Lock-Free and Wait-Free Slot Scheduling Algorithms

Pooja Aggarwal and Smruti R. Sarangi, *Member, IEEE*

Abstract—In this paper, we consider the design space of parallel non-blocking slot scheduling algorithms. Slot schedulers divide time into discrete quanta called *slots*, and schedule resources at the granularity of slots. They are typically used in high throughput I/O systems, data centers, video servers, and network drivers. We propose a family of parallel slot scheduling problems of increasing complexity, and then propose parallel lock-free and wait-free algorithms to solve them. In specific, we propose problems that can reserve, as well as free a set of contiguous slots in a non-blocking manner. We show that in a system with 64 threads, it is possible to get speedups of 10X by using lock-free algorithms as compared to a baseline implementation that uses locks. We additionally propose wait-free algorithms, whose mean performance is roughly the same as the version with locks. However, they suffer from significantly lower jitter and ensure a high degree of fairness among threads.

Index Terms—Wait free, lock free, scheduler, slot scheduling

1 Introduction

Large[1] shared memory multicore processors are becoming commonplace. Given the increased amount of parallel resources available to software developers, they are finding novel ways to utilize this parallelism. As a result, software designers have begun the process of scaling software to hundreds of cores. However, to optimally utilize such large systems, it is necessary to design scalable operating systems and middle-ware that can potentially handle hundreds of thousands of requests per second. Some of the early work on Linux scalability has shown that current system software does not scale beyond 128 cores [2], [3]. Shared data structures in the Linux kernel limit its scalability, and thus it is necessary to parallelize them. To a certain extent, the read-copy update mechanism [4] in the Linux kernel has ameliorated these problems by implementing wait-free reads. Note that writes are still extremely expensive, and thus the applicability of this mechanism is limited.

The problem of designing generic data structures that can be used in a wide variety of system software such as operating systems, virtual machines, and run-times is a topic of active research. In this paper, we focus on the aspect of scheduling in system intensive software. The traditional approach is to use a scheduler with locks, or design a parallel scheduler that allows concurrent operations such as the designs that use wait-free queues (see [5]). However, such approaches do not consider the temporal nature of tasks. For example, it is not possible to efficiently block an interval between $t+5$ and $t+7$ ms (where t is the current time) using a wait-free queue. Not only the arrival time, but the duration of the task should also be captured by the model.

Hence, in this paper, we look at a more flexible approach proposed in prior work called *slot scheduling* [6]. A slot scheduler treats time as a discrete quantity. It divides time into discrete quanta called *slots*. The Linux kernel divides time in a similar manner into *jiffies*. This model can support a diverse mix of scheduling needs of various applications. The key element of this model is an Ousterhout matrix [7] (see Fig. 1).

Here, we represent time in the x-axis, resources in the y-axis, and each slot(cell) represents a Boolean value—free or busy. *free* indicates that no task has been scheduled for that slot, and *busy* indicates the reverse. We can have several request patterns based on the number of requests that can be allotted per row and column. In this paper, we parameterize the problem of slot scheduling with three parameters—number of resources($capacity$), the maximum number of slots that a request requires ($numSlots$), and its progress condition (lock-free (LF), or wait-free(WF)). We consider four combinations of the capacity and number of slots: 1×1, $1 \times M$, $N \times 1$, and $N \times M$, where the format is $capacity \times numSlots$. For example, we can interpret the $N \times M - LF$ problem as follows. The number of rows in the Ousterhout matrix is equal to N, and a request requires up to M slots. These M slots need to be in contiguous columns. The 1×1 and $N \times 1$ problems are trivial. Our contribution is the non-blocking (lock-free and wait-free) implementation of the $1 \times M$ and $N \times M$ problems. The $N \times M$ formulation is the most generic version of the slot scheduling problem, and can be easily tailored to fit additional constraints such as having constraints on the rows.

We propose a novel, parallel, and linearizable (all operations appear to execute instantaneously [8]) data structure called *parSlotMap*, which is an online parallel slot scheduler. It is is well-suited for meeting the needs of both real-time

1. This is an extension of the conference paper by Aggarwal and Sarangi published in IPDPS 2013 [1].

• *The authors are with the Department of Computer Science & Engineering, Indian Institute of Technology, New Delhi, India. E-mail: {pooja.aggarwal, srsarangi}@cse.iitd.ac.in.*

Manuscript received 17 Sept. 2014; revised 13 May 2015; accepted 15 May 2015. Date of publication 19 May 2015; date of current version 13 Apr. 2016. Recommended for acceptance by S. S. Vadhiyar.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2435786

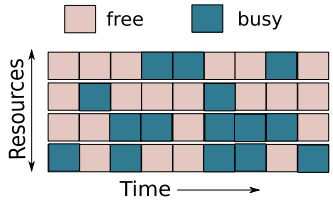

Fig. 1. The Ousterhout matrix for scheduling.

and gang-scheduled applications. It supports two operations—$schedule(request)$ and $free(request)$. Each request specifies the starting slot, and the number of slots it requires in the case of the $schedule$ operation, whereas in the $free$ operation, each request specifies the list of contiguous slots it wishes to free. Let us briefly motivate the need for such operations. Let us consider a scheduler for storage devices such as a set of hard drives or Flash drives [9]. Let us assume a set of requesting processes that need to access the storage devices. Since each device has a given number of ports (or channels), we can only service a small number of requests concurrently. Hence, a process that needs to access a storage device needs to *schedule* a set of slots in advance. Now, it is possible that, we might decide to cancel the request that was placed, or cancel the request of another process. This can happen for a variety of reasons such as termination of the process that needed to perform an I/O, a change in priorities, or because the driver got the value from a software cache (refer to [10] for more examples). We thus need a method to *free* slots. Thus, we provide two methods: $schedule$, and $free$.

To summarize, our contributions in this paper are as follows. We propose both lock-free and wait-free algorithms for different variants of the $free$ and $schedule$ operations. To the best of our knowledge, this has not been done before. Additionally, we prove that our algorithms are correct and linearizable. We implement them in Java, and for a 64-thread machine we find our non-blocking algorithms to be 1-2 orders of magnitude faster than algorithms that use locks. The wait-free algorithms are 3-8X slower than their lock-free counterparts. However, they have much better fairness guarantees, and for less than 16 threads have comparable system throughputs.

The paper is organized as follows. We give a brief overview of lock-free and wait-free algorithms in Section 2, discuss related work in Section 3, provide an overview of parallel slot scheduling in Section 4, show our algorithms in Section 5, sketch a proof in Section 6, present the evaluation results in Section 7, and finally conclude in Section 8.

2 Background

We assume a shared memory system where multiple independent threads see the same view of memory and share their address space. For a thread to successfully complete an operation on a concurrent data structure it needs to modify the state of some shared memory locations. To avoid correctness issues arising from simultaneous accesses to shared memory locations, the traditional approach is to encapsulate the critical region of code that accesses shared variables with locks. However, such approaches are slow primarily because they do not allow simultaneous access to disjoint memory regions and the thread holding the lock might get delayed indefinitely. A different paradigm is to allow the threads to go ahead, make their modifications to shared memory, and update crucial memory locations with read-modify-write operations such as *compareAndSet (CAS)*. The most common correctness guarantee for such *non-blocking* algorithms is *linearizability*(see Appendix A.2, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2015.2435786), which says that an operation is *linearizable* if it appears to take effect instantaneously at a point between its invocation and response. This point is called the point of linearizability.

Note that such non-blocking operations are not guaranteed to terminate in a finite number of steps. An algorithm is defined to be *lock-free* if at any point of time at least one thread is making forward progress and completing its operation. In a lock-free data structure, the system (all the threads) as a whole makes progress even though individual threads might suffer from starvation. In comparison, a *wait-free* algorithm guarantees that every operation will complete in a finite amount of time. This is typically achieved by faster threads helping the operations of slower threads.

3 Related Work

Scheduling is a classical problem. There has been a plethora of research in this area over the last few decades. Most of the work in classical parallel scheduling involves parallelizing different heuristics that are used in sequential scheduling (see the surveys by Wu [11] and Dekel and Sahni [12]). In this paper, we consider a popular paradigm of scheduling called *slot scheduling*, which provides a simple but powerful framework for building a wide range of schedulers that are especially useful in computer systems.

3.1 Slot Scheduling

In slot scheduling we divide time into discrete units called *slots*, and schedule tasks at the granularity of slots. Ousterhout [7] originally proposed a matrix representation of slots where time-slices(slots) are columns and processors are rows. This basic formulation has been used in later works for designing efficient multiprocessor schedulers (see[13] and Brandon Hall's thesis[6]). The main advantage of slot scheduling is that it makes scheduling simple, flexible, and easy to parallelize. For example, it is very easy to capture the following notion using slot schedulers: reserve m out of n subsequent slots for a job. We can additionally specify that m' out of m slots need to use resource 1, and the remaining can either use resources 1 or 2. Such mechanisms are very useful in storage systems [14], [15], [16]. In specific, Argon [14], uses slot schedulers to schedule disk I/O requests for a set of tasks running on a shared server using round robin scheduling. Anderson and Moir [15] propose a more generic scheme that also takes task priority into account. Park and Shen [16] propose a slot scheduler especially for flash drives that take preferential reads, and fairness into account. In addition slot schedulers have also been reported to be used in vehicular networks [17], ATM networks [18], and green computing [19]. It is important to note that all these slot schedulers are sequential. In our prior

work, we have proposed an software transactional memory (STM) based solution [20].

3.2 Non-Blocking Algorithms

To the best of our knowledge, lock-free and wait-free algorithms for parallel slot scheduling have not been proposed before. However, there has been a lot of work in the field of non-blocking parallel algorithms and data structures. Our algorithms have been inspired by some of the techniques proposed in prior work.

The problem that is the most closely related to slot scheduling is non-blocking multi-word compare-and-set (MCAS) [21], [22], [23]. Here, the problem is to atomically read k memory locations, compare their values with a set of k inputs, and then if all of them match, set the values of the k memory locations to k new values. This entire process needs to be done atomically, and additional guarantees of lock-freedom and linearizability are typically provided. The standard method for solving such problems is as follows. We have a two-pass algorithm. In the first pass we atomically mark the k memory locations as being temporarily reserved, and also read the values stored in them. Subsequently, we compare the values read from memory with the set of k inputs. If all the pairs of values match, we are ready to move to the second pass. Otherwise, we need to undo our changes, by removing the mark bit in the memory words that indicate temporary reservation. In the second pass, the thread goes through all the slots that it had temporarily reserved earlier, writes the new values, and removes the mark bits. There are many things that can go wrong in this process. It is possible that a thread i might encounter a memory location that has been temporarily reserved by another thread j. In this case, i cannot just wait for j to finish because there is a possibility that i might have to wait indefinitely. Instead, i needs to help j complete. While i is helping j, it might encounter a memory location that has been temporarily reserved by thread k. In this case, both i and j need to help k.

Such kind of issues thoroughly complicate such algorithms. Additionally, if we need to design an algorithm that bounds the number of steps that operation is allowed to take, then we need to ensure that no thread's request is left behind. It becomes the responsibility of faster threads to help all requests from other threads that are blocked. The reader can refer to the papers by Sundell [21] and Harris et al. [23] for a deeper discussion on the different heuristics that can be used to solve such problems. We need to acknowledge the fact that our slot scheduler uses similar high level ideas. It is however very different at the implementation level. Since we require the reserved slots to be in contiguous columns, and provide a choice for slots in a column, our helping, undo, and slot reservation mechanisms are very different. We additionally have the notion of freeing slots, which is not captured well by the MCAS literature.

4 Overview of Slot Scheduling

4.1 Definition of the Problem

Let us define the $parSlotMap$ data structure that encapsulates a 2D matrix of slots and supports two methods: $schedule$ and $free$. Every column is numbered and this number corresponds to time units (a column with a higher number denotes a later point in time). A schedule request (r) requires two parameters – the starting slot's column number ($slotRequested$), and the number of slots($numSlots$) to be reserved. Note that we start searching the slot matrix at the column with number $slotRequested$ till we are able to reserve $numSlots$ slots in contiguous columns (one slot per column). Note that the requested slots can be in same or in different rows, and the starting slot of the scheduled request can be ahead of the requested starting slot by an unbounded number of slots. The $free$ method takes only one parameter, which is the list of slots ($slotList$) that it wishes to release.

Both $schedule$ and $free$ need to be linearizable, which is a stronger correctness guarantee than sequential consistency. Sequential consistency means that the schedule generated by our concurrent scheduler should be the same as that generated by a purely sequential scheduler. Let us consider an example ($1 \times M$ problem($M = 3$)). Assume there are two requests that start at index 1 and want to book three slots. A sequential scheduler will try to book them at the earliest possible slots. There are two possible solutions: (request 1 (1-3), request 2 (4-6)), or (request 1 (4-6), request 2 (1-3)). The parallel scheduler should come up with one of these solutions. After the schedule request is completed the status of the slots 1 to 6 should be marked as $busy$. Let us now assume that request 1 completes before request 2 begins. In this case, the schedule is linearizable only if the schedule follows the real time order and is sequentially consistent i.e., request 1 gets 1-3 and request 2 gets 4-6. Similarly let us consider an example of a $free$ request. Assume there is a request r which wants to free the slots 2 to 4. After the request r is completed the status of the slots 2 to 4 should change from $busy$ to $free$. Any subsequent schedule request should be able to reserve these freed slots. We have designed another set of algorithms where we relax the constraint of reserving the earliest possible time slots and allow a request to get scheduled at some later slots (see Appendix E, available in the online supplemental material).

4.2 Basic Approach

Let us first look at the lock-free implementation of the schedule method. First, a thread starts searching from the first slot ($slotRequested$) for free slots in the next $numSlots$ contiguous columns. For each column, the thread iterates through all the rows till it finds a free slot. Once it finds a free slot, it changes the slot's status from EMPTY to TMP using an atomic CAS operation. After it has temporarily reserved the designated number of slots, it does a second pass on the list of slots that it has reserved(saved in the PATH array), and converts their status from TMP to HARD using CAS operations. This step makes the reservations permanent, and completes the schedule operation.

Note that there are several things that can go wrong in this process. A thread might not find enough free slots in a column or its CAS operations might fail due to contention. Now, in a column if all the slots are there in the HARD state then their status cannot be expected to change soon (unless there is a free request). Thus, the thread needs to undo all of its temporary reservations, and move beyond the column with all HARD slots (known as a *hard wall*). Alternatively, it is possible that some of the slots in a column might be in the

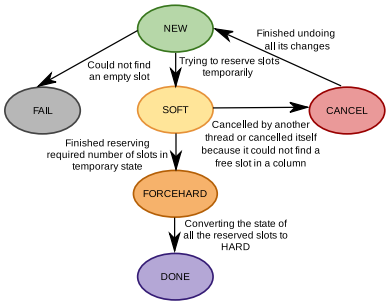

Fig. 2. Finite state machine of a *schedule* request.

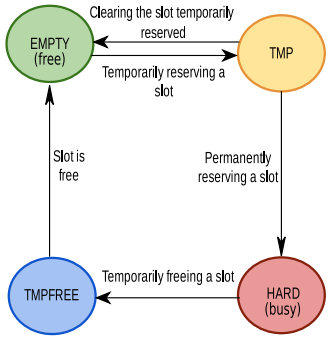

Fig. 3. Finite state machine of a slot.

TMP state, and there is no empty slot. In this case there are two choices. The first choice is to help the thread t (owner of the slot in the TMP state) to complete its operation (referred to as *internal helping*), and the second choice is to cancel thread t and overwrite its slot. This decision needs to be taken judiciously. After perhaps helping many threads, and getting helped by many other threads, a request completes. This algorithm is fairly complicated because we need to implement all the helping mechanisms, consider all the corner cases, and ensure linearizability.

The wait-free implementation is an extension of the lock-free approach by using a standard technique. Threads first announce their requests by creating an entry in a REQUEST array. Subsequently, they proceed to help older requests placed by other threads before embarking on servicing their own request. This ensures that no request remains unfinished for an indefinite amount of time. This is called *external helping*.

4.2.1 Details

The solution to the $N \times M$ *schedule* problem is broadly implemented in four stages. Each stage denotes a particular state of the request as shown in Fig. 2. The operation progresses to the next stage by atomically updating the state of the request.

1) At the outset, the request is in the NEW state. At this stage, a thread tries to temporarily reserve the first slot. If it is able to do so, the request moves to the SOFT state.
2) In the SOFT state of the request, a thread continues to temporarily reserve all the slots that it requires. When it has finished doing so, it changes the request's state to FORCEHARD. This means that the request has found the desired number of slots and it is ready to make its reservation permanent.
3) In FORCEHARD state, the temporary reservation is made permanent by converting the state of the reserved slots in the SLOT matrix to the HARD state. After this operation is over, the request transitions to the DONE state.
4) Finally in the DONE state, the thread collates and returns the list of slots allotted.

Let us comment on the need to have two separate phases. The main reason is that the atomic primitives can only operate on one memory word at a time. We can thus change the status of one slot at a time. Now, let's say that thread i needs to book five contiguous slots in an instance of the $1 \times M$ problem. After we have booked the first three slots, we cannot be sure of the availability of the last two slots. They might have been taken. We might have to undo the reservation of the first three slots. Meanwhile, assume that another thread, j, has seen one of the first three slots to be booked, and has changed its starting position. If thread i rolls back its changes, j will be deemed to have made the wrong decision, and linearizability will be violated. Hence, we found it necessary to first temporarily book a set of slots, then atomically set the state of the request to FORCEHARD (point of linearizability), and then change the status of the slots from TMP to HARD. The *free* operation is similarly implemented in two phases (see Fig. 4). Its steps are as follows.

1) The request is placed in the NEW state. At this stage, a thread indicates the first slot that it wishes to free. The state of the slot changes from HARD to TMPFREE. After doing so the request moves to the FSOFT state.
2) In the FSOFT state of the request, a thread temporarily frees the remaining slots in its list by converting the reserved slots in the HARD state to the TMPFREE state. When it has finished doing so, it changes the request's state to HELP.
3) In the HELP state, a thread t checks for the schedule requests that are not yet linearized. Now, if these requests can take the slots just freed by t, then thread t helps these requests in reserving the slots. After this operation is over, the request transitions to the FREEALL state.
4) In the FREEALL state, the temporarily freed slots are permanently freed by changing the state from TMPFREE to EMPTY. Finally, the request enters the DONE state, and the thread returns.

Next, we briefly discuss how the state of each slot in the SLOT matrix (see Fig. 3) changes. For the schedule request, the state of the slots in the SLOT matrix changes from EMPTY (*free*) to TMP and eventually to HARD. In the case of a free request, the state of the slot changes from HARD to TMPFREE to EMPTY.

5 Slot Scheduling Algorithm

5.1 Data Structures

The SLOT matrix represents the Ousterhout matrix of slots (see Fig. 5). Each entry in the matrix is 64 bits wide. When a slot is free, its state is EMPTY. When a thread makes a temporary reservation, the corresponding slots of the SLOT

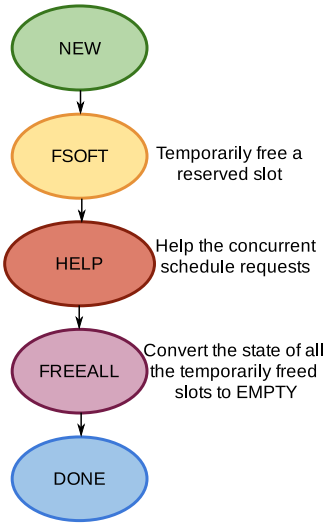

Fig. 4. Finite state machine of a *free* request.

matrix transition to the TMP state. We pack the following fields in each slot (64 bit long): *state* (2 bits), *requestId* (15 bits), *tid*(thread id) (10 bits), *slotNum* (6 bits), *round* (5 bits), and a *timestamp* (21 bits) (see Fig. 5). *slotnum* indicates the number of slots reserved by the thread. *round* indicates the iteration of a request. It is possible that a thread is able to reserve some slots, and is not able to proceed further because all the slots in a column are booked by other threads. In this scenario, the thread needs to start again with an incremented *round*. Lastly, the timestamp field is needed for correctness as explained in Section 5.5.

When a slot is temporarily freed, its state is TMPFREE. We pack the following fields: *state* (2 bits), *requestId* (15 bits), *tid*(thread id) (10 bits), and *slotNum* (6 bits). *slotNum* in this case indicates the number of subsequent slots a thread wishes to free. A slot in the HARD state has the same format. In this case, *slotNum* indicates the number of slots that have been reserved for that request in subsequent columns. We derive the sizing of different fields as described in Section 5.5.

Next, let us describe the REQUEST array that holds all the ongoing requests for all the threads in the system. An entry in the request array gets populated when a thread places a new request to reserve a set of slots or to free a set of reserved slots. It contains NUMTHREADS instances of the *Request* class. NUMTHREADS refers to the maximum number of threads in the system. The REQUEST array contains instances of the *Request* class (see Fig. 6). In the *Request* class, *requestId* and *tid*(thread id) are used to uniquely identify a request. *opType* is used to indicate whether it is a schedule request or a free request. *slotRequested* indicates the starting time slot number beyond which the request needs to be scheduled and *numSlots* denotes the number of slots a request wishes to reserve or free. The *iterState* field contains the current round of the request, the current index of a

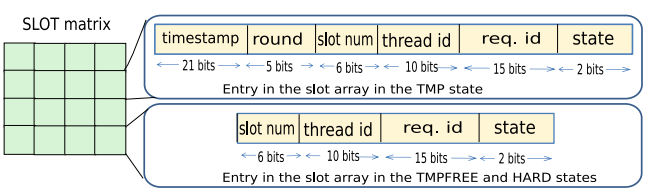

Fig. 5. The SLOT matrix.

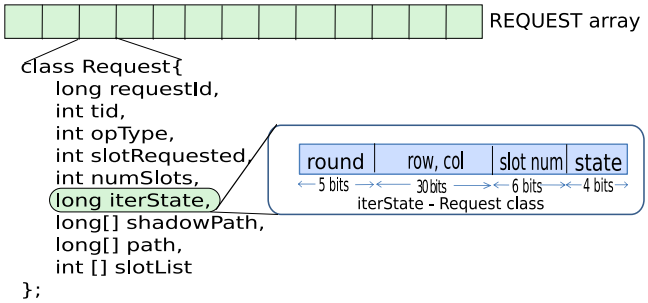

Fig. 6. The REQUEST array.

slot in the SLOT matrix (row, col), number of slots reserved, and the state of the request (as shown in Fig. 6).

Let us now describe two more fields of the *Request* class: the PATH and SHADOWPATH arrays (see Fig. 7). The PATH array stores the indices of the slots reserved by a thread. Whenever multiple helpers try to reserve a slot on behalf of a given thread, they first perform a CAS on a particular slot in the SLOT matrix and then save the entry in the PATH array atomically. To avoid the problem of different helpers booking different slots for the same request, we introduce the SHADOWPATH array. This is used by threads to announce their intention before booking a slot. Threads first search for a free slot, make an entry for it in the SHADOWPATH array, and then actually reserve it in the SLOT matrix. These two arrays are used for the schedule request. Each entry of the PATH array and SHADOWPATH array contains multiple fields as shown in Fig. 7.

The last field, *slotList* (in the *Request* class), is used for *free* requests. It contains the list of slots a thread wishes to free.

5.2 Entry Points

A thread t_i places a request to either schedule a request or to free a set of slots by calling the *applyOp* method (see Algorithm 1). Thread t_i first atomically increments a counter to generate the request id for the operation (Line 2). Then, it creates a new request with the time slots it is interested in booking or freeing, and sets the corresponding entry in the REQUEST array with a new request id (Line 8). In the lock-free algorithms, each thread tries to atomically book/free a slot for itself whereas in the wait-free case, it first helps other requests that meet the helping criteria. A thread helps only those requests for which the difference in the *requestId* is greater than REQUESTTHRESHOLD (Line 21). These functions are shared across the $1 \times M$ and $N \times M$ variants of our schedule and free algorithms. Each algorithm needs to implement its variant of the *processSchedule* and *processFree* functions. Note that in the *findMinReq* method (invoked in Line 20), the request *req* is passed as an argument. A request, which has *requestId* less than *req* and has not yet completed is returned (i.e. state not equal to DONE). This is later helped by the request, *req*, to complete its operation.

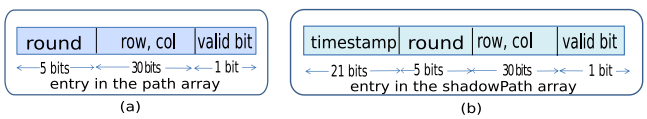

Fig. 7. The PATH and SHADOWPATH arrays.

Algorithm 1. Entry Points

```
1: function applyOp(tid,slotRequested,numSlots,optype)
2:   reqId ← requestId.getAndIncrement()
3:   if optype = schedule then
4:     req ← createRequest(reqId, tid, slotRequested,
       numSlots, NEW, optype)
5:   else if optype = free then
6:     req ← createRequest(reqId, tid, slotList, numSlots,
       NEW, optype)
7:   end if
8:   REQUEST.set(tid, req) /* announce the request */
9:   if WAITFREE then
10:    help(req) /* help other requests */
11:  end if
12:  if optype = schedule then
13:    return processSchedule(req)
14:  else if optype = free then
15:    return processFree(req)
16:  end if
17: end function

18: function help(req)
19:   while true do
20:     minReq ← findMinReq(req)
21:     if (req.getRequestId() - minReq.getRequestId() <
        REQUESTTHRESHOLD) then
22:       break
23:     end if
24:     if minReq.optype = schedule then
25:       return processSchedule(minReq)
26:     else if minReq.optype = free then
27:       return processFree(minReq)
28:     end if
29:   end while
30: end function

31: function findMinReq (req)
32:   minReq ← NULL
33:   for i ∈ [0, NUMTHREADS-1] do
34:     r ← requests[i]
35:     (state, reqid) ← unpack(r)
36:     if (state = DONE) || (reqid ≥ req.getRequest Id()) then
37:       continue
38:     end if
39:     minReq ← min(minReq, r)
40:   end for
41:   return minReq
42: end function
end
```

5.3 The $N \times M$ Schedule Problem

Here, we describe the implementation of the $N \times M$ algorithm. The implementation of the $1 \times M$ algorithm is discussed in Appendix D, available in the online supplemental material. The code for the *processSchedule* method is shown in Algorithm 2. We assume that the requested starting slot is in column *col*, and the number of slots requested is *numSlots* (can vary from 1 to M).

Algorithm 2. *processSchedule* $N \times M$

```
1: function processSchedule (Request req)
2:   Data
3:   col, row0 ← the requested slot
4:   state ← the current state of the requested
5:   nstate ← the new state of the request
6:   slotNum ← number of slots reserved so far
7:   while TRUE do
8:     (state,slotNum,round,row0,col) ← unpack
       (req.iterState)
9:     switch state
10:    case NEW :
11:      (res, row1, col1) ← bookFirstSlot(req, col, round)
12:      if res = FALSE then
13:        /* unable to find a free slot */
14:        /* set state to FAIL */
15:      else if res = TRUE then
16:        /* save the index of the slot in the PATH array
           and set state as SOFT */
17:        if pathCAS(req, round, slotNum, row1, col1)
           then
18:          nstate ← pack(SOFT,slotNum+1,round,
             row1,col1)
19:        else
20:          /* reset the slot in SLOT and SHADOWPATH array */
21:        end if
22:      end if
23:      break
24:    case SOFT :
25:      (round1, row1, col1) ← unpack(req.PATH.
         get(slotNum-1))
26:      /* reserve remaining required slots */
27:       (res,row2) ← bookMinSlotInCol(req, col1+1, slot-
          Num, round)
28:      if res = FALSE then
29:        /* changes its starting position */
30:        col3 = col1 +2
31:        /* request enters in cancel state */
32:        nstate ← pack(CANCEL,0,round,0,col3)
33:        req.iterState.CAS(state,nstate)
34:      else if res = TRUE then
35:        if pathCAS(req,round,slotNum,row2,col1+1)
           then
36:          if slotNum = numSlots then
37:            /* Point of linearization: If nstate is successfully
               set to FORCEHARD */
38:             nstate ← pack(FORCEHARD, numSlots, round,
                row0, col)
39:          else
40:            nstate ← pack(SOFT, slotNum+1, round, row2,
               col1+2)
41:          end if
42:        else
43:          /* reset the slot in SLOT and SHADOWPATH array */
44:        end if
45:      end if
46:      break
47:      case FORCEHARD :
48:        /* state of slots in SLOT matrix changes from TMP to
           HARD */
49:        forcehardAll(req)
50:        nstate ← pack(DONE, numSlots, round, row0, col)

51:      case DONE :
52:        /* return slots saved in the PATH */
53:        return req.PATH
54:      case CANCEL :
```

```
55:          /* slots reserved in SLOT matrix for request req are
             reset, PATH array and SHADOWPATH array get clear */
56:          undoPath (req, round)
57:          if cancelCount.get(req.getTid()) <
             CANCELTHRESHOLD then
58:            nround ← round +1
59:          else
60:            nround ← CANCELTHRESHOLD
61:          end if
62:            /* a request starts anew from NEW state */
63:            nstate ← pack(NEW, 0, nround, row0, col)
64:          case FAIL :
65:            return -1
66:          req.iterState.CAS(state, nstate)
67:          end switch
68:        end while
69:      end function

70: function getSlotStatus(req, value, round, slotNum)
71:   tid ← req.getTid()
72:   reqId ← req.getReqId()
73:   (reqId1, stat1, tid1, round1, slotNum1) ← unpack(value)
74:   if stat1 = HARD then
75:     return ((tid1,reqId1) = (tid,reqId)) ? BEHIND: HARD
76:   end if
77:   if stat1 = EMPTY then
78:     return EMPTY
79:   end if
80:   if stat1 = TMPFREE then
81:     return TMPFREE
82:   end if
83:   /* The state is SOFT and the tids are different */
84:   if tid ≠ tid1 then
85:     return (slotNum1 > slotNum) ? CANNOTCONVERT:
       ((cancelCount.get(tid1) < CANCELTHRESHOLD)?
       CANCONVERT: CANNOTCONVERT)
86:   end if
87:   /* tids are same */
88:   /* Give preference to the higher round */
89:   if round1 <= round then
90:     return AHEAD
91:   end if
92:   return BEHIND
93: end function

94: function bookMinSlotInCol (req, col, slotNum, round)
95:   while TRUE do
96:     (row, rank1, tstamp ) ← findMinInCol(req, col,
       slotNum, round)
97:     /* Set the SHADOWPATH entry */
98:     shadowPathCAS(req,row,col,tstamp,slotNum)
99:     /* currval is the value currently saved in
       SHADOWPATH array and def is the default value */
100:    /* the value finally saved in the SHADOWPATH array
       is (row1, col1) */
101:    /* compute the rank of the slot (row1,col1) */
102:    ...
103:    /* expSval is the expected value of the slot (row1,col1) */
104:    /* {newSval is the new value a thread wishes to save in the
       slot (row1,col1) */
105:      (reqid1, tid1, round1, slotNum1, stat1) ← unpack
         (SLOT [row1][col1])
106:    switchrank
107:    case BEHIND :
108:      /* undo SHADOWPATH array */
109:      req.SHADOWPATH.CAS(slotNum, currval, def)
110:      return (REFRESH, NULL)
111:    case AHEAD || EMPTY :
112:      /* reserve temporary slot */
113:        if (SLOT [row1][col1].CAS(expSval, newSval) =
          FALSE) ∧ (SLOT [row1][col1].get() ≠ newSval)
          then
114:          req.SHADOWPATH.CAS(slotNum, currval, def)
115:        continue
116:      end if
117:      return (TRUE, row1)
118:    case CANCONVERT :
119:      /* try to change other request's state to CANCEL */
120:      if otherCancel(tid1, round1) = FALSE then
121:        continue
122:      end if
123:      if (SLOT [row1][col1].CAS(expSval, newSval) =
         FALSE) ∧ (SLOT [row1][col1].get() ≠ newSval) then
124:        req.SHADOWPATH.CAS(slotNum, currval, def)
125:        continue
126:      end if
127:      return (TRUE, row1)
128:    case CANNOTCONVERT :
129:      req.SHADOWPATH.CAS(slotNum, currval, def)
130:      processSchedule(request.get(tid1))
131:      break
132:    case TMPFREE :
133:      req.SHADOWPATH.CAS(slotNum, currval, def)
134:      processFree(request.get(tid1))
135:      break
136:    case HARD :
137:      req.SHADOWPATH.CAS(slotNum,currval, def)
138:      return (FALSE, NEXT)
139:    end switch
140:  end while
141: end function
end
```

5.3.1 The *processSchedule* Method

We show an overall flowchart of the *processSchedule* function in Fig. 8. It extends Fig. 2 by listing the actions that need to be taken for each request state. The reader is requested to use this flowchart as a running reference when we explain the algorithm line by line.

First, we unpack the *iterState* of the request in Line 8, and execute a corresponding switch-case statement for each request state. In the NEW (starting) state, the *bookFirstSlot* method is used to reserve a slot $s_{[row1][col1]}$ in the earliest possible column, *col*1, of the SLOT matrix. We ensure that all the slots between columns *col* (requested starting column) and *col*1 are in the HARD state (permanently booked by some other thread). The *bookFirstSlot* method calls the method *bookMinSlotInCol* to reserve a slot. Since there can be multiple helpers, it is possible that some other helper might have booked the first slot. In this case we would need to read the state of the request again.

If we are able to successfully reserve the first slot, then the request enters the SOFT state; otherwise, it enters the FAIL state and the schedule operation terminates for the request. In specific, the request enters the FAIL state, when we reach the end of the SLOT matrix, and there are no more empty slots left. Now, in the SOFT state of the request, the rest of the

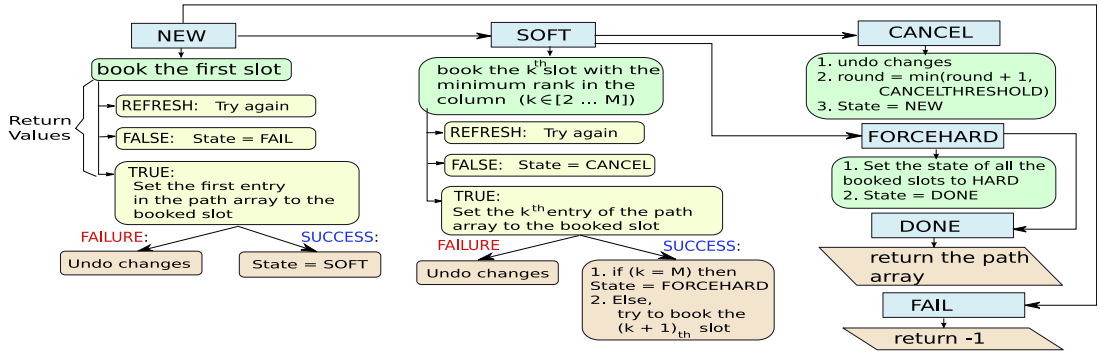

Fig. 8. The $processSchedule$ function.

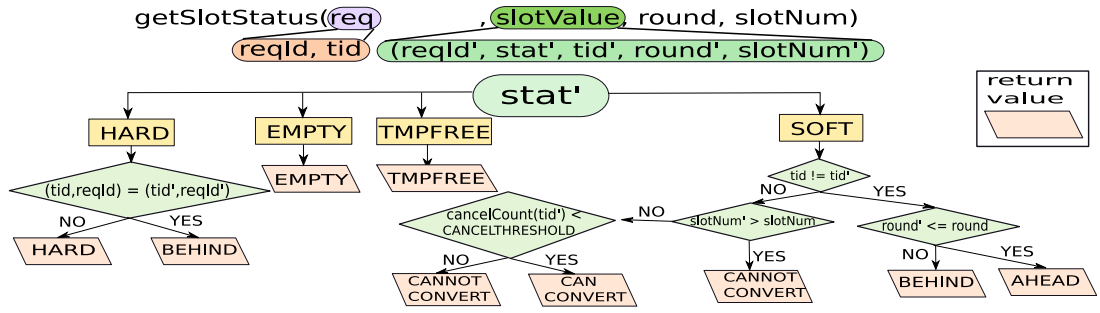

Fig. 9. The $getSlotStatus$ function.

slots are reserved in the TMP state by calling the $bookMinSlotInCol$ method iteratively (Line 27). The TMP state of a slot corresponds to a temporary reservation.

After reserving a slot (in the TMP state), we enter its index in the PATH array (Line 17). The state of the request remains SOFT (Line 40), and then becomes FORCEHARD after reserving the M^{th} (last) slot (Line 38). If the state of the request is successfully set to FORCEHARD, then it is the point of linearization for the successful $schedule$ call (see Appendix A, available in the online supplemental material).

In case a thread is unable to reserve a slot in the SOFT state, we set the state of the request to CANCEL (Lines 28 to 33). This happens because the request encountered a column full of HARD entries (hard wall). It needs to change its starting search position to the column after the hard wall (Line 29), which is column $col3$.

In the CANCEL state (Lines 54-63), the temporarily reserved slots are reset (i.e SOFT $\rightarrow$ EMPTY) along with the PATH and SHADOWPATH arrays. The state of the request is atomically set to NEW. We reset the starting column, and set the round to $min(round + 1,$ CANCELTHRESHOLD$)$. All this information is packed as one word and atomically assigned to the $iterState$ field of the request.

After a request has entered the FORCEHARD state, it is guaranteed that M slots have been reserved for the thread and no other thread can overwrite these slots. The state of all the slots reserved is made HARD and then the request enters the DONE state (Lines 49-50).

5.3.2 The $getSlotStatus$ Method

The $bookMinSlotInCol$ method used in Line 27 calls the $getSlotStatus$ method to rank each slot in a column, and chooses a slot with the minimum rank. Ranks are assigned to a slot based on its current state as shown in Fig. 9.

The $getSlotStatus()$ method accepts four parameters – req (request of thread t_j) for which the slot is to be reserved, current $round$ of t_j, the number of the slot($slotNum \in [1 \ldots M]$) that we are trying to book, and the $value$ stored at slot $s_{[row][col]}$. This method returns the rank of the slot $s_{[row][col]}$ (Lines 70-93).

The state of $s_{[row][col]}$ can be either HARD, TMP, TMPFREE or EMPTY. First, if $s_{[row][col]}$ is already in the HARD state and t_j owns the slot $s_{[row][col]}$ then it means that some other helper has already reserved this slot for t_j and has set it to HARD. The current thread is thus lagging behind; hence, we set the rank to BEHIND. If this is not the case, then the slot is permanently reserved for some other request, no other thread can take this slot, and we set the rank as HARD.

If the slot is in the TMP state and belongs to a different request, then we check if we can cancel the thread (say t_k) that owns the slot. We give a preference to requests that have already reserved more slots. If we decide to cancel the thread, then we return CANCONVERT, else we return CANNOTCONVERT. Note that if a thread has already been cancelled CANCELTHRESHOLD times, then we decide not to cancel it and return CANNOTCONVERT. In case both the threads t_j and t_k have been cancelled CANCELTHRESHOLD times then t_j helps t_k in completing its request and then proceeds with its own request. In this case the rank is returned as CANNOTCONVERT.

If the slot belongs to the same request, then the rank can be either AHEAD or BEHIND. The slot has rank AHEAD if it has been reserved by a previous cancelled run of the same request, or by another helper. Likewise, BEHIND means that the current run has been cancelled and another helper has booked the slot in a higher round.

If the slot is being freed by some other thread (t_f), then its state would be TMPFREE and we set the rank of the slot as

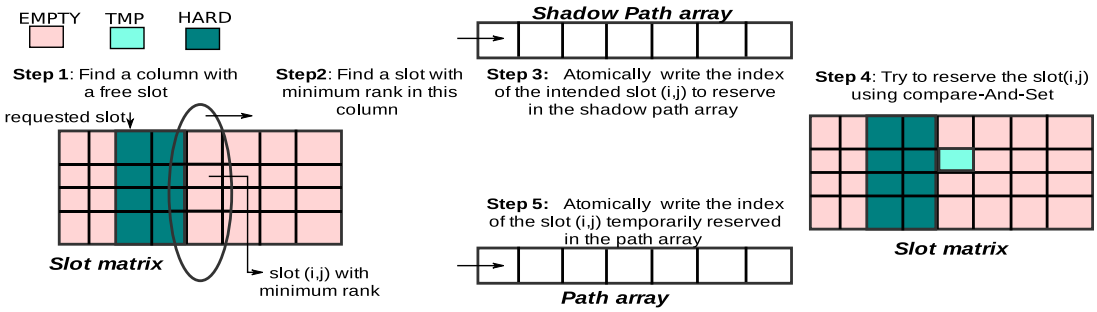

Fig. 10. Various steps involved in reserving a slot (temporarily).

TMPFREE. If the slot is already EMPTY then we set the rank as EMPTY.

The order of the ranks is as follows: BEHIND $<$ AHEAD $<$ EMPTY $<$ TMPFREE $<$ CANCONVERT $<$ CANNOTCONVERT $<$ HARD.

5.3.3 The *bookMinSlotInCol* Method

This method reserves a slot, with minimum rank, in the specified column in the SLOT matrix (Lines 94-141). First, this method calls the method $findMinInCol$, which in turn calls the $getSlotStatus$ method that returns the slot with the lowest rank (Line 96). Then, a thread records its intention to reserve the slot by trying to save its index in the SHADOWPATH array. Finally, it tries to reserve the slot in the SLOT matrix based on its rank as shown in Fig. 10, and if successful it records the row and column of the slot in the PATH array.

Let us now explain in detail. The $bookMinSlotInCol$ method accepts four parameters – request(req) of a thread t_j, column(col) to reserve a slot in, the number of the slot ($slotNum \in [1 \ldots M]$) that we are trying to reserve, and the current round($round$). The $findMinInCol$ method returns the lowest ranked slot along with its timestamp, $tstamp$. The timestamp is needed for correctness as explained in Section 5.5. Subsequently, all the helpers try to atomically update the SHADOWPATH array at index $slotNum$ (Line 98), and only one of them succeeds. We read the value of the entry that is finally stored in the SHADOWPATH array (by the thread or by its helpers) and compute its $rank$ (Line 100-101).

If the $rank$ is BEHIND, then it means that the thread should return to the $processSchedule$ method, read the current state of the request, and proceed accordingly. If the rank is AHEAD or EMPTY, we try to reserve the slot $s_{[row1][col1]}$ (Line 113). Simultaneously, other helper threads also observe the entry in the SHADOWPATH array and try to reserve the slot. Whenever we are not able to book the intended slot, the SHADOWPATH entry at index $slotNum$ is reset. If the $rank$ is CANCONVERT, then it means a lower priority thread (t_l) has reserved the slot $s_{[row1][col1]}$. In this case, the thread t_j tries to cancel the request of the thread t_j with the help of the method $otherCancel$ (Line 120). If thread t_j is successful in cancelling the request of thread t_l, then it proceeds to reserve the slot $s_{[row1][col1]}$.

If the $rank$ is CANNOTCONVERT, then it means that we have encountered a column that is full of temporary reservations of other threads, and we cannot cancel them. Hence, we start helping the request, which is the current owner of the slot $s_{[row1][col1]}$ (Line 130). If the $rank$ is TMPFREE, then it means that all the slots in that column are getting freed by some $free$ requests in progress. In this case, we help the $free$ request (Line 134), which is currently freeing the slot $s_{[row1][col1]}$. If the $rank$ is HARD, then it means that all the slots in that column are in the HARD state (already booked). We call such kind of a column a *hard wall*. In this case, we need to cancel the request of thread t_j. This involves converting all of its TMP slots to EMPTY, and resetting the PATH and SHADOWPATH arrays. Then the request needs to start anew from the column after the hard wall.

5.4 The $free$ Operation

The $processFree$ method captures the details of the $free$ operation (see Algorithm 3). The method accepts the request, req, as an argument and tries to free the intended slots i.e., change the state of slots to EMPTY. The initial state of the request is NEW. In the NEW state, a thread tries to temporarily free the first slot it wishes to free (change its state from HARD to TMPFREE). In doing so, a thread also saves the number of slots it will subsequently free (Line 11). This information is helpful for conflicting schedule requests. A schedule request will get to know the number of consecutive slots that will be freed and it can reserve those slots. After this, the request moves to the next state which is FSOFT (Line 12).

In the FSOFT state, a request continues to temporarily free the rest of the slots in its list ($slotList$). In this state of the request we use the $iterState$ field to additionally store the number of slots, $slotNo$, left to be freed. If the value of $slotNo$ is equal to 1, then it means that the required number of slots (M) are freed. Once the state of the desired number of slots is changed from HARD to TMPFREE, the state of the request is changed to HELP (Line 17).

Algorithm 3. Algorithm to Free Slots

```
1:  function processFree(req)
2:    Data
3:    state ← current state of the request req
4:    slotNo ← the number of slots to be freed by the thread
5:    startSlot (row,col) ← the slot to be freed
6:    while TRUE do
7:      expVal ← packSlot(req.reqTid,HARD)
8:      freVal ← packSlot(req.threadid, req.numSlot-1,
        TMPFREE)
9:      switch state
10:     case NEW :
11:       SLOT [row][col].CAS(expVal, freVal)
```

```
12:       req.iterState.CAS(reqState, packState(FSOFT,
          slotNo-1))
13:       break
14:     case FSOFT :
15:       SLOT [row][col].CAS(expVal, freVal)
16:       if slotNo == 1 then
17:         newState ← HELP
18:       else
19:         newState ← packState(TMPFREE, slotNo -1)
20:       end if
21:       req.iterState.CAS(reqState, newState)
22:       break
23:     case HELP :
24:       for i ∈ [0, req.numSlots-1] do
25:         checkHard ← checkBreakList(req.slotList(i))
26:         if checkHard = TRUE then
27:           /* find a schedule request that can be
              scheduled at column i */
28:           reqSch ← scanRequestArray(req)
29:           /* check state of the schedule request */
30:           reqState ← reqSch.iterState
31:           if reqState ≠ FORCEHARD ∧ reqState ≠
              DONE then
32:             /* help the request in getting scheduled at
                column i */
33:             notifyFreeSlot(req,reqSch,i)
34:           end if
35:         end if
36:       end for
37:       newState ← FREEALL
38:       req.iterState.CAS(state, newState) /* point of
          linearization */
39:     case FREEALL :
40:       for i ∈ [0, req.numSlots-1] do
41:         startSlot ← req.slotList(i)
42:         (row,col) ← unpack(startSlot)
43:         SLOT [row][col].CAS(packSlot(req.threadid,
            numSlot-(i+1),TMPFREE), EMPTY)
44:       end for
45:       req.iterState.CAS(state, DONE)
46:       break
47:     case DONE :
48:       return TRUE
49:       break
50:     end switch
51:   end while
52: end function

53: function notifyFreeSlot(req_f, req_s, index)
54:   (round, row1, col1) ← unpack(req_s.PATH. get(0))
55:   slot ← req_f.slotList(index)
56:   (row,col) ← unpack(slot)
57:   if col < col1 then
58:     nstate ← pack(CANCEL,0,round,0,
        max(req_s.slotRequested, col - req_s.numSlots + 1))
59:     otherCancel(req_s, nstate)
60:   end if
61: end function
end
```

Next, in the HELP phase, a request tries to help other conflicting schedule requests in the system to reserve the slots freed by it. To do so, a free request first checks while freeing a slot whether it has broken a hard wall or not (Line 25). Recall that a *hard wall* is a column of slots, where the state of all the slots is HARD. This check is required to enforce linearizability. Whenever, a schedule request encounters a hard wall (say at column k), it changes its starting index, and moves past the hard wall (to column $k+1$). Let us consider the case of a concurrent $free$ and $schedule$ request that access the same set of slots. Now, assume that the $schedule$ request finds a hard wall, moves ahead. Meanwhile, the $free$ request completes its operation. At a later point, the $schedule$ operation also completes. Since the point of linearizability of the $schedule$ operation is after the point of linearizability of the $free$ operation, it should have seen the $free$ operation. It should not have concluded that a hard wall exists. Instead, it should have used the slots freed by the $free$ operation. However, this has not been the case. To avoid this problem, it is necessary for the $free$ operation to wait till all concurrent, and conflicting schedule operations finish their execution. On the other hand, if no hard wall is broken, then it means that no $schedule$ request has moved passed the columns in which a free request is freeing the slots. Hence, in this case, it is not necessary to help $schedule$ requests.

In case a hard wall is broken, a free request first scans the REQUEST array to see if some schedule request is pending and conflicting. A request is considered as pending if its not linearized yet. Additionally, a request is considered conflicting if its ($slotRequested$) lies within [req.slotRequested, req.slotRequested + numSlots -1]. Next, we invoke the $notifyFreeSlot$ method if a conflicting schedule request req_s is found (Lines 54-60). It accepts three arguments — free request req_f, schedule request req_s and the $index$ indicating the slot number for which the hard wall was broken. We unpack the PATH array of the request req_s and find the column $col1$ at which req_s has reserved its first slot. If this column's number is greater than the number of the column (col) at which the free request freed its slot, then it means that the $schedule$ request needs to be cancelled. We thus cancel the schedule request, and make it start anew. The new starting position is important. If the $schedule$ operation needed to book $numSlots$ slots, then its new starting position can be as early as $col - numSlots + 1$ (see Line 58 for more details).

After helping the schedule request, a free request proceeds with it own request and enters the FREEALL phase. In the FREEALL phase, a request permanently frees all the slots. The status of the slots is changed from TMPFREE to EMPTY (Line 43). Lastly, a request enters the DONE state and returns successfully.

5.5 ABA Issues, Sizing of Fields, Recycling

The ABA problem represents a situation where a thread, t_i, may incorrectly succeed in a CAS operation, even though the content of the memory location has changed between the instant it read the old value and actually performed the CAS. For example, a process P_i has read a value of a shared memory (mem) as A and then sleeps. Meanwhile process P_j enters the system and modifies the value of shared memory mem to B and then back to A. Later, process P_i begins its execution and sees that the shared memory mem value has not changed and continues. This is know as the ABA

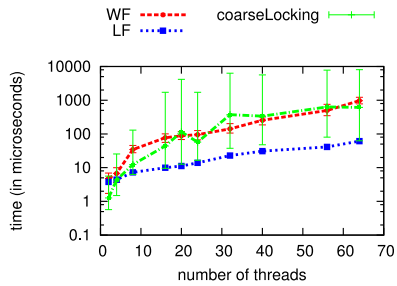

Fig. 11. t_{req} for the $N \times M$ algorithm.

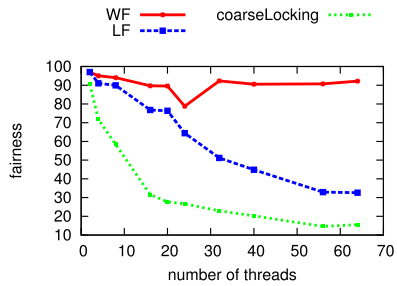

Fig. 12. Fairness (frn) for the $N \times M$ algorithm.

problem. The same thing can happen, when we are trying to reserve a slot. It is possible that the earliest thread might see an empty slot, enter it in the SHADOWPATH array, and then find the slot to be in the SOFT state. However, another helper might also read the same SHADOWPATH entry, and find the slot to be in the EMPTY state because the request holding the slot might have gotten cancelled. To avoid this problem, we associate a timestamp with every slot. This is incremented, when a thread resets a slot after a cancellation.

The maximum number of rounds for a request is equal to the CANCELTHRESHOLD. We set it to 32 (5 bits). We limit the number of slots (M) to 64 (6 bits). We can support up to 1,024 threads (10 bits). We note that the total number of timestamps required is equal to the number of times a given slot can be part of a cancelled request. This is equal to CANCELTHRESHOLD $\times$ NUMTHREADS $\times M$. The required number of bits for the timestamp field is $5 + 10 + 6 = 21$.

In our algorithm, we assume that the SLOT matrix has a finite size, and a request fails if it tries to get a slot outside it. However, for realistic scenarios, we can extend our algorithm to provide the illusion of a semi-infinite size SLOT matrix, if we can place a bound on the skew between requests' starting slots across threads. If this skew is $\mathcal{W}$, then we can set the size of the SLOT matrix to $\mathcal{S} > 2\mathcal{W}$, and assume the rows of the SLOT matrix to be circular.

6 Proof

We can prove that our all the algorithms obey sequential semantics (their execution matches that of a single thread), and are linearizable. We also prove that our algorithms lock-free and wait-free. The proofs are mentioned in detail in Appendix A, available in the online supplemental material.

7 Evaluation

7.1 Setup

We perform all our experiments on a hyper-threaded four socket, 64 bit, Dell PowerEdge R810 server. Each socket has eight 2.20 GHz Intel Xeon CPUs, 16 MB L2 cache, and 64 GB main memory. We have 64 threads visible to software. It runs Ubuntu Linux 12.10 using the generic 3.20.25 kernel. All our algorithms are written in Java 6 using Sun OpenJDK 1.6.0_24. We use the java.util.concurrent, and java.util.concurrent.atomic packages for synchronization primitives.

We evaluated the performance of our scheduling algorithms by assuming that the inter-request distances are truncated normal distributions (see Selke et al. [24]). We generated normal variates using the Box-Muller transform (mean = 5, variance = 3* tid). We run the system till the first thread completes κ requests. We define three quantities – mean time per request (t_{req}), *fairness* (frn) and throughput of the scheduler. The *fairness* is defined as the total number of requests completed by all the threads divided by the theoretical maximum. $frn = tot\_requests/(\kappa \times numthreads)$. frn measures the degree of imbalance across different threads. It varies from $1/$NUMTHREADS (min) to 1(max). If the value of frn is equal to 1, then all the threads complete the same number of requests – κ. The lower is the fairness, more is the discrepancy in performance across the threads.

We set a default REQUESTTHRESHOLD value of 50, and κ to 10,000. We varied the number of threads from 1 to 64 and measured t_{req} and frn for the $N \times M$ and $1 \times M$ variants of the problem. We perform each experiment 100 times, and report mean values. In our workload, 70 percent of the requests are $schedule$ operations, and the remaining 30 percent are $free$ operations (chosen at random). We vary the number of slots randomly from 2 to 64 (uniform distribution). We consider three flavors of our algorithms – lock-free (LF), wait-free (WF), and a version with locks (LCK). Further, we have implemented the version with locks in three ways – coarse grain locking ($coarseLocking$), fine grain locking ($fineLocking$) and fair coarse grain locking ($fairLocking$). The coarse grain locking algorithm performs better as compared to the fine grain and fair locking algorithms. Therefore, we have shown the comparison of our WF and LF algorithms with the $coarseLocking$ algorithm. Implementation details of different lock based algorithms and their results are presented in Appendix C, available in the online supplemental material.

7.2 Performance of the $N \times M$ and $1 \times M$ Algorithms

Fig. 11 and 12 present the results for the $N \times M$ problem. The LF algorithm is roughly 10X faster than all the algorithms (in terms of mean t_{req}). The performance of WF and $coarseLocking$ is nearly the same. However, the variation observed in the case of $coarseLocking$ is very high. For $coarseLocking$, t_{req} increases to up to 8,030 μs (mean: 620μs) for 64 threads. In the worst case, $coarseLocking$ is 7X slower than WF (for >32 threads), and it is two orders of magnitude slower than the LF algorithm. As explained in detail in Appendix C.4, available in the online supplemental material, there are two reasons for this trend: (1) the thread holding the lock might go to sleep, (2) and it might take a very long time to acquire the lock (even with fair locking algorithms). In comparison the jitter (max. value $-$ min. value)

TABLE 1
Number of Requests Helped by a Single Request

Number of Threads	WF external helping	WF internal helping	LF internal helping
2	0	0	0
4	1	0	0
8	1	0	1
16	2	1	1
20	4	1	1
24	4	1	1
32	5	2	1
40	7	2	1
56	10	3	2
64	9	2	2

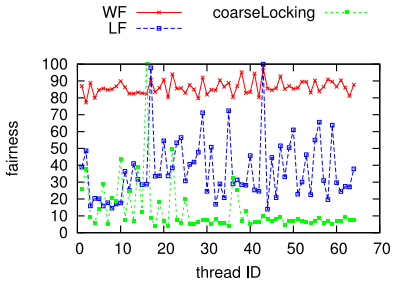

Fig. 13. Fairness (frn) for the $N \times M$ algorithm across threads.

observed in WF is 280 μs, and the jitter for LF is 30 μs for 64 threads.

The reasons for the roughly 7-10X difference between the LF and WF algorithms can be understood by taking a look at Table 1. It shows the number of requests on an average helped by any single request. We observe that in the WF algorithm, each request helps 7-13 other requests in its lifetime, whereas the number is much lower (1-2) for the LF algorithm. In WF, a request can perform both $external$ as well as $internal$ helping (see Section 4.2). As the number of threads increases $external$ helping increases dramatically. This also has a cascaded effect of $internal$ helping because if one request is stuck, multiple requests try to help it. As a result, the contention in the PATH and SHADOWPATH arrays increases. In comparison, LF algorithms have only internal helping. This results in lower contention and time per operation, at the cost of fairness.

Fig. 12 shows the results for fairness for a system with greater than 32 threads: ≈90 percent for WF, ≈35 percent for LF, and 15-25 percent for $coarseLocking$. In Fig. 13, we show the fairness values for each of the 64 threads in a representative 64-threaded run. From the figure, we can conclude that our wait-free implementation (WF) is very fair since the value of fairness is 85-95 percent for all the threads (due to external helping). In comparison, for LF, the mean fairness is 36 percent and in the case of $coarseLocking$ it is just 13 percent. The average deviation of $fairness$ for all the algorithms is within 5 percent (across all our runs).

A similar trend is observed in the case of the $1 \times M$ problem. Figs. 14 and 15 show the results for t_{req} and $fairness$. The LF algorithm is roughly 5-10× faster than the WF and $coarseLocking$ algorithms. The $coarseLocking$ algorithm is

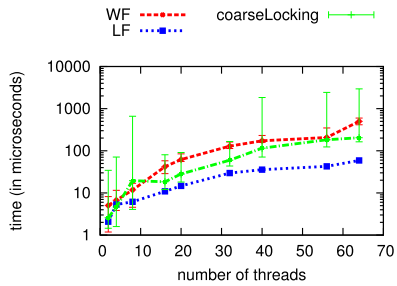

Fig. 14. t_{req} for the $1 \times M$ algorithm.

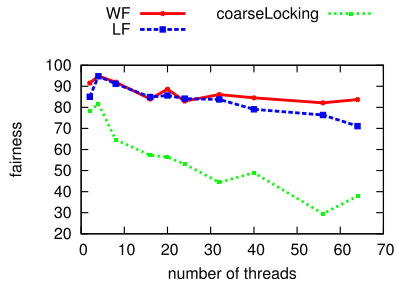

Fig. 15. Fairness (frn) for the $1 \times M$ algorithm.

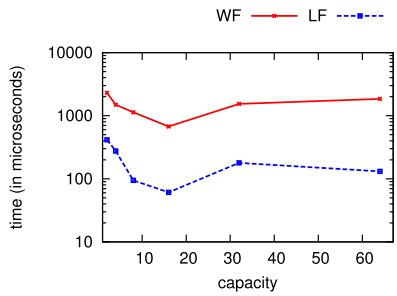

Fig. 16. t_{req} across different capacities.

nearly 2× faster than the WF algorithm on an average. However, the variation observed for $coarseLocking$, for more than 40 threads is very high. t_{req} is around 3,000 μs for 64 threads in the worst case (15 times the mean). Whereas, the LF and WF algorithms are very stable and the variations are within 15 percent of the mean values. In the worst case, t_{req} is around 70 μs and 600 μs for 64 threads for LF and WF respectively. Now, if we look at the $fairness$ of the system as a whole in Fig. 15, we observe that the $fairness$ values for WF remains at more than 80 percent. For up to 32 threads, $fairness$ of WF and LF is nearly same. Beyond 32 threads, $fairness$ is around 70 percent for LF. For $coarseLocking$, the $fairness$ remains within 30-50 percent (for $>$ 32 threads). The average deviation of $fairness$ for all the algorithms was within 3 percent.

7.3 Throughput

Next, we study the throughput of our slot scheduler by varying the average time between the arrival of two requests from 0 to 5 μs at intervals of 1 μs. Fig. 17 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 scales as the number of threads increases from 1 to 64. WF has comparable throughputs till 16 threads since

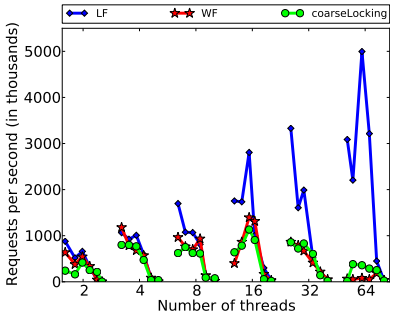

Fig. 17. Request throughput for the $N \times M$ algorithm.

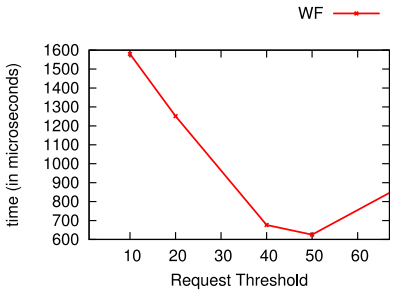

Fig. 18. t_{req} for WF across different values of REQUESTTHRESHOLD.

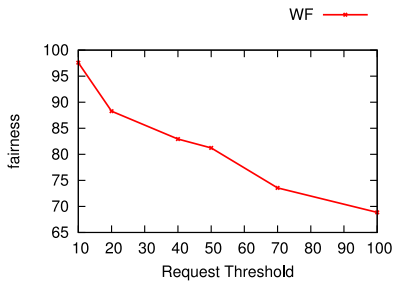

Fig. 19. Fairness (frn) for WF across different values of REQUESTTHRESHOLD.

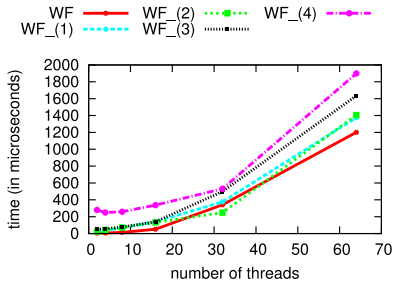

Fig. 20. t_{req} of WF for the $N \times M$ algorithm with varied CAS latencies.

the fairness value of each thread is as high as 94 percent, and then it becomes inferior to the LF algorithm. The throughput of WF does not scale beyond 16 threads because ensuring fairness proves to be very expensive. A lot of computational bandwidth is wasted in helping slower tasks, and thus throughput suffers. In comparison, LF keeps on getting better. For 64 threads its throughput is around 4,000k requests per seconds. The $coarseLocking$ algorithm has around 30 percent less throughput as compared to WF till 16 threads. Our wait-free algorithm has almost similar throughputs as the $coarseLocking$ algorithm (16-48 threads), with much stronger progress guarantees.

7.4 Sensitivity

7.4.1 Sensitivity: Varying Capacity (N)

Fig. 16 shows the time per request with different capacities (N) for LF and WF with the number of threads set to 64. We observe that as the number of resources increases, the time per request decreases. In the case of the LF algorithm, the time per request t_{req} drops to 60 from 414 μs as the number of resources increases(N) increase from 2-16. Similarly, we observe that t_{req} for the WF algorithm drops to around 675 μs as the number of resources(N) nears 16. When the number of resources is low, more threads compete with each other for the same time slot. This results in more cancellations, which leads to more wasted work. Hence, the time per request is more when we have fewer resources. Now, as we increase the number of resources(N) beyond 16, t_{req} increases since the time required to search for a free slot in a column increases.

7.4.2 Sensitivity: Varying REQUESTTHRESHOLD

We show the performance of WF, for 64 threads, for different values of the REQUESTTHRESHOLD. The parameter, REQUESTTHRESHOLD controls the amount of external helping being done by a thread. A lower value of REQUESTTHRESHOLD means that a thread needs to help more threads in its iterations. More helping would result in more time per request. Fig. 18 shows that as REQUESTTHRESHOLD varies from 10-50, t_{req} decreases from 1,550 μs to roughly 650 μs. We observe that *fairness* also decreases from 97 to 68 percent as the REQUESTTHRESHOLD varies from 10-100 (see Fig. 19). As REQUESTTHRESHOLD increases, a thread helps fewer requests in completing their operation. Thus, the *fairness* of the system decreases. We observe that 50 is the optimal value for the REQUESTTHRESHOLD.

We conclude our analysis by evaluating the sensitivity of the WF algorithm with respect to the compare-and-set instruction's latency. We added a few extra cycles to CAS latencies in our experiment by running a dummy loop. Fig. 20 shows the results. WF, $WF\_(1)$, $WF\_(2)$, $WF\_(3)$ and $WF\_(4)$ correspond to a system with 0, 1, 2, 3 and 4 additional μs for each CAS operation. t_{req} is nearly the same for WF, $WF\_(1)$ and $WF\_(2)$ (within 10 percent). Whereas $WF\_(3)$ and $WF\_(4)$ are 1.3× and 1.6× slower than WF for 64 threads. This experiment shows that our wait-free algorithm is fairly well tolerant to the latency of the underlying CAS instruction.

8 Conclusion

In this paper, we presented lock-free and wait-free algorithms for two variants of the generic slot scheduling problem. The solutions for the $1 \times M$ and $N \times M$ variants of the problem are fairly elaborate: they use recursive helping, and have fine grained co-ordination among threads. We consider both the $schedule$ and $free$ methods that can Dynamically reserve and free a set of slots in contiguous columns of the slot matrix. We additionally prove the linearizability correctness condition for all of our algorithms, and lastly experimentally evaluate their performance. The wait-

free and *coarseLocking* versions are slower than the lock-free version by 7-10× in almost all the cases. The performance of the wait-free algorithm is roughly similar to the version with locks. However, it provides significantly more fairness and suffers from 25× less jitter than the algorithms with locks.

REFERENCES

[1] P. Aggarwal and S. Sarangi, "Lock-free and wait-free slot scheduling algorithms," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, 2013, pp. 961–972.

[2] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich,"An analysis of linux scalability to many cores," in *Proc. 9th USENIX Conf. Operating Syst. Des. Implementation*, 2010, pp. 1–16.

[3] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: Architectural support for fine-grained parallelism on chip multiprocessors," in *Proc. 34th Annu. Int. Symp. Comput. Archit.*, 2007, pp. 162–173.

[4] P. E. Mckenney, J. Appavoo, A. Kleen, O. Krieger, O. Krieger, R. Russell, D. Sarma, and M. Soni,"Read-copy update," in *Proc. Ottawa Linux Symp.*, 2001, pp. 338–367.

[5] A. Kogan and E. Petrank, "Wait-free queues with multiple enqueuers and dequeuers," in *Proc. 16th ACM Symp. Principles Practice Parallel Program.*, 2011, pp. 223–234.

[6] B. Hall, "Slot scheduling: General purpose multiprocessor scheduling for heterogeneous workloads," Master's thesis, Univ. Texas, Austin, TX, USA, Dec. 2005.

[7] J. K. Ousterhout, "Scheduling techniques for concurrent systems," in *Proc. 3rd Int. Conf. Distrib. Comput. Syst.*, 1982, pp. 22–30.

[8] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Language Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[9] P. Aggarwal, G. Yasa, and S. R. Sarangi, "Radir: Lock-free and wait-free bandwidth allocation models for solid state drives, " in *Proc. IEEE Int. Conf. High Perform. Comput.*, 2014.

[10] S. W. Schlosser, J. Schindler, S. Papadomanolakis, M. Shao, A. Ailamaki, C. Faloutsos, and G. R. Ganger, "On multidimensional data and modern disks," in *Proc. 4th Conf. USENIX Conf. File Storage Technol.*, 2005, p. 17.

[11] M.-Y. Wu, "On runtime parallel scheduling for processor load balancing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, no. 2, pp. 173–186, Feb. 1997.

[12] E. Dekel and S. Sahni, "Parallel scheduling algorithms," *Operations Res.*, vol. 31, no. 1, pp. 24–49, 1983.

[13] J. E. Moreira, W. Chan, L. L. Fong, H. Franke, and M. A. Jette, "An infrastructure for efficient parallel job execution in terascale computing environments, " in *Proc. IEEE/ACM Conf. Supercomput.*, 1998, p. 50.

[14] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger, "Argon: Performance insulation for shared storage servers," in *Proc. 5th USENIX Conf. File Storage Technol.*, 2007, p. 5.

[15] J. H. Anderson and M. Moir, "Wait-free synchronization in multiprogrammed systems: Integrating priority-based and quantum-based scheduling," in *Proc. 18th Annu. ACM Symp. Principles Distrib. Comput.*, 1999, pp. 123–132.

[16] S. Park and K. Shen, "FIOS: A fair, efficient flash i/o scheduler," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, p. 13.

[17] J.-M.Liang, J.-J. Chen, H.-C. Wu, andY.-C. Tseng, "Simple and regular mini-slot scheduling for IEEE 802.16d grid-based mesh networks, " in *Proc. IEEE 71st Veh. Technol. Conf.*, 2010, pp. 1–5.

[18] S. R. Rathnavelu, "Adaptive time slot scheduling apparatus and method for end-points in an atm network," US Patent 6 205 118, Mar. 20, 2001.

[19] I. Goiri, K. Le, M. Haque, R. Beauchea, T. Nguyen, J. Guitart, J. Torres, andR. Bianchini, "GreenSlot: Scheduling energy consumption in green datacenters," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2011, pp. 1–11.

[20] P. Aggarwal and S. R. Sarangi, "Software transactional memory friendly slot schedulers," in *Proc. 10th Int. Conf. Distrib. Comput. Internet Technol.*, 2014, pp. 79–85.

[21] H. Sundell, "Wait-free multi-word compare-and-swap using greedy helping and grabbing," *Int. J. Parallel Program.*, vol. 39, no. 6, pp. 694–716, 2011.

[22] Y. Afek, M. Merritt, G. Taubenfeld, and D. Touitou, "Disentangling multi-object operations," in *Proc. 16th Annu. ACM Symp. Principles Distrib. Comput.*, 1997, pp. 111–120.

[23] T. L. Harris, K. Fraser, and I. A. Pratt, "A practical multi-word compare-and-swap operation," in *Proc. 16th Int. Conf. Distrib. Comput.*, 2002, pp. 265–279.

[24] S. Sellke, N. B. Shroff, S. Bagchi, and C.-C. Wang, "Timing channel capacity for uniform and gaussian servers," in *Proc. Allerton Conf.*, 2006.

[25] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*. Amsterdam, The Netherlands: Elsevier, 2012.

Pooja Aggarwal received the BE degree in information technology in 2009 from the Punjab Engineering College. She is currently working toward the PhD degree in the Department of Computer Science and Engineering, IIT Delhi, India. She was a software developer for two years with Aricent Technologies. Her main research interests include lock-free and wait-free algorithms.

Smruti R. Sarangi received the BTech degree in computer science from IIT Kharagpur, India, in 2002, and the MS and PhD degrees in computer architecture from the University of Illinois at Urbana-Champaign in 2007. He is an assistant professor in the Department of Computer Science and Engineering, IIT Delhi, India. He works in the areas of computer architecture, parallel and distributed systems. He is a member of the IEEE and ACM.