Lock-free and Wait-free Slot Scheduling Algorithms

Pooja Aggarwal
Computer Science Department
IIT Delhi
New Delhi, India
pooja.mcs11@cse.iitd.ac.in

Smruti R. Sarangi
Computer Science Department
IIT Delhi
New Delhi, India
srsarangi@cse.iitd.ac.in

***Abstract*—Scalable scheduling is being increasingly regarded as an important requirement in high performance systems. There is a demand for high throughput schedulers in servers, data-centers, networking hardware, large storage systems, and in multi-cores of the future. In this paper, we consider an important subset of schedulers namely slot schedulers that discretize time into quanta called slots. Slot schedulers are commonly used for scheduling jobs in a large number of applications. Current implementations of slot schedulers are either sequential, or use locks. Sadly, lock based synchronization can lead to blocking, and deadlocks, and effectively reduces concurrency. To mitigate these problems, we propose a set of parallel lock-free and wait-free slot scheduling algorithms. Our algorithms are immune to operating system jitter, and guarantee forward progress. Additionally, all our algorithms are linearizable and expose the scheduler's interface as a shared data structure with standard semantics. We empirically demonstrate the scalability of our algorithms for a setup with thousands of requests per second on a 24 thread server. The wait free algorithms are most of the time as fast as the lock-free versions (3X-8X slower in the worst case).**

***Keywords*-wait free, lock free, scheduler, slot scheduling**

I. Introduction

We are entering the era of large shared memory multicore processors with potentially hundreds of cores. Keeping these trends in mind, 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 middleware 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 [1], [2]. Shared data structures in the kernel limit its scalability, and thus it is necessary to parallelize them. To a certain extent the read-copy update mechanism [3] in the Linux kernel has ameliorated these problems by implementing wait free reads. Note that writes are still extremely expensive, and this approach has been predominantly used in the networking protocol stack, and memory management unit.

One of the remaining major bottlenecks to scalability is the scheduler [1], [2]. The quintessential approach to parallelizing the scheduler is to use a parallel wait free queue. It is possible to design a multiple enqueuer/dequeuer wait free queue with moderate overheads (see [4]). Note that parallel schedulers have limitations in their functionality. They typically do not consider dependences across tasks, or assume task deadlines. They try to schedule tasks in FIFO order.

In this paper, we look at a more flexible approach proposed in prior work called *slot scheduling* [5]. 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*. The beginning of each jiffy is indicated by a timer interrupt. Now, we can consider a two dimensional matrix of time and resources known as the Ousterhout matrix [6] as shown in Figure 1.

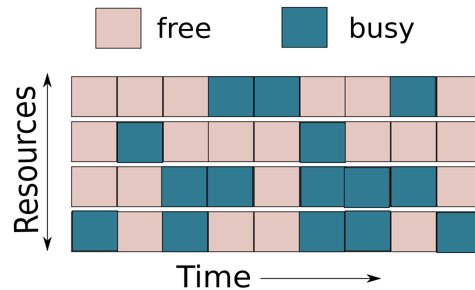

Figure 1. The Ousterhout matrix for scheduling

Here, we represent time in the x-axis, resources in the y-axis, and each cell(slot) represents a boolean value – empty or full. *Empty* indicates that no task has been scheduled for that slot, and *Full* indicates the reverse. We can have several request patterns based on the number of requests that can be allotted per row and column. In this paper, we parameterize the problem of slot scheduling with three parameters – number of resources($capacity$), the maximum number of slots that a request requires ($numSlots$), and its progress condition (lock-free (LF), or wait-free(WF)). We consider four combinations of the capacity and number of slots: 1×1, $1 \times M$, $N \times 1$, and $N \times M$, where the format is $capacity \times numSlots$. For example, we can interpret the $N \times M - LF$ problem as follows. The number of rows in the Ousterhout matrix is equal to N, and each request requires M slots. These M slots need to be in consecutive columns, there should be one slot per column, and the algorithm should be lock-free. 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 data structure called *parSlotMap*, which is an online parallel slot scheduler. In its current version, it supports just one operation – $schedule(request)$. Each request specifies the earliest possible starting slot, and the number of slots it requires. We propose both lock free and wait free algorithms for different variants of the scheduler. We experimentally show that the wait free algorithm is 3-8X slower than the lock free algorithm in the worst case. However, it ensures that the total work done by an ensemble of threads is more than the lock free variant, and both our parallel algorithms are orders of magnitude faster than an algorithm that uses locks.

Related Work

Scheduling is a classic problem. Most variants of scheduling that consider dependences between tasks, and assume multiple nodes have been proven to be NP hard. In this paper, we consider a simpler variant of the problem suitable for parallel schedulers. It admits simple sequential solutions.

To the best of our knowledge, lock free and wait free algorithms for parallel slot scheduling have not been proposed before. There is some related work in generic scheduling, and sequential slot scheduling.

Classical parallel scheduling involves parallelizing different heuristics for sequential scheduling. Some of the most common heuristics are: longest job first, earliest finish time, highest priority, and shortest job first. Wu [7] and Dekel et. al. [8] provide a survey of the algorithms. Most of the scheduling algorithms are for offline variants of the scheduling problem. In specific, they work on a set of requests that are known apriori. Subsequently, they use parallel ranking and sorting algorithms to compute schedules. All of these algorithms use locks. Some recent work has been done by Mhamdi et. al. [9] for distributed systems, and by Keller et. al. [10] for parallel systems. Mhamdi et. al. consider online variants of the scheduling problem for a message passing system. Keller et. al. propose an offline version of the algorithm that has a known task dependence graph. They propose heuristics to split the graph and distribute it across different processors. Each processor gets to schedule the subgraph assigned to it. Communication messages are infrequent.

Ousterhout [6] proposed the Ousterhout matrix that forms the basis of slot scheduling. His basic formulation has been used by the Rialto CPU scheduler [11] and the seminal work on slot scheduling in Brandon Hall's thesis [5]. Slot scheduling is now ubiquitous and finds uses in video streams [12], vehicular networks [13], optical networks [14], online advertisements [15], and green computing [16]. These slot schedulers will benefit from high performance parallel implementations.

II. Overview of Parallel Scheduling

A. Definition of the Problem

The $parSlotMap$ data structure supports just one method namely – $schedule(request)$. A request, r, has two parameters – starting index/time($start$), and the number of slots($numSlots$). $parSlotMap$ assigns the request to a set of empty slots, and returns a list of assigned $numSlots$ slots.

Now, at any point of time, let us consider all the requests that have been assigned slots. We want to characterize the state of the matrix at that point. One commonly used correctness condition is – *sequential consistency* (see Adve et. al. [17]). This means that it should be possible for some sequential scheduler to generate exactly the same output as the parallel 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. If the sequential scheduler produces an efficient schedule, then the parallel scheduler will also do the same. Let us now try to characterize what kind of schedules are *legal* for a single thread (sequential) scheduler.

B. 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, 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 should get slots (1-3), and request 2 should 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 [18], and is one of the most common correctness criteria for parallel shared data structures.

C. 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. 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 also.

To prove conditions 3 and 4, it is sufficient to show that there is a unique point between the invocation and response of every method at which it appears to execute instantaneously [19]. This point is known as the *point of linearization*. 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 a $parSlotMap$ is both legal (conditions 1 and 2), and linearizable (condition 5).

D. Lock Freedom and Wait Freedom

Furthermore, we want our parallel algorithms to be lock-free or wait-free. An algorithm is lock-free if any thread makes progress infinitely often. This means that we do not have a situation in which threads indefinitely wait for each other. At least one thread needs to be making forward progress at any point of time. Lock freedom ensures that one thread cannot block others. However, it is possible for a thread to wait indefinitely.

Wait-free algorithms rectify this problem. They guarantee that every thread successfully completes a method call in a finite number of steps. This is typically achieved by making threads that are faster help slower threads.

III. Slot Scheduling Algorithm

A. Overview

In the lock-free implementation of our algorithm, multiple threads atomically access a slot and try to reserve it. In the $N \times M$ variant, a thread needs to book upto M slots in multiple columns. If two threads contend for a slot, then one of the threads needs to back out. We give it two options. It can either cancel itself and start anew, or it can decide to help the winner thread. We call the latter scheme – *internal helping*. We give more priority to the request that has reserved more slots when two requests contend for a slot. We cancel the request that has a lower priority. This scheme reduces the amount of wasted work. Secondly, the cancelled request can now search for an alternative path. This increases the parallelism also. However, to keep starvation in control, we introduce a CANCELTHRESHOLD to limit the number of times a thread can be cancelled by other threads.

For the wait-free implementations, threads need to help each other at the top-most level (*external helping*). Before booking a slot in the 2-dimensional SLOT array (Ousterhout matrix), a thread creates a request, places it in a REQUEST array, and then chooses to help threads with older requests before proceeding. Once such a thread, t_j, is found, t_i helps t_j in completing its operation. After helping all candidate threads, t_i proceeds with processing its own request.

The solution to the $N \times M$ problem is broadly implemented in four stages. Each stage denotes a particular state of the request. The operation progresses to the next stage by atomically updating the state of the request. Concomitantly, the state of cells in the SLOT array changes from EMPTY to SOFT and eventually to HARD . This is shown pictorially in Figure 2.

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 it is ready to make its reservation permanent.
3) In FORCEHARD state, the temporary reservation is made permanent by converting the reserved slots in the SLOT array 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.

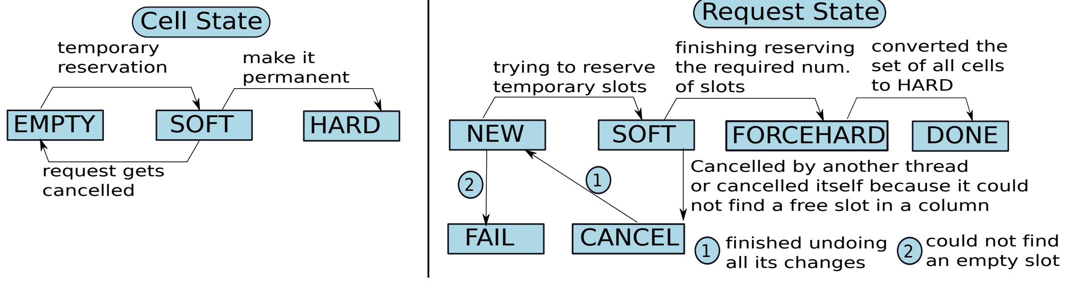

Figure 2. FSM for a cell and a request

B. Data Structures

Globally, $numSlots$ refers to the number of slots a request wants to book. For ease of explanation, our algorithms assume that each request needs the same number of slots. However, it is possible to easily change this assumption. NUMTHREADS refers to the total number of threads.

The data structures used by our implementation are shown in Figure 3 and Figure 4. These structures are mainly required for the $N \times M$ problem. The two-dimensional SLOT array represents the Ousterhout matrix which is used to keep track of free and vacant slots. Each entry in the array is 64 bits long. When a slot is free, its state is EMPTY . When a thread makes a temporary reservation, the slots of the SLOT array are in the SOFT state containing the $state$ (2 bits), tid(thread id) (10 bits), $slotNum$ (6 bits), $round$ (5 bits), $requestId$ (15 bits) and $timestamp$ (21 bits). $slotnum$ 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 some other threads. In this scenario, a thread cancels its request, clears the slots it has reserved, and starts reserving the slots again with an incremented $round$. It might be possible that other helpers are at different stages. The $round$ helps to co-ordinate between them. The $timestamp$ field is needed for correctness, as explained in Section III-E. Secondly, we derive the sizing of different fields shown in Figure 4 in Section III-E.

The REQUEST array gets populated when a thread places a new request to be scheduled. It contains NUMTHREADS instances of the $Request$ class, as shown in Figure 3. The $iterState$ field contains the current round of the request, number of slots reserved, the current index of the SLOT array, and the state of the request. The PATH array stores the slots reserved by the thread. Whenever multiple helpers try to reserve a slot on behalf of thread t_j, they first perform a CAS (Compare-and-set) on a particular slot in the SLOT array 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 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 array.

```
class Request{
      long requestId,
      long iterState,
      long[] shadowPath,
      long[] path,
      int tid,
      int slotRequested,
      int numSlots
};
```

Figure 3. The request class

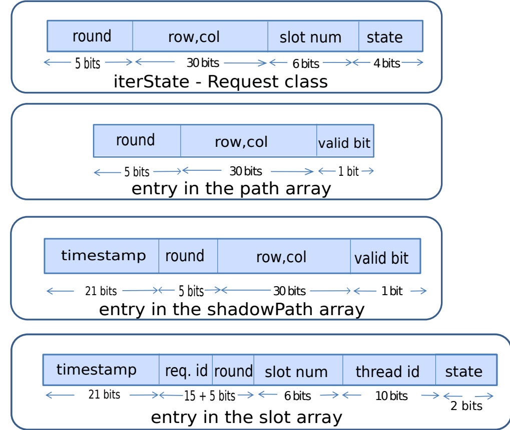

Figure 4. Data structures in the Request class and SLOT array

C. The schedule Operation

In the lock-free algorithms, each thread tries to atomically book a slot for itself whereas in the wait-free case, thread t_i first atomically increments a counter to generate the request id for the operation (Line 5). Then, it creates a new request with the time slots it is interested in booking, and sets the corresponding entry in the REQUEST array with a new request id (Line 7). It first helps any requests that meet the helping criteria as defined in the functions $findMinReq$, and $help$.

We only help requests that have a $requestId$ less than REQUESTTHRESHOLD . These functions are shared across all variants of our algorithms (see Algorithm 1). Each algorithm needs to implement its variant of the $process$ function.

Algorithm 1. *Schedule*

```
1:  function schedule(request)
2:    tid ← request.getTid()
3:    start ← request.getSlotRequested()
4:    if WAITFREE then
5:      reqId ← requestId.getAndIncrement()
6:      req ← createRequest(reqId, index, tid, numSlots,
        NEW )
7:      REQUEST .set(tid, req) /* announce the request */
8:      help(req) /* help other requests */
9:      return process(req)
10:   else if LOCKFREE then
11:     return process(request)
12:   end if
13: end function

14: function help(req)
15:   while true do
16:     minReq ← findMinReq(req)
17:     if (minReq = NULL) || (req.getRequestId() - min-
        Req.getRequestId() < REQUESTTHRESHOLD ) then
18:       break
19:     end if
20:     process(minReq)
21:   end while
22: end function
```

end

D. The $N \times M$ Problem

Here, we describe the implementation of the $N \times M$ algorithm. The code for the $process$ method is shown in Algorithm 2. We assume that the requested starting slot is $col0$, and the number of slots it requests is $numSlots$ (M).

1) $process$: We show an overall flowchart of the $process$ function in Figure 5. It extends Figure 2 by listing the list of 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 3, and execute a corresponding switch-case statement for each request state. In the NEW (starting) state, the $bookFirstSlot$ method reserves a slot $s_{[row1][col1]}$ in the earliest possible column, $col1$, of the SLOT array. We ensure that all the slots between $col0$ and $col1$ are in the HARD state (i.e booked by some other thread). $bookFirstSlot$ 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 (Line 11).

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. The SOFT state of a slot corresponds to a temporary reservation. It can either be undone, or later converted to the permanent HARD state. The request enters the FAIL state, when we reach the end of the SLOT array, and there are no more slots left. In the SOFT state of the request, the rest of the slots are reserved by calling the $bookMinSlotInCol$ method iteratively (Line 33). After reserving a slot, we enter it in the PATH array (Line 16). The state of the request remains SOFT (Line 49), and then becomes FORCEHARD after reserving the Mth slot (Line 47 and Line 57). If the CAS in Line 57 is successful then it is the point of linearization for the successful $schedule$ call (see Section IV).

In case a thread is unable to reserve a slot in the SOFT state, we set the state of the request to CANCEL (Lines 37 to 43). This happens because the request encountered a column full of HARD entries (hard wall). It changes its starting position to the slot after the hard wall.

In the CANCEL state (Lines 67-76), the $undoPath$ method resets (SOFT $\rightarrow$ EMPTY) the temporarily reserved slots, clears the PATH and SHADOWPATH arrays. The state of the request is atomically set to NEW . We set the starting column, and set $round = min(round + 1, CANCELTHRESHOLD)$. All this information is packed and atomically assigned to the $iterState$ field of the request.

Once the request is in the FORCEHARD state, it is guaranteed that M slots have been reserved for the thread and no other thread can overwrite these. All the slots reserved are made HARD and then the request enters the DONE state (Lines 61-63)

Algorithm 2. $process\ N \times M$

```
1:  function process (Request req)
2:    while TRUE do
3:      (state,slotNum,round,row0,col0)←unpack
        (req.iterState)
4:      /* Process according to the state of the request */
5:      switch (state)
6:      case NEW :
7:        (res, row1, col1) ← bookFirstSlot(req, col0,
          round)
8:        nstate ← state
9:        /* Some other helper has booked the first slot
          */
10:       if res = REFRESH then
11:         /* read the state again */
12:         break
13:       else if res = FALSE then
14:         nstate ← pack(FAIL , 0, 0, 0, 0)
15:       else if res = TRUE then
16:         if pathCAS(req, round, slotNum, row1,
            col1) then
17:           nstate ← pack(SOFT,slotNum+1,round,
              row1,col1)
```

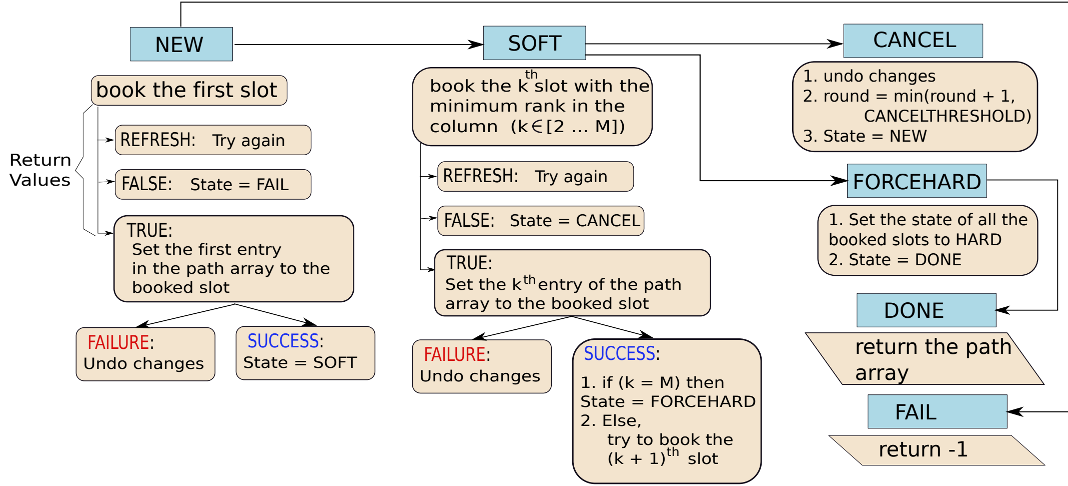

Figure 5. The *process* function

```
18:         else
19:           /* reset the slot in SLOT and SHADOWPATH
              array */
20:           undoSlot(req,round,slotNum,row1,col1)
21:           undoShadowpath(req,slotNum,row1,col1)
22:         end if
23:       end if
24:       /* Point of linearization: If nstate is FAIL and the
          CAS is successful */
25:       req.iterState.CAS(state, nstate)
26:       break
27:     case SOFT :
28:       (round1, row1, col1) ← unpack(req.PATH.
          get(slotNum-1))
29:       if round ≠ round1 then
30:         /* read the state again */
31:         break
32:       end if
33:       (res,row2) ← bookMinSlotInCol(req, col1+1,
          slotNum, round)
34:       if res = REFRESH then
35:         /* read the state again */
36:         break
37:       else if res = FALSE then
38:         col2 = col1 +1 /* changes its starting posi-
            tion */
39:         /* request enters in cancel state */
40:         nstate ← pack(CANCEL ,0,round,0,col2)
41:         if req.iterState.CAS(state,nstate) then
42:           cancelCount.getAndIncrement(req.getTid())
43:         end if
44:       else if res = TRUE then
45:         if pathCAS(req,round,slotNum,row2,col1+1)
            then
46:           if slotNum = numSlots then
47:             nstate ← pack(FORCEHARD , numSlots,
                round, row0, col0)
48:           else
49:             nstate ← pack(SOFT , slotNum+1,
                round, row2, col1+1)
50:           end if
51:         else
52:           undoSlot(req,round,slotNum,row2,col1+1)
53:           undoShadowpath(req,slotNum,row2,col1+1)
54:         end if
55:       end if
56:       /* Point of linearization: If nstate is FORCEHARD
          and the CAS is successful */
57:       req.iterState.CAS(state, nstate)
58:       break
59:     case FORCEHARD :
60:       /* state of cells in SLOT array changes to HARD
          */
61:       forcehardAll(req)
62:       nstate ← pack(DONE , numSlots, round, row0,
          col0)
63:       req.iterState.CAS(state, nstate)
64:     case DONE :
65:       /* return slots saved in the PATH */
66:       return req.PATH
67:     case CANCEL :
68:       /* slots reserved in SLOT array for req are reset,
          PATH array and SHADOWPATH array get clear */
69:       undoPath (req, round)
70:       if cancelCount.get(req.getTid()) < CANCEL
          THRESHOLD then
71:         nround ← round +1
72:       else
73:         nround ← CANCELTHRESHOLD
74:       end if
```

```
75:        nstate ← pack(NEW , 0, nround, row0, col0)
76:        req.iterState.CAS(state, nstate)
77:      case FAIL :
78:        return -1
79:      end switch
80:    end while
81: end function
```

2) $getSlotStatus$: The $bookMinSlotInCol$ method used in Line 33 calls the $getSlotStatus$ method to rank each slot in a column, and chooses a slot with the minimum rank.

The $getSlotStatus()$ method accepts four parameters – req of the thread t_j for which the slot is to be reserved, current $round$ of t_j, the number of the slot($slotNum \in [1 \ldots M]$) that we are trying to book, and the $value$ stored at slot $s_{[row][col]}$. This method returns the rank of $s_{[row][col]}$. Note that all the helpers of a thread have the same tid value. They do not use their original tids while helping other threads.

The state of $s_{[row][col]}$ can be either HARD , SOFT or EMPTY . If t_j owns the slot $s_{[row][col]}$, then there are two possibilities. First, if $s_{[row][col]}$ is already in the HARD state then it means that some other helper has already 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 has converted to HARD for some other request, and we set the rank as HARD . No other thread can take this slot.

If the slot is in the SOFT state and belongs to a different request, then we check if we can cancel the thread 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 .

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.

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

3) $bookMinSlotInCol$: This method reserves a slot in the SLOT array based on its rank. It accepts four parameters – request(req) of a thread t_j, column(col) to reserve a slot in, current round($round$) of t_j, and the slot number($slotNum$). First, we use the $findMinInCol$ method that uses the $getSlotStatus$ method to find the slot with the minimum rank in the column, col, (Line 87) with the corresponding timestamp, $tstamp$. The timestamp is needed for correctness as explained in Section III-E.

Subsequently, all the helpers try to update the SHADOWPATH array at index $slotNum$ with the value – (row, col, $tstamp$) (Line 88), and only one of them succeeds. We read the value ($row1$, $col1$, $tstamp1$) of the entry that is finally stored in the SHADOWPATH array (Line 91-93). We compute its $rank$ in Line 93.

If the $rank$ is BEHIND , then it means that the thread should return to the $process$ method, read the current state of the request, and proceed accordingly. If the rank is AHEAD or EMPTY , we try to reserve the slot $s_{[row][col]}$ (Line 109). Subsequent helpers 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 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 $s_{[row][col]}$ (Line 126). 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. This involves converting all of its SOFT slots to EMPTY , and resetting the PATH and SHADOWPATH arrays. Then the request needs to start anew from the column after the hard wall.

```
82: function bookMinSlotInCol (req, col, slotNum, round)
83:   while TRUE do
84:     /* Set the SHADOWPATH entry */
85:     tid ← req.getTid()
86:     reqid ← req.getReqId()
87:     (row, rank1, tstamp ) ← findMinInCol(req, col, slotNum, round)
88:     shadowPathCAS(req,row,col,tstamp,slotNum)
89:
90:     /* read the values stored in SHADOWPATH array */
91:     (row1,col1,tstamp1) ← req.shadowPath.get(slotNum)
92:     slotVal ← SLOT [row1][col1].get()
93:     rank ← getSlotStatus(req, slotVal, round, slotNum)
94:
95:     /* oldval is the value now saved in SHADOWPATH array and def is the default value */
96:     oldval ← pack(row1, col1, tstamp1, VALID)
97:     def ← pack(0, 0, 0, INVALID)
98:     (reqid1, tid1, round1, slotNum1, stat1) ← unpack (SLOT [row1][col1])
99:     expSval ← pack(reqid1, tid1, round1, slotNum1, tstamp1, stat1)
100:    newSval ← pack(reqid, tid, round, slotNum, tstamp1, SOFT )
101:
102:    switch (rank)
```

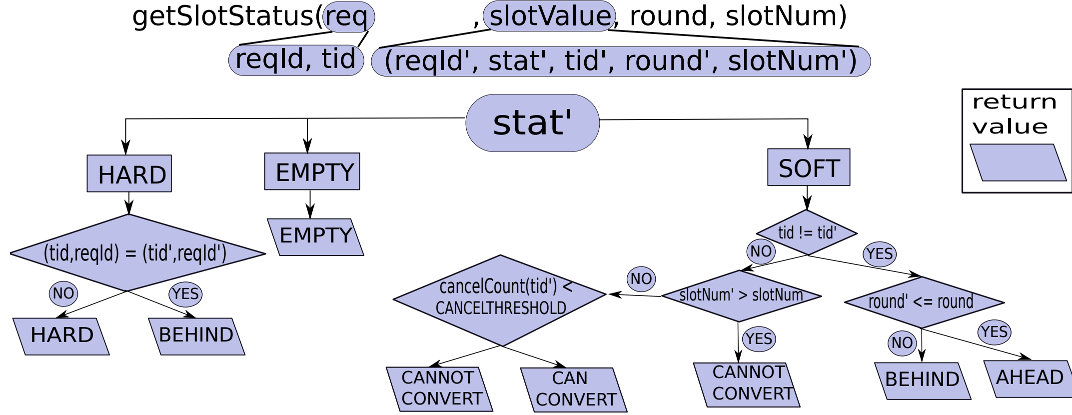

Figure 6. *getSlotStatus*

```
103:        case BEHIND :
104:            /* undo SHADOWPATH array */
105:            req.SHADOWPATH.CAS(slotNum, oldval, def)
106:            return (REFRESH, NULL)
107:        case AHEAD || EMPTY :
108:            /* reserve temporary slot */
109:            if (SLOT [row1][col1].CAS(expSval, newSval)
            = FALSE) ∧ (SLOT [row1][col1].get() ≠ newS-
            val) then
110:                req.SHADOWPATH.CAS(slotNum, oldval, def)
111:                continue
112:            end if
113:            return (TRUE, row1)
114:        case CANCONVERT :
115:            /* try to change other request's state to CANCEL
            */
116:            if otherCancel(tid1, round1) = FALSE then
117:                continue
118:            end if
119:            if (SLOT [row1][col1].CAS(expSval, newSval)
            = FALSE) ∧ (SLOT [row1][col1].get() ≠ newS-
            val) then
120:                req.SHADOWPATH.CAS(slotNum, oldval, def)
121:                continue
122:            end if
123:            return (TRUE, row1)
124:        case CANNOTCONVERT :
125:            req.SHADOWPATH.CAS(slotNum, oldval, def)
126:            process(request.get(tid1))
127:            break
128:        /* All the slots in the column are HARD */
129:        case HARD :
130:        req.SHADOWPATH.CAS(slotNum,oldval, def)
131:        return (FALSE, NEXT)
132:        end switch
133:    end while
134: end function
end
```

4) $otherCancel$: Here, we explain the idea of overwriting someone else's temporary slot. If the rank of the slot is CANCONVERT , it means that the request, r, which owns this slot has reserved less 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 . 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 . 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. If it cannot complete a request, it helps requests that are blocking its way, or changes its starting position upon encountering a hard wall.

E. 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. This 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

upto 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 $\times$ NUMTHREADS $\times$ M. The required number of bits is $5+10+6=21$. Here, we assume a request pattern that requests slots in monotonically increasing order.

In our algorithm, we assume that the SLOT array 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 array, if we can place a bound on the skew between requests' starting slots across threads. If this skew is $\mathcal{W}$, then we can set the size of the SLOT array to $\mathcal{S} > 2\mathcal{W}$, and assume the SLOT array to be circular.

IV. PROOFS

We outline a short proof in this section. Due to reasons of brevity, we omit a detailed formal proof.

Theorem 1. *The $N \times M$ – LF and WF algorithms are linearizable.*

Proof: We need to prove that there exists a point of linearization at which the $schedule$ function appears to execute instantaneously (see Section II-C). Let us try to prove that the point of linearization of a thread, t, is Line 57 when the state of the request is successfully changed to FORCEHARD , or it is Line 25 when the request fails because of lack of space. 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 state (see Section III-D4). Hence, the point of linearization (Line 57) 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 causes them to make permanent changes. Note that before this point, other threads can view temporarily reserved entries. They can perform two actions in response to a temporary reservation – decide to help the thread that has reserved the slot, or cancel themselves. In either case, the thread does not change its starting position.

A thread will change its starting position in Line 38, 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 7. In this figure there are three requests – 1, 2, and 3, and each of them needs 2 slots. Assume that request 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. We can thus conclude that before a thread is linearized, it cannot force other threads to alter its behavior. Thus, we have a linearizable implementation. ∎

Lemma 1. *The $N \times M$ – LF and WF algorithms obey condition 1.*

Proof: Since our algorithms are linearizable (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 38 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. ∎

Lemma 2. *The $N \times M$ – LF and WF algorithms obey condition 2.*

Proof: We need to prove that for a request, r, exactly $numSlots$ entries are allocated in consecutive columns with one entry per column. Line 33 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 88). 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 III-E) 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 array. 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 it up. This does not deter other requests, because they

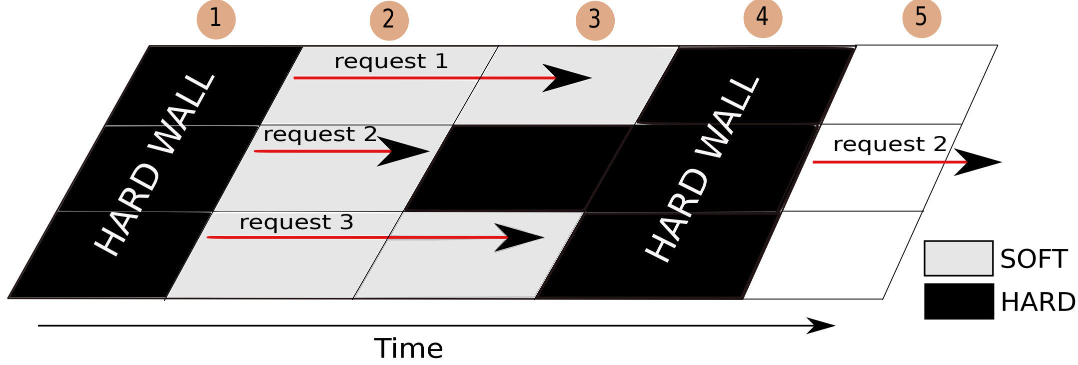

Figure 7. An example containing three requests

will see that the zombie slots belong to an older round of a request, and they can be recycled. Thus, our algorithms obey condition 2. ∎

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

Proof: Theorem 1, Lemma 1, and Lemma 2 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. ∎

Theorem 3. *The $N \times M$–WF algorithm is legal, linearizable, and wait-free.*

Proof: By Theorem 2, we have established that the $N \times M$–LF algorithm is lock-free. To make this algorithm wait-free, we use a standard technique based on the universal construction (see [19]). We need to note that in a lock-free algorithm a thread is unsuccessful, if some other thread is successful. This means that if a thread, t_i, cannot book a slot, then some thread is making progress. Ultimately, the difference in the $requestId$s will exceed the REQUESTTHRESHOLD , and other threads will help t_i to make it successful. ∎

V. EVALUATION

A. Setup

We perform all our experiments on a hyper-threaded dual socket, 64 bit, Dell PowerEdge R810 server. It has two six core 2.66 Ghz Intel Xeon GHz cpus, with 12 MB L2 cache, and 64 GB main memory. 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. [20]). We generated normal variates using the Box-Muller transform (mean = 5, variance = 3* tid). We run the system till the fastest thread completes κ requests. We define two quantities – mean time per request (t_{req}) and total work done (wk). The work done is defined as the total number of requests completed by all the threads divided by the theoretical maximum. $wk = tot_requests/(\kappa \times NUMTHREADS)$. wk measures the degree of imbalance across different threads. It varies from 0 to 1(max).

We set a default REQUESTTHRESHOLD value of 50, and κ to 10,000. We varied the number of threads from 1 to 24 and measured t_{req} and wk for the $N \times M$ variants of the problem. We perform each experiment 10 times, and report mean values. We consider three flavors of our algorithms – lock-free (LF), wait-free (WF), and a version with locks (LCK).

B. Performance of the $N \times M$ Algorithm

Figure 8 and 9 present the results for $N \times M$ ($M = 3$). We observe that LCK is three orders of magnitude slower than LF and WF. The performance of LF and WF is roughly equal, whereas, the work done is 1-10% more for WF for systems with more than 20 threads.

1) Sensitivity : Varying Capacity (N): Figures 10 and 11 show the time per request for different capacities for LF and WF respectively. We observe that the performance across different capacities is roughly similar till 15 threads. After that, there is some limited variation for the WF algorithm. However, for large capacities($N = 22$), the LF algorithm's delay increases significantly.

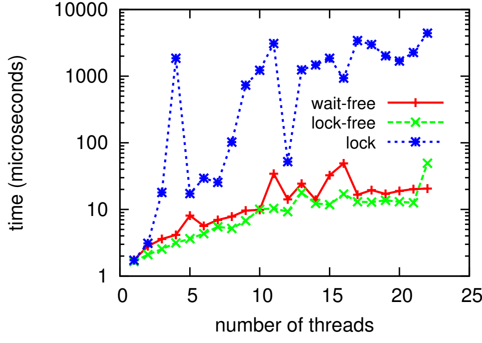

Figure 8. t_{req} for the $N \times M$ algorithm

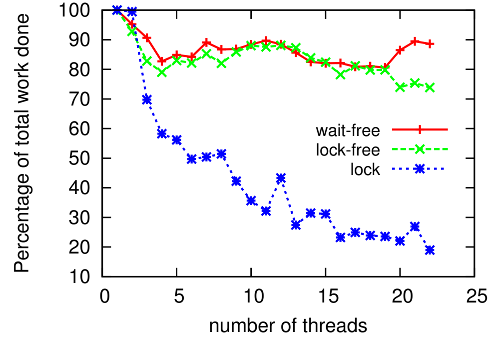

Figure 9. Work done (wk)

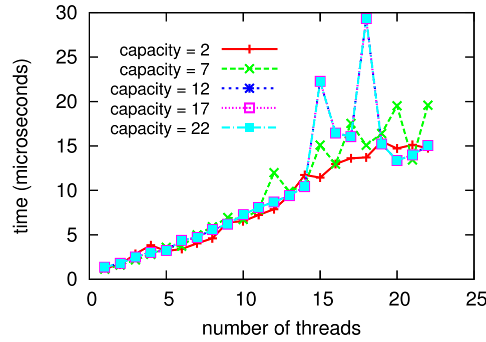

Figure 10. t_{req} for LF across different capacities

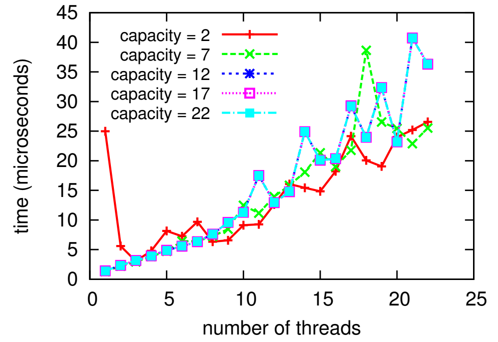

Figure 11. t_{req} for WF across different capacities

2) Sensitivity: Varying $numSlots(M)$: Next, we evaluated the performance of lock-free and wait-free by varying the number of slots per request (M) in Figures 12 and 13 respectively. We observe that with an increasing number of slots, the performance of LF varies from 1-20 μs, whereas the performance of WF varies from 1-160 μs. For high values of the number of threads and $numSlots$, WF is around 5-8X slower. We observe that the total work done(wk) is roughly similar for both LF and WF across different slots. Due to lack of space, we do not show the results for wk.

3) Sensitivity: Varying REQUESTTHRESHOLD : We show the performance of WF across two values of the REQUESTTHRESHOLD (10 and 50) in Figure 14. For a REQUESTTHRESHOLD of 10, t_{req} shoots up after we have more than 13 threads in our system. This configuration is 2.5X slower than the configuration that uses a value of 50. We observe that the work done is also higher for a REQUESTTHRESHOLD of 10.

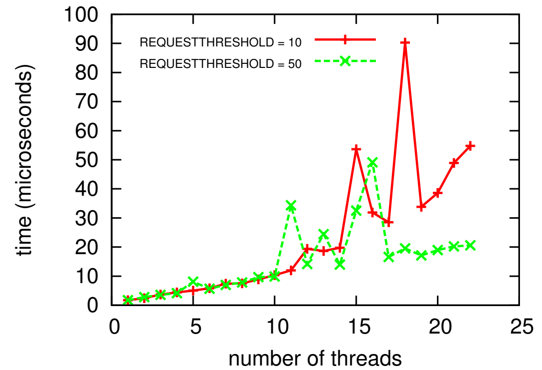

Figure 14. t_{req} for WF across different values of REQUESTTHRESHOLD

VI. CONCLUSION

In this paper, we presented a lock-free and wait-free algorithm for four variants of the generic slot scheduling problem. The single slot algorithms are simple, whereas, the complicated multiple slot algorithms use recursive helping and cancellation. We prove the linearizability correctness condition for all of our algorithms, and lastly experimentally evaluate their performance. The wait-free version is slower than the lock-free version by 3-8X in the worst case. However, it manages to do more work per unit time. Both the lock-free and wait-free versions are several orders of magnitude faster than algorithms that use locks.

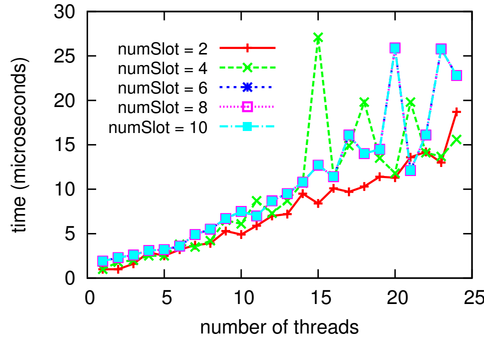

Figure 12. t_{req} for LF with different values of $numSlots(M)$

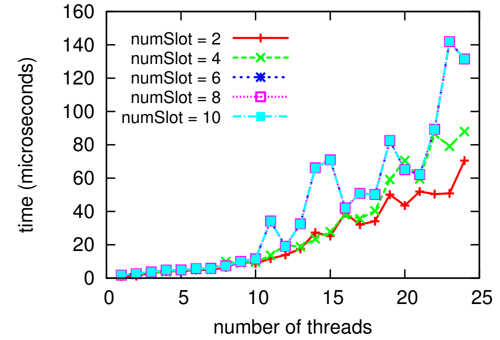

Figure 13. t_{req} for WF with different values of $numSlots(M)$

VII. Acknowledgments

We would like to express our thanks to Gaurav Makkar, Ajay Bakre, and Siddhartha Nandi from NetApp India, for the useful discussions that we had.

References

[1] 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.

[2] S. Kumar, C. J. Hughes, and A. Nguyen, "Carbon: Architectural Support for fFne-Grained Parallelism on Chip Multiprocessors," in *Proceedings of the 34th annual international symposium on Computer architecture*, ser. ISCA '07, 2007, pp. 162–173.

[3] 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.

[4] A. Kogan and E. Petrank, "Wait-free Queues With Multiple Enqueuers and Dequeuers," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, ser. PPoPP '11, 2011, pp. 223–234.

[5] B. Hall, "Slot Scheduling: General Purpose Multiprocessor Scheduling for Heterogeneous Workloads," Master's thesis, University of Texas, Austin, december 2005.

[6] J. K. Ousterhout, "Scheduling Techniques for Concurrent Systems," in *International Conference on Distributed Computing Systems*, October 1982, pp. 22–30.

[7] 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.

[8] E. Dekel and S. Sahni, "Parallel Scheduling Algorithms," vol. 31, pp. 31 – 49, 1983.

[9] L. Mhamdi and M. Hamdi, "Distributed Parallel Scheduling Algorithms for High-Speed Virtual Output Queuing Switches," in *Computers and Communications, 2009. ISCC 2009. IEEE Symposium on*, july 2009, pp. 944 –949.

[10] J. Keller and R. Gerhards, "PEELSCHED: a Simple and Parallel Scheduling Algorithm for Static Taskgraphs," in *PARS*, 2011.

[11] M. B. Jones, D. Roşu, and M.-C. Roşu, "CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities," in *Proceedings of the sixteenth ACM symposium on Operating systems principles*, ser. SOSP '97, 1997, pp. 198–211.

[12] C.-S. Lin, M.-Y. Wu, and W. Shu, "Efficient Algorithms for Slot-Scheduling and Cycle-Scheduling of Video Streams on Clustered Video Servers," *Multimedia Tools Appl.*, vol. 13, no. 2, pp. 213–227, Feb. 2001.

[13] 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 *Vehicular Technology Conference (VTC 2010-Spring), 2010 IEEE 71st*, may 2010, pp. 1 –5.

[14] X. Liu, N. Saberi, M. J. Coates, and L. G. Mason, "A Comparison between Time-slot Scheduling Approaches for All-Photonic Networks," in *Int. Conf. on Inf., Comm. and Sig. Proc (ICICS)*, 2005.

[15] J. Feldman, S. Muthukrishnan, E. Nikolova, and M. Pal, "A Truthful Mechanism for Offline Ad Slot Scheduling," *CoRR*, vol. abs/0801.2931, 2008.

[16] I. Goiri, 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 International Conference for*, nov. 2011, pp. 1 –11.

[17] S. V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *IEEE Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[18] 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.

[19] M. Herlihy and N. Shavit, *Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008.

[20] 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.