Scalable and Performance-Critical Data Structures for Multicores

(extended abstract of the MSc dissertation)

Mudit Verma
Departamento de Engenharia Informática
Instituto Superior Técnico

Abstract. In this work, we study the scalability, performance, design and implementation of basic data structure abstractions, such as a queue, for next generation multicore systems. We propose two algorithms for concurrent queue. Our first algorithm, a wait-free queue, provides an efficient replacement to a lock-free queue. Lock-free queue is considered very efficient, but does not provide local progress guarantee for each thread. It also performs badly under stressed conditions. Our wait-free queue, not only provides local progress guarantee, but also depicts high performance and positive scalability under highly stressed conditions. Our second algorithm, a sequentially consistent queue, further achieves high performance by changing the consistency model. All the queue algroithms provide linearizability, which orders the operations on a global time scale. However, our sequentially consistent queue orders the operations in a program order, which is local to a thread. Our experimental results shows that our algorithms outperforms existing *state-of-the-art* algorithm by a factor of 10 to 15.

1 Introduction

Contrary to the rapid development in multicore hardware architectures, softwares are still not able take full advantage of these massively parallel working units. Much of the problem lies in the way applications are designed and the way they utilize basic data structures. Parallel and concurrent programming is naturally hard. Therefore, programmers tend to pursue conservative approaches in order to avoid race conditions, while working on shared memory systems on multicores [1]. In a multithreaded application, threads running on different cores access the globally shared data structures. Concurrent access to these data structure becomes a bottleneck if the access rate is very high, which consequently results in degraded application performance. This problem increases as the number of threads in an application and number of cores in a machine increases [2].

Linear scalability is the ideal desired outcome for any application running on multicore machine. However, our experimental results shows us that few of the sophisticated concurrent data structure algorithms are far from achieving ideal linear scalability. Infact, they do not even achieve positive scalability. Figure 1 presents the scalability of *state-of-the-art* Michael and Scott's lock-free queue algorithm which is also implemented in *java.util.concurrent* library. This algorithm shows negative scalability under high concurrent access rate. On 32 cores, this algorithm is only able to perform 200 operation/ms as compared to 6000 operation/ms on a single core machine where it needs to perform all the operations sequentially. Therefore, the problem is not with running the code sequentially, but with the inherent underlying communication overheads, which these data structures suffer from on a multicore machines.

Considering the scalability and performance issues with existing algorithms, we foresee a fundamental shift in data structure's design for next generation multicores. Therefore, we are interested in designing and implementing efficient algorithms that are more scalable and perform better under stressed conditions. Currently, all the practiced algorithms provide semantics such as linearizability and lock-freedom. However, we look into alternate directions. We explore wait-free algorithms which can perform better than lock-free algorithms. Wait-free algorithms have gained the attention only recently, and they are not widely covered in the literature [3].

Unlike lock-free algorithms, which provide progress guarantee for one worker, wait-free algorithms provide a progress guarantee for all the workers. We emphasize that, with cloud computing being the future, where compute is governed by service level agreements (SLAs) and the quality of service (QoS), wait-free algorithms will replace lock-free algorithms. Therefore, we work on the design of efficient wait-free algorithms.

We also explore the trade-off between the correctness and performance. High performance can be obtained at the expense of correctness. Therefore, we consider the relaxation of consistency model from linearizability to sequential

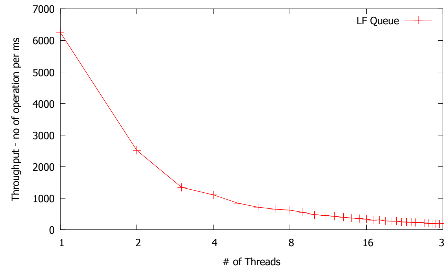

Fig. 1. Scalability of Michael and Scott's lock-free queue

consistency and apply it to basic data structures. We apply our techniques to concurrent queue and introduce two new efficient algorithms for a wait-free concurrent queue and a sequentially consistent concurrent queue. However, same techniques can also be applied to other concurrent data structures such as stacks, linked lists and skip lists.

Our primary aim is to target situations, where applications heavily access the concurrent objects and cause high contention. For other cases, existing algorithms provide fair results and is out of the scope of this work. This extended abstract summarizes my contributions related to scalability and performance of concurrent data structures with queue as a design example. Our main contributions are:

– We present a wait-free linearizable queue as a replacement to *state-of-the-art* Michael and Scott's lock-free linearizable queue, which is also implemented in JAVA concurrent library.
– We present a sequentially consistent queue. We achieve high performance by semantically relaxing the consistency model from linearizability to sequential consistency.

The remainder of this document is organized as follows. Section 2 covers the related work in the concurrent data structures on multicores, consistency models and *state-of-the-art* queue algorithms. Section 3 covers the algorithmic design and implementation of two algorithms, a wait-free queue and a sequentially consistent queue. Section 4 covers the performance and scalability evaluation of both the algorithms. Finally, section 5 concludes the extended abstract.

2 Related Work

In this section, we analyze the multicore architectures, consistency models and various algorithms and techniques for concurrent data structures in multicore machines. In 1st subsection, we analyze different types of multicore architectures and their associated problems with regard to the concurrent data structures. In 2nd subsection, we audit different types of consistency models in the literature. In 3rd subsection, we study various algorithms for concurrent queues.

2.1 Multicore Architecture

There are two main classes of multicore architectures. First and most common is *Symmetric Multiprocessing* (SMP) architecture. In this architecture two or more identical processing units are connected to a shared main memory. These types of machines can be found in mobile phones, laptops, desktops and servers. The second type of architecture is *Non-Uniform Memory Access* (NUMA) architecture. In NUMA architecture, many SMP machines (also called nodes) are connected using a network. This provides scaling over SMP machines.

Shared interconnect between the processors in SMP machines provides a homogeneous access to the resources. However, this design does not work well as the number of processors increases. The shared bus connecting the processors to memory becomes a bottleneck due to the limited bandwidth [4]. For example, a 4 processor SMP machines is only 2.7 times faster than a uniprocessor machine. This bottleneck at the hardware level does not allow applications to fully utilize the computational power of these machines.

Although NUMA architecture allows fast access to local memory, it becomes problematic for a multithreaded program running across the nodes that accesses a globally shared data structure. The problem becomes worse in case of cache coherent NUMA machine, where cache has to be kept consistent cross the nodes. If the threads running across the nodes, access the shared data structure too frequently, in becomes impossible for data to remain locally available. Therefore, it causes the remote memory access in almost every operation on the shared data structure. This creates a ping-pong type behavior which saturates the network. This increased contention results in severally reduced performance of an application [5].

2.2 Consistency Model

In a shared memory systems, Consistency model, provides a set rules for memory consistency. If these rules are followed by the programmers, system will provide a predictable outcome of memory operations. There are two main types of consistency models, linearizability and sequential consistency.

According to linearizability consistency model, for an observer, each operation applied by processes sharing a concurrent data structure (object), happens to take effect instantaneously at some point between the operation's invocation and the response. Therefore, all the operations can be given a pre-condition and a post-condition. Pre-condition defines the state of a concurrent object before the operation invocation event, whereas, post-condition defines the state of a concurrent object after the operation response event. Linearizability is considered one of the strongest level of consistency guarantee. It provides a global ordering of operations on a linear time scale.

Sequential consistency is a weaker consistency model as compared to linearizability. Unlike linearizability, which considers global time as the basis of ordering, sequential consistency considers program order as the basis of ordering. This model was introduced by Lamport as a consistency model for multiprocessor systems [6]. According to this model, for an observer, all the operations applied on a shared concurrent object, happens to come in effect according to program order.

Program order is defined by the order in which a process (thread in case of multithreaded program) applies the operations to a concurrent shared object. The preservation of program order in sequential consistency restricts an observer to view an inconsistent ordering of the operations issued by the same process. However, it does not restrict a global order to come in effect in which operations between the processes are reordered.

2.3 Concurrent Queues

In this subsection, we critically analyze various concurrent first-in first-out (FIFO) queue algorithms. First, we analyze the blocking queue algorithms which use locks for mutual exclusion, followed by the lock-free queue algorithms, which uses low level atomic primitive such as CAS (compare and swap). Finally, we analyze the wait-free queue algorithms.

A concurrent queue supports two fundamental operations, enqueue and dequeue. Enqueue operation appends the element to end (tail) of the queue, whereas a dequeue operation remove the element from the start (head) of the queue. Head holds the element which is present in the queue for the longest time, whereas, tail holds the element which is in queue for the shortest time.

2.3.1 Blocking queues:

Just like other blocking algorithms, blocking queue synchronizes the access to the queue. Queue has two access points, head and tail. In a conservative approach, a single lock can be used for the queue which synchronizes the enqueue and dequeue operations. This gives a safe implementation of a concurrent queue. However, Amdahl's law suggest that concurrent data structures, whose operations hold exclusive lock, fails to scale on a multicore machine [7].

2.3.2 Lock-free queues:

Lock-free algorithms presents an efficient alternative to blocking algorithms. Not only the lock-free algorithms provide better performance because of the lock freedom, they also provide a global progress guarantees. Most of the practical

lock-free algorithms are based on compare-and-swap (CAS) primitive.

One of the most efficient lock-free concurrent queue algorithm was presented by Michael and Scott [8]. In literature, it is considered as *state-of-the-art* lock-free algorithm [7,9,10].

A lock-free algorithm based on CAS works very efficiently as compared to lock based algorithms. It does not require any conservative locking while accessing the globally shared data structure, the queue. Also, it provides a global progress guarantee. However, the efficient functioning of queue depends a lot on the contention level and the multicore topology. Although it provides global progress guarantee, there can be cases where one of the threads will have to wait unfairly long to complete its operations as compared to other threads. This leads to starvation. One such case arises due to data locality.

2.3.3 Wait-Free queues:

Although, Lock-free concurrent queue algorithms provide global progress guarantee, they do not provide process (thread) level progress guarantee. To overcome this problem, wait-free queue algorithms were proposed. However, constructing a wait-free algorithm is not easy and they generally do not perform well as compared to lock-free queues.

One of the first wait-free algorithm was proposed by Lamport [11]. He presented a single producer (enqueuer) and single consumer (dequeuer) wait-free circular buffer. Subsequently, this circular buffer can also be used to implement a single producer and single consumer queue. Although, this implementation is wait-free, it limits the concurrency to only one consumer and one producer. Also, since the circular buffer is based on a static array, it limits the number of elements a queue can hold.

David proposed another wait-free algorithm that supports multiple concurrent dequeuers but only one enqueuer. This queue is based on an array which is infinitely large [12]. This requirement makes the algorithm impractical. There have been other proposals but none of them presented a multiple enqueuer and multiple dequeuer queue algorithm.

This first practical multiple enqueuer and multiple dequeuer concurrent queue algorithm is presented by Kogan and Petrank [3]. In this algorithm, faster threads try to help other slower threads in applying their operations. Each operation, enqueue or dequeue starts by choosing a phase number. This phase number is higher than all the phase numbers chosen previously by other threads. The thread performing the operation, records its operation information at a designated position in a state array. This algorithm has a very complex implementation. Though, it provides the wait-freedom, their results shows that it does not perform better than the Michael Scott's lock-free algorithm.

3 Scalable and Performance-Critical Queues

Having studied existing *wait-free* algorithms in literature, we have come to a conclusion that designing a simple and efficient *wait-free* algorithm is not easy. As covered in detail in related work sections, one of the most practical *wait-free* algorithm resort to use mutual helping during process execution. If few of the workers are not able to complete their operations, one of the workers which is fast enough, and is able to get hold of the object, helps other waiting workers in completing their operations. However, most of the *wait-free* algorithms are very complex in nature and do not perform better overall in throughput as compared to *lock-free* algorithms. Therefore, while designing this queue, our prime focus is to come up with a *wait-free* algorithm which is practical, simple and yet provides very efficient access to a concurrent queue.

3.1 Wait-Free Queue

Our queue is based on an underlying singly *linked list* which holds references to *head* and *tail* of the queue. Our idea extends the notion of mutual help in previous *wait-free* algorithms to an external help provided by a dedicated worker, whose job is to help other threads to complete their operations in a fair manner. Following are the two additions to our

wait-free queue algorithm.

- **External Helper**: Queue maintains an external helper for queue operations. The workers (enqueuers and dequeuers) do not directly interact with the underlying queue structure which is a *linked list*. Instead, they submit their requests to *Helper* which, in turn, completes their requests by enqueuing or dequeuing the elements to/from the queue.

- **State Array**: Queue internally maintains a *state array* that acts as a placeholder for incoming requests for *enqueues* and *dequeues*. The size of *state array* is equal to the number of workers in the program. Each position in the *state array* is dedicate to a worker. Whenever a worker needs to do an operation on the queue, it places the request at its designated position in the array.

The basic logic behind the algorithm is simple. Each worker puts its request for *enqueue* or *dequeue* in the *state array* and waits for its operation to be applied to the queue. *Helper* worker traverse through the *state array*, looking for new incoming requests. If a new request is found, it processes it on behalf of the requester. Workers wait until their operations are successfully picked up and applied by the *Helper*. Following is a brief description of how the enqueue and dequeue operations are performed:

- **Enqueue**: Algorithm for enqueue operation is given in Listing 1.1. Worker creates a *Request* with the element e and operation type as ENQUEUE. It then places this request in its designated position in the *state array* and waits for the operation to be picked up by the *Helper*. It continuously checks the status of the *Request* by checking *isCompleted* flag. If it is set true that means the operation has been completed by the *Helper*. Following that, it clears the request from *state array* and method returns as success. This operation never fails, assuming that the *Helper* will always complete the operation.

Listing 1.1. Algorithm for enqueue operation (Wait-Free Queue)

```
1 bool enqueue(e) {
2         id = getThreadId();
3         req = createRequest(e,ENQUEUE);
4         stateArr[id] = req;
5         while(!req.isCompleted);
6         stateArr[id] = null;
7         return true;
8 }
```

- **Dequeue**: Just like the enqueue operation, dequeue also creates a *Request* with empty element field and operation type as DEQUEUE. It places this request in its position in *state array*. The *Helper* threads picks this request and processes it. Once completed, the element field in *Request* structure contains the top element from the queue and *isCompleted* flag is marked as true. When the operation completes, the request is cleared from *state array* and the element is returned. Listing 1.2 presents the algorithm.

Listing 1.2. Algorithm for dequeue operation (Wait-Free Queue)

```
1  E dequeue() {
2          id = getThreadId();
3          req = createRequest(null,DEQUEUE);
4          stateArr[id] = req;
5          while(!req.isCompleted);
6          stateArr[id] = null;
7          return req.e;
8  }
```

The algorithm for *Helper* is given by Listing 1.3. It traverse through the *state array* in a round robin fashion in a tight loop. If it sees a *Request* from a worker it fulfills it before moving to next index in *state array*. It first reads the operation field in *Request*. If it is an enqueue operation, it creates a new node with the element in request field. It then appends the node to the *tail* and updates the *tail* of the *linked list*. In the end it marks the operation as completed.

If it is a dequeue operation, it checks if the *linked list* is empty. If it is empty, it returns null and marks the operation as completed. If the list is not empty, it removes the top element and updates the references to *head* and *tail* (if required) respectively. It changes the element field in *Request* from null to the removed element and marks the flag *isCompleted* as success.

Listing 1.3. Algorithm of Helper (Wait-Free Queue)

```
1  void helper() {
2          id = 0;
3          while(true) {
4            req = stateArr[id];
5            if(req!=null && !req.isCompleted) {
6                  if(req.operation == ENQUEUE) {
7                     n = new_node(req.e);
8                     tail.next =n;
9                     tail = n;
10                    size++;
11                    req.isCompleted = true;
12                 }        // enqueue completes
13                 // Dequeue or Poll
14                 if(req.operation == DEQUEUE) {
15                    if(head.next == null) {
16                      req.e = null;
17                      req.isCompleted = true;
18                    }
19                    else {
20                         n = head.next;
21                         head.next = n.next;
22                         if(n.next==null) {
23                                 tail = head;
24                         }
25                         req.e = n.e;
26                         size--;
27                         req.isCompleted = true;
28                   }
29                 } // dequeue completes
30           }
31           //increment index id++%worker
32           id = incrementId();
33         }
34 }
```

3.2 Sequentially Consistent Queue

It is practically impossible to provide a very fast FIFO Queue implementation that works on large multicore systems under stressed conditions. However, performance can be achieved by weakening the consistency. In our design, we explore the possibility of changing the consistency guarantees from linearizability to sequential consistency.

This algorithm is an optimized enhancement to our previous *wait-free* algorithm. The basic structural details of this algorithm remains similar to our previous design. Additionally, this queue maintains an internal *local linked list* structure for each worker. On enqueue, an element is appended to the local *linked list*. This local list is exclusive to the worker. Therefore, there is no possibility of contention and race among workers. This allows very fast enqueue operation. Queue also maintains a *global linked list* structure, where workers perform dequeue operations. However, this global structure is not directly accessed by the workers. Instead, they submit their request to *Helper* with the help of *state array* and wait until the operation is completed.

The *Helper* continuously checks for dequeue requests in *state array*, and if found it processes them. The *Helper* also, in the background, periodically merges the *local lists* with the *global list*. The *Helper* freezes (locks) the local linked list for a worker when it merges it with *global linked list*. This is done to avoid updating incorrect or stale references by enqueue operation. Listing 1.4 and 1.5 illustrates how enqueue and merge operations work for this *sequentially consistent* queue. Dequeue operation has same algorithmic steps as shown in wait-free algorithm.

Listing 1.4. Algorithm for enqueue operation (Sequentially Consistent Queue)

```
1  bool enqueue(E e) {
2          id = getThreadId();
3          while(!lock[id].CAS(false, true));
4          tail = localTails[id];
5          if(tail == null)
6                  return false;
7          tail.next = new_node(e);
8          localTails[id] = tail.next;
9          lock[id].set(false);
10         return true;
11 }
```

Listing 1.5. Algorithm for merge operation (Sequentially Consistent Queue)

```
1  void merge(id) {
2          localHead = localHeads[id];
3          localTail  = localTails[id];
4          if(localHead!=localTail){
5                  while(!lock[id].CAS(false, true))
6                  tail.next = localHead.next;
7                  tail = localTail;
8                  localTails[id]=localHeads[id];
9                  lock[id].set(false);
10         }
11 }
```

3.3 Correctness

We claim that our first algorithm implement a wait-free queue and our 2nd algorithm implements a sequentially consistent queue. Following is a brief discussion on the same.

– **Wait-Freedom**: Wait-freedom comes from the fact that an operation completes in a bounded number of steps. If we consider the number of workers predetermined and known (whose id ranges from 0 to n-1), we can instantiate the size of *state array* to a size proportional to the number of workers. In our first algorithm, any enqueue or dequeue operation will complete in maximum n steps, given there are n workers. This is so, because *Helper* iterates

over the state array. Let?s consider the worst case, where a worker T*i* places a request in *state array* when the pointer of *Helper* is at location *i+1*. Now, the helper needs to complete the whole round by jumping *n* places before it comes back to location *i*. Therefore, the upper bound for enqueue and dequeue operation is *n*, which is fixed.

– **Sequential Consistency**: Sequential consistency guarantees the program order. The order in which a worker (thread) issues operations is called program order. This order is unrelated to how and when operations are issued by different threads. We claim that our algorithm is sequentially consistent but not linearizable. This is so because *enqueue()* call returns as soon as the element is appended in the local linked list. *Helper* does not guarantee which local list will be merged first, therefore, it is possible that an element, enqueued at a later time by different worker, may be dequeued first by some other worker. This violates the linearizability property. Nonetheless, our program still follows the program order as two elements enqueued by the same worker will be dequeued in same order irrespective of who performed the dequeue operation. Our algorithm preserves this property because the local ordering is maintained when a worker issues enequeue operations in *local linked list*.

4 Evaluation

In this section, we evaluate the performance and scalability of both of our algorithms, the wait-free linearizable queue (WF Queue) and the sequentially consistent queue (SC Queue). We compare the performance and scalability of these algorithms with *state-of-the-art* lock-free queue algorithm presented by Michael and Scott.

We implemented both the algorithms in java in a controlled configurable environment. We have also implemented a benchmark framework for the experiments. All the experiments are run on a 48 core NUMA machine with x86 64 bit architecture running Linux with kernel 3.0.0. The machine has 4 sockets and 8 nodes where each node has 6 cores. All the micro benchmarks are taken in a careful manner in order to minimize the noise. Our evaluation framework emulates a multi worker (producer-consumer) workload where each worker is allowed to perform both enqueue and dequeue operations. Following are the benchmarks used.

– **Enqueue-Dequeue pair**: The queue is initially empty. At each iteration, each worker enqueues a random integer in the queue followed by a dequeue operation. This emulates the 50-50 percent producer-consumer workload under the stressed conditions. Each experiment performs approximately 2.5 million operations divided equally between all the workers. The number of maximum workers is 32 which is unchanged in all the experiments. Helper runs on a separate core.

– **Think-Time**: Think-Time is nothing but the work done by the worker between the two operations. We measure the performance of our algorithms as a function of increasing Think-Time. Ideally, with the increasing Think-Time contention in the system should decrease. Think-Time is presented in number of cycles. Each experiment performs approximately 2.5 million operations divided equally between all the workers. The number of maximum workers are 32.

4.1 Performance and Scalability

Following subsections present the scalability and performance experiments and its results.

4.1.1 Per operation Completion time:

In this experiment we have measured average per operation completion time for each of the three algorithms, lock-free (LF) Queue, wait-free (WF) Queue and sequentially consistent (SC) Queue under very high contention. Results are shown in Figure 2. Under such stressed conditions, is it important to known how long does each operation takes to complete. As we can see from the graph, it is evident that our algorithms takes far less time as compared to LF queue. With the increasing number of workers, per operation time increases for all the Queue implementations. For LF queue,

by doubling the number of workers, per operation time increases by approximately 4 times, however, for WF queue and SC queue it increases approximately twice which is understandable.

With 32 workers, one single operation in LF queue takes as much as 0.1 ms. This converts to 300000 cycles of work, which LF queue does in a single enqueue or a dequeue operation. In comparison to that, our WF queue takes 9 times lesser time and our SC queue takes 16 times lesser time. SC queue perform better than WF queue because of weaker consistency semantics. If we increase the number of workers, this performance gap between LF queue and WF queue increases.

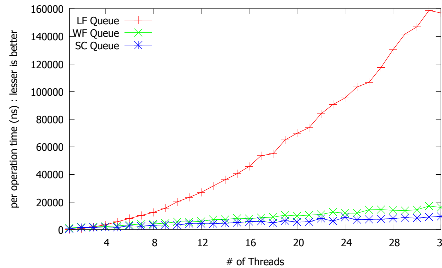

Fig. 2. A per operation completion time comparison with the increasing number of threads (workers)

4.2 Total Execution Time

The performance of an application depends on how fast it completes a given task irrespective of how workers perform it individually. In this experiment, we evaluate the time taken to complete the whole execution of the program. Results are shown in Figure 3. We can see from the graph that upto 2 threads (workers), WF Queue and SC Queue take more time to complete the task. However, for LF queue, as the number of workers increase, the total execution time increases drastically. This depicts the worst kind of negative scalability and goes against the very purpose of parallel programming. On the contrary, WF depicts some positive scalability, where total execution time either remains stable or decreases. SC Queue performs best, where its total execution is least among all. Also, similar to WF Queue, it depicts positive scalability. With 32 workers, WF queue performs approximately 10 times faster and SC Queue performs approximately 15 times faster.

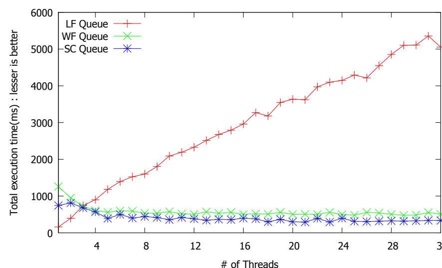

Fig. 3. A total execution time comparison with the increasing number of threads (workers)

4.3 Throughput

Scalability and performance of a queue also depends on how many operations it can successfully perform in a unit time. In this experiment, we show the throughput as a function of increasing number of workers. Results are presented in

Figure 4. In this case also, LF queue depicts negative scalability and the throughput decreases with increasing number of workers. WF Queue and SC queue shows sharp positive scalability upto 6 workers, after which throughput remains stable with the increasing number of workers. However, in comparison with the LF Queue, throughput remains approximately 10 times more for WF queue and 15 times more for SL queue. This higher throughput directly translates to better performance for an application using the concurrent queue.

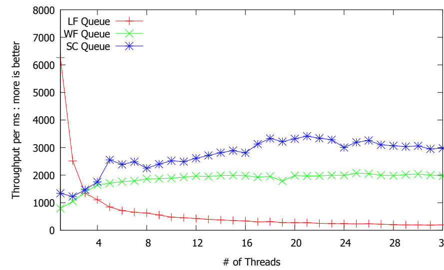

Fig. 4. Throughput comparison with the increasing number of threads (workers)

4.4 Think Time

In real scenarios, applications do some work between two operations. For some applications, this work can be computationally intensive, for others it can be computationally low. We call this work as think time. In this experiment we compare the performance of the Queues with increasing think time. In our experiment, it ranges from 0 to 1 million cycles. The number of workers are fixed to 32. Helper runs on a separate core. Results are shown Figure 5. Up to 1000 cycles of think time, per operation time remains stable for all three queues. Also, in comparison with LF queue, WF queue and SL queue take 10 to 15 times lesser time to complete an operation. After 10,000 cycles of think time, contention starts to reduce and performance improves for all three algorithms. Nonetheless, our algorithms still perform significantly better in comparison of LF queue.

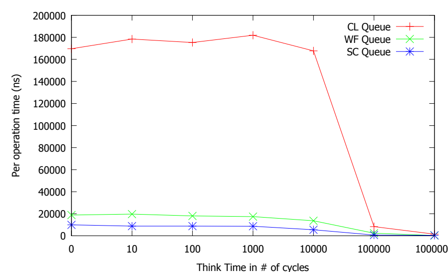

Fig. 5. Performance comparison with increasing Think Time (work done between two operations)

4.5 Analysis

High performance and wait freedom both are necessary properties required for next generation softwares running on multicore machines. Wait-free algorithms are gaining momentum and are being considered a replacement for lock-free algorithms. Yet, we did not find any wait-free algorithm in literature for queues, which can perform 8 to 10 times better

than lock-free algorithms. We have achieved this performance gap with just 32 workers. If, we increase the number of workers, this gap will further increase. We believe that this is the first time an algorithm is able to provide wait-freedom and high performance at the same time. Similarly, we also did not find any sequentially consistent version of a queue. It is a known fact that high performance can be achieved by compromising strict notions of correctness. Yet there was no implementation, which exploit the trade-off between the performance and correctness in terms of the consistency model.

5 Conclusions

This extended abstract summarized our work on the scalability and performance of concurrent data structures on multi-core machines. We showed that it is possible to implement a wait-free algorithm that can perform better than a lock-free algorithm. We also showed that it is possible to exploit the trade-off between the performance and the notion of correctness. We presented our algorithms for concurrent queue which is one of the widely used basic data structure. Our wait-free queue outperforms existing *state-of-the-art* Michael and Scott's lock-free queue while providing stricter local progress guarantee for each worker. We further achieved high performance by relaxing the FIFO ordering of a queue. Both of our algorithms also depict positive or stable scalability, which was non existent in previous algorithms.

In our algorithmic design, we introduced the concept of external *Helper*. This concept can be applied to other concurrent data structures such as stack, skip-lists and linked-lists. For example, a stack is very similar to a queue. The only difference is the order in which elements are inserted and removed. Queue provides FIFO ordering while stack provides LIFO ordering. We believe that our techniques and algorithms will become highly beneficial in future, as the number of cores continues to grow on multicore machines.

Acknowledgments

This work was supported by the *International Internship program 2013* at INRIA Paris. This work have been performed under the guidance of Dr. Marc Shapiro from INRIA Paris and Prof. Luis Veiga from INESC-ID Lisboa. I would also like to thank Gael Thomas and Lokesh Girda from LIP6 laboratory who participated in discussions regularly and provided their invaluable feedback.

References

1. M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.
2. C. M. Kirsch and H. Payer, "Incorrect systems: it's not the problem, it's the solution," in *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012, pp. 913–917.
3. 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*. ACM, 2011, pp. 223–234.
4. R. C. Unrau, "Scalable memory management through hierarchical symmetric multiprocessing," Ph.D. dissertation, Citeseer, 1993.
5. W. Bolosky, R. Fitzgerald, and M. Scott, "Simple but effective techniques for numa memory management," in *ACM SIGOPS Operating Systems Review*, vol. 23, no. 5. ACM, 1989, pp. 19–31.
6. L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *Computers, IEEE Transactions on*, vol. 100, no. 9, pp. 690–691, 1979.
7. M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann, 2012.
8. M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. ACM, 1996, pp. 267–275.
9. P. Tsigas and Y. Zhang, "A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems," in *Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*. ACM, 2001, pp. 134–143.
10. E. Ladan-Mozes and N. Shavit, *An optimistic approach to lock-free fifo queues*. Springer, 2004.
11. M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati, "An efficient unbounded lock-free queue for multi-core systems," in *Euro-Par 2012 Parallel Processing*. Springer, 2012, pp. 662–673.
12. M. David, "A single-enqueuer wait-free queue implementation," in *Distributed Computing*. Springer, 2004, pp. 132–143.