RADIR: Lock-free and Wait-free Bandwidth Allocation Models for Solid State Drives (Technical Report)

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*

I. 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 I-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 1: 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
```

```
      end switch
    end while
  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
```

II. 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. ■