An optimistic approach to lock-free FIFO queues

Abhishek Kumar

CSE, IIT Delhi, India

1. Introduction

The limitations of increasing clock speed beyond a threshold accelerated the growth of shared memory multiprocessors architectures. Multi core processors require multiple threads to communicate and synchronize through data structures in shared memory. Designing such scalable data structures is a challenging task. Concurrent data structures that use locks suffer from a sequential bottleneck in high contention. Concurrent data structures that do not use locks (Non-blocking) provide great alternatives and work relatively well in high contention, but there is a price to pay, as they use higher-order synchronization primitives such as compare-and-swap instructions (CAS). The hardware implementation of these instructions prevents any instruction-level parallelism, even when there is no dependency between issued operations.Moreover the use of CAS also introduces performance overhead due to its failed operations. It is sensible to create algorithms that use fewer numbers of such atomic instructions while building non-blocking data structures. The titled paper works along the same direction, where it reduces the use of CAS in building lock-free queues. It improves the existing concurrent MS queue that uses two instances of CAS for enqueuing to use only one CAS, optimistically

2. Implementaion

This project implements the title paper and demonstrates the new queue performance with existing concurrent queues. Although the paper primarily deals with optimistic queue algorithm, It uses other data structures for comparison so I have implemented all the necessary data structures required for comparing and evaluating the benchmark.

2.1 Two lock queue:

The Two-Lock Concurrent Queue algorithm uses two separate locks to manage access to the head and tail of the queue, minimizing contention in multi-threaded environments. The queue head and tail are initialised with dummy nodes. During the enqueue process, a new node is created and then tail lock is acquired and the node inserted to the end of the queue. For dequeuing, the head lock is secured to exclusively access and update the head of the queue. If the queue is not empty, the next node's value is retrieved, the head is updated, and the old head node is released after the head lock is freed. As compared with single lock queue, This method allows concurrent enqueue and dequeue operations, enhancing performance by reducing the waiting time associated with lock acquisition.

Algorithm 1 Two-Lock Concurrent Queue

```
Class Node
    data: value, next* (pointer to Node object)
    constructor: Node(T value) initializes value and next

Class TwoLockQueue
    Private:
        Node* head, tail
        mutex head_lock, tail_lock

    Public:
        TwoLockQueue() initializes head and tail with dummy node

function ENQUEUE(T value)
    Node* node = new Node(value)
    tail_lock.lock()
    tail.next = node
    tail = node
    tail_lock.unlock()
end function

function DEQUEUE
    head_lock.lock()
    Node* front = head
    Node* next = front.next
    if next == nullptr then
        head_lock.unlock()
        return false
    else
        T value = next.value
        head = next
        head_lock.unlock()
        return true
    end if
end function
```

2.2 Michael-Scott Queue

The Michael-Scott (MS) Lock-Free Queue algorithm enhances concurrency through the use of atomic operations, avoiding the need for separate locks on the head and tail.

Initially, both the head and the tail of the queue point to a dummy node. In the enqueue process, the tail is accessed,and new node is appended atomically only if the tail's next pointer is null, and the tail pointer is then moved to the new node atomically. For dequeuing, if the queue's head is the same as the tail, an attempt is made to shift the tail forward before removing the head node. This is done basically to help enqueuers as general recurring pattern of help in lock free data structures.If queue head is not tail, then the dequeuing thread reads the head ,then read the next node ahead of head, then reads the value of that node next to head, and after these three steps, the thread does CAS to check if the head is still the same as it read last time and then advanced the head to next node and now the next node becomes the new head.This method eliminates the need for explicit locks, reducing potential bottlenecks and allowing for high throughput in multi-threaded scenarios. However as shown below in picture the enqueuing requires two CAS instructions while dequeing uses single CAS.

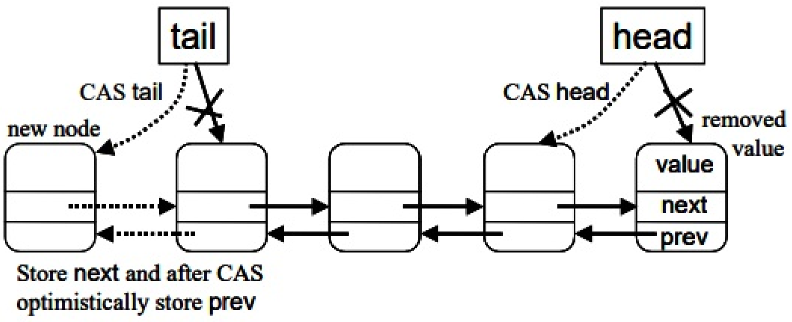

Figure 1: The single CAS dequeue and costly two CAS enqueue of the MS-Queue algorithm.

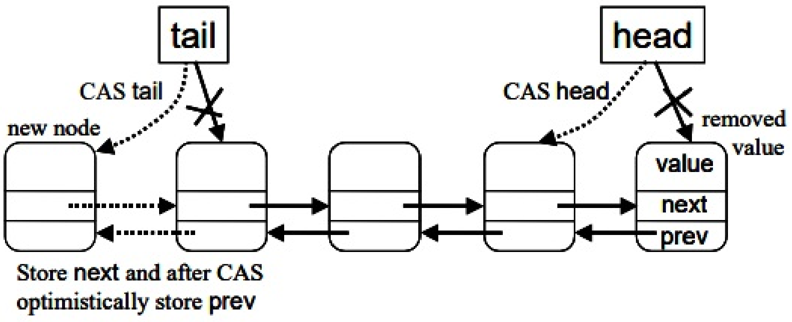

Figure 2: The single CAS dequeue and costly two CAS enqueue of the MS-Queue algorithm.

Algorithm 2 Michael-Scott Lock-Free Queue Implementation (Part 1)

```
Class Node
    data: value: T, next: std::atomic<Node*>
    constructor: Node(T value) initializes value and next to nullptr

Class MSqueue
    Private:
        std::atomic<Node*> head, tail

    Public:
        MSqueue() initializes with a dummy node
            Node* dummy = new Node(T())
            head.store(dummy)
            tail.store(dummy)


function ENQUEUE(T value)
    Node* node = new Node(value)
    Node* last
    Node* next
    while true do
        last = tail.load()
        next = last->next.load()
        if last == tail.load() then
            if next == nullptr then
                if last->next.compare_exchange_weak(next, node) then
                    tail.compare_exchange_weak(last, node)
                    break
                end if
            else
                tail.compare_exchange_weak(last, next)
            end if
        end if
    end while
end function
```

Algorithm 2 Michael-Scott Lock-Free Queue Implementation (Part 2)

```
function DEQUEUE
    while true do
        Node* front = head.load()
        Node* last = tail.load()
        Node* next = front->next.load()
        if front == head.load() then
            if front == last then
                if next == nullptr then
                    return false
                else
                    tail.compare_exchange_weak(last, next)
                end if
            else
                T value = next->value
                if head.compare_exchange_weak(front, next) then
                    return value
                end if
            end if
        end if
    end while
end function
```

2.3 Optimistic Lockfree Queue

The new algorithm proposes a novel approach by logically reversing the enqueue and dequeue operations in a queue, utilizing a doubly linked list for traversing from tail to head when needed. In this setup, enqueueres add elements at tail by simple store operation to set the new node's next pointer to the current tail, followed by a single CAS operation to update the tail pointer to its node. enqueuing threads further store the prev pointer of the node next to tail, to its current node. Here slow threads could create problem by not linking prev pointer on time. The dequeue method is similar to MS queue, except it calls fixlist when the head prev pointer is inconsistent. In this case This causes problem, as dequeuing thread may not be able to access the first node, after the head(dummy node). For this a method fixlist is called , that traverses from tail to head (this is where doubly linked list feature gets used ,and links all the unlinked node prev pointers to its previous nodes. Since calls to fixlist method rarely happens, the new algorithm proves effective overall, compared to MS queue.

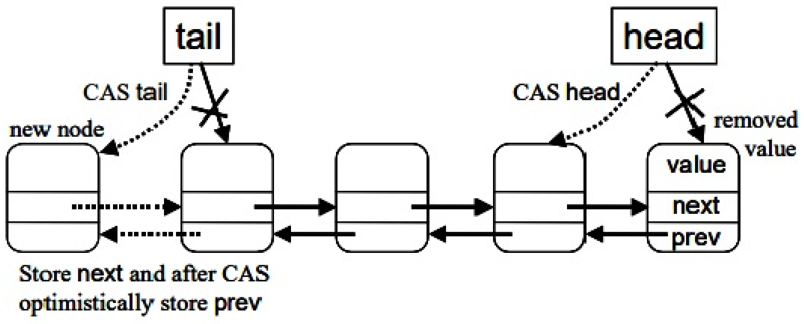

Figure 3: The single CAS dequeue and single CAS enqueue of the Optimistic-Queue algorithm.

Algorithm 3 OptimisticQueue Implementation

```
Class Node
    data: value: T, next*: Node*, prev*: Node*
    constructor: Node(T value) initializes value, next, and prev to nullptr

Class OptimisticQueue
    Private:
        std::atomic<Node*> head, tail

    Public:
        OptimisticQueue() initializes head and tail with a dummy node
            Node* dummy = new Node(T())
            head.store(dummy)
            tail.store(dummy)


function ENQUEUE(T value)
            Node* node = new Node(value)
            Node* last
    while true do
                last = tail.load()
                node->next = last
        if tail.compare_exchange_weak(last, node) then
                    last->prev = node
                    break
        end if
    end while
end function
```

Algorithm 3 OptimisticQueue Implementation (continued)

```
17: function DEQUEUE
18:     while true do
19:         Node* front = head.load()
20:         Node* last = tail.load()
21:         Node* second = front->prev
22:         if front == head.load() then
23:             if front != last then
24:                 if second == nullptr then
25:                     fixlist(front, last)
26:                     continue
27:                 else
28:                     T value = second->value
29:                     if head.compare_exchange_weak(front, second) then
30:                         return value
31:                     end if
32:                 end if
33:             else
34:                 return 0                                    ▷ Queue is empty
35:             end if
36:         end if
37:     end while
38: end function
39:
40: function FIXLIST(Node* front, Node* last)
41:     Node* curr = last
42:     Node* succ
43:     while front == head.load() and curr != front do
44:         succ = curr->next
45:         succ->prev = curr
46:         curr = succ
47:     end while
48: end function
```

3. ABA Problem and memory reclamation:

All the algorithms above, I discussed, I abstracted concept of ABA just as to explain the algorithm at hand in simple manner. In all queues above dequeue method basically involves deleting the node.After the node has been deleted, that memory address might be used for some other purpose or reused in queue operation again. While the above code will work perfectly fine in garbage collector supported languages such as JAVA because in JAVA, there is no free operation and all the memory management is handled by the garbage collector tracing the memory addresses. as long as any thread have references for the addresses, it will not be sent back to free space for reuse. In C++ however ,based on memory reuse there will be some problem that can arise. 1. segmentation fault: there may be case that some thread still have reference to that deleted head, while that node is deleted and when they try to deference it, they might find some different data type at place of address space or any garbage value, resulting in segmentation fault. 2. ABA scenario: In case of addressed reused in the queue, we might face this.

An ABA scenario may arise when a process reads a segment of the shared memory in a specific state and is then temporarily suspended. Upon resuming, the segment initially read might appear unchanged, even though numerous additions and removals might have occurred meanwhile. This can lead the process to erroneously complete a CAS operation, potentially leading to inconsistencies within the data structure. To detect and address such issues, we implement a conventional tagging mechanism. In this mechanism, instead of representing a pointer as an address, we represent it as a pair of serial number and address <count, addr> , and serial number change every-time we update the pointer. Since this requires comparing 2 parameters, we need double word wide CAS and not all architecture have double word CAS available, hence I have utilised the bit packing technique.

Algorithm 4 Tagged Pointer Template Structure

```
template<typename T>
struct TaggedPtr
    uintptr_t ptr                                        ▷ full 64-bit pointer
    static constexpr uintptr_t PTR_MASK = 0x0000FFFFFFFFFFFF
    static constexpr uintptr_t TAG_MASK = 0xFFFF000000000000

    Constructor: TaggedPtr() : ptr(0) {}
    Constructor: TaggedPtr(T* p, uint16_t tag = 0)
        begin
            set(p, tag)
        end

    Function T* getPtr() const
        return reinterpret_cast<T*>(ptr & PTR_MASK)

    Function uint16_t getTag() const
        return static_cast<uint16_t>((ptr & TAG_MASK) » 48)

    Procedure set(T* p, uint16_t tag)
        begin
            uintptr_t p_int = reinterpret_cast<uintptr_t>(p)
            ptr = (p_int & PTR_MASK) | (static_cast<uintptr_t>(tag) « 48)
        end
```

Algorithm 5 OptimisticQueueABA Implementation - Part 1

```
template<typename T>
bool cas(TaggedPtr<T>* addr, TaggedPtr<T> oldVal, TaggedPtr<T> newVal)
    uintptr_t expected = oldVal.ptr
    uintptr_t desired = newVal.ptr
    return __sync_bool_compare_and_swap(&addr->ptr, expected, desired)

template<typename T>
class OptimisticQueueABA
    private:
        struct Node
        T value
        TaggedPtr<Node> next
        TaggedPtr<Node> prev
        Node(T value) : value(value), next(nullptr), prev(nullptr) {}

    public:
        TaggedPtr<Node> Head
        TaggedPtr<Node> Tail
        OptimisticQueueABA()
            Node* dummy = new Node(0)
            TaggedPtr<Node> tp(dummy)
            Head = (tp)
            Tail = (tp)


function ENQUEUE(T value)
            Node* node = new Node(value)
            TaggedPtr<Node> tail
    while true do
                tail = Tail
                node->next = TaggedPtr<Node>(tail.getPtr(), tail.getTag() + 1)
        if cas(&Tail, tail, TaggedPtr<Node>(node, tail.getTag() + 1)) then
                    tail.getPtr()->prev = TaggedPtr<Node>(node, tail.getTag() + 1)
                    break
        end if
    end while
```

Algorithm 6 OptimisticQueueABA Implementation - Part 2

```
function DEQUEUE
    while true do
        Node* head = Head.load()
        Node* tail = Tail.load()
        Node* firstNodeprev = head->prev
        if head == Head.load() then
            if tail != head then
                if firstNodeprev.getTag() != head.getTag() then
                    fixlist(head, tail)
                    continue
                else
                    T value = firstNodeprev->value
                    if cas(&Head, head, TaggedPtr<Node>(firstNodeprev, head.getTag() + 1)) then
                        return value
                    end if
                end if
            else
                return T()                                        ▷ Queue is empty
            end if
        end if
    end while
end function

function FIXLIST(Node* head, Node* tail)
    while head == Head.load() and tail != head do
        Node* curr = tail
        Node* succ = curr->next
        while curr != head do
            succ->prev = TaggedPtr<Node>(curr, curr.getTag() - 1)
            curr = succ
        end while

```

In Intel x86 architecture, size of virtual memory is 48bit hence 16bit is available for use. I have utilised that upper 16 bit for tag. Now one can argue that problem of roll over might happen, however chances are extrememly rare. Assuming one operation takes roughly 1 microseconds, for the tag to rollover to reach same value, a thread need to suspended for (2**16-1) microsecond , that would be around half an hour. For a any good operating systems to hold a thread that long is impractical. This will prevent any threads seeing the same value twice with tag rollover phenomenon.

In my paper for safe memory reclamation, concept of thread pool was coined. Although paper did not have any code/implementation , I thought of using Treiber stack after reading on internet as lockfree linkedlist stack for memory pool for each thread. It will be easy and effective solution at the same time, it would give further performance boost due to preserving cache locality for each core. However while trying to make it run, It did not work and I was continuously getting segmentation fault. I have added the stack code which I implemented but did not use. In addition to that I also write CAS bit packing in assembly directive however I continued with my earlier approach. both was working and I have added code for that too in my folder.

4. Performance analysis:

I have evaluated all the results on my system Intel x86 64bit architecture with 4 core @1.60 GHz supporting 2 threads per core.

4.1 Experiment results:

As mentioned in the paper, I created two different benchmarks to evaluate the performance.

Benchmark1: "enqdeqpair": A group of n threads does total 1 million enqueue and dequeue operations, each thread enqueues and dequeues for 10**6 /k times enqueue and dequeue operation alternatively .

Benchmark2: "50percenteq": A group of n threads does total 1 million enqueue and dequeue operations, however each thread randomly chooses to enqueue or dequeue hence roughly 50 percent of the operations are enqueue and this adds randomness to the process, which is later on shown produces different outcomes in some cases.

In both the benchmark I have given small amount of delay to simulate the local works done by threads in the process. This I have done with incrementing a variable for certain amount of time in the loop. loop counter is controlled with limit OTHERWORK variable in utill.py file.

Observation1 : From the given data and graphs below, we can see Optimistic queue performs much better as opposed to Twolockqueue and MS queue. While at low contention when the no of threads are equal to no of cores (4), the performance is somewhat similar , the minor differences are mostly coming from overhead in code complexity of all queues enqueue and dequeue operations, However as contention increases, it becomes prominent how lock based queue suffers from sequential bottleneck. The gap between MS queue and new optimistic queue also widens due to higher overhead, as a result of increased failed CAS calls.

Table 1: Performance Metrics of Different Queuing Algorithms across Multiple Threads

No of threads	Twolockqueue	MSQueue	Optimisticqueue
1	47.3056	70.8774	69.5924
2	125.915	104.061	83.7207
3	115.49	116.906	96.8181
4	126.522	171.312	151
5	135.934	154.996	122.217
6	169.494	148.351	122.973
7	166.557	155.812	133.196
8	164.75	160.399	126.042
9	171.654	148.645	111.851
10	173.303	147.065	115.987
11	171.812	136.019	109.901
12	181.079	146.399	115.191
13	174.352	145.604	106.218
14	172.914	149.306	120.9
15	172.373	149.073	113.987
16	179.931	146.63	114.568

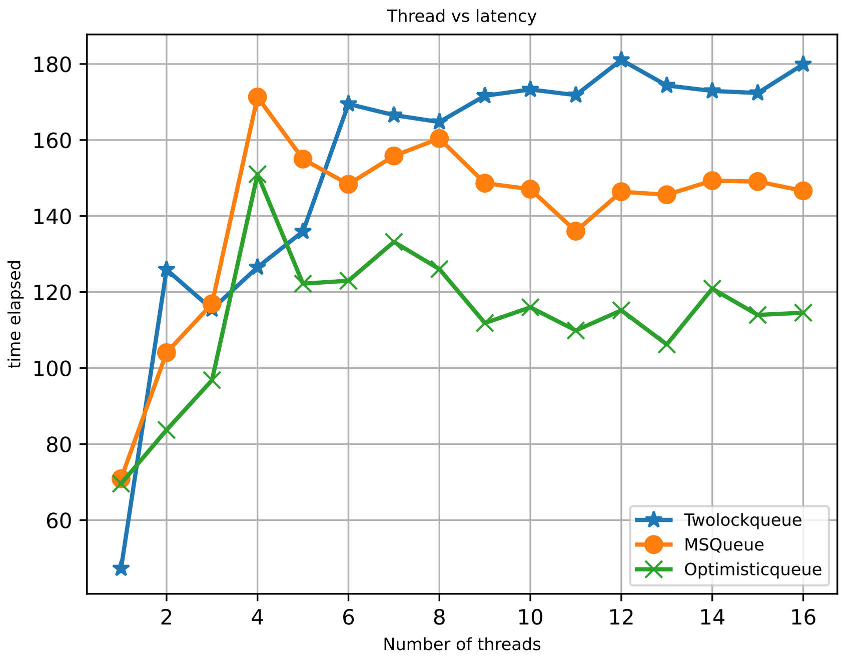

Figure 4: Relative performance for 1M enqueue/dequeue on concurrent FIFO queues for benchmark1

Observation2: As I mentioned earlier, I have created two benchmarks for evaluating the performances and as we can see the results are similar in both the figure 3 and figure 4, however the only difference we could see when number of threads are one or two. it is because as we discussed in new algorithm, additional 1 nos. CAS is required when the queue is empty, this introduces additional overhead in case when queue is empty. in benchmark1, as the number of threads increases, their scheduling time will also increase hence the chances of queue getting empty will get less frequent so that overhead will not reflect, on the other hand, benchmark2 does operation in random way so the overhead of queue getting empty will be eliminated at even at lower level of concurrency.

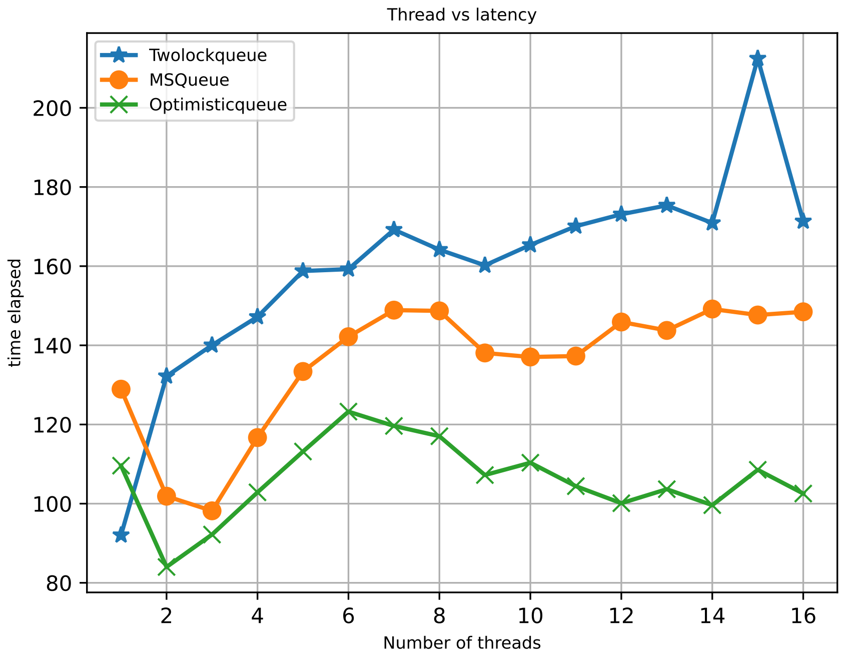

Figure 5: Relative performance for 1M enqueue/dequeue on concurrent FIFO queues for benchmark2

Observation3: Choice of atomic instructions as well as its extent of use does play a fundamental role in non blocking algorithms. The difference however does not necessarily happen due to difference in latency or bandwidth. Almost all atomics have comparable latency and bandwidth. The difference happens because of atomics instruction nature in operation. for example FetchandAdd(FAA) will always succeed however, CompareandSwap(CAS) may succeed or fail. The additional overhead created due to CAS failed operation heavily affects the performance. The below plot demonstrates the reason behind better performance of optimistic algorithm.

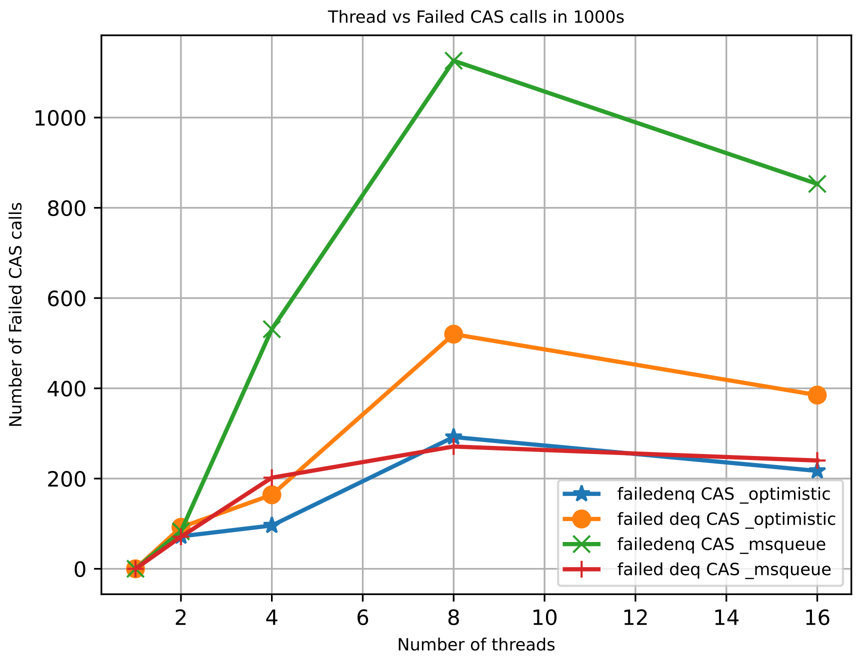

Figure 6: Relative performance for 1M enqueue/dequeue on concurrent FIFO queues for benchmark2

Observation4: One very interesting pattern we see is that, as we increase the amount of local work by tuning OTHERWORK variable, it imposes different schedulling pattern on operations.Increasing work between operations causes more contention and hence more failed CAS operation and that as a result shows higher time taken , as opposed to case where "otherwork" is low. In comparison to Figure 3 the average time increases from around 100ms to 250ms in Figure5

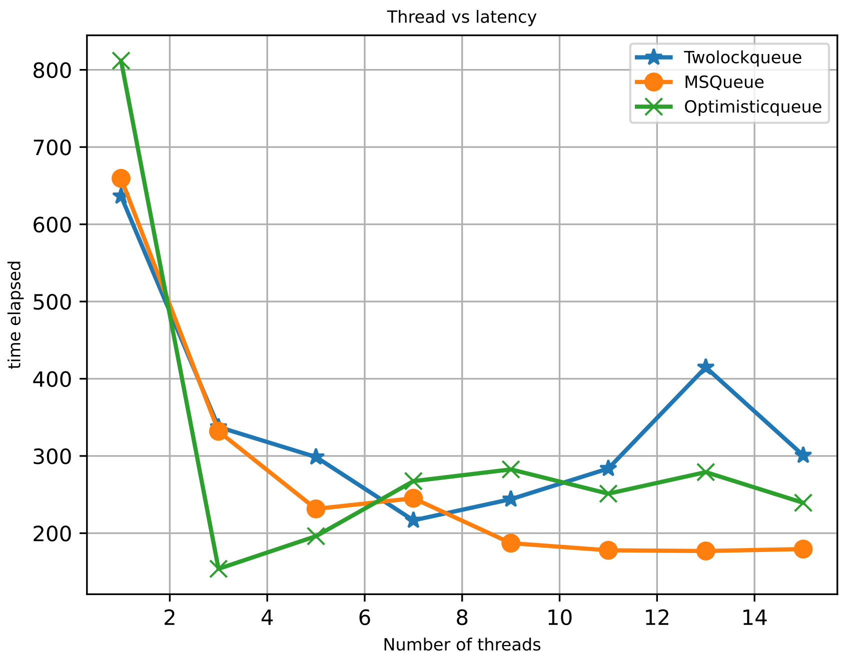

Figure 7: Relative performance for 1M enqueue/dequeue on concurrent FIFO queues for benchmark1 with OTHERWORK increased 10X from figure3

Observation5: As mentioned in the new algorithm that it uses to fixlist() calls optimistically restore the prev pointers of each object to its previous node, when the thread doing enqueue in past fails to set the prev of the node during enqueue operation. Since this is done only when the normal dequeu fails, We would want the calls to fixlist() as minimum as possible for our algorithm to work effectively. plot given below demonstrates that the total number of the function calls remain below 300 calls for rougly half a million operations for benchmark1 of enqdeqpair. for benchmark2 of "50percentenq" there are neglible calls to fixlist() due to its random nature of queue methods.

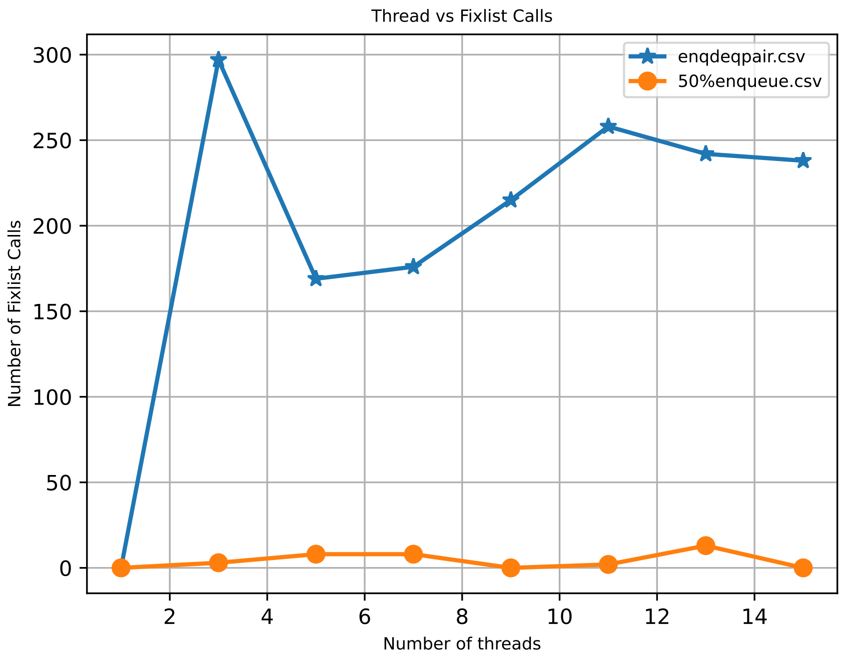

Figure 8: No of fixlist method calls for enqdeqpair and 50percentenq benchmarks