

Lock-free and Wait-free Slot Scheduling Algorithms

Pooja Aggarwal and Smruti R. Sarangi
 Department of Computer Science & Engineering,
 Indian Institute of Technology, New Delhi, India
 e-mail: {pooja.aggarwal,srsarangi}@cse.iitd.ac.in

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.

1 INTRODUCTION

Large¹ 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 Figure 1).

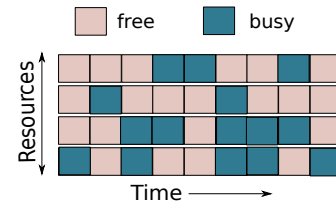


Fig. 1: The Ousterhout matrix for scheduling

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

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

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 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*. The most common correctness guarantee for such *non-blocking* algorithms is *linearizability* (see Appendix A.2), 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 et. al. [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 et al. [15] propose a more generic scheme that also takes task priority into account. Stan et al. [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).

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 **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 Figure 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 5 contiguous slots in an instance of the $1 \times M$ problem. After we have booked the first 3 slots, we cannot be sure of the availability of the last 2 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 Figure 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 Figure 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

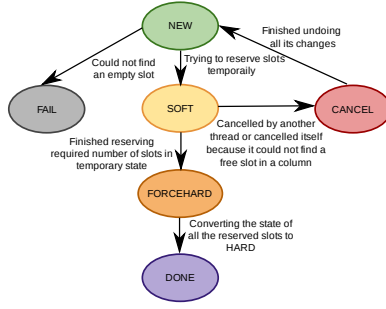


Fig. 2: Finite state machine of a *schedule* request

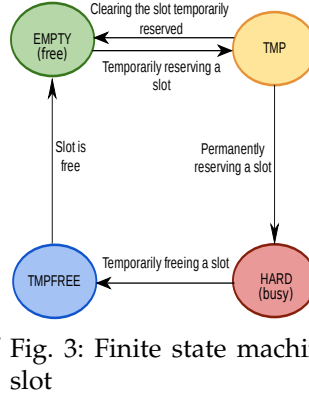


Fig. 3: Finite state machine of a *slot*

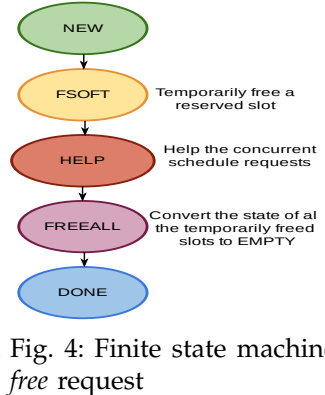


Fig. 4: Finite state machine of a *free* request

HARD to TMPFREE to EMPTY.

5 SLOT SCHEDULING ALGORITHM

5.1 Data Structures

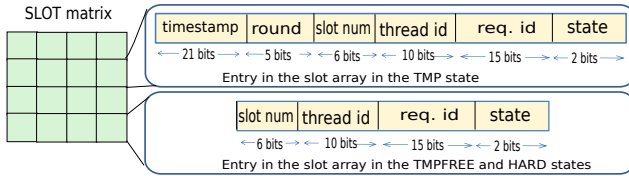


Fig. 5: The SLOT matrix

The SLOT matrix represents the Ousterhout matrix of slots (see Figure 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 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 Figure 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

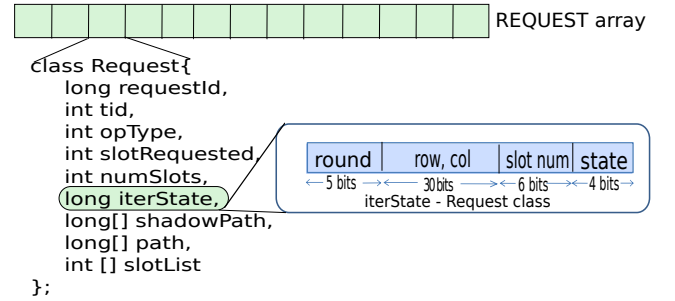


Fig. 6: The REQUEST array

number of threads in the system. The REQUEST array contains instances of the *Request* class (see Figure 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 slot in the SLOT matrix (row, col), number of slots reserved, and the state of the request (as shown in Figure 6).

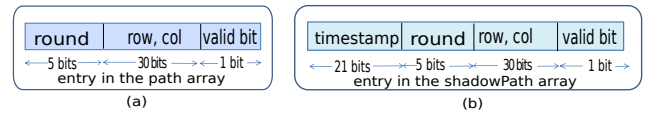


Fig. 7: The PATH and SHADOWPATH arrays

Let us now describe two more fields of the *Request* class: the PATH and SHADOWPATH arrays (see Figure 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 (compare-and-set) 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 Figure 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.

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.getRequestId())
        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. 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).

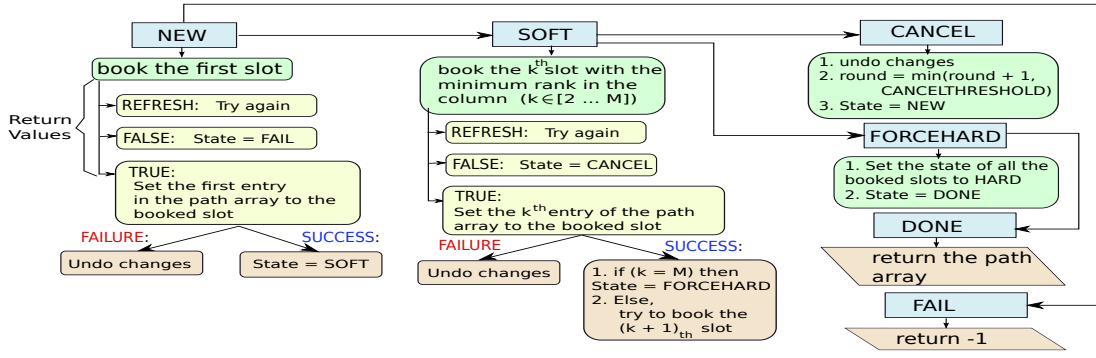
5.3.1 The `processSchedule` method

We show an overall flowchart of the `processSchedule` function in Figure 8. It extends Figure 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, `col1`, of the `SLOT` matrix. We ensure that all the slots between columns `col` (requested starting column) and `col1` 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 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

Fig. 8: The *processSchedule* function

request is successfully set to **FORCEHARD**, then it is the point of linearization for the successful *schedule* call (see Appendix A).

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** → **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 $\min(\text{round} + 1, \text{CANCELTHRESHOLD})$. 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).

Algorithm 2: processSchedule $N \times M$

```

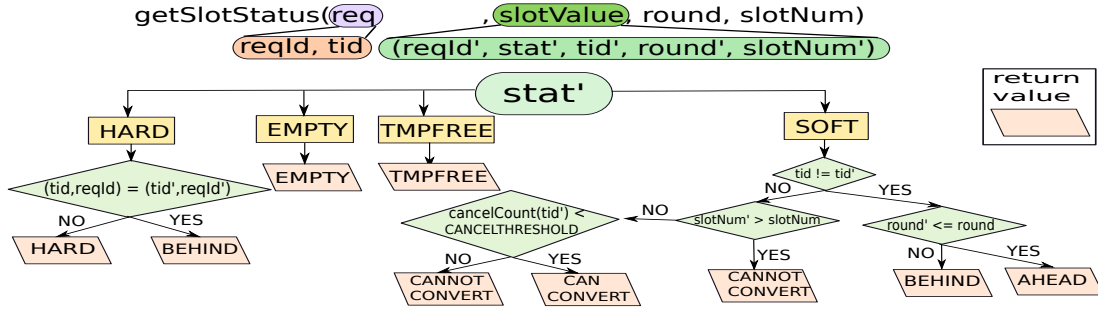
1: function processSchedule (Request req)
2:   Data
3:   col, row0 ← the requested slot
4:   state ← the current state of the requested
5:   nstate ← the new state of the request
6:   slotNum ← number of slots reserved so far
7:   while TRUE do
8:     (state, slotNum, round, row0, col) ← unpack
       (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 suc-
              cessfully 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 if
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()) < CANCELTHRESH-
           OLD 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)

```

Fig. 9: The *getSlotStatus* function

```

64: case FAIL :
65:     return -1
66: req.iterState.CAS(state, nstate)
67: end switch
68: end while
69: end function

```

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 Figure 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 ($slotNum \in [1 \dots M]$) 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 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.

```

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)? CAN-
      CONVERT: 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

```

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 Figure 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

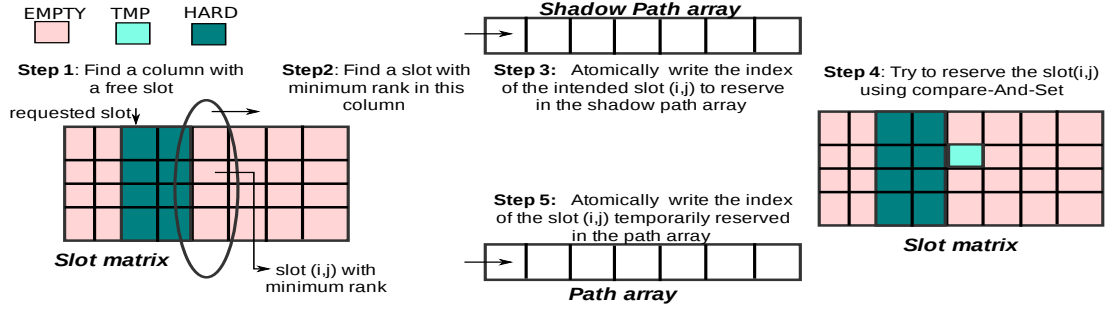


Fig. 10: Various steps involved in reserving a slot (temporarily)

thread t_j , column(col) to reserve a slot in, the number of the slot($slotNum \in [1 \dots M]$) 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.

```

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

```

5.4 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 *TMPPFREE*). 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 *TMPPFREE*, 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,TMPPFREE)
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(TMPPFREE, 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 for
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),TMPPFREE), 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(reqf, reqs, index)
54:   (round, row1, col1) ← unpack(reqs.PATH.get(0))
55:   slot ← reqf.slotList(index)
56:   (row,col) ← unpack(slot)
57:   if col < col1 then
58:     nstate ← pack(CANCEL,0,round,0,
       max(reqs.slotRequested, col - reqs.numSlots + 1))
59:     otherCancel(reqs, 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 its 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 known as the ABA 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 1024 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 × NUMTHREADS × 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 *W*, then we can set the size of the `SLOT` matrix to $S > 2W$, 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.

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 2.20GHz 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 = $3 \times \text{tid}$). 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. $\text{frn} = \text{tot_requests} / (\kappa \times \text{NUMTHREADS})$. *frn* measures the degree of imbalance across different threads. It varies from $1/\text{NUMTHREADS}$ (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% of the requests are *schedule* operations, and the

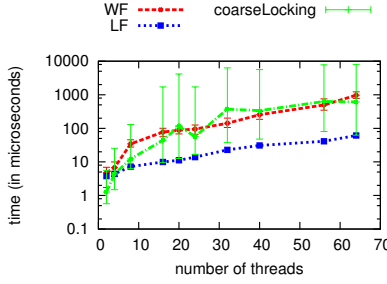


Fig. 11: t_{req} for the $N \times M$ algorithm

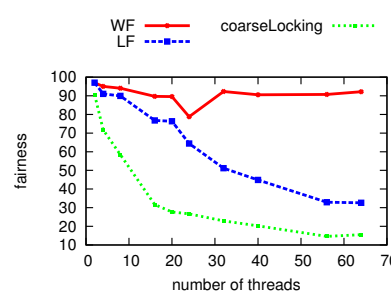


Fig. 12: Fairness (frn) for the $N \times M$ algorithm

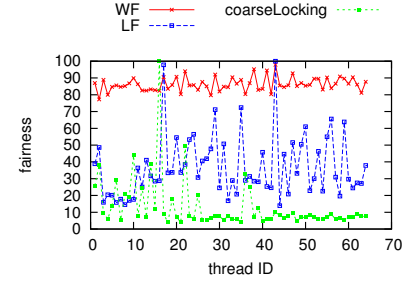


Fig. 13: Fairness (frn) for the $N \times M$ algorithm across threads

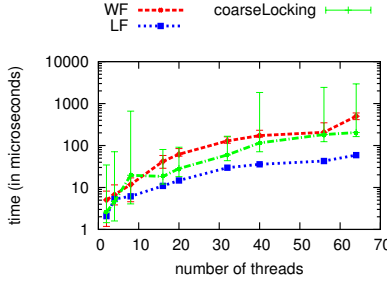


Fig. 14: t_{req} for the $1 \times M$ algorithm

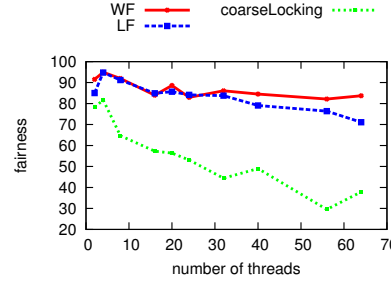


Fig. 15: Fairness (frn) for the $1 \times M$ algorithm

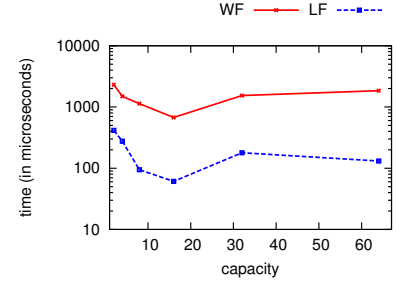


Fig. 16: t_{req} across different capacities

remaining 30% 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.

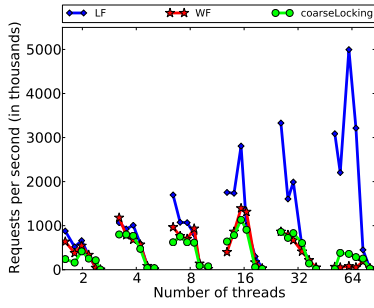
7.2 Performance of the $N \times M$ and $1 \times M$ Algorithms

Figures 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 8030 μs (mean: 620 μs) 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, 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) 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.

Figure 12 shows the results for fairness for a system with greater than 32 threads: $\approx 90\%$ for *WF*, $\approx 35\%$ for *LF*, and 15-25% for *coarseLocking*. In Figure 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% for all the threads (due to external helping). In comparison, for *LF*, the mean fairness is 36% and in the case of *coarseLocking* it is just 13%. The average **deviation** of *fairness* for all the algorithms is within 5% (across all our runs).

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

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

coarseLocking algorithm is nearly 2X 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 3000 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% 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 Figure 15, we observe that the *fairness* values for *WF* remains at more than 80%. For up to 32 threads, *fairness* of *WF* and *LF* is nearly same. Beyond 32 threads, *fairness* is around 70% for *LF*. For *coarseLocking*, the *fairness* remains within 30-50% (for > 32 threads). The average deviation of *fairness* for all the algorithms was within 3%.

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 . Figure 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 the *fairness* value of each thread is as high as 94%, 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 4000k requests per seconds. The *coarseLocking* algorithm has around 30% 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)

Figure 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

Number of Threads	<i>WF</i>		<i>LF</i>
	external helping	internal helping	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

TABLE 1: Number of requests helped by a single request

resources increases, the time per request decreases. In the case of the *LF* algorithm, the time per request t_{req} drops to 60 μs 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. Figure 18 shows that as REQUESTTHRESHOLD varies from 10-50, t_{req} decreases from 1550 μs to roughly 650 μs . We observe that *fairness* also decreases from 97% to 68% as the REQUESTTHRESHOLD varies from 10-100 (see Figure 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(CAS) instruction's latency. We added a few extra cycles to CAS latencies in our experiment by running a dummy loop. Figure 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%). Whereas *WF*_(3) and *WF*_(4) are 1.3X and 1.6X 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

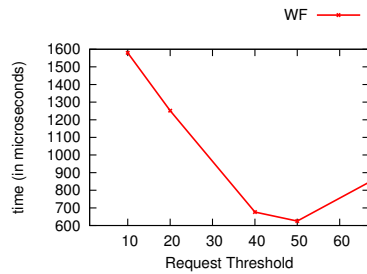


Fig. 18: t_{req} for WF across different values of REQUESTTHRESHOLD

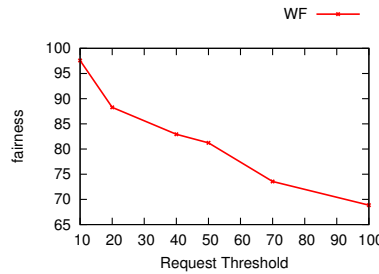


Fig. 19: Fairness (frn) for WF across different values of REQUESTTHRESHOLD

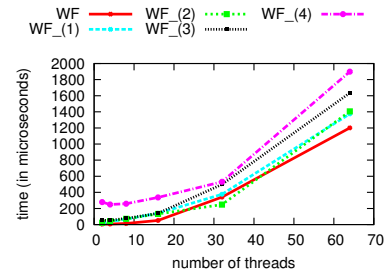


Fig. 20: t_{req} of WF for the $N \times M$ algorithm with varied CAS latencies

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 7-10X 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 25X less jitter than the algorithms with locks.**

REFERENCES

- [1] P. Aggarwal and S. Sarangi, "Lock-free and wait-free slot scheduling algorithms," in *IPDPS*, 2013.
- [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 *OSDI*, 2010, pp. 1–16.
- [3] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors," in *ISCA*, 2007.
- [4] P. E. Mckenney, J. Appavoo, A. Kleen, O. Krieger, O. Krieger, R. Russell, D. Sarma, and M. Soni, "Read-Copy Update," in *In Ottawa Linux Symposium*, 2001, pp. 338–367.
- [5] A. Kogan and E. Petrank, "Wait-free Queues With Multiple Enqueuers and Dequeuers," in *PPoPP*, 2011.
- [6] B. Hall, "Slot Scheduling: General Purpose Multiprocessor Scheduling for Heterogeneous Workloads," Master's thesis, University of Texas, Austin, december 2005.
- [7] J. K. Ousterhout, "Scheduling Techniques for Concurrent Systems," in *ICDCS*, 1982.
- [8] 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.
- [9] P. Aggarwal, G. Yasa, and S. R. Sarangi, "Radir: Lock-free and wait-free bandwidth allocation models for solid state drives," in *HiPC*, 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 *FAST*, 2005.
- [11] M.-Y. Wu, "On Runtime Parallel Scheduling for Processor Load Balancing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 8, no. 2, pp. 173–186, feb 1997.
- [12] E. Dekel and S. Sahni, "Parallel scheduling algorithms," *Operations Research*, 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 *SC*, 1998.
- [14] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger, "Argon: Performance insulation for shared storage servers," in *FAST*, 2007.
- [15] J. H. Anderson and M. Moir, "Wait-free synchronization in multiprogrammed systems: Integrating priority-based and quantum-based scheduling," in *PODC*, 1999.
- [16] S. Park and K. Shen, "Fios: a fair, efficient flash i/o scheduler," in *FAST*, 2012.
- [17] J.-M. Liang, J.-J. Chen, H.-C. Wu, and Y.-C. Tseng, "Simple and Regular Mini-Slot Scheduling for IEEE 802.16d Grid-based Mesh Networks," in *VTC*, 2010.
- [18] S. R. Rathnavelu, "Adaptive time slot scheduling apparatus and method for end-points in an atm network," Mar. 20 2001, US Patent 6,205,118.
- [19] I. Gori, K. Le, M. Haque, R. Beauchea, T. Nguyen, J. Guitart, J. Torres, and R. Bianchini, "GreenSlot: Scheduling Energy Consumption in Green Datacenters," in *High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.
- [20] P. Aggarwal and S. R. Sarangi, "Software transactional memory friendly slot schedulers," in *Distributed Computing and Internet Technology*. Springer, 2014, pp. 79–85.
- [21] H. Sundell, "Wait-free multi-word compare-and-swap using greedy helping and grabbing," *International Journal of Parallel Programming*, vol. 39, no. 6, pp. 694–716, 2011.
- [22] Y. Afek, M. Merritt, G. Taubenfeld, and D. Touitou, "Disentangling multi-object operations," in *PODC*, 1997.
- [23] T. L. Harris, K. Fraser, and I. A. Pratt, "A practical multi-word compare-and-swap operation," in *Distributed Computing*, 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 *Proceedings of the Allerton Conference*, 2006.
- [25] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.



Pooja Aggarwal is a Ph.D student in the Department of Computer Science and Engineering, IIT Delhi, India. She has worked as a software developer for 2 years with Aricent Technologies. She received a B.E in Information Technology in 2009 from the Punjab Engineering College. Her main research interests are lock-free and wait-free algorithms.



Smruti R. Sarangi is an Assistant Professor in the Department of Computer Science and Engineering, IIT Delhi, India. He graduated with a M.S and Ph.D in computer architecture from the University of Illinois at Urbana-Champaign in 2007, and a B.Tech in computer science from IIT Kharagpur, India, in 2002. He works in the areas of computer architecture, parallel and distributed systems. Prof. Sarangi is a member of the IEEE and ACM.

APPENDIX A PROOFS

In this appendix we present the proof of correctness and the proof of lock/wait-freedom of the algorithms described in Section 5. Let us start with some definitions.

Definition 1: A requestId is a unique identifier for each request.

Definition 2: A request is an operation to either *reserve* or *free* a set of slots in contiguous columns in the `SLOT` matrix using the methods `processSchedule` or `processFree`.

Let us now try to characterize what kind of schedules are *legal* for a single thread (sequential) scheduler.

A.1 Legal Sequential Specification

For producing legal schedules, a parallel schedule needs to obey conditions 1 and 2.

Condition 1

Every request should be scheduled at the earliest possible time.

Condition 2

Every request should book only `numSlots` entries in consecutive columns: one slot per each column.

However, for a parallel scheduler, sequential consistency and producing legal schedules is not enough. Let us consider the previous example (Section 4.1), and assume that request 2 arrives a long time after request 1, and these are the only requests in the system. Then we intuitively expect request 1 to get slots (1-3), and request 2 to get slots (4-6). However, sequential consistency would allow the reverse result. Hence, we need a stronger correctness criteria that keeps the time of request arrival in mind. This is called *linearizability* [8], and is one of the most common correctness criteria for parallel shared data structures.

A.2 Linearizability

Let us define a *history* as a chronological sequence of events in the entire execution. Formally, history $H \in (T, E, i, V^*)^*$. Here, T denotes the thread id, E denotes the event (invocation or response), i denotes a sequence number, and V denotes the return value/arguments of a method. We define only two kinds of events in our system namely invocations(inv) and responses(resp). A matching invocation and response have the same sequence number, which is unique. We refer to a invocation-response pair with sequence number i as request r_i . Note that in our system, every invocation has exactly one matching response, and vice versa, and needless to say a response needs to come after its corresponding invocation.

A request r_i precedes request r_j , if r_j 's invocation comes after r_i 's response. We denote this by $r_i \prec r_j$. A

history, H , is *sequential* if an invocation is immediately followed by its response. We define the term *subhistory* $(H|T)$ as the subsequence of H containing all the events of thread T . Two histories, H and H' , are equivalent if $\forall T, H|T = H'|T$. Furthermore, we define a complete history – *complete*(H) – as a history that does not have any pending invocations.

Let the set of all sequential histories that are correct, constitute the *sequential specification* of a scheduler. A history is *legal* if it is a part of the sequential specification and it is characterized by conditions 1 and 2. Typically for concurrent objects, we define their correctness by a condition called *linearizability* given by the following conditions.

Condition 3

A history H is linearizable if *complete*(H) is equivalent to a legal sequential history, S .

Condition 4

If $r_i \prec r_j$ in *complete*(H), then $r_i \prec r_j$ in S (sequential history) also.

To prove conditions 3 and 4, it is sufficient to show that there is a unique point between the invocation and response of a method at which it appears to execute instantaneously [25]. This point is known as the *point of linearization* or the *point of linearizability*. This further means that before the point of linearization, changes made by the method are not visible to the external world, and after the point, all the changes are immediately visible. These changes are irrevocable. Let us call this condition 5.

Condition 5

Every method call appears to execute instantaneously at a certain point between its invocation and response.

To summarize, we need to prove that the execution history of the `parSlotMap` data structure is both legal (conditions 1 and 2), and linearizable (condition 5).

Theorem 1: The $N \times M$ schedule – LF and WF algorithms take effect instantaneously.

Proof: We need to prove that there exists a point at which the `processSchedule` function appears to execute instantaneously.

Let us try to prove that this point of linearization of a thread, t , is Line 38 when the state of the request is successfully changed to `FORCEHARD`, or it is Line 14 when the request fails because of lack of space (see Algorithm 2). Note that before the linearization point, it is possible for other threads to cancel thread t using the `otherCancel` function. However, after the status of the request has been set to `FORCEHARD`, 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 `NEW` and `SOFT` states (see Appendix B.4).

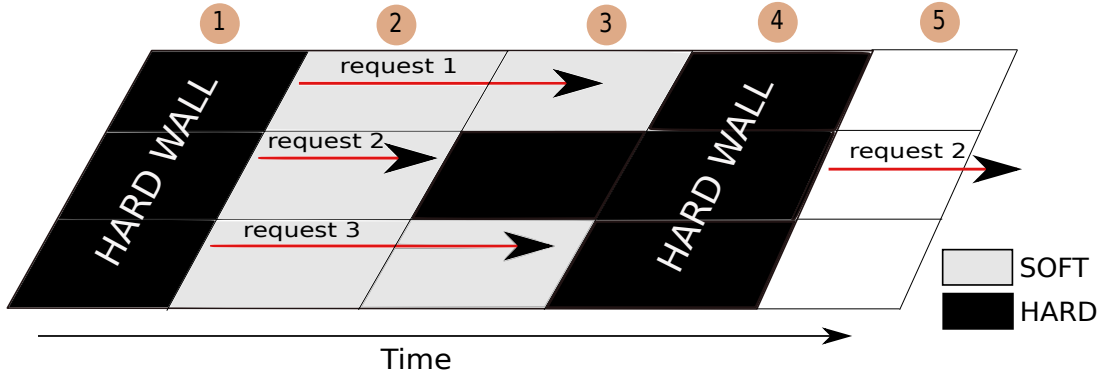


Fig. 21: An example containing three requests

Hence, the point of linearization (Line 38) 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 matrix.

Likewise, we need to prove that before the point of linearization, no events visible to other threads cause them to make permanent changes. Before the point of linearization, another thread (say t) can view temporarily reserved entries. It can perform two actions in response to a temporary reservation – decide to help the thread that has reserved the slot, or cancel the thread. In either case, t does not change its starting position.

A thread will change its starting position in Line 29, only if it is not able to complete its request at the current starting position because of a hard wall. Recall that a *hard wall* is defined as a column consisting of only **HARD** entries. We show an example in Figure 21. In this figure there are three requests – 1, 2, and 3, and each of them needs 2 slots. Assume that requests 1 and 3 are able to complete and convert the state of their slots to **HARD**. Then request 2 will find column 3 to be a hard wall. Since column 4 is also a hard wall it will restart from column 5.

Note that a hard wall is created 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 next column after the hard wall without sacrificing linearizability. Additionally, it cannot force other threads to change their starting position (unalterable change in their behavior).

Thus, we can treat the point of linearization as Lines 38 and 14. We have thus proven that the schedule function appears to execute instantaneously. Alternatively, this means that the execution history is equivalent to a sequential history. However, we have still not proven that it is a legal sequential history. Once, we prove the latter, we can prove that the *processSchedule* operation is linearizable. \square

Let us now prove that in the absence of *free* requests, we always produce legal schedules.

Lemma 1: Every request is scheduled at the earliest possible time slot when there are no concurrent free operations. Alternatively, the equivalent sequential history of an execution with respect to the *schedule* requests is legal.

Proof: Since our algorithms take effect instantaneously (Theorem 1), the parallel execution history is equivalent to a sequential history. We need to prove that in this sequential history, a request is scheduled at the earliest possible slot, or alternatively, the starting slot has the least possible permissible value. If a request is scheduled at its starting slot, then this lemma is trivially satisfied. If it is not the case, then we note that the starting slot changes in Line 29 (of Algorithm 2) only if the request encounters a hard wall. This means that it is not possible to schedule at the given slot. The earliest possible starting slot, is a slot in a column immediately after the hard wall. If the request can be satisfied with this new starting slot, then the lemma is satisfied. Using mathematical induction, we can continue this argument, and prove that the slot at which a request is finally scheduled is the earliest possible slot. \square

Let us now prove two lemmas for the *schedule* operation, when there are concurrent *free* requests. We define a *free* request to be *conflicting* with a *schedule* request, if the *schedule* request can possibly reserve slots freed by the *free* request.

Lemma 2: A linear scan of the **REQUEST** array is sufficient to notify a concurrent *schedule* request to take the slots being freed by a *free* request.

Proof: A *free* request after temporarily freeing all the slots enters its **HELP** phase (Line 17:Algorithm 3). In this phase the request checks each entry in the **REQUEST** array and tries to find a *schedule* request, which is not linearized yet, and can reserve the slots freed by the *free* request. Assume that while scanning the **REQUEST** array, the *free* request reaches the i^{th} entry. Let a new *schedule* request, req_j , get placed at the j^{th} entry, where $j < i$. Assume it can reserve the slots freed by the *free* request.

Since a *free* request has already reached the i^{th} entry, it has not notified req_j about the slots being freed. We need to prove that req_j will still be able to see the slots freed by the *free* request even if req_j was not notified by the *free* request.

The list of actions in the algorithm are as follows:

Step 1: The *free* request first changes the status of the slots it wishes to free from **HARD** to **TMPFREE** and changes its state to **HELP**.

Step 2: The *free* request starts scanning the **REQUEST** array.

Step 3: A new *schedule* request (req_j) enters the system using an atomic set instruction.

All these three operations are atomic. Step 1 follows Step 2 since these two operations are issued by the same thread. Step 2 and 3 can either be ordered as $2 \rightarrow 3$ or $3 \rightarrow 2$. Let us first discuss the case $2 \rightarrow 3$. In this case the *schedule* request req_j will see the slots freed by the *free* request, and they will either be in the **TMPFREE** or **EMPTY** states. If the slot is in **EMPTY** state, request req_j will try to reserve it and if the state of the slot is **TMPFREE**, req_j will help the *free* request.

Let us now consider the case: $3 \rightarrow 2$. If a *schedule* request is placed before a *free* request starts the scan then the *free* request will find req_j in the **REQUEST** array, and it will be able to take the slots freed by it. Hence, in both the cases, a *schedule* request is aware about a concurrent and conflicting *free* request. \square

Lemma 3: A *schedule* request does not miss any valid slots in the presence of concurrent *free* operations.

Proof: Slots which are in the progress of getting freed due to a *free* request are valid slots for scheduling any other request. For a *schedule* request req_s to not miss any valid slots, the slots freed by a concurrent and conflicting *free* request should be made available to the *schedule* request. In other words, a *schedule* request should be able to reserve the slots freed by a *free* request, which has executed before it as per the equivalent sequential history.

Let us say a *schedule* request req_s enters the system and changes its starting slot on seeing a hard wall at column col and then sleeps. Meanwhile a *free* request req_f comes in and breaks the hard wall at column col , while freeing a set of slots, and finishes its operation. Later, req_s wakes up and finishes its operation and leaves the system. Now, as per the equivalent sequential history request, req_f completed before req_s . So if these two requests are conflicting, req_s should reserve a slot in column col . As request req_s has already changed its starting slot to a column after col , it will not get the slot freed by req_f in the column col . To avoid this problem, every *free* request helps conflicting *schedule* requests upon breaking a hard wall and then it moves to the **FREEALL** state.

Each *free* request will check whether while freeing a

slot it breaks a hard wall or not (Line 25:Algorithm 3). Assume that a *free* request, req_f , breaks a hard wall while freeing a slot s in column col . In this scenario req_f linearly scans the **REQUEST** array to see if some *schedule* request (not linearized yet (Line 29:Algorithm 3)) can be scheduled in a slot s in column col . As proved in Lemma 2, a linear scan ensures that request req_s will not miss the slots freed by req_f . Now, req_f will help the request req_s in getting scheduled earlier.

In case the *schedule* request req_s sees a slot in the **TMPFREE** state, then it tries to help the request req_f in completing its operation (Line 134:Algorithm 2). In both the cases, the *free* request is linearized before the *schedule* request. Thus if a *schedule* request takes the slots freed by a *free* request, correctness is not violated. Hence, we can say that a *schedule* request reserves the earliest possible slots even in the presence of concurrent *free* requests. \square

Now, that we have proved that we get the starting point right for each request, let us prove that the rest of the slots in a request are reserved correctly.

Lemma 4: Every request books $numSlots$ entries in consecutive columns: one slot per each column.

Proof: We need to prove that for a request, r , exactly $numSlots$ entries are allocated in consecutive columns with one entry per column. Line 27 (of Algorithm 2) ensures that the columns are consecutive because we always increment them by 1.

To ensure that exactly one request is booked per column by a thread and all of its helpers, we use the *path* and *shadowPath* arrays. Different threads first indicate their intent to book a certain slot by entering it in the *shadowPath* array (Line 98:Algorithm 2). One of the threads will succeed. All the other helpers will see this entry, and try to book the slot specified in the *shadowPath* entry. Note that there can be a subtle ABA issue (see Section 5.5) here. It is possible that the thread, which set the *shadowPath* entry might find the slot to be occupied, whereas other helpers might find it to be empty because of a cancellation. This is readily solved by associating a timestamp with every slot in the **SLOT** matrix. Since we are not booking an extra slot in the column for the current round, we can conclude that a column will not have two slots booked for the same request and round. It is possible that some slots might have been booked in previous cancelled rounds, and would not have been cleaned up. However, the thread that was cancelled will ultimately clean them up. This does not deter other requests, because they will see that the zombie slots belong to an older round of a request, and they can be recycled. Thus, the statement of the lemma is true. \square

Theorem 2: The $N \times M$ *schedule* -*LF* algorithm is legal, linearizable, and lock-free.

Proof: Theorem 1 and, Lemma 1, 3 and 4 establish the fact that $N \times M$ -*LF* is legal and linearizable. We need

to prove lock freedom. Since we use only non-blocking primitives, it is possible to have a live-lock, where a group of threads do not make forward progress by successfully scheduling requests. Let us assume that thread, t_i , is a part of a live-lock. For the first CANCELTHRESHOLD times, t_i will get cancelled. Subsequently, it will start helping some other request, r . t_i can either successfully schedule r , or transition to helping another request. Note that in every step, the number of active requests in the system is decreasing by one. Ultimately, thread t_i will be helping some request that gets scheduled successfully because it will be the only active request in the system. This leads to a contradiction, and thus we prove that $N \times M-LF$ is lock-free. \square

Theorem 3: The $N \times M$ schedule-WF algorithm is wait-free.

Proof: By Theorem 2, we have established that the *processSchedule* algorithm is lock-free. To make this algorithm wait-free, we use a standard technique based on an universal construction (see [25]). The threads first announce their requests by entering their requests in the REQUEST array. When a new request arrives it checks if there is any request whose id differs by more than REQUESTTHRESHOLD. If there is any request, then this request is helped.

Assume that it is possible for a request to wait indefinitely. Let the request with the lowest request id in the system by req_{min} . Since our algorithms are lock-free some requests are always making progress. Their request ids are increasing. Ultimately, their request ids will exceed the request id of req_{min} by more than REQUESTTHRESHOLD. Thus, these threads will begin to help req_{min} . It is not possible for any subsequent request to complete without helping req_{min} to complete. Ultimately, a time will come when all the requests would be helping req_{min} . Since our algorithm is lock-free, one of the threads will succeed and req_{min} will get completed. Hence, we can conclude that no request will wait indefinitely, and thus our algorithm is wait-free. \square

Let us now prove that all the *free* methods are legal, linearizable, and lock-free (or wait-free).

Theorem 4: The $N \times M$ free-LF and WF algorithms appear to execute instantaneously.

Proof: We need to prove that there exists a point (called the point of linearization) at which the *free* function appears to execute instantaneously. Let us try to prove that the point of linearization of a thread, t , is Line 37 (of Algorithm 3) when the state of the request is successfully changed from HELP to FREEALL. Note that before the point of linearization, it is possible that for only some slots it is visible that the *free* operation is in progress (their status is TMPFREE). However, after the status of the request has been set to FREEALL, the slots which a *free* request wishes to free are made available to

all the conflicting *schedule* operations. Hence, the point of linearization (Line 37:Algorithm 3) ensures that after its execution, changes made by the request are visible as well as irrevocable: the slots will eventually get freed.

Let us now consider all the concurrent *schedule* requests that find slots in the TMPFREE states before the *free* operation has linearized. First, the *free* operation needs to get linearized, and then these *schedule* operations should get linearized. This will ensure that in the equivalent sequential history, the *free* appears first, and the *schedule* operations that use the slots that it has freed appear later. This is required to ensure a legal sequential history. In order to do so, a *schedule* request first helps the concurrent and conflicting *free* request in completing its operation and then the *schedule* request tries to reserve the slot. Note that no *schedule* request directly changes the state of the slot from TMPFREE to TMP. This ensures that the *schedule* request only takes the slots of those *free* requests which have been linearized before it. To summarize, this particular step ensure that if any *schedule* request sees a *free* request that is in progress and has not reached its point of linearizability, then it suspends itself, and helps the *free* request to complete. From the point of view of the *schedule* request, the effect is as if it has not seen the *free* request at all. This means that before a *free* request has linearized its temporary changes are effectively not visible.

A *schedule* request, which is already linearized will not alter its behavior on seeing a *free* request that is in progress. Next, we consider concurrent *free* requests. We assume that two *free* requests never contend on a slot, so one *free* request will not alter the behavior of another *free* request.

Thus, the *free* request appears to execute instantaneously at Line 37. \square

Theorem 5: The $N \times M$ free-WF algorithm is legal, linearizable, and wait-free.

Proof: Theorem 4 and Lemma 3 establish the fact that the *free* operation is legal and linearizable.

To prove that *free-WF* is wait-free, we need to prove that a *free* request is completed in a finite number of steps. After a *free* request enters the system, it starts by temporarily freeing the desired slots. During this phase a *free* and *schedule* request can collide. Firstly, a *free* request will continue temporarily freeing the desired slots (change their state from HARD to TMPFREE). Next, the *free* request helps colliding *schedule* requests in starting from new positions. The number of requests, which a *free* request can help is equivalent to the number of requests it collides with. The point of collision of two requests is the slot one aims to either *free* or *schedule*. No two *free* requests contend with each other. Now at each slot only one request can *schedule* so for each slot a *free* request will help only one *schedule* request. A *free* request will therefore help M requests, where M is the number of slots it wishes to free. This process a finite

number of steps.

After helping the requests it proceeds with permanently freeing the slots (changing the state from `TMPPFREE` to `EMPTY`). Since, at each phase the *free* request has a finite number of steps. Thus, *free-WF* is wait-free. \square

APPENDIX B

AUXILIARY FUNCTIONS

In this appendix, we give details of the functions used in the algorithm described in Section 5.

B.1 Functions to Book Slots

First, we discuss the *bookFirstSlot* method. This method is used for reserving the first slot of the *schedule* request with the help of the function *bookMinSlotInCol* (see Section 5.3.3). The `SLOT` matrix is scanned to find an available slot in a column. The search terminates as soon as an available slot (for more information see the *getSlotStatus* and *bookMinSlotInCol* functions in Section 5.3.2 and 5.3.3) is found or the state of the request is not `NEW`.

Next, the *findMinInCol* function is used to find a slot in a column with the minimum rank. It internally uses the *getSlotStatus* method to find the rank of each slot in a column. The slot with the minimum rank along with its timestamp is returned (see Algorithm 4 Lines 18- 30).

The *forcehardAll* function is used to change the status of all the temporarily reserved slots from `TMP` to `HARD`. We also save number of slots that have been reserved for a request (say *req*) in subsequent columns. This is useful for other schedule requests (in $1 \times M$ problem). As a thread can directly jump to the column next to the last reserved slot by the request (*req*) (see Algorithm 4 Lines 31- 36).

Lastly, the *HardWall* method is used to check whether a column (*index*) is a `HARD` wall or not. We need to check whether all entries in the column are in the `HARD` state or not. If all the entries are in the `HARD` state then we return `TRUE` otherwise, the column (*index*) is not a hard wall.

Algorithm 4: *bookFirstSlot*

```

1: function bookFirstSlot (req, startCol, round)
2:   for col  $\in$  [startCol, SLOT.length()]  $\wedge$  (col +
   req.numSlot < SLOT.length()) do
3:     reqState  $\leftarrow$  req.iterState
4:     if reqState  $\neq$  NEW then
5:       return (REFRESH, NULL, NULL)
6:     end if
7:     (res, row)  $\leftarrow$  bookMinSlotInCol(req, col,
   slotNum, round)
8:     if (res = FALSE)  $\wedge$  (row = NULL) then
9:       return (REFRESH, NULL, NULL)
10:    else if (res = FALSE)  $\wedge$  (row = NEXT) then
11:      continue
12:    else
13:      return (TRUE, row, col)
14:    end if
15:  end for
16:  return (FALSE, NULL, NULL)
17: end function

```

```

18: function findMinInCol (req, col, slotNum, round)

```

```

19: minRank  $\leftarrow$  MAX
20: for  $i \in [0, N - 1]$  do
21:   val  $\leftarrow$  SLOT [ $i$ ][col].get()
22:   otherRank  $\leftarrow$  getSlotStatus(req.gettid(), val,
   round, slotNum)
23:   if minRank > otherRank then
24:     minRank  $\leftarrow$  otherRank
25:     row  $\leftarrow$  i
26:   end if
27: end for
28: tstamp  $\leftarrow$  SLOT [row][col].getTstamp()
29: return (row, minRank, tstamp)
30: end function

31: function forcehardAll (req)
32:   for  $i \in [1, \text{length}(\text{req.path})]$  do
33:     (row,col)  $\leftarrow$  unpack(req.path[i-1])
34:     SLOT [row][col].set(pack(HARD,req.getTid(),
   req.requestid,req.numSlots -i))
35:   end for
36: end function

37: function HardWall(index)
38:   for  $i \in [0, N-1]$  do
39:     state  $\leftarrow$  SLOT [ $i$ ][index].getState()
40:     if state = HARD then
41:       continue
42:     else
43:       return FALSE
44:     end if
45:   end for
46:   return TRUE
47: end function

end

```

B.2 Functions to Reset the State

Now, we describe a set of functions *undoSlot*, *undoPath* and *undoShadowpath*, which are used to reset the state of various fields such as a slot in the SLOT matrix, the PATH array and the SHADOWPATH array respectively (see Algorithm 5). These functions are mainly used when a request either gets cancelled or some other helper has already reserved a slot. In these scenarios we need to undo the changes done by a particular thread.

In the method *undoSlot*, a slot is atomically reset to its default value with a higher time stamp (Line 5). The index and the time stamp of the slot which a thread wishes to reset is saved in the SHADOWPATH array. Before resetting the slot, the value of SHADOWPATH array is read. A thread resets the value of a slot only if the entry in the SHADOWPATH array is valid and the time stamp matches. Otherwise, it means some other thread has already cleared the slot.

Next, the method *undoShadowpath* is used to set an entry in the SHADOWPATH array to its default value and mark it as INVALID. This is done when a thread is unable to reserve a slot which it has saved in its SHADOWPATH array.

Lastly, when a thread gets cancelled then it has to clear its PATH array, SHADOWPATH array and the slots reserved so far. This is done with the help of *undoPath* method. In each entry of the PATH array, along with the index of a slot of the SLOT array, a round of an request is saved. This round indicates the iteration of a request. If the round saved in the PATH array is different from the one passed as an argument, it means that some other helper has already cleared the PATH array (Line 21). If the round matches, then the PATH array is cleared and each entry has updated round. This updated round is the current iteration of a request. The SHADOWPATH array and the slots are cleared as explained in the method *undoShadowpath* and *undoSlot*.

Algorithm 5: Undo Methods

```

1: function undoSlot(req,slotNum)
2:   (storeRound, row, col, status)  $\leftarrow$ 
   req.path.get(slotNum)
3:   tstamp  $\leftarrow$  req.SHADOWPATH.getTstamp(slotNum)
4:   if status = VALID then
5:     newVal  $\leftarrow$  pack(0,0,0,tstamp+1)
6:     oldVal  $\leftarrow$ 
       pack(req.getTid(),tstamp,round,slotNum)
7:     slots[row][col].compareAndSet(oldVal,
       newVal)
8:   end if
9: end function

10: function undoShadowpath(req,slotNum,row,col)
11:   tstamp  $\leftarrow$  req.SHADOWPATH.getTstamp(slotNum)
12:   expVal  $\leftarrow$  pack(tstamp, row, col, VALID)
13:   newVal  $\leftarrow$  pack(0,0,0,INVALID)
14:
   req.SHADOWPATH.compareAndSet(slotNum,expval,
   newVal)
15: end function

16: function undoPath (req, round)
17:   cancelVal  $\leftarrow$  pack(round+1, 0, 0,INVALID)
18:   for  $i \in [0, \text{length}(\text{path}) - 1]$  do
19:     (storeRound, row, col, status)  $\leftarrow$ 
       req.path.get(i)
20:     if round  $\neq$  storeRound then
21:       continue
22:     end if
23:     tstamp  $\leftarrow$  req.SHADOWPATH.getTstamp(i)
24:     req.path.compareAndSet(i, pack(storeRound,
       row, col, status), cancelVal)
25:     req.SHADOWPATH.compareAndSet(i,
       pack(tstamp, row, col, status),
       pack(0,0,0,INVALID))
26:     if status = VALID then
27:       newVal  $\leftarrow$  pack(0,0,0,tstamp+1)
28:       oldVal  $\leftarrow$  pack(req.getTid(),tstamp,round,i)
29:       SLOT [row][col].compareAndSet(oldVal,
       newVal)
30:     end if

```



```

31:   end for
32: end function

```

end

B.3 Functions to Co-ordinate Among Helpers

The method *pathCAS* is used to save the index of the slot reserved by a thread in the TMP state in the reservation PATH array. In the case of internal helping, multiple threads try to reserve a slot for a particular thread. Lets assume that a request is in the SOFT state and is trying to reserve a slot in the column *col*. All the entries in the column *col* are in the HARD state. In this case, the request *req* will move to the CANCEL state and start anew with a new round. It is possible that some slow helper may read the old state of the request, *req*, as SOFT, try to reserve a slot, and make an entry in the PATH array using a CAS operation. As the request has entered a new round, the thread will fail and it needs to undo the reservation in the SLOT matrix (see Algorithm 6 Line 11).

Next, multiple helpers can find different slots in the same column (*col*), which can be reserved for a request *req*. If each of the threads first tries to reserve a slot in the SLOT matrix and later updates the PATH array, then only one of them will succeed in updating the path and the rest will fail. But this causes more than one slot in the SLOT matrix to be in the TMP state for the same request *req* which can unnecessarily delay other low priority requests. In order to avoid this a SHADOWPATH array is used by the threads to announce their intention to book 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. This is done using the function *shadowPathCAS*.

In the FORCEHARD state, a thread makes the temporary reservation as permanent by converting the reserved slots in the SLOT matrix to the HARD state using the function *forcehardAll*. The list of slots to be reserved in the HARD state is saved in the PATH array.

Algorithm 6: pathCAS and shadowPathCAS

```

1: function pathCAS (req, round, slotNum, row,
   col)
2:   expVal ← pack(round, 0, 0, INVALID)
3:   newVal ← pack(round, row, col, VALID)
4:   if (req.path.compareAndSet(slotNum, expVal ,
   newVal) = FALSE) || (req.path.get(slotNum) ≠
   newVal) then
5:     tstamp ← SLOT [row][col].getStamp()
6:     e ← pack(row,col,tstamp,VALID)
7:     d ← pack(0,0,0,INVALID)
8:     req.SHADOWPATH.compareAndSet(e, d)
9:     es ← pack(req.getTid(), slotNum, round,
   tstamp)
10:    ds ← pack(0,0,0,tstamp+1)
11:    SLOT [row][col].compareAndSet(es, ds)

```

```

12:   end if
13: end function

```

```

14: function shadowPathCAS (req, row, col, tstamp,
   slotNum)
15:   expVal ← pack(0, 0, 0, INVALID)
16:   newVal ← pack(row, col, tstamp, VALID)
17:   req.SHADOWPATH.compareAndSet(slotNum,
   expVal, newVal)
18: end function

```

end

B.4 The otherCancel Function

The *otherCancel* method is used to cancel other requests (see Algorithm 7). This function is invoked if we find the rank of the lowest ranked slot to be CANCONVERT. This means that the request, *r*, which owns this slot has reserved a lesser number of slots than the current request. Intuitively, we would want to give more priority to a request that has already done more work (does not affect correctness). In this case, we will try to set the state of request *r* to CANCEL. Recall that one thread can change the state of another thread's request to CANCEL only if the current request state of that thread is either NEW or SOFT (Line 6 and 12). It might be possible that the same request keeps on getting cancelled by other threads. To avoid this a CANCELTHRESHOLD is set, which means that a request can get cancelled at the most CANCELTHRESHOLD times by other threads. After this it cannot be cancelled anymore by other threads. After cancelling a request, the thread can take its slot and continue. The cancelled thread needs to start anew.

Algorithm 7: cancel method

```

1: function otherCancel (tid, round)
2:   req ← request.get(tid)
3:   state ← req.getState()
4:   (stat, slot, round, row, col) ← unpack(state)
5:   newState ← pack(CANCEL, slot, round, row, col)
6:   if stat = NEW ∧ cancelCount(tid).get() <
   CANCELTHRESHOLD then
7:     if req.state.compareAndSet(state, newState)
   then
8:       cancelCount(tid).getAndIncrement()
9:       return TRUE
10:    end if
11:  end if
12:  while stat = SOFT ∧ cancelCount(tid).get() <
   CANCELTHRESHOLD do
13:    if req.state.compareAndSet(state, newState)
   then
14:      cancelCount(tid).getAndIncrement()
15:      return TRUE
16:    end if
17:    state ← req.getState()
18:    (stat, slot, round, row, col) ← unpack(state)

```

```

19:     newState ← pack(CANCEL, slot, round, row,
20:         col)
21:   end while
22:   return FALSE
end function
end

```

APPENDIX C

SLOT SCHEDULING – LOCKED VERSION

In this appendix, we discuss the implementation details of different lock based algorithms. In specific, we look at coarse grain locking with and without fair locks, and a fine grain locking algorithm.

C.1 Coarse Grain Locking

To implement the lock based version of the slot scheduling algorithm, let us first look at a very simple and naive approach (see Algorithm 8), which is also our most high performing lock based implementation.

Assume that thread t_i places a request to book $numSlots$ slots starting from time slot $slot$. Thread t_i first scans the SLOT matrix to find $numSlots$ consecutive EMPTY slots lying between $[slot, slot + numSlots - 1]$ (Line 4). When a slot is found to be EMPTY, t_i stores its index in a local array *record* (Line 6). Once the required number of slots are found, t_i books these slots by changing the state of the slots from EMPTY to HARD (Line 15). This method is annotated with the keyword *synchronized*, which makes it execute as a critical section. Only one thread can execute this function at a time. All other threads attempting to call the *processSchedule* function (i.e trying to enter the synchronized block) are blocked until the thread inside the synchronized block exits the block.

Algorithm 8: processSchedule- locked

```

1: synchronized
2: function processSchedule (tid, slot, numSlots)
3:   slotCount ← 0, index ← 0
4:   for i ∈ [slot, SLOT.length] ∧ (slotCount <
      numSlots) do
5:     if ∃ j, SLOT [i][j] = EMPTY then
6:       record[index++] ← j
7:       slotCount++
8:     else
9:       reset slotCount and record, start searching
        again
10:    end if
11:  end for
12:  if slotCount = numSlots then
13:    /* reserve the slot by setting the threadId in
        SLOT array */
14:    for i ∈ [0, numSlots] do
15:      SLOT [row + i][record[i]] ← threadId
16:    end for
17:    return record;
18:  else
19:    return FAIL;
20:  end if
21: end function

22: synchronized
23: function processFree (tid, slotList[], numSlots)
24:   for i ∈ [0, numSlots] do
25:     (row, col) ← slotList[i]

```

```

26:     SLOT [row][col] ← EMPTY
27: end for
28: end function
end

```

There are three steps in this method: (1) acquiring the lock, (2) searching for *numSlots* free slots, and (3) and changing their status from *EMPTY* to *HARD*. Step (3) is very fast. In fact for a lock based algorithm this step is faster than non-blocking algorithms. The reason is that non-blocking algorithms use the atomic *get*, *set*, and *CAS* primitives to access the *SLOT* matrix. These are slow operations because they need to execute slow memory fence instructions. Whereas, in the case of the lock based algorithm we use regular reads and writes to access the *SLOT* matrix. These memory operations are significantly faster than their atomic (*get*, *set*, *CAS*) counterparts.

However, the speed of step (3) is compensated by steps (1) and (2). The execution time of steps (1) and (2) depends on the amount of contention and the memory access patterns of the threads. Since the number of software threads is always less than or equal to the number of hardware threads, OS scheduling is not an issue. However, in the hardware, the order and timing of atomic memory operations can dictate the order in which threads acquire the lock, and subsequently get a chance to reserve their slots. We have observed that these steps are very jitter prone and the variance in execution times is very large. We shall discuss this point in more detail in Section C.4.

Let us now consider the *free* operation. The *processFree* method is invoked whenever a thread issues a free request. Similar to the *processSchedule* method it executes in a critical section, and gets an exclusive lock on the *SLOT* matrix.

Note that our lock based implementation is fairly naive. We made an effort to make it more sophisticated by trying to make the granularity of locking finer. In specific, we tried to split the *SLOT* matrix and use a separate lock for each part. Alternatively, we tried to remove some steps out of the critical section such as the step to search for a possible set of free slots. Different combinations of these approaches did not yield any significant improvements in terms of performance. Hence, we decided to stick with this particular implementation (as shown in Algorithm 8).

C.2 Slot Scheduling with Fair Locking

We had observed in our experiments with locks that the process of acquiring the lock takes a variable amount of time and is very jitter prone. We thus decided to look at fair locks that have stronger guarantees. To implement slot scheduling algorithms by using fair locking mechanisms, we used reentrant locks that have native support in Java. The fair version of reentrant locks ensures that the time a thread needs to wait for acquiring a lock is finite as long as the thread that is executing a critical section does not go off to sleep for an indefinite amount of

time [Cederman, 2013]. Using this particular method for implementing fair locking is a standard recommended practice as mentioned in Java’s online documentation.

C.3 Slot Scheduling with Fine Grain Locking

Lastly, we briefly discuss the implementation of the slot scheduling algorithm with fine grain locking mechanisms. In this case, we divide the whole *schedule* operation into multiple stages similar to our lock-free implementation. Each stage is executed as a critical section annotated with the keyword, *synchronized*. In our approach, we divide the total computation into 3-5 stages (see [20] for more detail). Each state of the request corresponds to a stage in the computation. To allow for greater parallelism, and overlap between threads, we do not store any state in any thread’s local storage. Instead, all the state is stored in the *SLOT* matrix itself similar to our non-blocking algorithms.

The approach is thus as follows. A request first searches for the first available slot, at which it can book a request. Then, it acquires a lock, and temporarily reserves the slots, and releases the lock. Subsequently, it acquires a lock again, checks the slots that it had temporarily reserved, and then makes their status *HARD*. Our initial thinking was that by dividing the entire operation into finer chunks and executing each chunk as a critical section will increase the overlap between threads, reduce starvation, and will be beneficial. However, this did not turn out to be the case. We tried many more variants such as acquiring locks on smaller portions of the *SLOT* matrix, and varying the number of stages into which we split the computation. However, the results for this particular method with the best combinations were not encouraging.

C.4 Results: Characterization of Locking

Before we take a look at the results for slot scheduling with different kinds of locking mechanisms, let us look at the performance aspects of the process of acquiring and releasing locks. It is very important to understand this process because we shall use the crucial insights learned in this section to explain the results for slot scheduling. We shall observe in Section C.5 that the results have a lot of variance (or jitter). We want to prove in this section, that the jitter is because of the variable and non-deterministic delays suffered while acquiring a lock.

Let us start out by defining the term “OS jitter”, which refers to the interference experienced by an application due to activities such as the running of daemon processes, periodic tasks such as timer interrupts and asynchronous interrupts received by the operating system. Various studies have shown the effects of OS jitter on parallel applications running on high performance systems. Our aim here is to first measure the interference caused by the OS/JVM and show that it is **not responsible** for the variance that we observe in our results. We

measure the OS jitter in our system by using the methodology suggested by [Mann et al, IEEE International Conference, 2007] after following his recommendations to minimize OS jitter in the system. His approach to measure OS jitter is to measure the time it takes to run a simple set of arithmetic computations, thousands of times. The duration of each iteration should be of the order of hundreds of microseconds. We performed this experiment. However, instead of running one thread, we ran 64 threads, where each thread measures the time it takes for each iteration of a *for* loop to run. Figure 22 shows the result of this experiment aggregated across all the threads. We aimed for a mean request duration of 450 μ s. We plot the results for a randomly chosen set of 1000 iterations. We observe that there is a swing of $\pm 150\mu$ s in the worst case. On the basis of this experiment, we concluded that the baseline system can at the most create jitter limited to 20% of the total execution time for sub-millisecond tasks (similar results obtained by [Chandran et al, IEEE TPDS, 2014]).

Next, we did the same experiment by running each iteration in a critical section. We used different types of locks such as MCSlocks (*MCS*), fair locking by using reentrant locks (*Reentrant*) and traditional locks that are declared by using the *synchronized* keyword (*Synchronized*). The threads do not share any memory. The *Baseline* implementation is similar to our previous experiment. It does not have locks. We run each experiment for 1 million iterations for 32 and 64 threads.

Figure 23 shows the results. In the *Baseline* method, the average time per operation is the same for 32 and 64 threads (around 450 μ s). Increasing the number of threads has no impact on the performance of *Baseline*. Next, we observe the time per operation for the lock based algorithms. These algorithms are 7-12X slower than the *Baseline* method for a system with 32 threads. Even though there is no shared structure among the threads, the time per operation for the lock based algorithms is more as compared to the *Baseline* algorithm. This is because the threads contend with each other to acquire the lock. As the number of threads increases to 64, the time per operation for *Synchronized* and *Reentrant* increases by 2-3X. Moreover, the jitter observed in the case of lock based algorithms is in the order of milliseconds: 70 to 430ms in the worst case (mean is 3ms and 7ms for *Synchronized* and *Reentrant* respectively).

Recall that we had established with the help of Figure 22 that the jitter introduced by the OS and JVM is limited to 150 μ s, whereas the jitter by adding locks is at least 20X more. Thus, we can conclude that the process of acquiring a lock is extremely jitter prone at sub-millisecond time scales.

C.5 Results for Slot Scheduling

We implemented three kinds of slot scheduling: coarse locking (*coarseLocking*), fine grain locking

(*fineLocking*) and coarse locking with fair locks(*fairLocking*).

Figure 24 shows the values of t_{req} for the $1 \times M$ algorithm. We observe that *fineLocking* is three orders of magnitude slower than the rest of the algorithms. This is mainly because acquiring a lock is a slow and jitter prone process. The *LF* algorithm is roughly 5-10X faster than the *WF*, *coarseLocking* and *fairLocking* algorithms. The performance of *WF*, *coarseLocking* and *fairLocking* is nearly the same. However, the variation in the *coarseLocking* and *fairlocking* algorithms (shown with error bars) is very high (order of 2-3ms). We have not shown the variance of *WF* and *LF* for the sake of readability in this figure. However, their variance is limited to 100-200 μ s, which is 20X better than the algorithms with locks. Additionally, note that *fairLocking* has a noticeably smaller variance for less than 20 threads; however, the mean values of t_{req} for both the coarse locking algorithms (with and without fair locks) is comparable.

Similar trends are observed for the $N \times M$ problem (see Figure 25). In the worst case, *fairLocking* and *coarseLocking* are 8-10X slower than *WF*. The time per request goes up to 8000 μ s and 13000 μ s for *coarseLocking* and *fairLocking* respectively (for 64 threads). The jitter in both the cases, is of the order of thousands of microseconds.

From Figures 24 and 25, we can conclude that *coarseLocking* is the best algorithm with locks. *fairLocking* might be better in some cases, especially with regards to the jitter; however, it is not uniformly better. Additionally, its mean performance is inferior to the *coarseLocking* algorithm.

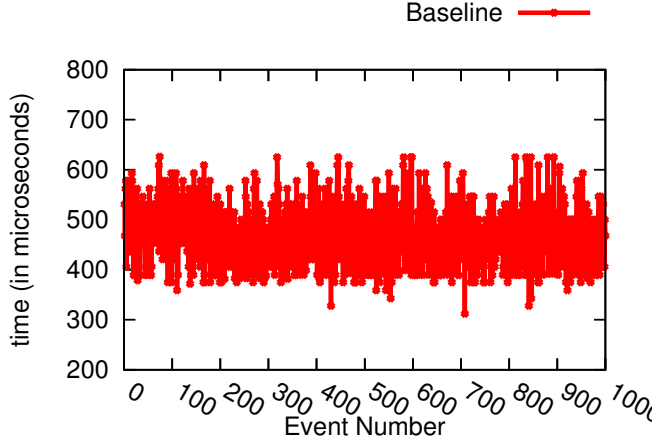
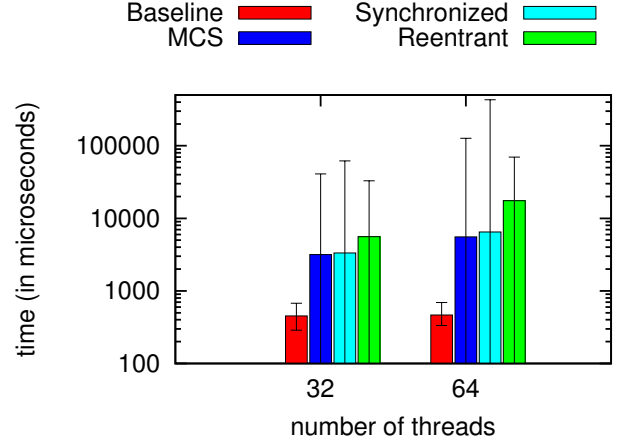
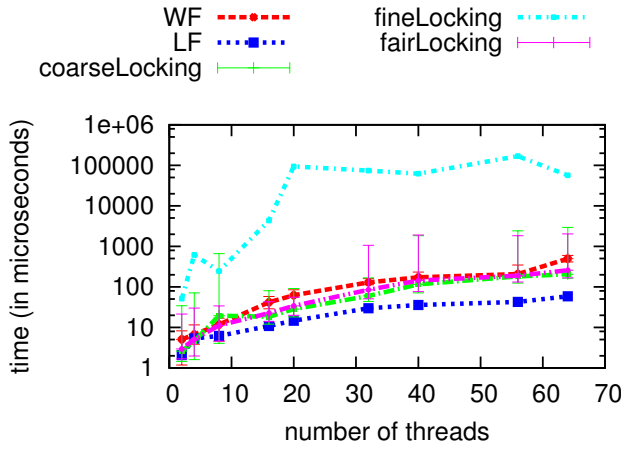
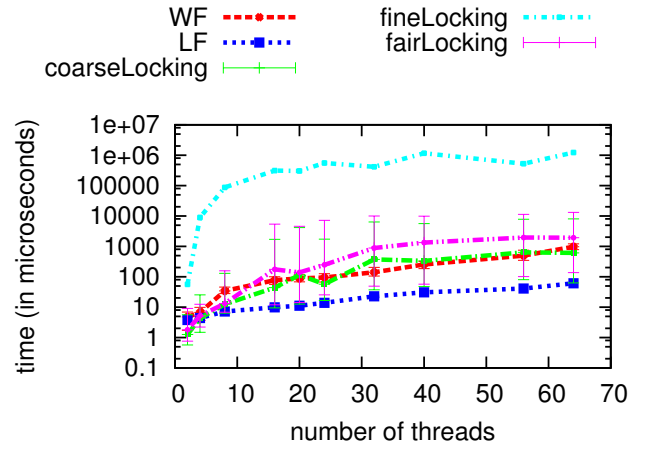


Fig. 22: Baseline jitter

Fig. 23: t_{req} Fig. 24: t_{req} for the $1 \times M$ algorithmFig. 25: t_{req} for the $N \times M$ algorithm

APPENDIX D

THE $1 \times M$ SCHEDULE PROBLEM

Here we present the algorithm for reserving the slots in the $1 \times M$ schedule problem, where the number of resources is 1 and a request wishes to reserve M slots (see Algorithm 9). A new request is placed in the `NEW` state. In this state a thread (t) temporarily reserves a slot in the `SLOT` array with the help of the function `reserveSlots` (explained in Appendix D.1). On successfully reserving the first slot, the request moves to the `SOFT` state (Line 14) using a CAS operation. In the `SOFT` state the remaining slots are reserved temporarily (change the state of the slot from `EMPTY` to `TMP`). Once it finishes doing so, the request moves to the `FORCEHARD` state (Line 28). In this state the reservation made by the thread t (or by some helping thread on behalf of t) is made permanent (Line 40) by changing the state of a slot from `TMP` to `HARD`. Lastly, the request enters the `DONE` state. The first slot reserved by the thread is returned. The rest of the slots reserved can be calculated based on it. A thread is unable to reserve its slots if all the slots are permanently reserved till the thread reaches the end of the `SLOT` array

(Line 10). In each `SOFT` state, a thread tries to book the next consecutive slot (Line 30). The request continues to stay in the `SOFT` state till the required number of time-slots are not reserved temporarily.

If a thread is unable to temporarily reserve a slot in its `SOFT` state, we move the request to the `CANCEL` state. In the `CANCEL` state (Lines 45- 48), the temporarily reserved slots are reset to their default values (`EMPTY`). The `request.slotAllocated` field is also reset. The request again starts from the `NEW` state with a new round (`round`) and starting slot (`index`). The `round` field is used to synchronize the helpers. Once the request enters the `FORCEHARD` state, it is guaranteed that M slots are reserved for the thread and no other thread can overwrite these slots.

Algorithm 9: *process $1 \times M$*

```

1: function processSchedule(request)
2:   while TRUE do
3:     iterState  $\leftarrow$  request.iterState.get()
4:     (reqState, round, index)  $\leftarrow$  unpackState(iterState)
5:     switch (reqState)
6:     case NEW :
7:       (status, res)  $\leftarrow$  reserveSlots(request, index, round, reqState)
```



```

8:   if status = FAIL then
9:     /* linearization point */
10:    request.iterState.CAS(iterState,
11:    packState(0,0,0,FAIL))
12:  else if status = RETRY then
13:    /* read state again */
14:  else
15:    if request.iterState.CAS(iterState,
16:    packState(1,round,res+1,SOFT)) then
17:      request.slotAllocated.CAS(-1,res)
18:    else
19:      /* clear the slot reserved */
20:    end if
21:  end if
22:  break
23: case SOFT :
24:   slotRev ← getSlotReserved(reqState)
25:   /* reserve remaining slots */
26:   (status, res) ← reserveSlots(request,
27:   index,round,reqState)
28:   if status = SUCCESS then
29:     if slotRev+1 = req.numSlots then
30:       /* linearization point */
31:       newReqState ← Request.packState(req.
32:       numSlots,round,index,FORCEHARD)
33:     else
34:       newReqState ← Request.packState(slot
35:       Rev+1,round,index+1,SOFT)
36:     end if
37:     request.iterState.CAS(iterState, newReqState)
38:   else if status = CANCEL then
39:     request.iterState.CAS(iterState,pack
40:     State(slotRev, round, index, CANCEL))
41:   else
42:     RETRY /* read state again */
43:   end if
44:   break
45: case FORCEHARD :
46:   /* make the reservation permanent */
47:   /* change state of slot from TMP to HARD */
48:   request.iterState.CAS(iterState, DONE)
49:   break
50: case CANCEL :
51:   /* reset the slots reserved till now */
52:   /* Increment the request round */
53:   /* reset the slot Allocated field of request */
54:   request.iterState.CAS(iterState,
55:   packState(0,round+1,index+1,NEW))
56:   break
57: case DONE :
58:   return request.slotAllocated
59: case FAIL :
60:   return -1
61: end switch
62: end while
63: end function

```

D.1 Reserve Slots

This method accepts four parameters — request(*req*) of a thread *t*, the slot to be reserved *currSlot*, current round *round* of the request and the state of the request *reqState*. Depending upon the rank of the slot (*currSlot*) we execute the corresponding switch-case statement. The field *round* indicates the iteration of a request. It is used to synchronize all the helpers of a request. If the slot is in the EMPTY state, we try to

temporarily reserve the slot and change the state of the slot from EMPTY to TMP (Line 64). If the slot is in the TMPFREE state, it means some free request is trying to free a slot. In this case, we help the free request (Line 72) and then we proceed with reserving the rest of the slots.

Next, when the state of the slot is TMP it indicates that some other thread has temporarily reserved the slot *currSlot*. If the *threadid* saved in the slot is the same as that of the request *req* (Line 75), we simply return to the *processSchedule* method and read the state of the request again. Otherwise, the slot is temporarily reserved for another request, *otherReq*. Now, we have two requests *req* and *otherReq* contending for the same slot *currSlot*. 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* thus changing the state of the request *otherReq* to CANCEL atomically (Lines 80 - 87). The request *req* will help the request *otherReq* in case *req* has a lower priority. We increment the priority of a request to avoid starvation (Line 95). The request which has reserved more slots is assigned a higher priority.

Next, we discuss the case where the slot is found in the HARD state. In the NEW state of the request, a request tries to search for the next empty slot (Line 106). When the state of the slot is made HARD, along with the *threadId*, we save the number of consecutive slots (*slotMove*) reserved in the HARD state. The search index moves directly to slot (*currSlot* + *slotMove*) instead of incrementing by 1. The search terminates when either a slot is successfully reserved or the request reaches the end of the SLOT array (Line 114). In the SOFT state, we return CANCEL (Line 108). On receiving the result of the function *reserveSlots* as CANCEL, the request moves to the CANCEL state. Lastly, it is possible that some other helper has reserved the slot for request *req* (Line 100) and changed its state to HARD. In this case the thread returns to the *processSchedule* method and reads the state of the request *req* again.

57: **function** reserveSlots(request,currSlot, round, reqState)

```

58:   for i ∈ [currSlot, SLOT.length()] do
59:     slotState ← getSlotState(SLOT.get(i))
60:     (threadid,round1,state) ← unpackSlot(SLOT.get(i))
61:     newVal ← packTransientState(request)
62:     switch (slotState)
63:     case EMPTY :
64:       res ← SLOT.CAS(currSlot,EMPTY,newVal)
65:       if res = TRUE then
66:         return (SUCCESS, currSlot)
67:       end if
68:     break
69:     case TMPFREE :
70:       otherReq ← REQUEST.get(threadid)
71:       /* res = HELP */
72:       processFree(otherReq)
73:     break
74:     case TMP :
75:       if threadid = req.threadid then

```

```

76:     return (RETRY, null)
77: else
78:   otherReq ← REQUEST.get(threadid)
79:   res ← getHighPriorityRequest(req, otherReq, i)
80:   if res = req then
81:     /* preempt lower priority request */
82:     if cancelReq(otherReq) then
83:       oldValue ← packTransientState(
84:         threadid, round1, state)
85:       newValue ← packTransientState(req.
86:         threadid, round, SOFT)
87:       res1 ← SLOT.CAS(currSlot, oldValue,
88:         newValue)
89:       if res1 = TRUE then
90:         return (SUCCESS, currSlot)
91:       end if
92:     end if
93:   else
94:     /* res = HELP */
95:     processSchedule(otherReq)
96:     /* increase priority to avoid starvation */
97:     req.priority.getAndIncrement()
98:   end if
99:   end if
100:  break
101: case HARD :
102:   if threadid = req.threadid then
103:     /* slot reserved on req's behalf */
104:     return (RETRY, null)
105:   else
106:     if req.iterState = NEW then
107:       slotMove ← getReserveSlot(SLOT.get(i))
108:       i ← i + slotMove
109:     else
110:       return (CANCEL, null)
111:     end if
112:   end if
113:   break
114: end switch
115: end for
116: return (FAIL, -1)
117: end function
end

```

APPENDIX E

RELAXED SLOT SCHEDULING

In this appendix, we discuss 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. We refer these set of algorithms as *relaxed lock-free* and *relaxed wait-free* algorithms respectively. Here we sacrifice linearizability to gain performance.

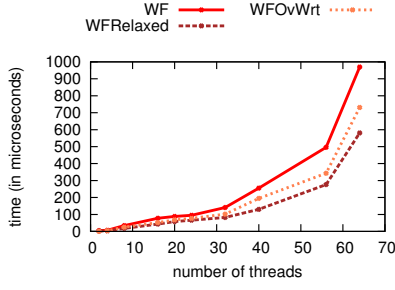
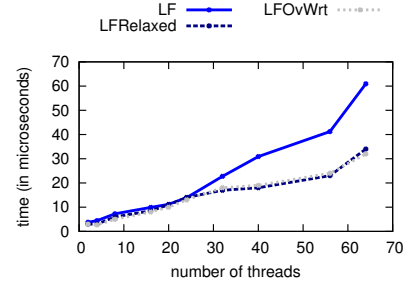
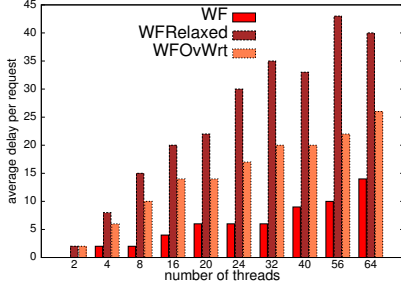
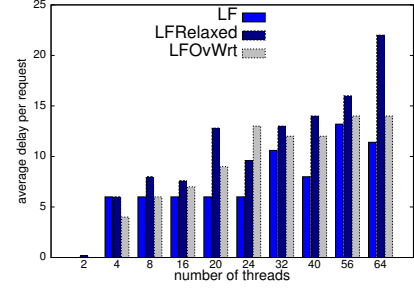
Lock-free and wait-free algorithms are fairly complicated. These algorithms ensure that a request is scheduled at the earliest possible slot and does not miss a valid slot. In the relaxed version of lock-free and wait-free algorithms, we allow a thread to change its starting position even if the threads encounter a *soft wall*. A column full of SOFT entries is referred to as a *soft wall*. This helps in reducing contention. A thread neither internally helps another thread nor it overwrites the slot (*currSlot*) temporarily reserved by some other thread. In case of the wait-free version of *relaxed* scheduling, helping is limited to only *externally* helping. We refer to this implementation of wait-free and lock-free algorithms as *WFRelaxed* and *LFRelaxed* respectively. It is possible that a thread t_j , which has currently reserved a slot, *currSlot*, gets cancelled later. In this scenario, another thread t_i which had changed its starting position, seeing that thread t_j had temporarily reserved the slot *currSlot*, misses a valid slot (which is *currSlot*). This implementation is not linearizable but it improves the performance of the algorithms roughly by a factor of 2.

We designed another non-blocking slot scheduling algorithm where a thread having a higher priority can overwrite the slots temporarily reserved by some other low priority thread but does not help a higher priority thread (i.e. *internal* helping is not done). We refer to this implementation of wait-free and lock-free algorithms as *WFOvWrt* and *LFOvWrt* respectively. These versions also have much better performance at the cost of sacrificing linearizability.

E.1 Results

Next, we compare the performance of the wait-free (*WF*) algorithm with relaxed wait-free (*WFRelaxed*) and *WFOvWrt*. A thread places a request for the time slot immediately next to the last allocated slot in the *LF* and *WF* algorithms. This introduces high contention among the threads. Figure 26 shows that *WFRelaxed* is nearly 2x faster than *WF* as contention in the case of the *WFRelaxed* algorithm is resolved by one thread moving ahead in case of a conflict. The performance of *WFOvWrt* lies in the middle of *WF* and *WFRelaxed* (i.e. *WFOvWrt* is 30% faster than *WF* for less than 24 threads). Recall that in the *WFOvWrt* algorithm, a thread cancels another thread upon a conflict.

In the case of lock-free algorithms, the performance of *LFRelaxed* and *LFOvWrt* are nearly 2x better than *LF* beyond 32 threads (see Figure 27). The difference in the performance of *LFRelaxed* and *LFOvWrt* is roughly

Fig. 26: t_{req} for relaxed version of WF Fig. 27: t_{req} for relaxed version of LF Fig. 28: $delay$ per request for WF Fig. 29: $delay$ per request for LF

similar because in both the cases, we avoid internal helping. For less than 32 threads the difference in the performance is within 10-12% as there are less number of requests conflicting with each other.

The schedule generated by relaxed lock-free and wait-free algorithms is not optimal as the requests are not being scheduled at the earliest possible slot. The quality of the schedule can be measured in terms of delay in scheduling a request. $delay$ signifies the deviation/difference in the starting slot requested and the actual starting slot allotted to a request. Figures 28 and 29 show the average delay incurred in scheduling a request. This is 8 time slots for 32 threads and 12 time slots for 64 threads in the case of WF . Whereas, for $WFRelaxed$ and $WFOvWrt$ the delay is around 40 time slots per request because we move forward due to contention. This results in missing some valid time slots for scheduling a request. For lock-free algorithms, the delay is around 5 to 12 time slots for LF and it goes up to 22 time slots for $LFRelaxed$ for 64 threads.