Software Transactional Memory Friendly Slot Schedulers

No Author Given

No Institute Given

Abstract. In this paper, we discuss the design space of highly concurrent linearizable data structures for slot scheduling. We observe that it is not possible to have high *fairness* across threads, and maximize *throughput* of the entire system simultaneously. Lock free algorithms are very fast, yet very *unfair*, and wait free algorithms follow the reverse trend. We thus propose a class of algorithms using software transactional memory (STM) that are in the middle of the spectrum. They equitably balance fairness and throughput, and are much simpler to design and verify.
Keywords software transactional memory (STM), transactions, wait-free, lock-free, schedulers, slot scheduling

1 Introduction

Multicore and manycore processors are increasingly replacing ASICs in large scale enterprise level systems. For example, multicore processors are beginning to be extensively used in high throughput networking systems [9], high volume storage systems [10], and for implementing fast software base switches in the Telecom industry. An integral component in all of these systems is the *scheduler*, whose role is to accept requests from multiple producers, and dispatch them to multiple consumers with respect to a given optimality criteria. For example, a scheduler in a wireless networking application accepts requests from different ingress links, schedules them, and dispatches them along different egress links. To support these novel applications of multicore processors, it is necessary to implement fast schedulers that have high throughput, and can simultaneously schedule requests for a large number of tasks.

In this paper, we consider an important subset of schedulers namely *slot schedulers* [1] that discretize time into fixed-length timeslices called *slots*. A slot is either completely occupied by a task, or it is empty. We design a parallel *scheduler* data structure in this paper that can form the core logic of slot schedulers. It can schedule requests for 64 threads in parallel.

In this paper, we first discuss the design space of slot scheduling algorithms, and classify them on the basis of three attributes – *throughput*, *fairness*, and *complexity*. The number of requests scheduled per second is the throughput, and the ratio of the average number of requests scheduled per thread and the maximum number of requests scheduled per thread is referred to as *fairness*. We show that lock free algorithms are fast, yet are not very fair, whereas, wait free algorithms have a high degree of fairness, yet are slow. Both of these algorithms are fairly complicated are at least an order of magnitude faster than the algorithms with locks (conclusion similar to Sarangi and Aggarwal [4]).

The main contribution of this paper is to propose parallel scheduling algorithms using software transactional memory(STM). These algorithms are simpler than the corresponding non-blocking versions, provide strong consistency guarantees (linearizability), and equally balance fairness and throughput. Moreover, we demonstrate that creating high performance parallel data structures using transactional memory is non trivial. There is a trade-off between the size of a transaction, time, and fairness.

2 Background

2.1 Basics of Slot Scheduling

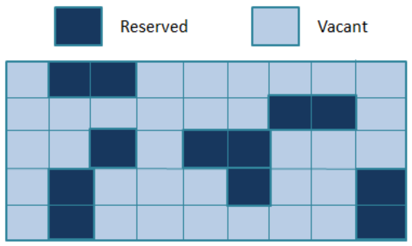

Fig. 1. The Ousterhout matrix for scheduling

Figure 1 shows an Ousterhout matrix that is typically used for slot scheduling. Here, the columns represents time (in slots) and the rows represents resources (core, disk, network link), where we assume that resources are homogeneous. Each cell(slot) can either be *free* or *busy*. We consider one of the generic versions of the parallel slot scheduling problem, where we have M rows (see [4]) (each row corresponding to a resource), and we wish to schedule a request that occupies N consecutive slots (1 in each column) starting from a given slot number. We can model multiple types of resources by considering multiple instances of this problem (one for each type of resource). Furthermore, it is possible to consider dependencies between tasks by annotating each cell with the task id, and ensuring that a task is scheduled after all of its predecessors are scheduled. In this paper, we just focus on solving the kernel of the scheduling problem, which is to place N requests as early as possible.

2.2 Parallel Programming Paradigms

Traditionally, parallel programming is done with the help of locks, which are slow in practice because tasks that might be holding the locks might get swapped out, and sometimes we are overly conservative in placing locks. Hence, researchers are increasingly considering non-blocking algorithms that do not use locks.

Non-blocking *lock free* algorithms ensure that at any point of time, at least one thread is making forward progress. They thus guarantee forward progress of the entire system as a whole. However, lock free algorithms do not guarantee fairness across threads, and it is possible to have *starvation.* In comparison, non-blocking wait free algorithms explicitly guarantee fairness across threads. They ensure that every thread completes its request in less than n internal steps. Faster threads help slower threads to complete their requests. This is known as *external helping.* This strategy ensures that no thread waits indefinitely. Wait free algorithms are typically slower than their lock free counterparts, and are much more complicated. *Linearizability* is a commonly used correctness criteria for such concurrent parallel data structures. An operation is *linearizable* if it appears to execute in an infinitesimally small instant of time (details given in [8]).

2.3 Software Transactional Memory (STM)

Transactional memory [11] is a programming abstraction that treats a block of code as an atomic unit akin to a database transaction. It obeys the ACID properties similar to database transactions. We use the DEUCE STM framework developed by Korland, Shavit, and Felber [3], which is a complete, compiler independent, STM framework in Java. DEUCE does not need to modify the JVM, nor does it require any language extensions, or additional APIs.

Deuce implements atomic blocks at the granularity of methods. The methods that should execute as transactions are annotated by adding the *@Atomic* keyword. The applications layer consists of user's classes with annotated atomic functions. Next comes the DEUCE runtime layer, which detects atomic methods, and implements an STM framework to execute these methods as transactions. By default, it allows disjoint access parallelism, which means that transactions that access disjoint sets of data can execute in parallel.

2.4 Related Work – Slot Scheduling

Scheduling is a classic problem. Most variants of scheduling that consider dependencies between tasks, assume multiple nodes, and try to minimize the makespan, have been proven to be NP hard. However, most parallel schedulers simply look at simple *earliest possible* scheduling, and their main aim is to minimize the time that it takes to schedule a request.

Aggarwal and Sarangi [4] have recently proposed lock-free and wait-free implementations of slot schedulers. Figure 2 shows the sequence of steps in a successful iteration of the lock free algorithm. We first try to temporarily reserve slots by marking them. If we can successfully mark all the slots that we require, then we proceed to permanently reserve them. It is possible that multiple threads may try to reserve the same slot. If two threads contend for a slot, then one of the threads needs to back out. This can happen in two ways - either it can cancel itself and start anew, or it can decide to help the high priority thread. Helping another thread in this manner is know as – *internal helping.*

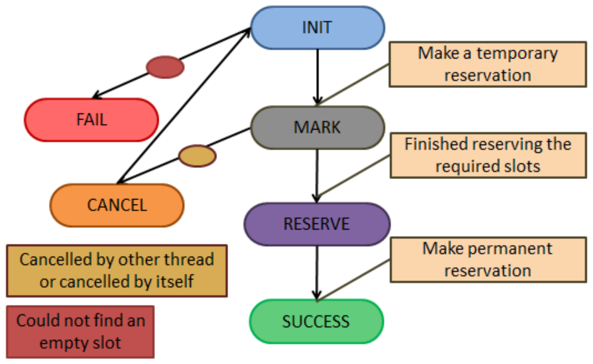
INIT
Make a temporary reservation
FAIL
MARK
Finished reserving the required slots
CANCEL
RESERVE
Make permanent reservation
Cancelled by other thread or cancelled by itself
Could not find an empty slot
SUCCESS

Fig. 2. FSM of Slot Scheduling Algorithm

The authors propose to first try canceling one of the threads, and then resort to internal helping.

Their algorithm broadly consists of three stages.

1. The request is created in the INIT state. The thread subsequently tries to temporarily reserve the first available slot. After doing so, the request moves to the MARKED state.
2. In the MARKED state of the request, a thread marks the rest of the slots specified in the request. Once the required number of slots are marked, the request progresses to the RESERVE state.
3. In the RESERVE state, the marked slots are finally permanently booked by the thread. After this, the request moves to the SUCCESS state where the list of slots allotted is returned

For the *wait-free* implementations, threads need to help each other at the highest level *(external helping)*. Before proceeding with its own request, a thread needs to help all the threads that have been waiting for a longer time, or alternatively have a higher priority. The presence of multiple helpers significantly complicates the algorithm and it becomes necessary to ensure that the state of a request is not polluted. The authors of [4] propose several data structures to handles these issues. The goal of our work is to develop parallel slot scheduler algorithms with STMs that solve the same problem proposed by Aggarwal and Sarangi [4], and are linearizable.

3 Parallel Slot Scheduling using STMs

3.1 Data Structures

We use a 2-dimensional array called SCHEDULEMAP similar to the Ousterhout matrix. This array is used to maintain the status of all the slots. The inputs to the *slotSchedule* operation are *threadId*, number of slots to be reserved (*numSlots*) and the starting time slot(*startSlot*) from which the thread wishes to book the slots. The *slotSchedule* operation returns the list of slots allotted to the thread. The state of a cell in the SCHEDULEMAP array can take three values: VACANT , MARKED and RESERVED . VACANT means that the slot is free and a request can be scheduled for this slot. MARKED means that the cell is temporarily reserved, and RESERVED means that the slot is permanently reserved. As a request gets scheduled for a slot, the state of the slot in the SCHEDULEMAP array changes from VACANT to MARKED and eventually it becomes RESERVED . This is shown in Figure 4. However, in one of our algorithms, the state of the slot directly changes from VACANT to RESERVED . Whenever a thread places a new request to reserve the slots, an instance of the REQUEST class gets populated (see Figure 3). The *state* field maintains the state of the request, number of slots already marked (*slotCount*) and index of the last marked slot in the SCHEDULEMAP array as shown in Figure 4. The RECORD array stores the slots reserved by the thread.

```
class Request{
     long state,
     int [][] record,
     int startSlot,
     int numSlots
};
```

Fig. 3. The REQUEST class

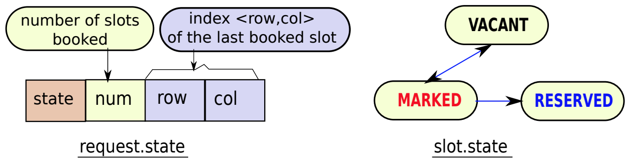

Fig. 4. request.state and slot.state

3.2 The *SimpleScan* Algorithm

A thread needs to book *numSlots* slots in adjacent columns. There are various ways in which we can solve this problem using an STM framework. First and the simplest of all the methods is shown in Algorithm 1, which we call *SimpleScan*. Suppose thread t_i places a request to book n slots starting from time slot x. t_i first scans the SCHEDULEMAP array to find n consecutive VACANT slots starting from slot x (Line 3). When a slot is found to be VACANT , t_i stores its index in a local array RECORD (Line 5). Once the required number of slots are found, t_i books these slots by changing the state of the slot from VACANT to RESERVED (Line 11). This method is annotated with the keyword, *@Atomic* (Line 1) indicating that all the instructions in this method are enveloped in one transaction and are executed atomically. The STM system monitors the transactions of the threads and if two or more threads concurrently access the SCHEDULEMAP array at the same index, it handles the conflict by allowing one thread to commit its transaction and abort/restart the other transaction(s) involved in the conflict.

Algorithm 1 *SlotSchedule- SimpleScan*

```
@Atomic
function slotSchedule (tid, slot,$numSlots$)
    for i $\in$ [slot, SCHEDULEMAP .length] $\wedge$ (slotCount $<$ $numSlots$) do
        if $\exists$ j, SCHEDULEMAP [i][j] = VACANT then
            record[index++] $\leftarrow$ j , slotCount++
        else
            reset $slotCount$ and $record$, start searching again
        end if
    end for
    if slotCount = $numSlots$ then
        /* reserve the slot by setting the threadId in SCHEDULEMAP array */
        return record;
    else
        return FAIL ;
    end if
end function
```

end

3.3 The *SoftVisible* Algorithm

The main issue with the *SimpleScan* algorithm is that it creates one large transaction for processing the entire request. Since processing transactions frequently requires $O(n^2)$ operations, and the chance of conflicts is high with larger transactions, we wish to design an algorithm that breaks the schedule operation into several mini-transactions. Instead of keeping track of VACANT slots in a local structure, a thread saves some temporary information in cells of the SCHEDULEMAP array and this is made visible to concurrent threads when the transaction commits. The purpose of doing this is to notify other transactions of the state of the SCHEDULEMAP array so that other threads can reserve their slots accordingly.

Algorithm 2, shows the *SoftVisible* algorithm that divides the schedule operation into 5 steps. At the outset, a thread starts a transaction (Line 7) by calling the method *findFirstSlot* in which it tries to find the earliest possible VACANT slot available at or after the requested starting slot (*startSlot*). We start out by invoking the *markSlot* (Line 25) method that marks a VACANT slot, and changes its state to MARKED . Concomitantly, the request moves to the (MARKED , *slotCount*) state. After successfully finding the first slot, the *markRestSlots* method is called (Line 9) to mark the rest of the required slots. This method tries to mark ($numSlots - 1$) slots in the SCHEDULEMAP array, where the slots are in adjacent columns.

Once *numSlots* slots are marked, a thread enters the RESERVE state. In this state the marked slots are reserved by changing their state from MARKED to RESERVED (Line 60), as shown in method *reserveAllSlots*. The state of the request changes from RESERVE to SUCCESS indicating that the request is successfully scheduled. Subsequently, the *slotSchedule* operation returns the list of the allotted slots (in the RECORD array (Line 15).

markSlot(): This method is invoked to change the state of a slot from VACANT to MARKED in a given column of the SCHEDULEMAP array. This method accepts two parameters- the request and the column id. It returns the booked slot (if possible), and the *status*, which can be – SOFTRETRY , HARDRETRY , TRUE or FALSE . SOFTRETRY means that all the slots in the column are in the MARKED state. In this case, we wait for the slots to turn into either VACANT or RESERVED (Line 24 and Line 40). HARDRETRY means that all the slots in a column are already in the RESERVED state. TRUE indicates that a slot is successfully marked, and FALSE indicates that the request cannot be scheduled because we reach the end of the SCHEDULEMAP array (thus transition to the FAIL state (Line 29)).

If the *markSlot* returns HARDRETRY , then it is clear that no slot in the specified column can be booked. If we are still looking to book our first slot, then we can start from the next column. However, if we have already booked some slots, then we need to cancel the request by moving to the CANCEL state (Line 45). We need to convert the state of all the MARKED slots to VACANT and clear the RECORD array.

All these methods, which are executed as transactions run in a loop (Line 2) until the state of the request becomes either SUCCESS or FAIL . SUCCESS state means that required number of slots are reserved by a thread and FAIL state means that the thread is not able to schedule its request. Breaking the schedule operation in this format makes the changes done at cell level in SCHEDULEMAP array immediately visible. This is shown with help of a flowchart in Figure 5.

Algorithm 2 *SlotSchedule-SoftVisible*

```
 1: function slotSchedule (request)
 2:   while TRUE do
 3:     state ← request.getState()
 4:     switch (state)
 5:     /* a request is created in the START state */
 6:     case START :
 7:       findFirstSlot(request) /* find the first slot (if possible) */
 8:     case MARKED :
 9:       markRestSlots(request) /* mark slots in the SCHEDULEMAP array */
10:     case RESERVE :
11:       reserveAllSlots(request) /* reserve the marked slots */
12:     case CANCEL :
13:       cancelSlots(request) /* clear the marked slots */
14:     case SUCCESS :
15:       return request.record /* return the reservation path array */
16:     end switch
17:     return FAIL
18:   end while
19: end function
20:
```

```
@Atomic
function findFirstSlot(request)
    startSlot ← request.getStartSlot() , status ← SOFTRETRY
    while status ∈ (SOFTRETRY , HARDRETRY ) do
        (status, row, col) ← markSlot(request,startSlot.col) /* index of the slot */
        (status = HARDRETRY ) ? startSlot++ : startSlot)
    end while
    if status = FALSE then
        newState ← FAIL /* unable to fulfill the request */
    else if status = TRUE then
        /* save the slot in the record array and move to next state */
        newState ← (MARKED , 1, row, col)
    end if
    request.state ← newState /* change the state of the request */
end function

@Atomic
function markRestSlots(request)
    (row, col) ← state.getLastSlot(), status ← SOFTRETRY
    while status = SOFTRETRY do
        (status, row, col) ← markSlot(request, col+1)
    end while
    if status = HARDRETRY then
        newCol ← col +1
        newState ← (CANCEL , 0, 0, newCol) /* could not find VACANT slot */
    else if status = TRUE then
        /* save the slot allotted in the record array */
        if numSlots = slotNum then
            newState ← RESERVE /* all the slots are marked */
        else
            newState ← (MARKED , slotNum+1, row, col+1) /* some slots are left
            to be marked */
        end if
    end if
    request.state ← newState /* set the new request state */
end function

@Atomic
function reserveAllSlots(request)
    for slot ∈ request.record do
        slot.state ← RESERVED
    end for
    request.state ← SUCCESS
end function

@Atomic
function cancelSlots(request)
    (row, col) ← state.getLastSlot()
    undo (request) /* Clear record and SCHEDULEMAP array */
    newState ← (START ,0,0,col+1) /* set the request state as START */
```

```
70: end function
end
```

3.4 The *SoftVisibleMerge* Algorithm

Another way of implementing the *SoftVisible* algorithm is to merge the RESERVE and CANCEL stages into one RESERVE stage. Thus, instead of having four to five transactions/stages per *slotSchedule* operation there can be three stages, as shown in Figure 5. This strategy performs slightly better (see Section 4) on our test system.

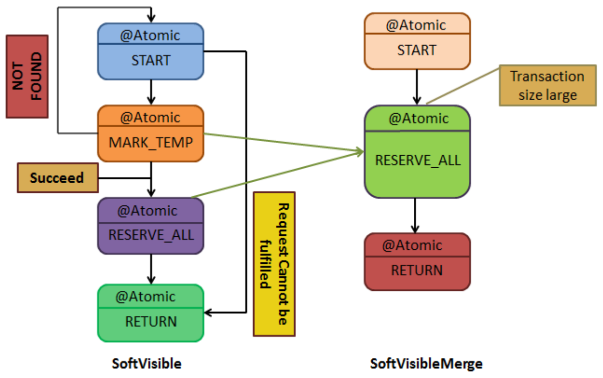

Fig. 5. Flowchart of the *SoftVisible* and *SoftVisibleMerge* algorithms

Proof: We can prove that irrespective of the STM model, all three of our algorithms obey sequential semantics (their execution matches that of a single thread), and are linearizable. The main idea of the proof is that we start searching for a new set of slots only when we encounter a column that has all of its entries in the RESERVED state. This ensures that we do not take decisions based on the temporary state of unfinished requests. Secondly, once a request has finished, its state in the SCHEDULEMAP array cannot be overwritten. We can use these two observations to prove linearizability. Due to lack of space, we request the interested reader to kindly look at a technical report posted anonymously at [12].

4 Evaluation

4.1 Experimental Setup

The performance of the proposed scheme was evaluated on a hyper-threaded four socket, 64 bit, Dell PowerEdge R820 server. It has four eight core 2.20GHz

Intel(R) Xeon(R) cpus, 16 MB L2 cache, and 64 GB main memory. It runs on Ubuntu Linux 12.10 using the generic 3.5.0-17 kernel. All our algorithms are written in Java 6 using Sun OpenJDK 1.6.0_27. We use the DEUCE STM framework.

We use a synthetic distribution to evaluate the performance of our slot scheduling algorithms (similar to [5, 4]). The assumption is that the request inter-arrival time is normally distributed. We use the Box-Muller transform (mean = 10, variance = 5**threadid*) to generate normal variates, and run the experiment till the fastest thread completes κ requests. We define two quantities – mean time per request (*time*) and *fairness* (frn). The *fairness* is defined as the total number of requests completed by all the threads divided by the theoretically maximum number of requests that could have been completed. $frn = tot\_requests/(\kappa \times NUMTHREADS)$. If the value of our fairness metric is equal to 1, then all the threads complete the same number of requests. The lower is the fairness, more is the discrepancy in performance across threads.

We set $\kappa = 10,000$, and vary the number of threads from 1 to 64. We repeat each experiment ten times to reduce the variance in the results and report the mean values. We compare all our algorithms SimpleScan($SimScan$), SoftVisible($SofVis$) and SoftVisibleMerge($SofVisMer$) with wait-free(WF) and lock-free(LF) algorithms as presented in [4]. We have not included locked version for slot scheduling algorithm as it is an order of magnitude slower than the rest.

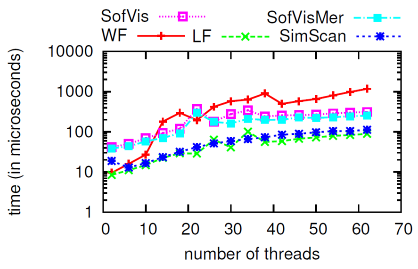

Fig. 6. Time

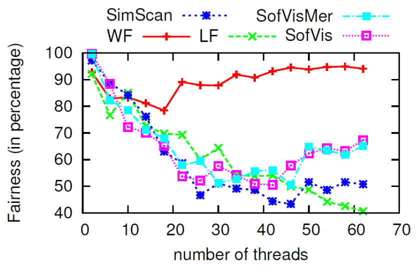

Fig. 7. *fairness*

4.2 Performance of the Algorithm

Figures 6 and 7 show the results for $numslots = 3$. We observe that the WF implementation is 5-10 times slower (1170 μs for 64 threads) than others. In terms of time taken per request, $SimScan$ is as fast as LF. Both the algorithms take around 90-110 μs for 64 threads. $SofVis$ and $SofVisMer$ lie in the middle of the spectrum with $SofVisMer$ being faster by 20%. They are 2-3 times slower than $SimScan$ or LF, and are 3-4 times faster than WF. The reason for this trend is that WF and STM based algorithms have a fair amount of overhead. This slows them down as compared to LF. Among the STM algorithms, $SimScan$ has the least amount of overhead because we have a single transaction, whereas $SofVis$

and $SofVisMer$ create 3-5 transactions. However, $SofVis/SofVisMer$ also have a lesser number of aborts per commit (Figure 8). Because of their smaller transaction size, chances of conflicts among the transactions reduce. The relative reduction in aborts does not offset the overheads of creating additional transactions, and thus STM algorithms with a larger number of transactions are slower in practice.

The fairness results shown in Figures 7 and 9 show a reverse trend. WF is the most fair algorithm ($frn > 90\%$). LF and $SimScan$ are the most unfair algorithms because they follow a "winner take all" strategy. Their fairness values decrease from 92% for 2 threads till 50% for a system of 50 threads. Beyond 50 threads, $SimScan$ is better than LF by 5-20%. In comparison, $SofVis$ and $SofVisMer$ start outperforming $SimScan$ and LF in terms of fairness, and their frn values jump to 60-65% beyond 50 threads. This is because these algorithms create multiple transactions, and it becomes easier for all the threads to make some incremental progress. We have observed that as the granularity of transactions gets finer, the fairness across threads increases.

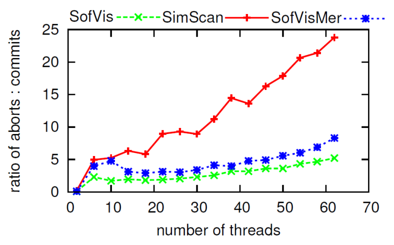

Fig. 8. Ratio of aborts to commits

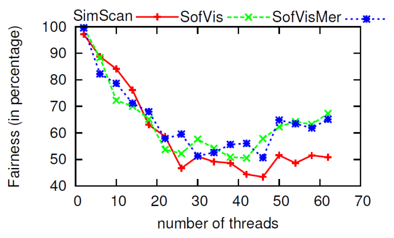

Fig. 9. Fairness (STM based algorithms only)

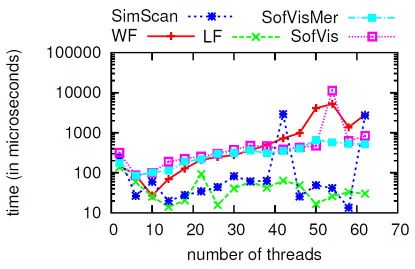

Fig. 10. Time (with jitter)

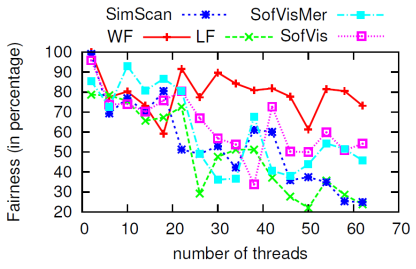

Fig. 11. Fairness (with jitter)

Subsequently, we performed experiments to evaluate the performance of our algorithms with some degree of jitter by running another copy of the same experiment in parallel (with 5 threads). Figure 10 shows the time taken per request. In this case, we see that all the algorithms become slower by a factor of 2-3, and the trends remain the same. The fairness values for all the algorithms also dip; however, the least amount of degradation is shown by $SofVis$ and $SofVisMer$. The fairness value for WF dips by 22% for 64 threads, whereas their fairness value dips by only 14%. We also tried adding the *helping* feature in STM based implementations. There was no performance benefit as the read-write set for the various helpers overlap and this led to more aborts.

5 Conclusion

We demonstrated that there is a trade off between throughput and fairness. In a concurrent system, if each thread is suppose to do a pre-specified amount of work, then we aim to achieve high fairness across the threads. But if the aim is to complete as much of work as possible, then it is desirable to have high throughput. Experiments demonstrate that lockfree slot schedulers provide high throughput, whereas waitfree slot schedulers have very high fairness. STM friendly Slot Schedulers represent an equitable trade-off between fairness and throughput. The *SimpleScan* algorithm is as fast as lock-free. Moreover, all the STM based algorithms have much simpler designs and are thus less error prone. Hence, it is easier to reason about their correctness.

References

[1] Brandon Hall: Slot Scheduling: General Purpose Multiprocessor Scheduling for Heterogeneous Workloads, Ph.D Thesis, University of Texas, Austin, 2005

[2] J. K. Ousterhout: Scheduling Techniques for Concurrent Systems, in International Conference on Distributed Computing Systems, 1982

[3] G. Korland, N. Shavit, and P. Felber. : Non invasive concurrency with Java STM. in MultiProg 2010

[4] Pooja Aggarwal and S.R Sarangi: Lock-free and Wait-free Slot Scheduling Algorithms. in IPDPS 2013

[5] Sarah Sellke and Ness B. Shroff and Saurabh Bagchi and Chih-Chun Wang: Timing Channel Capacity for Uniform and Gaussian Servers. in Allerton Conference, 2006

[6] M. Herlihy and E. Moss: Transactional memory: Architectural support for lock-free data structures, in ISCA 1993

[7] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In OOPSLA, 2006

[8] Herlihy, Maurice P. and Wing, Jeannette M.: Linearizability: A Correctness Condition for Concurrent Objects. ACM Trans. Program. Lang. Syst. Volume 12, pages 463-492.

[9] Jia-Ming Liang and Jen-Jee Chen and Ho-Cheng Wu and Yu-Chee Tseng: Simple and Regular Mini-Slot Scheduling for IEEE 802.16d Grid-based Mesh Networks, in Vehicular Technology Conference, 2010

[10] Q. Zhu, Z. Chen and L. Tan, Y. Zhou and K. Keeton and J. Wilkes. Hibernator: Helping disk arrays sleep through the winter, in SOSP 2005

[11] N. Shavit and D. Touitou. Software transactional memory. Distributed Computing, Special Issue(10):99116, 1997.

[12] Anonymous: Proof of Linearizability of STM Algorithms, posted at https://sites.google.com/site/slotstmproof/home/proof.pdf

6 Proofs

Lemma 1. *The Deuce STM system implements linearizable transactions.*

Proof. Linearizability means that the method appears to execute instantaneously. This means that at a certain point between the invocation, and the final termination of a transaction, there is a point, at which it appears to execute instantaneously. This point is known as the *point of linearizability.* Linearizability has been proved to be equivalent to the case of a sequential execution, where the sequential order is consistent with the real time order [8]. In the case of the Deuce STM system, the execution of transactions that mutually conflict with each other is equivalent to a sequential execution. We are not concerned about non-conflicting transactions because there is no way to detect a violation of the sequential order of executions. Secondly, before a transaction finishes, the Deuce STM system ensures that its write set is committed, and is made visible to the transactions that start after it in real time order. In this sense the order of committing a transaction is equivalent to its real time order. This means that if transaction A begins after transaction B finishes, B is guaranteed to see the writes performed by A. The Deuce STM system, ensure this semantics. Since both the conditions of linearizability – equivalent sequential execution, and adherence to a real time order are satisfied by the native Deuce STM system – we can conclude that the execution of transactions in Deuce is linearizable. Furthermore, the point of linearizability is at which the transaction commits.

Lemma 2. *The $SimpleScan$ algorithm is linearizable.*

Proof. The entire *SimpleScan* algorithm is encapsulated in one transaction. According to Lemma 1, transactions in Deuce are linearizable. Consequently, the *SimpleScan* method is also linearizable.

Theorem 1. *The $SoftVisible$ and $SoftVisibleMerge$ algorithms are linearizable.*

Proof. Let us try to prove that the point of linearizability for a thread, t_i, is when the state of the request is changed to RESERVE (In Algorithm 2 line 49), or when the request fails to reserve any slots (i.e the state of the request is changed to FAIL (line 29)). When the status of the request becomes RESERVE , it means that t_i has successfully marked the required number of slots (Line 48) and now the request can finally reserve the marked slots.

Now, after the state of the request has turned to RESERVE , the thread needs to create a new transaction to change the status of the MARKED slots to RESERVED . It is possible that this transaction gets aborted multiple times. We need to ensure that the slots which thread t_i has already set as MARKED are not overwritten by other threads since they are visible to all the concurrent threads. In our implementation, a thread tries to modify only the VACANT slots (in method $markSlot()$) or the slots marked by itself (in method $reserveAllSlots()$). This ensures that in our implementation no thread overwrites the slots marked or reserved by some other thread. Irrespective of the fact that the transaction can abort, it will eventually (disregarding starvation) be able to reserve the slots it has marked earlier. We can thus conclude that when the state of a request is set to RESERVE , the list of slots marked by the thread is fixed and these slots will eventually transition to the RESERVED state. If a request is failing, then this outcome is independent of other threads, since the request has reached the end of the SCHEDULEMAP array.

Likewise, we need to prove that before the point of linearizability, no events visible to other threads causes them to make permanent changes. Note that before this point, other threads can view the intermediate state of the slots (i.e MARKED) as the *slotSchedule* operation is divided into mini-transactions. Seeing this state of the slots, other threads will wait for the state to turn into either VACANT or RESERVED (as can be seen in method $findFirstSlot()$ and $markRestSlots()$ at Lines 24 and 40 respectively).

The idea here is that, we start searching for a new set of slots only when we encounter a column that has all of its entries in the RESERVED state (functions $findFirstSlot()$ and $markRestSlots()$). In the case of marking the first slot, the index *startSlot* is incremented only when the function *markSlot* returns HARDRETRY (Line 26), indicating all the entries in column *startSlot* are in the RESERVED state. Whereas, we enter the CANCEL state if *markSlot* returns HARDRETRY while marking rest of the slot (Line 45). This means that the threads that have made the entries in a column as RESERVED can alter the behavior of other threads. This ensures that we do not take decisions based on the temporary state of unfinished requests but only on the basis of those threads that have already passed their point of linearizability We can thus conclude that before a thread is linearized, it cannot force other threads to alter its behavior.

We have thus proved that before the point of linearizability, it is not possible for a thread to make its changes permanent, and after the point of linearizability its changes are permanent. Secondly, we have also proved that before a thread reaches its point of linearizability, other thread do not consider its changes as permanent. These three results ensure that the point of linearizability is the point at which a thread makes its changes permanent, and henceforth, the changes are visible to all the threads. Thus, we have a linearizable implementation.

Now, let us prove that every request executes correctly in the sense that it obeys sequential semantics.

Lemma 3. *Every request books only numSlots entries in consecutive columns. One slot per each column.*

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, for a request, r, exactly *numSlots* entries are allocated in consecutive columns with one entry per column.

First, we need to ensure that whenever a transaction aborts and restarts, it should start afresh. All the changes made by it in the SCHEDULEMAP array should be undone. The STM system ensures that the changes made by a transaction that commits are eventually reflected in the system. This ensures that even though there can be multiple transactions reserving the slots for a thread, eventually only one of them will commit. And, in each transaction, only one slot is reserved by a thread in one column. As a slot gets reserved in a column of the SCHEDULEMAP array, we increment the index to point to the next column. We continue doing this till *numSlots* slots are reserved.