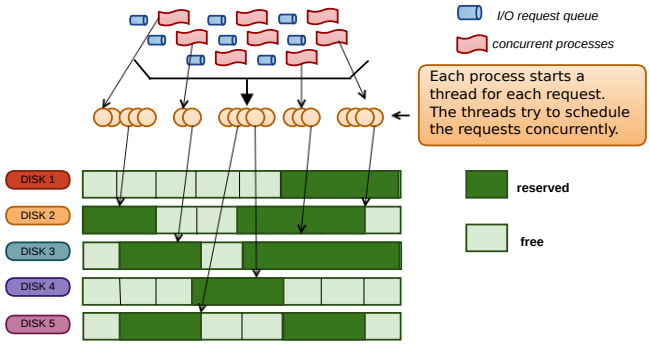

Figure 2: The Resource allocation model

storage device as a 1-dimensional array, where each cell represents a *time-slot*. A cell can either be *reserved* or *free*. Each process that wishes to access the storage device first tries to reserve the bandwidth for a specified number of slots. Figure 2, generalizes the idea to a matrix of slots (also known as the Ousterhout matrix [8]). Here, the columns represent slots, and the rows represent ports of a set of storage devices. To realize a request, we typically need to reserve a sequence of contiguous slots.

We propose, implement, and evaluate a request scheduling layer for storage devices called RADIR. It provides a parallel and linearizable data structure for the reservation record, $diskRevRec$, which provides lock-free and wait-free guarantees. Each flash drive has its dedicated $diskRevRec$ data structure to record its requirements. Each request specifies the time for which it wants to access the resource in terms of the number of slots, the starting slot number and the time by which the request should be scheduled (slot-deadline). In the case of RAID 1, a request can be fulfilled by reserving slots in a set of redundant disks. We demonstrate that we can sustain a request throughput of 800-900K IOPS with commercial I/O traces.

II. Background and Related Work

A. Scheduling

Most resource management schemes that optimally allocate resources, and try to minimize the makespan, energy consumed, or average execution time are known to be NP-complete [9]. Nevertheless, there are very effective heuristics for scheduling problems in general, and for storage drives in specific [10, 11]. Several scheduling methods are known for optimizing data retrieval and storage operations in disk drives [12, 13, 14]. Most of these sequential methods [15] rely on one request queue per disk, which uses locks. Secondly, the time to process all the jobs and their requirements, and to compute an optimal schedule is prohibitive for very short jobs [5]. Sadly, these I/O schedulers are not effective for SSDs because they have a different model of operation. Traditional I/O schedulers optimized for HDDs have been

retuned for SSDs [16, 17, 18, 19]. These schedulers are generally designed to reduce random write operations and not focus on the fairness and throughput of the scheduler. SSDs are organized into multiple banks that can be independently accessed [20]. It is possible to have a large amount of parallelization within the drive itself. We need a high throughput scheduler which considers the internal parallelism of the flash drives also.

For scheduling lots of tiny tasks, the reservation records have to export a very simple interface to threads such that we can minimize the time it takes to schedule a request. Slot based schedulers fit this category, and have been widely discussed in the storage literature [17, 21, 22, 23, 24, 25, 26]. By making time a discrete quantity, we can make the design of the scheduler more efficient and elegant (similar to the way paging helps in managing virtual memory). The authors of [21](Argon) argue that having the notion of slots also helps in maintaining fairness across the threads. Unfortunately, similar to paging in virtual memory, slot scheduling also suffers from the problem of internal fragmentation. However, if the size of the slot is chosen appropriately [21], this can be minimized.

B. Non-blocking Slot Schedulers

Aggarwal et al. [7] were the first to propose a parallel slot scheduler that did not use locks. It was shown to be at least 10X faster than conventional solutions. Our aim in this paper is to make it storage aware. Let us briefly introduce their contributions in this section.

In their slot scheduler, multiple threads place requests for a given number of contiguous slots. Each request specifies the starting time slot ($startSlot$), and the number of *slots* it requires. The *lock-free* implementation guarantees that regardless of the contention caused by concurrent processes, always at least one request makes progress. However, there is a risk of starvation. The *wait-free* implementation prohibits starvation because it guarantees that a request completes its computation in a finite number of steps, regardless of the behavior of other requests. This is achieved by making threads with newer requests *help* older requests to complete.

Their basic approach is as follows. At the outset, a thread t tries to temporarily reserve the first slot that is closest to $startSlot$. Next, t continues to temporarily reserve the required number of slots in consecutive columns of the slot array. When the thread t finishes doing so, it changes the state of the request, and makes the reservation permanent. After this operation is over, the thread collates and returns the list of slots allotted. It is possible that while the request is temporarily reserving slots it might get cancelled by another request, or it might not be able to reserve slots because they have already been committed to another request. In this case, the request needs to wind up its state, and start anew.

In spite of all of this complexity, they guarantee the strictest form of consistency in concurrent systems – linearizability [27]. A method call is said to be *linearizable* if it appears to execute instantaneously between its invocation and completion. We extend this slot scheduler and make it aware of dependencies, deadlines, request lengths, priorities, and RAID levels. Additionally, we provide generic mechanisms to take into account the page/block copy operations in SSD drives.

III. Bandwidth Allocation Model for I/O Scheduling

A. Request Parameters

Our aim is to schedule higher priority requests, among a set of concurrent requests. We propose a set of heuristics, which are considered in the scheduling process. The heuristics consider: (i) deadline of a request, (ii) request size, (iii) number of requests depending on a given request, (iv) request type and (v) preferential writes to blocks. Next, we discuss each of these heuristics and their role in dynamically updating the priorities of the requests placed by the various processes.

1) Deadline of a Request: There are many real time applications that have deadline constraints. For example, a military radar application may need to retrieve radar signatures of aircrafts from a storage device, or a stock market software may need to finish a trade by a certain deadline. In our scheduling logic, the concurrent requests are prioritized depending upon their deadlines similar to the Earliest Deadline First (EDF) [28] scheduling paradigm. Deadlines are specified in terms of time slots. We service a request with lower priority only if serving this request is not going to violate the deadline constraints of a higher priority request. However, if this is not the case, then the lower priority request is cancelled and scheduler later.

2) Request Size: The processing time of a request is equal to the number of slots it requires. This is dependent on two factors. The first is the amount of data that needs to be read or written. We sometimes may need to insert some extra time slots, called $dummySlots$, to take write amplification into account (copy-erase operations). In our algorithm, the variable $diskSlots$ is the sum of the $requestSize$ (original number of slots requested) and the number of $dummySlots$ required. If a request requires more than 24 slots for reading/writing, we classify this request as a *long request*. The rest are short requests. Short requests are preferred over long requests if doing so does not result in a long request missing its deadline. However, this can lead to starvation for the long request. To keep starvation in control, we introduce a ROLLBACKLIMIT to limit the number of times a request can be cancelled by other requests. Every time a request is cancelled by some other request, its priority is increased. Once the cancellation count for a request req reaches the ROLLBACKLIMIT, then other threads help to complete the request.

3) Dependent Requests: Sometimes, I/O accesses have dependences between them. We explicitly model the notion of a *request packet*, which consists of a set of requests arranged as a directed acyclic graph (DAG). For example, in the case of RAID 5, a write operation can be broken down into 4 requests: read original block, read parity block, write to the original block, and write to the parity block. Figure 3 shows the dependences.

4) Request Type: Flash writes are often substantially slower than reads and a reader may experience excessive slow-down, when other concurrent writes are present. Therefore most storage system prioritize reads over writes because reads are typically on the critical path. We can trivially model this by increasing the priority of reads. Again to avoid starvation, we gradually increase the priority of writes such that they will ultimately have a higher priority than reads.

5) Block Preferential Writes: In flash drives writing to a new block imposes significant penalty. A scheduler should aim at minimizing the number of writes to a new block [19]. This is achieved by giving higher priority to the requests that are in the same block as the previous request.

B. Redundant Disk Arrays

It is possible that the data is distributed across the drives as in the case of RAID levels. Let us consider a highly reliable system where $drive_1$, $drive_2$ and $drive_3$ are redundant (RAID 1). In such a scenario a read request can be served from multiple disks. We term such type of requests as *Snake Requests*. In this case a single request for process P_3 requesting for 7 time $slots$ can be scheduled as shown in Figure 4. In the case of a RAID 5 implementation, a write request needs to write its data to a particular block as well as update the parity block. Such a request needs to reserve time slots at two drives for the same duration. We term such type of requests as *thick requests*. As shown in Figure 4, drives 4, 5 and 6 are connected in a RAID 5 configuration. Process P_1 reserves slots for both drives 4 and 6 – one for writing the data and the other for updating the parity.

C. Dummy Slots

Data within the SSDs needs to be erased before the blocks can be used for new writes. As a result, most SSDs employ copy-on-write mechanisms and remap blocks through the Flash Translation Layer (FTL). FTL also performs garbage collection, which issues many reads and writes internally. To take into account these additional copy-erase operations for wear levelling and mitigating read disturbance, a request occasionally might need some extra time slots, called dummy slots.

D. Load Balancing

For drives with redundant configurations (RAID 1), we maintain the notion of a *load factor*. We try to balance the usage across the drives such that flash blocks take longer to wear out.

IV. Design And Implementation

A. Data Structures

We maintain the schedule of the various incoming requests in a 1 dimensional array, $diskRevRec$. Each cell in this array can be in one of the following states: VACANT, TRANSIENT and RESERVED.

The VACANT state means that the slot is free and a request can be scheduled in it. The TRANSIENT state refers to a temporary reservation made by some thread, and the RESERVED state indicates that the slot is permanently booked. Figure 5 shows the information saved in the slot for each state. The REQUEST array is an array of atomic object references indexed by the thread id. In the REQUEST array, each thread places its request to reserve flash bandwidth. Each index of REQUEST has the following fields: $requestType \rightarrow$ read or write, $startSlot \rightarrow$ starting slot number beyond which the request needs to be scheduled, $startAddr \rightarrow$ starting address of the block in the storage device, $slotDeadline \rightarrow$ the largest possible value of the starting slot, $reqSize \rightarrow$ request size, $depLisLen \rightarrow$ the number of requests depending on the request. The field, $iterationState$, maintains the current phase of the request. A request has broadly four phases which are: INIT, TEMP, PERM and FINISH (explained in Section IV-B).

The priority of the request is saved in the field, $priority$. $reservationPath$ and $diskNumber$ are used for the case of redundant drives. The slots currently reserved by a request are saved in the array, $reservationPath$. It has $diskSlots$ entries. In case there is only one flash drive, we need to only record the first slot reserved by a request, and thus we save it in $slotAllocated$. The rest of the time-slots that are reserved are the next consecutive $diskSlots\text{-}1$ slots. We use the variable, $diskSlots$ to represent the number of time-slots for which a drive is to be reserved for a particular request. Globally, we maintain the load factor $loadFactor$ and the last blocked written ($lastBlockAccessed$) of each drive in an array. Apart from this we also maintain the list of redundant drives.

B. Reserve Operation

In the lock-free algorithm, a thread iteratively reserves slots for itself. Whenever a thread collides with any other thread, it either decides to overwrite that thread's booking or help that thread in completing its request. Whereas, in the wait-free algorithm, a thread begins its operation by first helping all the slow threads, which have placed their request and then it proceeds with its own request [7]. We mainly focus on the case when there is no redundancy and the schedule is generated for each drive separately. The flash bandwidth is reserved by placing a new request in the INIT phase. In this phase a thread (t) temporarily reserves a slot in the $diskRevRec$ array with the help of the function $reserveSlots$ (explained in Section A-1). On successfully reserving the first slot, the request moves to

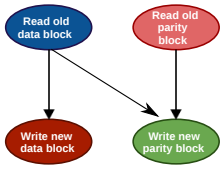

Figure 3: Request dependence graph

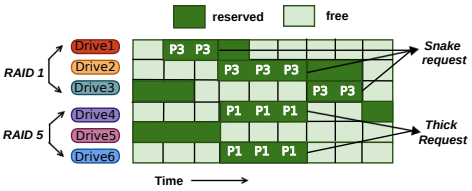

Figure 4: Various request types

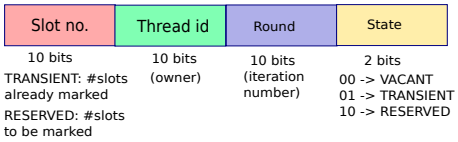

Figure 5: Slot state

```
class Request {
 final int    requestType,
 final int    startSlot,
 final int    threadId,
 final int    request_size,
 final int    slotDeadline,
 final int    depList,
 final int    startAddr,
 AtomicLong   slotAllocated,
 AtomicLong   iterationState,
 AtomicInteger priority,
 AtomicLong [] reservationPath,
 AtomicInteger diskNumber
}
```

Figure 6: The Request class

the TEMP phase. In the TEMP phase, the remaining slots are reserved temporarily. Once it finishes doing so, the request moves to the PERM phase. In this phase the reservation made by the thread (or by some other helper on its behalf) is made permanent. For lack of space, we do not show the code for this operation and its proof of linearizability (interested readers can look at the associated technical report [29]). Next, we explain how the slots are reserved by a thread and the methods for assigning and updating priorities.

1) Reserve Slots: We start by explaining how a slot in the $diskRevRec$ array is reserved by a thread, which currently has highest priority among all the contending threads. This method accepts four parameters - request(req) of a thread t, the current slot to reserve $currSlot$, current round $round$ of the request and the phase of the request $reqState$. In case their are redundant drives, first we find the drive ($minDisk$) which requires the minimum number of dummy slots and also has least load factor. This is done with the help of the method $getMinDiskSlot$. Once we have the desired drive, depending upon the status of the slot ($currSlot$) in that drive we execute the corresponding switch-case statement. $round$ indicates the iteration of a request. It is used to synchronize all helpers of a request [7]. If the slot is in the VACANT state, we try to temporarily reserve the slot and change the state of the slot from VACANT to TRANSIENT (Line 9) and update the load factor of the drive $minDisk$.

Next, we discuss the case when the state of the slot is TRANSIENT. It indicates that some thread has temporarily reserved the $currSlot$ slot. If the thread id saved in the slot is the same as that of the request req (Line 16), we simply return and read the phase of the request again. Otherwise, the slot is temporarily reserved by some other thread for another request, $otherReq$. Now, we have two requests req and $otherReq$ contending for the same slot $currSlot$ in the drive $minDisk$. If the priority of the request req is higher than $otherReq$, request req wins the contention and will overwrite the slot after cancelling the request $otherReq$ i.e changing the state of the request $otherReq$ to CANCEL atomically (Lines 22 - 29). Request req will help request $otherReq$ in case req has a lower priority. We increment the priority of a request to avoid starvation (Line 35). Requests priorities are decided in the method $getHighPriorityRequest()$ as explained in Section IV-C.

Let us now discuss the case where the slot is found to be in the RESERVED state. In the INIT phase of the request, a request tries to search for the next vacant slot (Line 48). The search terminates when either a slot is successfully reserved or the request hits its slot deadline (Line 56). In the TEMP phase, we return CANCEL (Line 50). On receiving the result of the function $reseveSlots$ as CANCEL, the request moves to the CANCEL phase. Lastly, it is possible that some other helper has reserved the slot for request req (Line 42). In this case the thread refreshes and reads the phase of the request req again.

Algorithm 1: Reserve Slots

```
1:  function reserveSlots(request,currSlot, round, reqState)
2:    for i ∈ [currSlot, req.slotDeadline] do
3:      minDisk ← getMinDiskSlot(request, i)
4:      diskRevRec ← diskArray[minDisk]
5:      slotState ← getSlotState(diskRevRec.get(i))
6:      (threadid,round1,state) ←
        unpackSlot(disk − RevRec. get(i))
7:      switch (slotState)
8:      case VACANT :
9:        res ← diskRevRec.CAS(currSlot,
          packTransientState(request), VACANT)
10:       updateDiskLoad(minDisk)
11:       if res = TRUE then
12:         return (SUCCESS, currSlot)
13:       end if
14:       break
15:     case TRANSIENT :
16:       if threadid = req.threadid then
17:         /* slotState = MYTRANSIENT */
18:         return (RETRY, null)
19:       else
```

```
20:            otherReq ← REQUEST.get(threadid)
21:            res ← getHighPriorityRequest(req,otherReq,i,minDisk)
22:            if res = req then
23:              /* preempt lower priority request */
24:              if cancelReq(otherReq) then
25:                oldValue ← packTransientState( threadid, round1, state)
26:                newValue ← packTransientState(req. threadid, round, TRANSIENT)
27:                res1 ← diskRevRec.CAS(currSlot, oldValue, newValue)
28:                if res1 = TRUE then
29:                  return (SUCCESS, currSlot)
30:                end if
31:                break
32:              end if
33:            else
34:              /* res = HELP */
35:              reserveDiskBandwidth(otherReq)
36:              /* increase priority to avoid starvation */
37:              req.priority.getAndIncrement()
38:            end if
39:          end if
40:          break
41:        case RESERVED :
42:          if threadid = req.threadid then
43:            /* slot reserved on req's behalf */
44:            return (RETRY, null)
45:          else
46:            if req.iterationState = INIT then
47:              slotMove ← getReserveSlot(diskRevRec. get(i))
48:              i ← i + slotMove
49:            else
50:              return (CANCEL, null)
51:            end if
52:          end if
53:          break
54:        end switch
55:      end for
56:      return (FAIL, req.slotDeadline)
57:  end function
```

C. Request Priority Rule

Request issuance order is the same as the arrival order but the order in which the requests are serviced depends on their relative priorities. As discussed in Section III-A, there are various parameters based on which the priorities of the requests are updated to permute the order of service of incoming I/O requests in order to generate a near optimal schedule. Each parameter is assigned a weight, we examine each parameter along with their weights to calculate an overall priority for a request. These weights can be customized by the user. As shown in Figure 7, contending requests pass through various checks such as the deadline, request type, request size, and dependency list length checkers. At each level, some of the requests get filtered out and the request with the highest priority wins the contention and is ready to be scheduled.

Now, we give a detailed description of the $getHighPriorityRequest$ method (Lines 58 - 96). We have given the deadline parameter the highest weightage in deciding the relative priorities of the requests so that the requests with lesser weights for other parameters do not miss their deadlines. One can change this, depending upon the target applications. If the given requests are within a deadline range, then we look at other heuristics to decide request priorities. Otherwise we schedule a request, which has its deadline closest to the slot under consideration (Line 61).

Each request can have some default priority associated with itself. Whenever a request helps some other request, we increase its priority by 1 to keep starvation in control. Once the difference between the priorities of the contending requests reaches a threshold value (PRIORITY_THRESHOLD) we choose the request with higher priority to schedule next. Read requests are given higher priority over write requests to minimize read-blocked-by-write situations in concurrent workloads. We check the request type of the contending requests and if one of them is a read and the rest are writes, we give higher priority to the read request. If both the requests are writes then we check the blocks these requests wish to access. We prefer writes to the blocks that were accessed in the recent past. This is because in DRAM backed drives, the writes get absorbed by the DRAM.

Next we consider the request size (Line 86). Whether to give high priority to small sized requests or to large sized requests depends on the user. We assign higher priority to smaller requests by default, in order to avoid unnecessary delays due to large requests, and to maximize request throughput. Let $reqA$ and $reqB$ be two contending requests. We calculate the dummy slots required for each request based on the block it wishes to access, or garbage collection requirements in SSDs. The request with the lower value of $diskSlots$ is assigned a higher priority. If the difference between disk time slots required by each request lies within a specified range, then we give higher priority to the request which has reserved more slots till now. This is done in order to reduce the wastage of useful work done by a thread so far (Line 95).

```
58:  function getHighPriorityRequest(reqA , reqB, slot, diskId)
59:    /* priority takes into account some default priority if specified and age based priority */
60:    if abs(reqA.slotDeadline - slot)- abs(reqB.slotDeadline - slot) > deadlineRange then
61:      return reqA.slotDeadline < reqB.slotDeadline ? reqA:reqB
62:    end if
63:    if abs(reqA.priority - reqB.priority) > PRIORITY_THRESHOLD then
64:      return reqA.priority > reqB.priority ? reqA:reqB
65:    end if
66:    if reqA.requestType = READ ∧ reqB.requestType = WRITE then
```

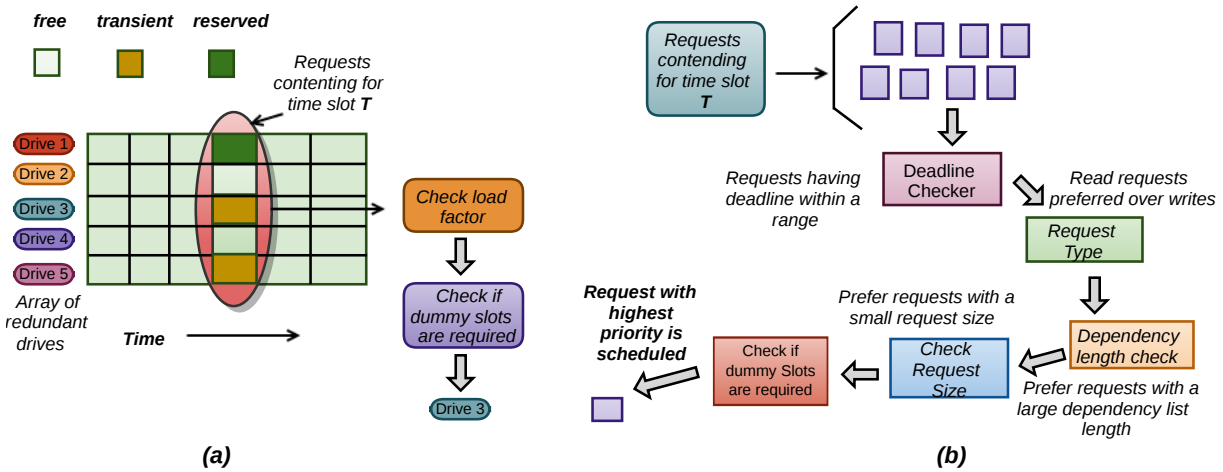

Figure 7: Request to drive slot mapping

```
67:        /* write request is within its deadline range so we can
           schedule it later */
68:        return reqA /* request with READ access is given
           preference */
69:     end if
70:     /* both are write requests */
71:     if reqA.requestType = WRITE ∧ reqB.requestType =
        WRITE then
72:        blockA ← getBlock(reqA.startAddr)
73:        blockB ← getBlock(reqB.startAddr)
74:        if blockA = lastAccessedBlock ∧ blockB ≠
           lastAccessedBlock then
75:           return reqA
76:        else
77:           if blockB = lastAccessedBlock then
78:              return reqB
79:           end if
80:        end if
81:     end if
82:     dummySlotA ← calculateDummySlots(reqA)
83:     dummySlotB ← calculateDummySlots(reqB)
84:     diskSlotA ← reqA.requestSize + dummySlotA
85:     diskSlotB ← reqB.requestSize + dummySlotB
86:     if abs(reqA.diskSlotA - reqB.diskSlotB) >
        diskSlotRange then
87:        return diskSlotA < diskSlotB ? reqA : reqB
88:     end if
89:     if abs(reqA.depLength - reqB.depLength) >
        depLengthRange then
90:        return reqA.depLength > reqB.depLength ?
           reqA:reqB
91:     end if
92:     /* find request which has booked more slots */
93:     slotA ← getSlotBooked(reqA)
94:     slotB ← getSlotBooked(reqB)
95:     return slotA > slotB ? reqA:reqB
96: end function
```

D. Disk Redundancy

In this section, we look at the scenario where we have an array of redundant drives. In this case we need to find a disk, which is the most suitable to schedule an incoming request. As shown in Figure 7, we have considered two parameters - $dummySlots$ and $loadfactor$ to select a drive among an array of drives. The way in which a flash drive is selected is captured in the $getMinDiskSlot$ method (Lines 87 to 99). It is possible that a read request is scheduled partly on one drive and partly on the other. Recall the $snake\,request$ type as explained in Section III-B.

```
87: function getMinDiskSlot(request, slotId)
88:    minDisk ← MAX
89:    redDiskArr ← findRedundantDisk(request.disk
       Number)
90:    for i ∈ [0, redDiskArr.length] do
91:       dummySlot ← getDummySlots(redDiskArr[i],
          request)
92:       loadFactor ← getLoad(diskId)
93:       /* select a disk with minimum load factor and dummy
          slots */
94:       if minDisk.dummySlot > dummySlot ∧
          minDisk.loadFactor > loadFactor then
95:          minDisk ← redDiskArr[i]
96:       end if
97:    end for
98:    return minDisk
99: end function
end
```

V. Performance Evaluation

This section examines the performance of RADIR. We performed all our experiments on a Dell PowerEdge R820 server running the Ubuntu Linux 12.10 operating system with the generic 3.5.0-17 kernel. It is a hyper-threaded four socket, 64 bit machine. Each socket has eight 2.20GHz Intel Xeon CPUs, 16 MB L2 cache, and 64 GB main memory. All our algorithms are written in Java 6 using Sun OpenJDK 1.6.0_27.

Let us now describe our experimental methodology. Let there be T threads in the system. We execute the experiments till the fastest thread completes N requests. At this point,

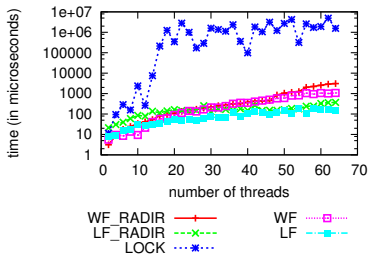

Figure 8: Time/request

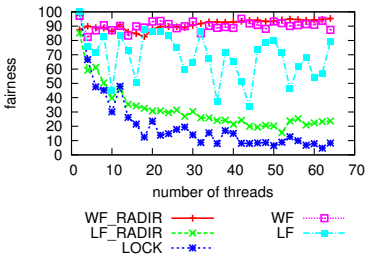

Figure 9: Fairness

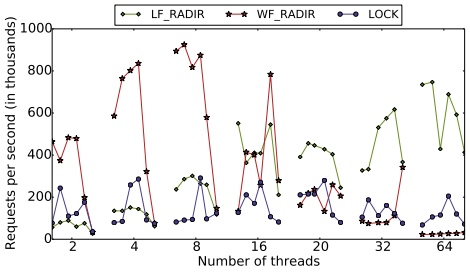

Figure 10: Request throughput

let the total number of requests completed (by all the threads) be K. We define three quantities – time per request $time$, *fairness* (frn) and $delay$. $time$ means average time taken to schedule a request. The *fairness* is defined as $frn = K/(N \times T)$. If the value of our fairness metric is equal to 1, then all the threads complete the same number of requests – N. The lower is the fairness, more is the discrepancy in performance across the threads. $delay$ signifies the deviation/difference in the starting slot requested ($startSlot$) and actual starting slot alloted ($slotAllocated$) to a request. It is measured in terms of time-slots. Higher priority requests are scheduled as early as possible so that they have less delay (or nearly zero delay) as compared to low priority requests. Delay in scheduling occurs when there is more than one request contending for the same slot. Only one request gets the slots and other requests need to try again for subsequent slots.

A. Storage Device Model

We used synthetic workloads based on the Microsoft SNIA I/O traces [30] for evaluating the performance of our scheduling algorithms. We run each experiment for 10 times, and report the mean values. We have modeled the Seagate 600 SSD drive (see Table I for its characteristics). The request size governs the number of slots required to access the device since most SSDs support non-sequential accesses very well. We consider the slot size as 100μs. This can be decided based on the underlying storage device.

Drive Capacity	480 GByte
Burst Transfer Rate	600MB/s(Max)
Flash Memory Type	NAND MLC
Page Size	8192 Bytes
Average read latency	158μs
Average write latency	125μs

Table I: SSD characteristics (Seagate 600 Pro SSD)

B. Performance of RADIR

We evaluated the performance of our lock-free (LF_RADIR) and wait-free (WF_RADIR) RADIR algorithms by comparing them with the lock-free (LF) and wait-free (WF) slot scheduling algorithms proposed by Aggarwal and Sarangi [7], and also with an algorithm that uses locks ($LOCK$). We set $N = 100,000$, and vary the number of threads from 1 to 64.

Figure 8 and 9 show the results for the $time$ and $fairness$ of each of the algorithms. After adding I/O awareness, the performance of our algorithms is comparable to the LF and WF algorithms. The time per operation of (WF_RADIR) is 2-3 times more than WF for more than 56 threads. All the algorithms are at least 3 orders of magnitude faster than the version with locks. We can also see that the wait free algorithms maintain a high degree of fairness ($\approx$ 90%), and the fairness for lock free algorithms varies from 20-80%. The algorithm with locks has very low fairness ($\approx$ 10% for $>$ 32 threads).

Next, we study the throughput of our slot scheduler by varying the average time between the arrival of two requests from 0 to 5μs at intervals of 1 μs. Figure 10 shows the results. Note that the six points for each line segment correspond to an average inter-request arrival time of 0, 1, 2, 3, 4, and 5 μs respectively. The first noteworthy trend is that LF_RADIR scales as the number of threads increases from 1 to 64. WF_RADIR has the highest throughput till 16 threads, and then it becomes inferior to the LF_RADIR algorithm. With 8 threads its throughput touches the 800-900K IOPS range, which is sufficient for todays fastest SSDs with large DRAM based caches (refer to Figure 1). The throughput of WF_RADIR does not scale beyond 16 threads because ensuring fairness becomes a very onerous task. A lot of computational bandwidth is wasted helping slower tasks, and thus throughput suffers. In comparison, the LF_RADIR keeps on getting better. Lastly, it is important to note that the algorithm with locks does not support a throughput that is more than 200-300K IOPS, and has a lower throughput than at least one of our algorithms by 5-10X.

C. Impact of Heuristics

We study the impact of each parameter on time, fairness and delay with the number of threads set to 64.

1) Sensitivity: Slot Deadline: We start by investigating the impact of the slot deadline. Each request is associated with an expiration time. Read requests have a lower

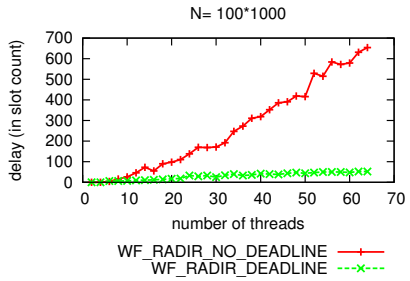

Figure 11: Request scheduling delay

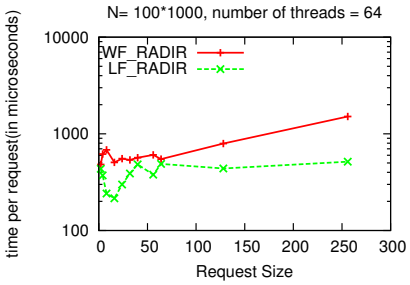

Figure 12: $time$ with varying request size

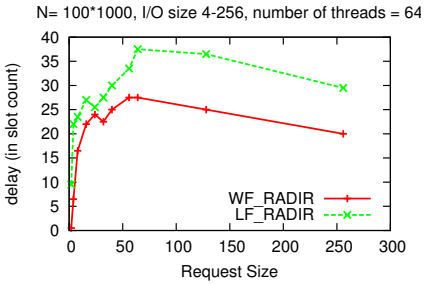

Figure 13: $delay$ with varying request size

expiration time as compared to write requests. We set a deadline of 60 slots for read requests, and 600 slots for write requests (follow the standard 1:10 thumb rule for reads and writes). When there is no deadline to schedule a request, the delay per request for WF_RADIR is as high as 600-800 time slots since there is no bound on when to schedule the request. Other parameters also play a role in deciding the request priority. This results in some requests getting scheduled as early as possible and others suffering from huge delays. Whereas, when deadlines are considered, the $delay$ is within 50-60 time slots as shown in Figure 11. From this experiment, we can conclude that the request deadline plays an important role in deciding the request priority, and requests should have associated deadlines in our system.

2) Sensitivity : Request Size: We vary the request size from 4 slots to 256 slots. We observe that when large requests get interleaved with small requests, $time$ and $delay$ get affected. The time per request for the LF_RADIR algorithm increases from 35-120μs for small requests (< 24 slots) to 300-520μs for long requests (24-256 slots). For WF_RADIR, the time varies from 140-175μs to 600-1520μs for 64 threads as shown in Figure 12. As the request size increases, the number of slots to be reserved per request also increases, which leads to more contention resulting in more request cancellations. Therefore, the time per request increases when there are long requests.

Figure 13 shows the impact of the request size on $delay$. Recall that $time$ is measured in seconds, and represents the wall clock time for scheduling a request, and $delay$ is simulated time (measured in the number of slots). As the request size increases from 2-64, the delay increases from 1-27 for (LF_RADIR) and 8-38 for (WF_RADIR). However, the delay for request sizes beyond 64 decreases since a lesser number of requests get scheduled due to high contention.

3) Sensitivity: I/O Intensity: The term *I/O intensity* refers to the duty cycle of the scheduling threads. It is defined as the average time a thread spends in scheduling requests divided by the total time. The total time includes *idle time* in which the thread does not do anything. We simulate idle time by inserting dummy computations with a known execution time. This experiment is to evaluate the quality of the schedule with varying I/O intensity. We assume a situation in which all the threads contend for the same set of slots (have proximate starting slots).

Figure 14 shows that for an I/O intensity less than 50%, the $delay$ is within 8 time-slots. The delay in scheduling the request increases with an increase in I/O intensity. As the I/O intensity increases to 90% the $delay$ rises to roughly 30 slots. There is no significant difference between the WF_RADIR and LF_RADIR algorithms.

4) Sensitivity: Dependency Length: The dependency length of a request (req) represents the number of requests depending on the given request, req. The impact of the dependency length is only visible in the case of $fairness$ of LF_RADIR. Figure 15 shows that the $fairness$ of LF_RADIR drops to 25-15% for 64 threads as the dependency length increases from 32 to 64. Requests with large values of the dependency length have higher priorities and are scheduled first, and this affects fairness negatively. This indicates that it is advisable to assign a lesser weightage to the dependency length if fairness across threads is desired.

We conclude our analysis by evaluating the performance with a redundant array of drives (RAID 1). Figure 16 shows the results. $WF\_RADIR(1)$, $WF\_RADIR(2)$ and $WF\_RADIR(3)$ correspond to a system with 1, 2 and 3 drives (latter two are in RAID 1). We use a similar terminology for LF_RADIR. As the number of drives increases to 3, the time to schedule a request decreases by 2x in the case of WF_RADIR since the contention per drive decreases and our algorithm can handle $snake$ requests very well. Whereas, in the case of LF_RADIR there is no significant change in the time per request. This is because in the lock free algorithm, only a few threads make rapid progress, and additional redundancy does not benefit them.

VI. Conclusion

In this paper, we proposed a highly scalable set of algorithms for scheduling I/O requests on SSDs. Our algorithms can incorporate a wide variety of request and device characteristics. We showed how different request parameters such

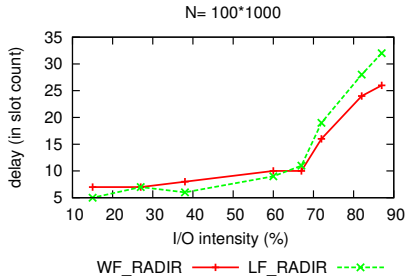

Figure 14: Impact of I/O intensity on *delay*

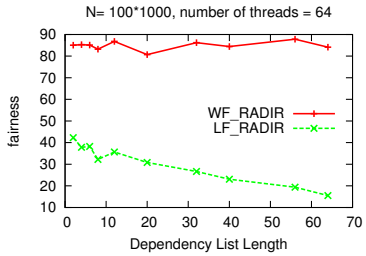

Figure 15: $fairness$ with varying dependency length

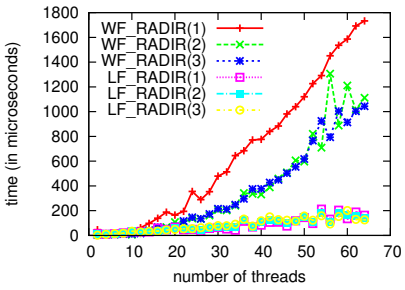

Figure 16: $time$ with redundant disks

as the slot deadline and request size play a significant role in setting request priorities. On a 64 threaded machine, we can support a scheduling throughput of 800-900K IOPS, which is sufficient for the fastest storage devices to be released in the near future. Our wait-free algorithm (WF_RADIR) has a fairness of $\approx 90\%$ as the threads are multiplexed at the level of individual requests, thus preventing jitter. The results show that our RADIR scheduler is at least 3 orders of magnitude faster than the version with locks.

REFERENCES

[1] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica, "Sparrow: distributed, low latency scheduling," in *ACM Symposium on Operating Systems Principles*, 2013.

[2] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, 2010.

[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012.

[4] M. Kornacker and J. Erickson, "Cloudera impala: real-time queries in apache hadoop, for real," 2012.

[5] K. Ousterhout, A. Panda, J. Rosen, S. Venkataraman, R. Xin, S. Ratnasamy, S. Shenker, and I. Stoica, "The case for tiny tasks in compute clusters," in *HotOS*, 2013.

[6] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[7] P. Aggarwal and S. R. Sarangi, "Lock-free and wait-free slot scheduling algorithms," in *IPDPS*, 2013.

[8] J. K. Ousterhout, "Scheduling Techniques for Concurrent Systems," in *ICDCS*, October 1982.

[9] J. Błażewicz, *Scheduling under resource constraints: Deterministic models*. JC Baltzer, 1986, vol. 7.

[10] R. Jain, K. Somalwar, J. Werth, and J. C. Browne, "Heuristics for scheduling i/o operations," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 8, no. 3, pp. 310–320, 1997.

[11] B. Li, Y. Pei, H. Wu, and B. Shen, "Resource availability-aware advance reservation for parallel jobs with deadlines," *The Journal of Supercomputing*, pp. 1–22, 2012.

[12] D. M. Jacobson and J. Wilkes, *Disk scheduling algorithms based on rotational position*. Citeseer, 1991.

[13] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource kernels: A resource-centric approach to real-time and multimedia systems," in *Photonics West'98 Electronic Imaging*, 1997.

[14] M. Seltzer, P. Chen, and J. Ousterhout, "Disk scheduling revisited," in *Winter USENIX Technical Conference*, 1990.

[15] M. J. Gallagher and R. M. Jantz, "Method for scheduling the execution of disk i/o operations," Jul. 1 1997, uS Patent 5,644,786.

[16] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Disk schedulers for solid state drivers," in *International conference on Embedded software*, 2009.

[17] S. Park and K. Shen, "Fios: a fair, efficient flash i/o scheduler," in *FAST*, 2012.

[18] S. Kang, H. Park, and C. Yoo, "Performance enhancement of i/o scheduler for solid state devices," in *Consumer Electronics (ICCE), 2011 IEEE International Conference on*, 2011, pp. 31–32.

[19] M. P. DUNN, "A new i/o scheduler for solid state devices," Ph.D. dissertation, Texas A&M University, 2009.

[20] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design tradeoffs for ssd performance." in *USENIX Annual Technical Conference*, 2008.

[21] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger, "Argon: Performance insulation for shared storage servers." in *FAST*, 2007.

[22] W. Jin, J. S. Chase, and J. Kaur, "Interposed proportional sharing for a storage service utility," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 32, no. 1, 2004, pp. 37–48.

[23] C. A. Waldspurger and W. E. Weihl, "Lottery scheduling: Flexible proportional-share resource management," in *OSDI*, 1994, p. 1.

[24] P. Valente and F. Checconi, "High throughput disk scheduling with fair bandwidth distribution," *Computers, IEEE Transactions on*, vol. 59, no. 9, pp. 1172–1186, 2010.

[25] L. Abeni, G. Lipari, and G. Buttazzo, "Constant bandwidth vs. proportional share resource allocation," in *Multimedia Computing and Systems*, vol. 2, 1999, pp. 107–111.

[26] J. H. Anderson and M. Moir, "Wait-free synchronization in multiprogrammed systems: Integrating priority-based and quantum-based scheduling," in *PODC*, 1999.

[27] M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[28] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, 1973.

[29] P. Aggarwal, G. Yasa, and S. R. Sarangi. Appendix of radir: Lock-free and wait-free resource allocation model for flash drive bandwidth reservation. [Online]. Available: http://www.cse.iitd.ac.in/~srsarangi/tr/stpaperreport.pdf

[30] M. LLC. (2011) Msr cambridge traces. [Online]. Available: http://iotta.snia.org/traces/388

APPENDIX A.
RADIR ALGORITHM

Here we present the algorithim for reserving the disk bandwidth. The disk bandwidth is reserved by placing a new request in INIT phase. In this phase a thread (t) temporarily reserves a slot in the $diskRevRec$ array with help of the function $reserveSlots$ (explained in Section A-1). On successfully reserving the first slot, request moves to TEMP phase (Line 14) using compare-And-Set(CAS). In TEMP phase the remaining slots are reserved temporarily. Once it finishes doing so, the request moves to PERM phase (Line 28). In this phase the reservation made by the thread (or by some other helper on behalf of t) is made permanent(Line 40). Lastly, the request enters the FINISH phase where the list of reserved slots is returned. A thread is unable to reserve its slots if all the slots are permanently reserved till the thread reaches end of array or it reaches its slot deadline (Line 10). In each TEMP phase, a thread tries to book the next consecutive slot (Line 30). Request continues to stay in TEMP phase till the required number of time-slots are not reserved temporarily.

In case a thread is unable to temporarily reserve a slot, we move the request to CANCEL phase. In CANCEL phase (Lines 45- 48), the temporarily reserved slots are resettled to its default value (i.e FREE). The $request.slotAllocated$ field is also reset. The request again starts from the INIT phase with a new $round$ and $index$. $round$ field is used to synchronize the helpers and $index$ field indicates which slot to reserve in the $diskRevRec$ array. Once the request enters PERM phase, it is guaranteed that $diskSlot$ number of slots are reserved for the thread and no other thread can overwrite these slots. The disk head position is also updated to the last address which the thread (t) wishes to access (Line 41).

Algorithm 2: reserveDiskBandwidth

```
1:  function reserveDiskBandwidth(request)
2:    while TRUE do
3:      iterState ← request.iterationState.get()
4:      (reqState,round,index) ← unpackState(iterState)
5:      switch (reqState)
6:      case INIT :
7:        (status,res) ← reserveSlots(request,index, round,
          reqState)
8:        if status = FAIL then
9:          /* linearization point */
10:         request.iterationState.CAS(iterState,
            packState(0,0,0,FAIL))
11:       else if status = RETRY then
12:         /* read state again */
13:       else
14:         if request.iterationState.CAS(INIT,
            packState(1,round,res+1,TEMP)) then
15:           request.slotAllocated.CAS(-1,res)
16:         else
17:           /* clear the slot reserved */
18:         end if
19:       end if
20:       break
21:     case TEMP :
22:       slotRev ← getSlotReserved(reqState)
23:       /* reserve remaining slots */
24:       (status, res) ← reserveSlots(request,
          index,round,reqState)
25:       if res < request.slotDeadline ∧ status = SUCCESS
          then
26:         if slotRev+1 = req.diskSlots then
27:           /* linearization point */
28:           newReqState ← Request.packState(req.
              diskSlots,round,index,PERM)
29:         else
30:           newReqState ← Request.packState(slot
              Rev+1,round,index+1,TEMP)
31:         end if
32:         request.iterationState.CAS(iterState,
            newReqState)
33:       else if status = CANCEL then
34:         request.iterationState.CAS(iterState,pack
            State(slotRev, round, index, CANCEL))
35:       else
36:         RETRY/* read state again */
37:       end if
38:       break
39:     case PERM :
40:       /* make the reservation permanent */
41:       /* update the disk head position */
42:       request.iterationState.CAS(iterState, FINISH)
43:       break
44:     case CANCEL :
45:       /* reset the slots reserved till now */
46:       /* Increment request round */
47:       /* reset the slot Allocated field of request */
48:       request.iterationState.CAS(iterState,
          packState(0,round+1,index+1,INIT))
49:       break
50:     case FINISH :
51:       return request.slotAllocated
52:     end switch
53:   end while
54: end function
```

1) Reserve Slots: We start by explaining how a slot in the $diskRevRec$ array is reserved by a thread, which currently has highest priority among all the contending threads. This method accepts four parameters - request(req) of a thread t, the current slot to reserve $currSlot$, current round $round$ of the request and the phase of the request $reqState$. In case their are redundant drives, first we find the drive ($minDisk$) which requires the minimum number of dummy slots and also has least load factor. This is done with the help of the method $getMinDiskSlot$. Once we have the desired drive, depending upon the status of the slot ($currSlot$) in that drive we execute the corresponding switch-case statement. $round$ indicates the iteration of a request. It is used to synchronize all helpers of a request. If the slot is in the VACANT state, we try to temporarily reserve the slot and change the state of the slot from VACANT to TRANSIENT (Line 9) and update the load factor of the drive $minDisk$. Next, we discuss the case when the state of the slot is TRANSIENT. It indicates that some thread has temporarily

reserved the $currSlot$ slot. If the thread id saved in the slot is the same as that of the request req (Line 16), we simply return and read the phase of the request again. Otherwise, the slot is temporarily reserved by some other thread for another request, $otherReq$. Now, we have two requests req and $otherReq$ contending for the same slot $currSlot$ in the drive $minDisk$. If the priority of the request req is higher than $otherReq$, request req wins the contention and will overwrite the slot after cancelling the request $otherReq$ i.e changing the state of the request $otherReq$ to CANCEL atomically (Lines 22 - 29). Request req will help request $otherReq$ in case req has a lower priority. We increment the priority of a request to avoid starvation (Line 35).

Let us now discuss the case where the slot is found to be in the RESERVED state. In the INIT phase of the request, a request tries to search for the next vacant slot (Line 48). The search terminates when either a slot is successfully reserved or the request hits its slot deadline (Line 56). In the TEMP phase, we return CANCEL (Line 50). On receiving the result of the function $reseveSlots$ as CANCEL, the request moves to the CANCEL phase. Lastly, it is possible that some other helper has reserved the slot for request req (Line 42). In this case the thread refreshes and reads the phase of the request req again.

```
1:  function reserveSlots(request,currSlot, round, reqState)
2:    for i ∈ [currSlot, req.slotDeadline] do
3:      minDisk ← getMinDiskSlot(request, i)
4:      diskRevRec ← diskArray[minDisk]
5:      slotState ← getSlotState(diskRevRec.get(i))
6:      (threadid,round1,state) ←
        unpackSlot(disk − RevRec. get(i))
7:      switch (slotState)
8:      case VACANT :
9:        res ← diskRevRec.CAS(currSlot,
          packTransientState(request), VACANT)
10:       updateDiskLoad(minDisk)
11:       if res = TRUE then
12:         return (SUCCESS, currSlot)
13:       end if
14:       break
15:     case TRANSIENT :
16:       if threadid = req.threadid then
17:         /* slotState = MYTRANSIENT */
18:         return (RETRY, null)
19:       else
20:         otherReq ← REQUEST.get(threadid)
21:         res ←
            getHighPriorityRequest(req,otherReq,i,minDisk)
22:         if res = req then
23:           /* preempt lower priority request */
24:           if cancelReq(otherReq) then
25:             oldValue ← packTransientState( threadid,
                round1, state)
26:             newValue ← packTransientState(req.
                threadid, round, TRANSIENT)
27:             res1 ← diskRevRec.CAS(currSlot,
                oldValue, newValue)
28:             if res1 = TRUE then
29:               return (SUCCESS, currSlot)
30:             end if
31:             break
32:           end if
33:         else
34:           /* res = HELP */
35:           reserveDiskBandwidth(otherReq)
36:           /* increase priority to avoid starvation */
37:           req.priority.getAndIncrement()
38:         end if
39:       end if
40:       break
41:     case RESERVED :
42:       if threadid = req.threadid then
43:         /* slot reserved on req's behalf */
44:         return (RETRY, null)
45:       else
46:         if req.iterationState = INIT then
47:           slotMove ← getReserveSlot(diskRevRec.
              get(i))
48:           i ← i + slotMove
49:         else
50:           return (CANCEL, null)
51:         end if
52:       end if
53:       break
54:     end switch
55:   end for
56:   return (FAIL, req.slotDeadline)
57: end function
end
```

APPENDIX B.
PROOF

Theorem 1: The LF_RADIR and WF_RADIR algorithms are linearizable.

Proof: We need to prove that there exists a point of linearization at which the $reserve$ function appears to execute instantaneously. Let us try to prove that the point of linearization of a thread, t, is Line 28 when the request enters PERM phase, or it is Line 10 when the request fails because of lack of space or it misses it deadline. Note that before the linearization point, it is possible for other threads to cancel thread t using the $cancelReq$ function it they have higher priority than t. However, after the status of the request has been set to PERM, it is not possible to overwrite the entries reserved by the request. To do so, it is necessary to cancel the request. A request can only be cancelled in the INIT and TEMP phase. Hence, the point of linearization (Line 28) ensures that after its execution, changes made by the request are visible as well as irrevocable. If a request is failing, then this outcome is independent of other threads, since the request has reached the end of the $diskRevRec$ array.

Likewise, we need to prove that before the point of linearization, no events visible to other threads causes them to make permanent changes. Note that before this point, other threads can view temporarily reserved entries. They can perform two actions in response to a temporary reservation –

decide to help the thread that has reserved the slot (Line 35), or cancel themselves. In either case, the thread does not change its starting position.

A thread will change its starting position in Line 48, only if it is not able to complete its request at the current starting position because of a slot that is in RESERVED state.

Note, that the state of a slot is changed to RESERVED only by threads that have already passed their point of linearization. Since the current thread will be linearized after them in the sequential history, it can shift its starting position to the slot next to the reserved slot without sacrificing linearizability. We can thus conclude that before a thread is linearized, it cannot force other threads to alter its behavior. Thus, we have a linearizable implementation. ■