JKSCI

한국컴퓨터정보학회논문지
Journal of The Korea Society of Computer and Information
Vol. 30 No. 12, pp. 1-14, December 2025
https://doi.org/10.9708/jksci.2025.30.12.001

Time-Stamp with Window Decoupled: A Fast and Practical Lock-Free Relaxed Queue

Yong-Joon Cho*, Nai-Hoon Jung**

*Student, Dept. of Game & Multimedia Engineering, Tech University of Korea, Siheung, Korea
**Professor, Dept. of Game & Multimedia Engineering, Tech University of Korea, Siheung, Korea

[Abstract]

To address the limitations of strict FIFO lock-free queues, relaxed queues have been actively studied as a way to loosen the FIFO constraint and improve parallelism. In this work, we present the Time-Stamped Window Decoupled Queue, a novel relaxed queue that integrates time-stamping with the window-decoupled approach. Experimental results demonstrate that our design outperforms existing relaxed queues in both micro- and macro-benchmarks. Notably, in microbenchmarks, the proposed queue achieves significantly higher performance as the enqueue rate increases, surpassing other relaxed queues. In a microbenchmark with 72 threads, our queue delivers 60.67% higher throughput than the d-Choice Balanced Operations Queue.

▸ **Key words**: Parallel Processing, Data structure, Lock-free, Queue, Relaxation

[요 약]

엄격한 FIFO Lock-Free Queue의 한계를 극복하기 위해, FIFO 제약을 완화하고 병렬성을 향상시키는 방법으로 Relaxed Queue가 활발히 연구되어 왔다. 본 연구에서는 타임스탬핑 알고리즘과 Window Decoupled 구조를 통합한 새로운 Relaxed Queue인, Time-Stamped Window Decoupled Queue를 제안한다. 실험 결과, 제안한 Relaxed Queue는 모든 마이크로/매크로벤치마크에서 기존 Relaxed Queue보다 뛰어난 성능을 보였다. 특히, 마이크로벤치마크에서 Enqueue Rate가 증가할수록 다른 Relaxed Queue를 압도적으로 능가하는 성능을 달성하였다. 또한 스레드 72개로 수행하는 마이크로벤치마크에서, 제안한 Relaxed Queue는 d-Choice Balanced Operations Queue보다 60.67% 높은 Throughput을 보였다.

▸ **주제어**: 병렬 처리, 자료 구조, 무잠금, 큐, 릴렉세이션

• First Author: Yong-Joon Cho, Corresponding Author: Yong-Joon Cho
*Yong-Joon Cho (yongjun1108j@naver.com), Dept. of Game & Multimedia Engineering, Tech University of Korea
**Nai-Hoon Jung (nhjung@tukorea.ac.kr), Dept. of Game & Multimedia Engineering, Tech University of Korea
• Received: 2025. 10. 17, Revised: 2025. 11. 22, Accepted: 2025. 12. 02.

http://www.ksci.re.kr pISSN:1598-849X | eISSN:2383-9945

I. Introduction

멀티코어 아키텍처의 발전으로 병렬 작업 실행 효율성이 크게 향상하였다. 이러한 시스템을 최대한 활용하기 위한, 효율적인 동시성 자료구조에 대한 수요가 증가하고 있다. 상호배제에 의존하는 블로킹(Blocking) 알고리즘은 멀티스레드 환경에서 발생하는 데이터 레이스(Data Race)와 같은 문제를 효과적으로 방지할 수 있으나, 컨보잉(Convoying)과 우선순위 역전(Priority Inversion)[1]과 같은 본질적인 한계는 극복하기 어렵다는 문제점이 있다. 이러한 문제점을 해결하기 위해 상호배제를 사용하지 않는, Lock-Free 알고리즘이 제안되었다. 그러나 Lock-Free 알고리즘조차도 항상 최적의 병렬성을 보장하는 것은 아니다. 특정 지점에 대해 스레드 간 경쟁이 심해질 경우, 잦은 자원 접근 실패로 인해 병렬성이 낮아지고 성능도 떨어진다. Michael-Scott Queue(MS)[2]와 같은 Strict FIFO(First-In First-Out)의 Lock-Free 큐가 바로 대표적인 예시이다. 이러한 문제점을 해결하기 위해서 Relaxation[3] 알고리즘이 연구되고 있다. Relaxation이란 정확성을 어느 정도 희생하는 대신, 병렬성을 높이는 알고리즘이다. 이를 기반으로 한 Relaxed Queue는, 기존의 Strict FIFO를 완벽히 보장하지 아니하고 어느 정도의 순서 섞임을 허용함으로서 성능 향상을 이루어냈다.

II. Related Works

엄격한 선입선출의 본질적인 문제점은 Head와 Tail에 집중되는 스레드 간의 경합으로 인한 성능 저하이다. Distributed Queue(DQ)[4]는 큐를 여러 개의 부분 큐(Partial Queue)로 나누어 이러한 문제를 해결하였다. DQ는 Enqueue 또는 Dequeue 연산을 실행할 때 부분 큐 중에서 가장 적절한 부분 큐를 선택하여 실행한다. DQ의 성능은 적절한 큐를 선택하는 알고리즘에 따라 결정된다. 해당 논문에서는 3가지 알고리즘, d-Random(d-RA), Least Recently Used(LRU), b-Round Robin(b-RR)가 제안되었다.

d-RA는 부분 큐 d개를 무작위로 선택하고, Enqueue를 할 때는 그 중에서 가장 원소가 적은 부분 큐를, Dequeue를 할 때에는 그 중에서 가장 원소가 많은 부분 큐를 선택한다. 이러한 방법은 Throughput은 가장 높지만 Relaxation Bound가 유한하지 않아 정확성이 매우 떨어진다.

LRU는 연산을 할 때, 가장 오랫동안 연산을 적용하지 않은 부분 큐를 찾아 연산을 진행한다. LRU는 Relaxation 정도를 자유롭게 조정할 수 있으나, 조정할 수 있는 방법은 오직 부분 큐의 개수를 조정하는 것뿐이다. 부분 큐의 개수를 과하게 늘리면 적절한 부분 큐를 찾기 위해 탐색해야 하는 부분 큐의 수 또한 증가하여 성능 하락으로 이어질 수 있다.

b-RR은 b개의 Round Robin 카운터를 이용하는 알고리즘이다. 부분 큐의 개수를 늘리는 것 외에도 Round Robin 카운터의 개수인 b의 값을 조정하여 Relaxation을 조정할 수 있으나 스레드 개수 이상으로 b의 값을 늘리는 것은 의미가 없다는 문제점을 가진다.

d-Choice Balanced Operations Queue(d-CBO)[5]는 d-RA를 개선한 자료구조이다. 똑같이 랜덤 기반 알고리즘을 사용하면서도, Relaxation Bound가 유한하지 않음에도 불구하고 d-RA보다 정확성이 훨씬 높음이 증명되었다.

2D Window Decoupled Queue(2Dd)[6]는 Window Decoupled(WD)를 이용하는 자료구조로서, 2개의 파라미터인 Width와 Depth를 이용해 Relaxation을 보다 자유롭게 조정할 수 있는 자료구조이다. Relaxation을 자유롭게 조정할 수 있어 확장성이 뛰어날 뿐만 아니라 동일 Relaxation Bound에서 LRU, RR 등보다 높은 성능을 보였다.

Time-Stamped Queue(TS)[7]는 Time-Stamped Stack을 큐로 변형한 자료구조이다. Enqueue 연산을 Wait-Free 하게 구현이 가능하다는 장점이 있으나, 스택을 큐로 변형하면서 소거(Elimination)[8]를 적용할 수 없어졌으며, 최소 타임스탬프를 찾기 위해 무조건 모든 부분 큐를 탐색해야 하는 문제점이 존재한다. 또한 TS-CAS(Compare and Swap)과 TS-interval은 TS Family 중에서 높은 성능을 보였으나 타임스탬프를 겹치기 위해 연산 도중 Pause를 해야 하는 문제점이 있다.

따라서 본 논문에서는 상술한 TS의 문제점을 해결하여 기존보다 더 나은 알고리즘인 Time-Stamped Window Decoupled Queue(TSWD)를 제안할 것이다.

III. Time-Stamped Window Decoupled Queue

1. Overview

앞서 언급된 TS의 문제점은 WD를 적용함으로서 해결할 수 있다. WD를 적용함으로서 Depth의 개념을 적용할

수 있으며, 타임스탬프를 겹치기 위해 연산 도중 Pause 할 필요가 없어지고, 타임스탬프의 최솟값을 찾기 위해 모든 부분 큐를 탐색할 필요가 없어진다. 또한 TS의 Enqueue 연산을 Wait-Free 하게 구현할 수 있다는 장점을 그대로 가져올 수 있어, 같은 WD를 사용하는 2Dd와의 차별화를 기대할 수 있다.

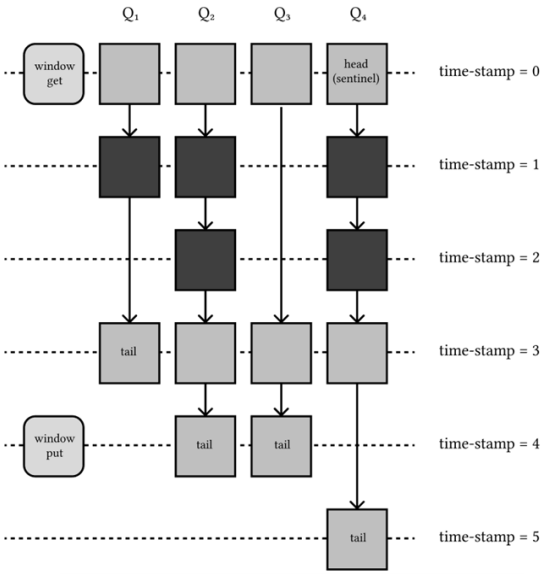

Fig. 1. TSWD as an example ($depth = 2$)

Fig. 1은 TSWD의 구조를 나타낸다. TSWD는 Window Structure 2개, Window Get(WG)과 Window Put(WP)를 가지며 스레드의 개수만큼의 부분 큐를 가지며, 각 스레드가 부분 큐 하나를 담당한다. 각 스레드는 각자의 부분 큐에만 Enqueue를 할 수 있다. 단, Dequeue를 할 때에는 이러한 제약이 없다. 완전(Total) 무제한(Unbounded) 큐를 구현하기 위하여 부분 큐는 연결 리스트(Linked List)로 구현한다. 각 노드는 삽입될 때 논리적 시간을 기반으로 한 타임스탬프를 할당한다. 각 부분 큐의 Head 노드의 타임스탬프는 0부터 시작한다.

WP와 WG 또한 타임스탬프를 가진다. WP의 타임스탬프는 새로 삽입되는 노드의 타임스탬프를 결정할 수 있으며, WG의 타임스탬프는 Dequeue의 대상이 될 수 있는 노드의 타임스탬프를 결정한다.

Dequeue의 대상이 될 수 있는 노드의 타임스탬프의 Lower Bound는 $Ts(WG) + depth$로 정의한다. $Ts(x)$는 x의 타임스탬프라고 하자. 즉, 임의의 노드 e에 대하여 $Ts(e) \leq Ts(WG) + depth$를 만족하면 노드 e는 Dequeue 가능한 노드이다. 예를 들어, Fig. 1의 경우, WG의 타임스탬프의 값은 0이고, Depth의 값은 2이므로 Dequeue 가능한 노드의 타임스탬프 하한선은 2가 된다. 즉, 어두운 색으로 표시된 원소들이 Dequeue의 대상이 될 수 있는 것이다.

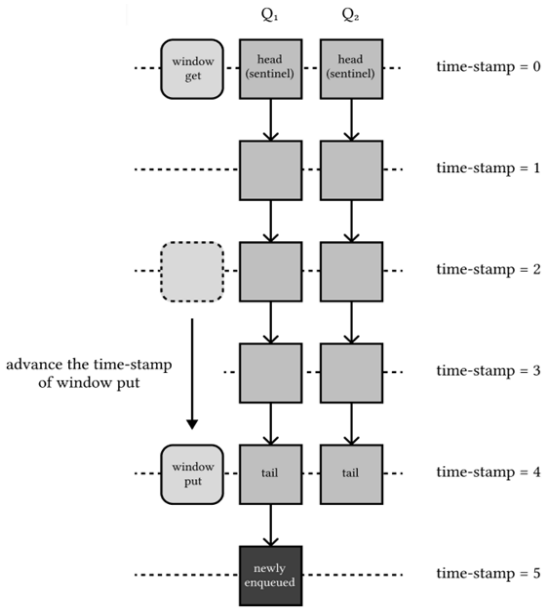

Fig. 2. Advancement of the time-stamp of WP ($depth = 2$)

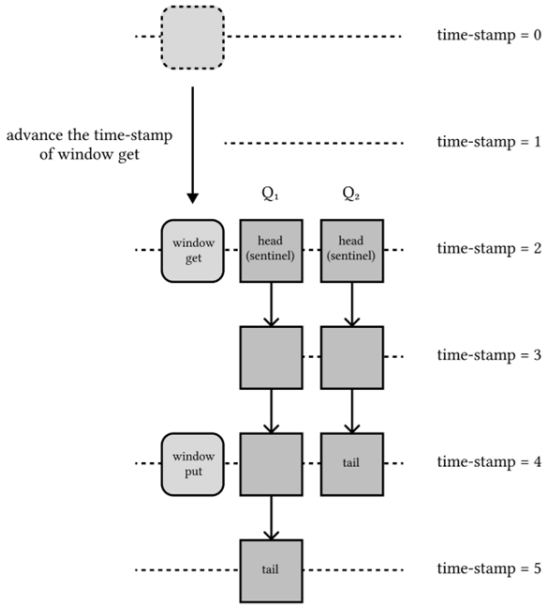

Fig. 3. Advancement of the time-stamp of WG ($depth = 2$)

Enqueue 실행 시 삽입되는 노드의 타임스탬프는, WP의 타임스탬프와 Enqueue를 하는 스레드가 담당하는 부분 큐의 Tail의 타임스탬프를 기반으로 하는, 휴리스틱 함수를 기반으로 결정된다. 새로 삽입되는 노드 e에 대하여 $Ts(e)$의 값은 $\max(Ts(WP), Ts(Tail(q))) + 1$이다.

$Tail(q)$는 부분 큐 q의 Tail이라고 하자. 예를 들어, Fig. 1의 경우, Q_1에 노드가 삽입된다면, WP의 타임스탬프가 4이고, Q_1의 Tail의 타임스탬프가 3이므로 새로 삽입되는 노드의 타임스탬프는 5가 된다.

2Dd와 유사하게, WG와 WP의 타임스탬프는 특정 조건 만족하에 전진한다. Fig. 2-3은 각각 WP와 WG의 타임스탬프가 전진하는 경우를 나타낸다. 원소 e가 부분 큐 q에 삽입되기 직전, $Ts(Tail(q)) = TS(WP) + depth$를 만족한다면, WP의 타임스탬프는 Depth만큼 전진한다. 모든 원소가 Dequeue 가능한 범위 내에 없으면 WG의 타임스탬프는 Depth만큼 전진한다. 단, 원소가 하나도 없다면 WG의 타임스탬프는 전진하지 않는다.

이를 바탕으로 순서도를 그리면 다음과 같다. 우선 첫 번째로 Fig. 4는 Enqueue의 과정을 나타낸다. 상술한 WP의 타임스탬프 전진 조건을 만족한다면 WP의 타임스탬프를 전진시킨 다음, 해당 스레드가 담당하는 부분 큐에 새로운 원소를 추가한다. 그렇지 않다면 WP의 타임스탬프 전진 없이 곧바로 부분 큐에 원소를 추가하면 된다.

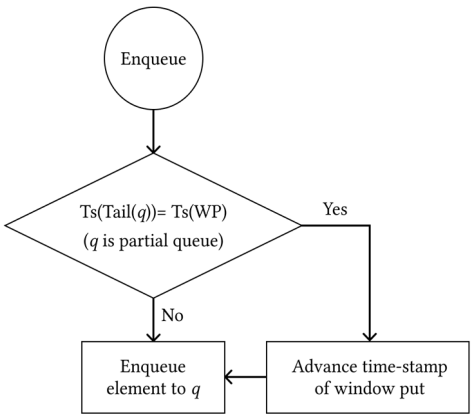

Fig. 4. Flow chart: enqueue of TSWD

다음으로 Fig. 5은 Dequeue의 과정을 나타낸다. 모든 부분 큐를 순회하며 각 부분 큐에 대해 Dequeue를 시도한다. Dequeue 시도에서 유효한 값이 반환되었다면, Dequeue는 곧바로 그 값을 반환한다. 어느 부분 큐도 유효한 값을 반환하지 못했다면 모든 부분 큐가 비어있는지 확인하고, 만약 모든 부분 큐가 비어있다면 Dequeue는 DEQUEUE_FAILED를 반환한다. 그렇지 않다면 위의 과정을 재시도한다.

하지만 실제로는 멀티스레드 환경에서 실행될 것이므로 동시성을 지키기 위하여 몇 가지 조치가 취해져야 한다. 이를 테면, WP 또는 WG의 타임스탬프를 전진시킬 때는 동시성이 보장되어야 하므로 이를 위해서 원자적 연산인 CAS(Compare-and-Swap)를 이용해야 한다. 자세한 구현은 바로 다음 절인 3장 2절에서 이어진다.

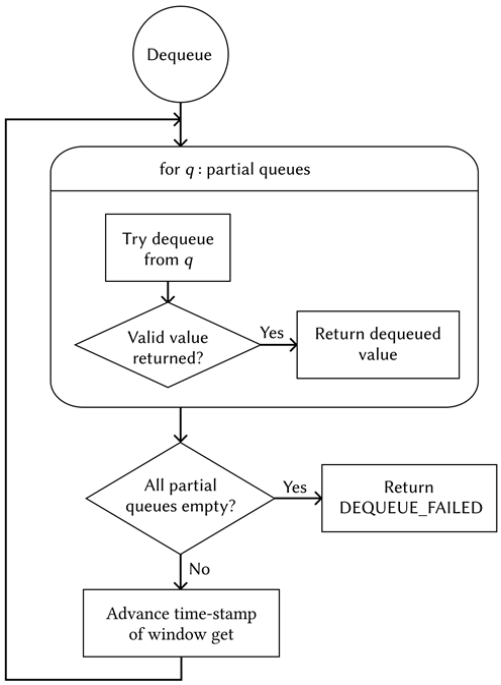

Fig. 5. Flow chart: dequeue of TSWD

2. Implementation

```
01      struct TimeStampedWindowDecoupledQueue:
02              Window window_get
03              Window window_put
04              PartialQueue queues[NUM_THREAD]
05
06      struct Window:
07              uint64 time_stamp
08
09      method Enqueue(value):
10              node := Node(value)
11              put_ts := window_put.time_stamp
12              if queues[THREAD_ID].tail.time_stamp = put_ts + depth:
13                      CAS(window_put.time_stamp, put_ts, put_ts + depth)
14              queues[THREAD_ID].Enqueue(node, put_ts)
```

Fig. 6. Algorithm: data structures and enqueue of TSWD

타임스탬프는 Overflow가 발생하여선 안 되므로 이를 최대한 방지하기 위해 자료형을 부호 없는 64비트 정수으로 하였다.

Enqueue 메소드는 먼저 노드를 생성하고 WP의 타임스탬프를 읽는 것으로 시작한다. 만약 Tail의 타임스탬프가 읽어온 WP의 타임스탬프(put_ts)와 Depth의 합과 같다면 WP의 타임스탬프는 CAS를 통해 Depth만큼 전진한다. 최종적으로 생성된 노드는 해당 스레드가 담당하는 부분 큐에 Enqueue 된다.

```
15  method Dequeue():
16          old_heads := Array(NUM_THREAD)
17          id := THREAD_ID
18          while TRUE:
19                  cnt_empty := 0
20                  put_ts := window_put.time_stamp
21                  get_ts := window_get.time_stamp
22                  for i in 0 to NUM_THREAD - 1:
23                          value, old_heads[id] := queues[id].TryDequeue(get_ts)
24                          if value = DEQUEUE_FAILED:
25                                  cnt_empty := cnt_empty + 1
26                          elif value != RETRY_REQUIRED:
27                                  return value
28                          id := (id + 1) % NUM_THREAD
29                  if cnt_empty = NUM_THREAD:
30                          is_empty := TRUE
31                          for i in 1 to NUM_THREAD - 1:
32                                  id := (i + THREAD_ID) % NUM_THREAD
33                                  next := old_heads[id].next
34                                  if next != NULL:
35                                          is_empty := FALSE
36                                          break
37                          if is_empty = TRUE:
38                                  return DEQUEUE_FAILED
39                  else:
40                          id := THREAD_ID
41                  if get_ts < put_ts:
42                          CAS(window_get.time_stamp, get_ts, get_ts + depth)
```

Fig. 7. Algorithm: dequeue of TSWD

```
43  struct PartialQueue:
44          Node head
45          Node tail
46
47  struct Node:
48          Value value
49          Node next
50          uint64 time_stamp
51
52  method Enqueue(node, put_ts):
53          tail_ts := tail.time_stamp
54          node.time_stamp := max(put_ts, tail_ts) + 1
55          tail.next := node
56          tail := node
57
58  method TryDequeue(get_ts):
59          while TRUE:
60                  loc_head := head
61                  first := loc_head.next
62                  if first = NULL:
63                          return { DEQUEUE_FAILED, loc_head }
64                  if first.time_stamp > get_ts + depth:
65                          return { RETRY_REQUIRED, loc_head }
66                  value := first.value
67                  if CAS(head, loc_head, first):
68                          return { value, loc_head }
```

Fig. 8. Algorithm: partial queue

Dequeue 메소드는 각 부분 큐를 탐색하면서 원소를 삭제하거나 삭제할 원소가 없음을 확인할 때까지 반복 시도한다. 전체 루프는 WP와 WG의 타임스탬프를 읽어, 타임스탬프의 스냅샷(put_ts와 get_ts)을 만드는 것으로 시작한다. 부분 큐 탐색 순서는, 스레드 간 충돌을 줄이기 위하여 현재 스레드의 ID 번째 부분 큐부터 시작하여 전체 부분 큐를 순회한다. 부분 큐의 TryDequeue 메서드를 통해 부분 큐로부터 Dequeue를 시도한다. 모든 부분 큐로부터 Dequeue를 시도한 결과가 DEQUEUE_FAILED(큐가 비어있어 Dequeue 실패)라면, Second Check(Line 29-38)를 통하여 전체 큐가 비어있는 것이 맞는지 확인한다. Second Check에 대한 것은 4장 1절에서 설명한다. 만약 모든 부분 큐로부터 유효한 값을 얻지 못하였으나 최소 한 번의 부분 큐의 Dequeue 시도의 결과가 RETRY_REQUIRED(Dequeue 가능 범위 밖에만 원소가 존재함)라면, WG의 타임스탬프를 Depth만큼 CAS를 통해 전진시키고 이러한 과정을 재시도한다.

새로 Enqueue 하고자 하는 노드의 타임스탬프는 put_ts와 Tail의 타임스탬프에 기반한 휴리스틱에 의해 결정된다 (line 54). 부분 큐의 Enqueue 메소드는 CAS 연산을 사용하지 않아도 된다. 이는 4장 1절에서 설명된다.

TryDequeue 메소드는 MS의 Dequeue 메소드와 마찬가지로, 원소의 삭제는 CAS를 통해 Head를 다음 노드로 전진시키는 것으로 구현한다. 메소드의 반환 값은, 큐가 비어있다면 DEQUEUE_FAILED, 성공적으로 원소를 삭제했다면 첫 번째 원소의 값, 해당 부분 큐에 Dequeue 가능 범위 밖에만 원소가 존재한다면 RETRY_REQUIRED이다. 또, Second Check를 위한 Head의 스냅샷(loc_head)도 같이 반환한다.

상술한 알고리즘은 메모리 재사용으로 발생하는 ABA 문제[9]를 고려하지 않았음에 주의한다. 실제 구현에서는 Epoch-Based Reclamation(EBR)[10-11] 또는 Hazard Pointer[12]과 같은 메모리 관리 기법을 추가로 적용해야 한다.

IV. Correctness

1. Lock-freedom and Thread-safety

Theorem 1. Enqueue 메소드는 스레드-세이프(Thread-Safe) 하며 정확히 하나의 원소만 삽입한다.

Proof 1. MS와는 달리, TryDequeue 메소드는 Head가 Tail을 앞지르는 것을 막지 아니한다. 하지만 각 부분 큐의 Tail에는 오직 스레드 1개만이 접근이 가능하기 때문에, Enqueue 메소드를 실행하기 직전의 Tail은 큐의 마지막 노드를 가리키고 있음이 보장된다. 따라서 CAS를 사용하지 않더라도 Enqueue 메소드는 스레드-세이프 하다. 같은 원리로 새로운 노드는 마지막 노드에 데이터 레이스 없이 정확히 한 번씩만 Link 된다.

Theorem 2. Enqueue 메소드는 Wait-Free 하다.

Proof 2. WP의 타임스탬프에 적용하는 CAS는 최대 1번만 실행되며, 다른 메소드와 충돌로 인한 재시도 과정 자체가 없다. 따라서 Enqueue 메소드는 항상 유한 스텝 내에 끝남이 보장되어 Wait-Free 하다[13].

Theorem 3. Dequeue 메소드는 스레드-세이프 하며

최대 하나의 원소만 삭제한다.

Proof 3. 원소를 삭제하는 것은 첫 번째 노드를 Head로 설정하는 것으로 구현되었다. 이는 CAS를 통해 최대 1번씩만, 스레드-세이프 하게 적용된다.

Theorem 4. Dequeue 메소드가 DEQUEUE_FAILED를 반환했다면, 모든 부분 큐가 비어있던 순간이 존재한다.

Proof 4. 루프를 통해 각 부분 큐가 전부 비어있음을 확인하고, 루프 중에 모든 부분 큐의 상태가 변하지 않았다면, 모든 부분 큐가 비어있던 순간이 존재함이 입증되며[14], TSWD는 Second Check를 통해 각 부분 큐의 상태가 변하지 않았음을 확인한다. 단, Dequeue를 실행하는 스레드가 관리하는 부분 큐의 경우는 확인할 필요가 없다. 자신이 관리하는 부분 큐에는 자신만 Enqueue 할 수 있기 때문이다.

Theorem 5. Dequeue 메소드는 Lock-Free 하다.

Proof 5. Dequeue 도중 재시도를 하는 경우는, Head에 대한 CAS가 실패하거나, 모든 원소가 Dequeue 가능한 범위 밖에 존재하는 경우, 두 가지뿐이다. 첫 번째의 경우, 어떤 스레드의 Head에 대한 CAS의 실패는 곧 다른 어떤 스레드의 CAS 성공을 의미한다. 두 번째의 경우, WG의 타임스탬프를 전진시키고 재시도를 하게 되는데, 노드의 타임스탬프의 값은 고정되어 있으므로, 최소 하나의 스레드의 WG의 타임스탬프 전진 시도 횟수는 유한함이 보장된다. 두 가지 경우 모두, 최소 하나의 스레드가 Dequeue 메소드를 유한 스텝 내에 끝낼 수 있음이 보장되므로 Dequeue 메소드는 Lock-Free 하다.

2. Application of EBR

C++과 같은 언어는 가비지 콜렉터를 지원하지 않아 메모리 재사용으로 인한 ABA 문제와 댕글링 포인터 역참조 문제가 발생할 수 있다. 따라서 여기서는 EBR을 통하여 두 문제를 해결하는 방법을 소개하고 증명한다.

MS나 DQ 기반의 Relaxed Queue에 EBR을 적용하는 경우, 각 메소드 실행 전에 Reservation을 등록하는 것으로 해당 문제를 해결할 수 있다. 이는 TSWD에도 적용된다. 하지만 TSWD의 경우, Enqueue 메소드에 대해 Reservation을 등록할 필요가 없다. 왜냐하면 각 부분 큐에는 스레드 하나만 Enqueue가 가능하므로, Enqueue 도중 참조하는 노드는 적어도 Sentinel 노드로서 남아있게 되어 절대로 회수되지 아니하기 때문이다.

3. Relaxation Analysis

k-out-of-order[3] 선형화 가능성(Linearizability)을 증명하기 위해서는 임의의 원소에 대해 최대 몇 개의 원소가 Dequeue 우선순위가 같은지를 확인해야 한다. 각 연산의 선형화 포인트를 정의한다면 직관적인 증명이 가능하지만, TSWD와 같이 타임스탬프 기반 알고리즘의 경우는 선형화 지점(Linearization Point)이 Non-Fixed[15]하여 해당 방식을 그대로 적용하기 어렵다. 따라서 여기서는 Lazy Element라는 개념을 정의함으로서 TSWD의 k-out-of-order 선형화 가능성을 증명한다.

Lemma 6.1. Enqueue와 Dequeue 연산 도중 다른 스레드의 연산이 끼어들지 않는다는 가정하에, 어떤 원소 e에 대해서, e와 Dequeue 우선순위가 같은 원소의 최대 개수는 $depth(p-1)$이다. (p는 부분 큐의 개수이다.)

Proof 6.1. 주어진 가정하에서는, Tail에 Link 되는 노드의 타임스탬프의 값은 모두 WP의 타임스탬프의 값보다 크다. 또한 WG의 타임스탬프의 최댓값은 곧 WP의 타임스탬프의 값이므로 Tail에 Link 되는 노드의 타임스탬프의 값은 모두 WG의 타임스탬프의 값보다 크다. Dequeue 메소드는 타임스탬프가 $Ts(WG)+depth$ 이하인 원소만을 제거하므로, Dequeue 우선순위가 같은 두 원소 x와 y에 대해서 식 (1)이 성립한다. $Q(e)$는 원소 e가 삽입된 큐이다.

$$\left\lceil \frac{Ts(x)}{depth} \right\rceil = \left\lceil \frac{Ts(y)}{depth} \right\rceil \wedge Q(x) \neq Q(y) \quad \cdots\cdots (1)$$

이에 따라 어떤 원소 e에 대해서, e와 Dequeue 우선순위가 같은 원소의 최대 개수는 $depth(p-1)$이다. 하지만 실제로는 당연히 Enqueue와 Dequeue 연산 도중 다른 스레드의 연산이 끼어들 수 있기 때문에, 해당 경우에 대한 추가적인 고려가 필요하다.

Lemma 6.2. Dequeue 우선순위가 같은 두 원소 x와 y에 대해서, 식 (1)을 만족하지 않는 원소 x와 y가 존재한다.

Proof 6.2. 식 (1)을 만족하지 않으면서 식 (2)를 만족하는 원소 x와 y 또한 Dequeue 순서가 Unordered 하다.

$$Q(x) \neq Q(y) \wedge Ts(x) \leq Ts(WG)+depth \wedge \cdots \quad (2)$$
$$Ts(y) \leq Ts(WG)+depth$$

실제로 이러한 경우가 존재한다. 다음 History를 고려해 보자. 가로축은 실행 시간을 나타내며, 위쪽 화살표는 메소드의 Call을, 아래쪽 화살표는 메소드의 Return을 나타낸다. $enq(q,e)$는 원소 e가 q번째 부분 큐에 Enqueue 되었음을 나타내며, $deq()\rightarrow d$는 원소 d가 Dequeue 되었음을 나타낸다. (A)는 큐가 생성된 순간, (B)와 (C)는 WG 또는 WP의 타임스탬프 값이 전진한 순간을 나타낸다.

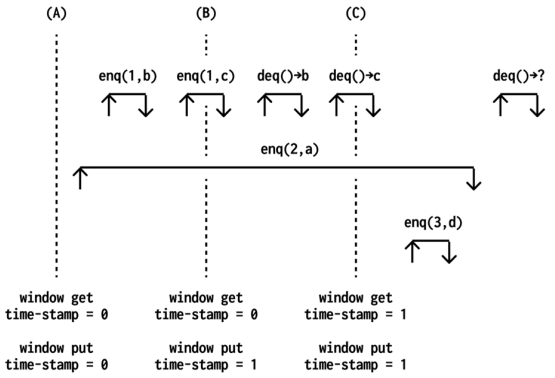

Fig. 9. Example execution history ($depth = 1$)

Fig. 9의 경우는 $depth$의 값이 1일 때의 예시이다. 만약 원소 a의 타임스탬프가 할당된 순간이 (A)와 (B) 사이라면, 원소 a의 타임스탬프는 1이 된다. 하지만 이렇게 될 경우, 원소 a는 타임스탬프가 2인 원소 d와 Dequeue 순서가 Unordered 하게 된다.

Definition 1. 어떤 원소 l에 대해서 식 (3)을 만족한다면 원소 l은 Lazy Element이다.

$$Ts(l) \leq Ts(WG) \quad \cdots\cdots (3)$$

Lemma 6.3. 각 부분 큐에는 Lazy Element가 최대 1개만 존재할 수 있다.

Proof 6.3. 이를 증명하려면 한 부분 큐에 Lazy Element가 2개 이상 존재할 수 없음을 보여야 한다. 해당 경우를 재현하려면, 타임스탬프가 $Ts(WG) + depth$ 이하인 원소를 2개 이상 관측하지 못한 채 WG의 타임스탬프를 전진시켜야 한다. 증명을 위하여 Enqueue와 Dequeue의 내부 연산 중 일부를 Table 1과 같이 정의한다.

Table 1. Orders on Enqueue and Dequeue

Order	Description
$pread$	Reads on the time-stamp of WP
$pcas$	Advances the time-stamp of WP via CAS
$link$	Links new tail
$td(q)$	Tries to dequeue from qth queue
$gcas$	Advances the time-stamp of WG via CAS

해당 경우를 Fig. 10에 도식화하였다. $enq(1,a)$의 $pread$가 실행되는 시점에서, WG와 WP의 타임스탬프 값은 동일하다고 하자. 만약 해당 경우를 재현할 수 있다면 원소 c와 원소 a, b는 Dequeue 순서가 Unordered 하게 되어, 한 부분 큐에 Lazy Element가 2개 이상 존재할 수 있게 된다. 해당 경우가 성립하려면 다음을 만족해야 한다.

(1) $enq(2,c)$의 $pcas$는 $enq(1,b)$의 $pread$가 실행된 뒤에 실행되어야 한다. 원소 c가 Enqueue 될 적의 WP의 타임스탬프는, 원소 a, b가 Enqueue 될 적의 WP의 타임스탬프와 값이 달라야 하기 때문이다.

(2) $td(1)$은 $enq(1,c)$의 $link$보다 먼저 실행되어야 한다. 원소 a와 원소 b를 관측하지 못해야 하기 때문이다.

(3) $deq()\rightarrow?$의 $pread$는 $enq(2,c)$의 $pcas$가 실행된 뒤에 실행되어야 한다. 그렇지 않으면 $gcas$는 실행되지 않는다. 전진한 WP의 타임스탬프를 읽지 않으면 WG의 타임스탬프를 전진시킬 수 없기 때문이다.

(1), (2), (3)을 모두 만족하는 것은 불가능하다. Fig. 10에서 볼 수 있듯, (1), (2), (3)을 만족하면 내부 연산 순서에 사이클이 발생하기 때문이다. 따라서 부분 큐 하나당 Lazy Element는 2개 이상 존재할 수 없다.

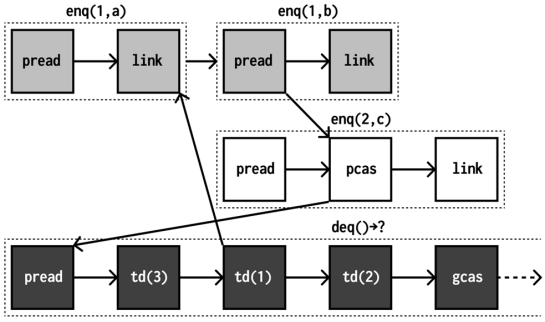

Fig. 10. Linearizability simulation for the concerned case

Lemma 6.4. TSWD에는 최대 $p-1$개의 Lazy Element가 존재한다.

Proof 6.4. Lazy Element가 생성되려면 다른 스레드가 Enqueue를 하는 동안, 적어도 스레드 하나는 WP의 타임스탬프를 전진시켜야 한다. 따라서 Lazy Element의 최대 개수는 $p-1$개가 된다.

Theorem 6. TSWD는 k-out-of-order로 선형화가 가능하다.

Proof 6. Enqueue와 Dequeue 연산 도중 다른 스레드의 연산이 끼어들지 않는다면 임의의 원소 e와 Dequeue 우선순위가 같은 원소의 최대 개수는 $depth(p-1)$이다. 또 Enqueue와 Dequeue 연산 도중 다른 스레드의 연산이 끼어들 때 발생할 수 있는 추가적인 경우, 즉 Lazy Element의 최대 개수는 $p-1$개다. 이에 따라 임의의 원소 e와 Dequeue 우선순위가 같은 원소의 최대 개수의 총합은 $depth(p-1)+p-1=(depth+1)(p-1)$개이다. 즉, $k=(depth+1)(p-1)$일 때의 k-out-of-order 선형화가 가능하다.

V. Experimental Evaluation

모든 알고리즘은 C++ 23으로 작성되었으며 g++ 13.2.0에서 -Ofast 최적화 옵션으로 컴파일 하였다. ABA 문제와 댕글링 포인터 역참조 문제는 EBR을 이용해 해결하였다. 실험은 Intel 기반 x86-64 서버 머신에서 진행되었다. 이 머신은 Linux 6.8.0-49에서 구동되며, Intel® Xeon® Gold 6240 CPU(2.60 GHz, 36코어, 코어당 2개의 하이퍼스레드), NUMA 아키텍처, 250 GiB RAM, 그리고 49.5 MiB 공유 L3 캐시 2개를 갖추고 있다.

1. Microbenchmark Performance

실험을 위해 각 스레드가 무작위 연산, Enqueue 또는 Dequeue를 수행하는 마이크로벤치마크를 설계하였다. 연산 횟수는 도합 36,000,000회이며, 빈 큐에 대해 Dequeue 연산을 실행하는 것을 방지하기 위해 실험 전에 큐에 원소를 10^5개를 Pre-fill 하였다. 벤치마크는 Enqueue Rate가 50%, 65%, 95%로 설정된 3가지 시나리오에 대해 실험을 진행한다. 측정하고자 하는 값은 Throughput과 평균 Relaxation Distance이다. Throughput을 측정할 때에는 각 연산마다 1.2 마이크로초만큼의 딜레이를 주었다. 마이크로벤치마크 실험 데이터는 5회 실험의 평균값이다. Relaxation Distance를 측정할 때에는 정확한 측정을 위해 선형화 지점에 전역 락을 걸었다. 다만 TSWD의 Enqueue 연산의 선형화 지점은 임의의 지점인 $pread$의 시점으로 지정하였다. 빈 큐에서 Dequeue 연산이 발생하지 않는 한 선형화 지점을 $pread$로 지정하여도 선형화 가능성에 영향을 주지 않는다.

1.1 General Scalability (Microbenchmark)

첫 번째로, 스레드의 개수를 증가시켜 가며 벤치마크를 진행한다. 스레드 개수에 따라 Relaxation의 정도 또한 커지도록 하였다.

대조군으로서, 2Dd와 d-CBO를 채택하였다. 2Dd는 역대 Relaxed Queue 중에서 충분한 높은 성능을 갖고 있음과 동시에 TSWD와 Relaxation Bound를 동일하게 맞추기 용이함에 채택하였으며, d-CBO는 Relaxation Bound는 유한하지 않지만, 역대 Relaxed Queue 중에서 Relaxation Distance 대비 Throughput이 매우 높은 편이기 때문에 채택하였다. 각 자료구조에 쓰인 파라미터의 값은 Table 2에 명시되어 있다. 부분 큐의 개수는 스레드의 개수(n)으로 설정했으며, TSWD와 2Dd의 Depth의 값은 각각 14와 15로 설정했다. 이 경우, TSWD와 2Dd의 Relaxation Bound는 $15(n-1)$로 동일하다.

Fig. 12-14는 평균 Relaxation Distance와 Throughput의 측정값을 보여준다. TSWD는 다른 Relaxed Queue보다 평균 Relaxation Distance 대비 Throughput이 높았다. Enqueue Rate가 50%인 시나리오에서, TSWD는 d-CBO보다 평균 Relaxation Distance는 약 6.79% 낮았으며 Throughput은 d-CBO보다 약 60.67% 높았다. 2Dd와 비교했을 때는, TSWD의 평균 Relaxation Distance는 2Dd보다 약 71.25% 낮았을뿐더러 Throughput은 약 55.68% 높았다. TSWD의 Enqueue 연산은 매우 가벼우며, Wait-Free이므로 다른 알고리즘보다 성능이 뛰어난 것으로 보인다. 이러한 이유로 Enqueue Rate가 높아질수록 Throughput 또한 상승함을 확인할 수 있다. 특히 Enqueue Rate가 95%인 시나리오에서, TSWD는 d-CBO보다 평균 Relaxation Distance는 약 22.68% 높았으나, Throughput은 d-CBO보다 약 284.26% 높았다. 2Dd와 비교했을 때는, TSWD의 평균 Relaxation Distance는 2Dd보다 약 58.39% 낮았을뿐더러 Throughput은 약 242.09% 높았다.

Table 2. Microbenchmark parameter

Data Structure	Parameter
TSWD	$depth = 14$
2Dd	$(width, depth) = (n, 15)$
d-CBO	$(width, d) = (n, 2)$

1.2 Relaxation Scalability (Microbenchmark)

다음으로, Relaxation Bound(k)를 증가시켜 가며 벤치마크를 진행한다. 이때 스레드 개수는 41개로 고정이다. 대조군으로서 2Dd를 채택하였다. TSWD와 Relaxation Bound를 동일하게 맞추기 용이함과 동시에 Relaxation Bound를 자유롭게 조정할 수 있기 때문이다. 각 자료구조에 쓰인 파라미터의 값은 Table 3에 명시되어 있다.

Fig. 15-17에서 볼 수 있듯, 모든 경우에서 TSWD는 2Dd보다 정확성과 Throughput이 높았다. Enqueue Rate가 50%인 시나리오에서, Relaxation Bound가 10240일 때, 평균 Relaxation Distance는 TSWD가 Width의 값이 41인 2Dd보다 약 45.36% 낮았으며, Throughput은 약 89.43% 높았다.

Table 3. Depth configuration (Microbenchmark)

Data Structure	Depth
TSWD	$depth = \frac{k}{40} - 1$
2Dd	$depth = \frac{k}{width - 1}$

다만, Enqueue Rate가 높은 시나리오에서는 TSWD의 Relaxation Bound의 값이 커져도 Throughput은 늘어나지 않았다. TSWD는 Enqueue Rate가 높을수록 스레드 간 경합이 줄어들게 되는데, Enqueue Rate가 극단적으로 높은 시나리오에서는 이미 스레드 간 경합이 최소이기 때문에, Relaxation Bound의 값을 크게 해도 Throughput 증가에 영향을 주지 못한 것이다. 그러나 이 점을 감안하여도, Enqueue Rate가 95%인 시나리오에서, Relaxation Bound가 10240일 때, 평균 Relaxation Distance는 TSWD가 Width의 값이 41인 2Dd보다 약 27.07% 낮았으며, Throughput은 약 148.66% 높았다.

2. Macrobenchmark Performance

또한 매크로벤치마크를 통해 TSWD가 충분히 실용적임을 보일 수 있다. 실험을 위해 경로 탐색 알고리즘인 Relaxed Breadth-First Search(RBFS)을 통하여 임의의 두 정점 간 거리를 계산하는 매크로벤치마크를 설계한다. RBFS는 Concurrent Breadth-First Search(CBFS)[5]와 다소 다른 방식으로 진행된다. CBFS와는 다르게 RBFS는 더 효율적인 경로를 발견할지라도 이미 방문한 노드는 방문하지 아니하며 어떤 스레드가 목적지에 도착할 경우 탐색을 중지한다. 이에 따라, 과한 Relaxation이 발생할 경우 비효율적인 경로를 찾게 될 것이며, 최적의 성능을 위해서는 Throughput만 고려할 것이 아닌, 적절하게 조절된 Relaxation이 필요하다. Fig. 11의 알고리즘은 시점과 종점 간 경로가 존재할 때만 정상 동작함에 주의한다.

실험에 사용된 그래프는 총 6종류(Alpha, Beta, Gamma, Delta, Epsilon, Zeta)이며, 모든 그래프는 Unweighted Graph임과 동시에 Undirected Graph이다. 각 그래프에 대한 정보는 Table 4에 명시되어 있다. 매크로벤치마크 실험 데이터는 10회 실험의 평균값이다.

2.1 General Scalability (Microbenchmark)

첫 번째로, 스레드의 개수를 증가시켜 가며 매크로벤치마크를 진행한다. 스레드 개수에 따라 Relaxation의 정도 또한 커지도록 하였다. 대조군으로서, 1.1절과 동일한 이유로 2Dd와 d-CBO를 채택하였다. 추가적으로 이번에는 TSWD와 같이 타임스탬프 기반 알고리즘을 사용하는 TS Family(TS-interval, TS-Stutter, TS-atomic, TS-CAS) 또한 채택하였다. 각 자료구조에 쓰인 파라미터의 값은 Table 5에 명시되어 있다.

```
method RelaxedBFS(src, dst):
        found := Array(NUM_THREAD)
        Fill(found, INF)
        dists := Array(NUM_VERTEX)
        Fill(dists, INF)
        dists[src] := 0
        has_ended := FALSE
        queue.Enqueue(src)
        threads := Array(NUM_THREAD)
        for i in 0 to NUM_THREAD - 1:
                threads[i] := Thread(WorkerThread(dst))
        for i in 0 to NUM_THREAD - 1:
                threads[i].Join()
        return Min(found)

method WorkerThread(dst):
        while has_ended = FALSE:
                curr := queue.Dequeue()
                if curr != DEQUEUE_FAILED:
                        dist := dists[curr] + 1
                        for adj in adj_vertices[curr]:
                                if adj = dst:
                                        has_ended := TRUE
                                        found[THREAD_ID] := dist
                                        break
                                if dists[adj] = INF:
                                        if CAS(dists[adj], INF, dist):
                                                queue.Enqueue(adj)
```

Fig. 11. Algorithm: relaxed breadth-first search

Table 4. Graphs used in macrobenchmark

Graph	Vertices	Edges	Shortest Distance
Alpha	4,500,000	53,608,016	386
Beta	4,500,000	320,150,816	60
Gamma	12,000,000	214,066,426	659
Delta	18,000,000	1,282,592,852	236
Epsilon	21,000,000	1,745,718,670	236
Zeta	25,000,000	2,474,029,252	238

Table 5. Macrobenchmark parameter

Data Structure	Parameter
TSWD	$depth = 56$
2Dd	$(width, depth) = (n, 57)$
d-CBO	$(width, d) = (n, 2)$
TS-CAS	$delay = 24\mu s$
TS-interval	$delay = 15\mu s$

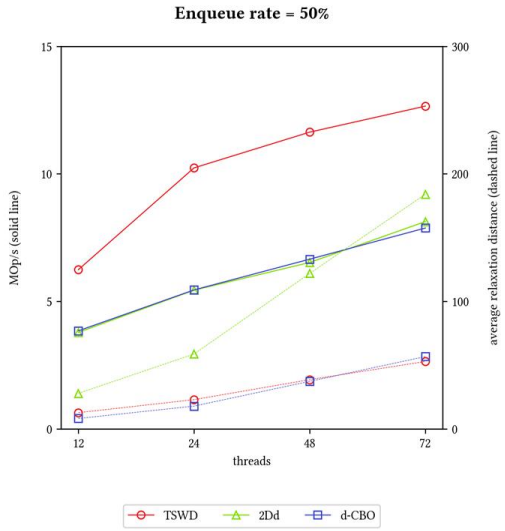

Fig. 12. Enqueue rate = 50% (general scalability)

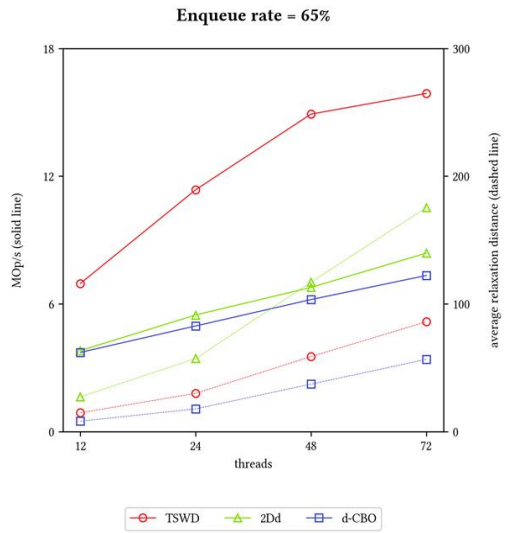

Fig. 13. Enqueue rate = 65% (general scalability)

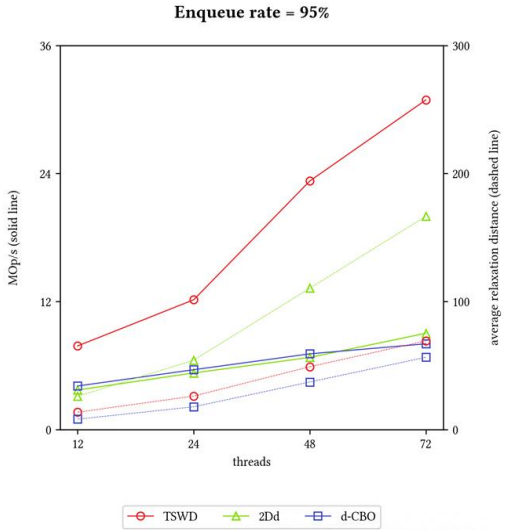

Fig. 14. Enqueue rate = 95% (general scalability)

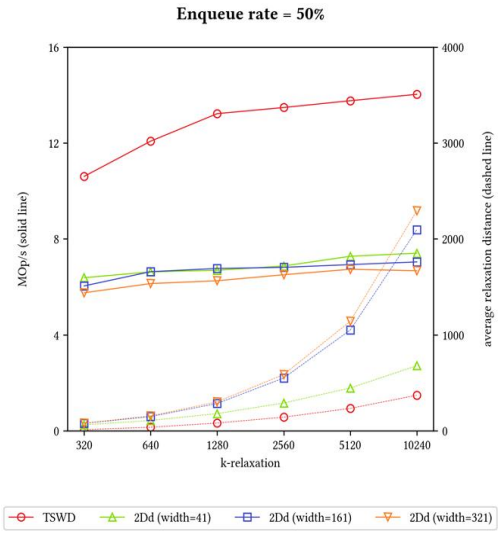

Fig. 15. Enqueue rate = 50% (relaxation scalability)

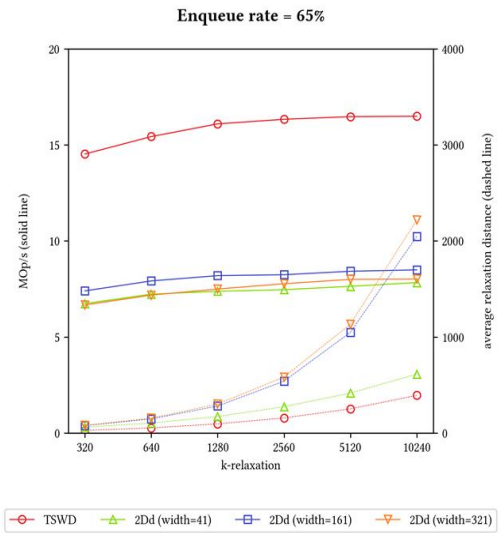

Fig. 16. Enqueue rate = 65% (relaxation scalability)

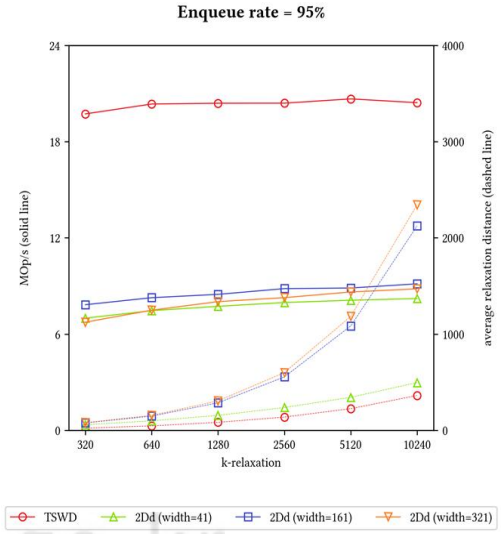

Fig. 17. Enqueue rate = 95% (relaxation scalability)

Fig. 18-23에서 볼 수 있듯, 모든 경우에서 TSWD는 다른 Relaxed Queue보다 오차 대비 처리 시간이 짧았다. 또한 동일하게 타임스탬프 기반 알고리즘을 이용하는 TS Family보다도 처리 시간이 매우 짧았다. 특히 간선과 정점이 매우 많은 그래프 Zeta에서, 72 스레드에서의 TSWD의 처리 시간은 TS Family 중 가장 처리 시간이 짧았던 TS-interval의 처리 시간보다 약 70.52% 낮았으며, d-CBO의 처리 시간보다 약 49.10% 낮았고, 2Dd의 처리 시간보다 약 61.89% 낮았다.

대체로 정확성은 비슷했으나 2Dd는 상대적으로 낮은 정확성을 보였다. 이를 테면, 간선과 정점이 가장 적은 그래프 Alpha에서는, 오차는 TSWD가 2Dd보다 약 2.41%p 낮았다.

2.2 Relaxation Scalability (Microbenchmark)

다음으로, Relaxation Bound를 증가시켜 가며 매크로벤치마크를 진행한다. 이때 스레드 개수는 41개로 고정이다. 1.2절과 동일한 이유로, 대조군으로서 2Dd를 채택하였다. 각 자료구조에 쓰인 파라미터의 값은 Table 6에 명시되어 있다.

Table 6. Depth configuration (Macrobenchmark)

Data Structure	Depth
TSWD	$depth = \frac{k}{40}$
2Dd	$depth = \frac{k}{width - 1}$

Fig. 24-29에서 볼 수 있듯, 모든 경우에서 TSWD는 2Dd보다 오차 대비 처리 시간이 짧았다. 특히 간선과 정점이 매우 많은 그래프 Zeta에서, 72 스레드에서의 TSWD의 처리 시간은 Width 값이 41인 2Dd의 처리 시간보다 약 58.41% 낮았다.

간선과 정점이 적을 경우, 전체적으로 TSWD가 2Dd보다 안정적인 정확성을 보였다. 특히 간선과 정점이 가장 적은 그래프 Alpha에서, Relaxation Bound가 5120일 때, TSWD가 Width 값이 41인 2Dd보다 오차가 약 2.93%p 낮았다. Relaxation Bound가 커질수록 이 차이는 더 커진다.

VI. Conclusion

본 논문에서는 WD와 타임스탬프를 이용한 새로운 Relaxed Queue인 TSWD를 제안하였다. TSWD의 모든 연산은 스레드-세이프 하게 동작함을 증명하였으며, Dequeue 연산은 Lock-Free 하게, Enqueue 연산은 Wait-Free 하게 수행됨을 보였다. 또한 TSWD의 Relaxation Bound는 유한함을 증명하였다. 뿐만 아니라, TSWD는 마이크로벤치마크와 매크로벤치마크를 통해 기존의 Relaxed Queue보다 성능이 더 우수함을 확인하였다. 특히 Enqueue 연산이 Wait-Free 함에 따라, Enqueue Rate가 높은 시나리오에서는 월등히 높은 Throughput을 보였다.

향후 연구 방향은 TSWD에 Lateral 구조[16]를 적용했을 때 Elastic Relaxation의 효용성을 확인하는 것이다.

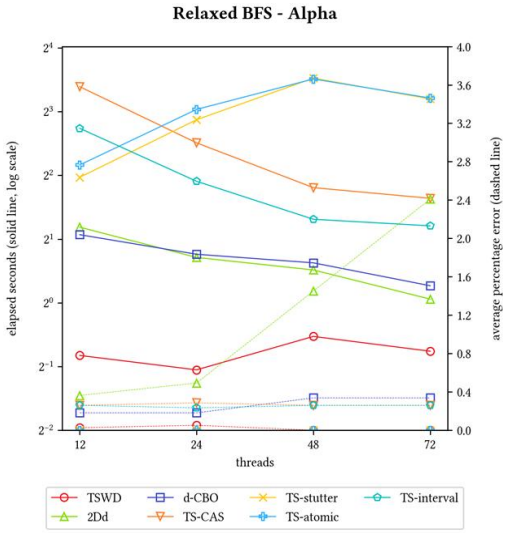

Fig. 18. Relaxed BFS – Alpha (general scalability)

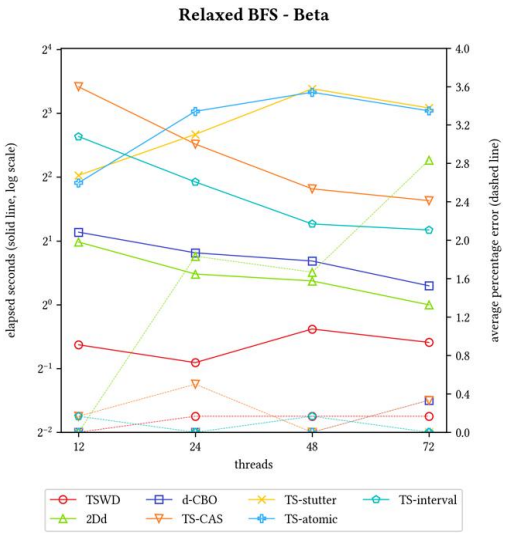

Fig. 19. Relaxed BFS – Alpha (general scalability)

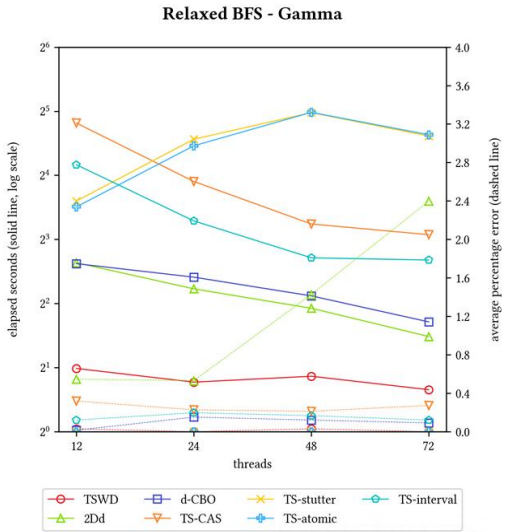

Fig. 20. Relaxed BFS – Gamma (general scalability)

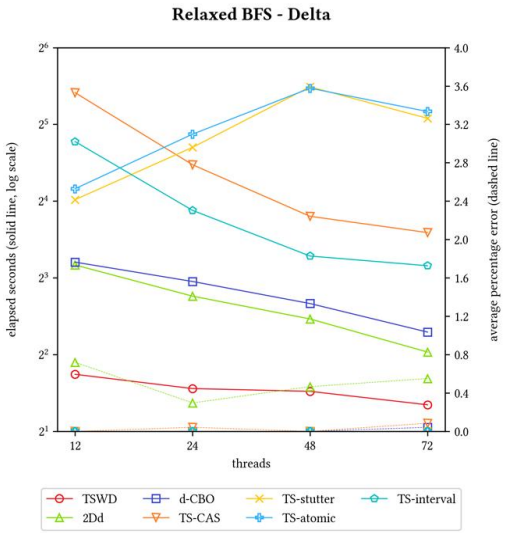

Fig. 21. Relaxed BFS – Delta (general scalability)

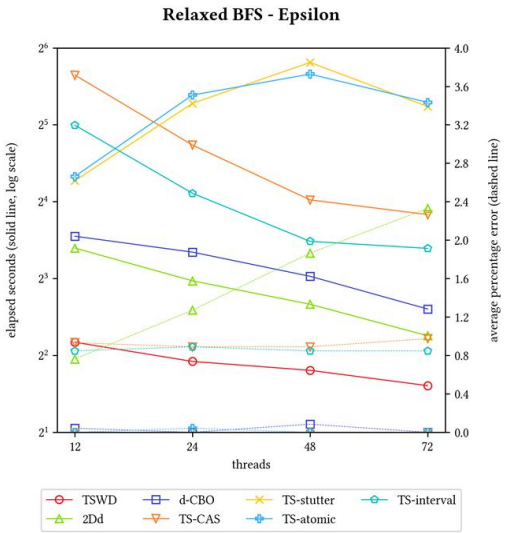

Fig. 22. Relaxed BFS – Epsilon (general scalability)

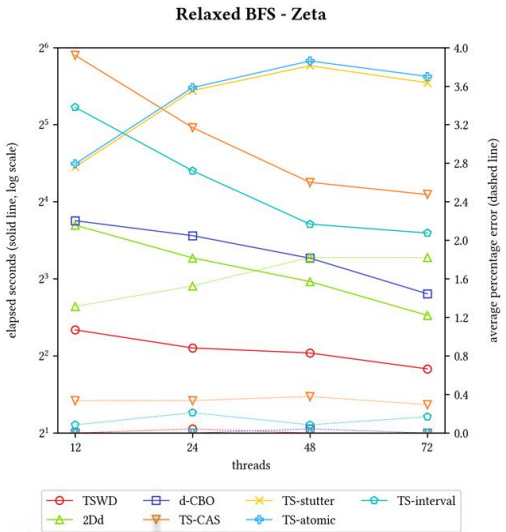

Fig. 23. Relaxed BFS – Zeta (general scalability)

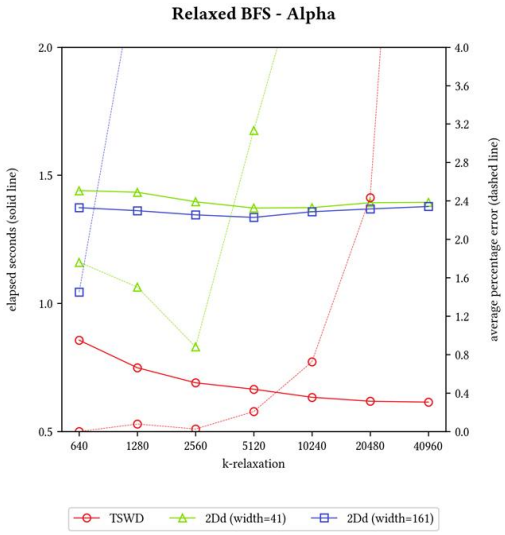

Fig. 24. Relaxed BFS – Alpha (relaxation scalability)

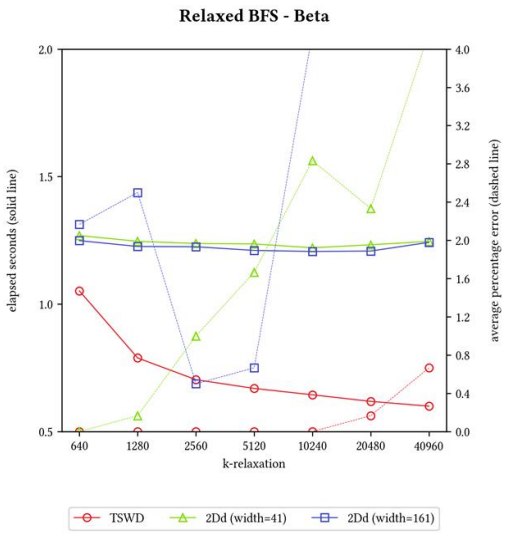

Fig. 25. Relaxed BFS – Beta (relaxation scalability)

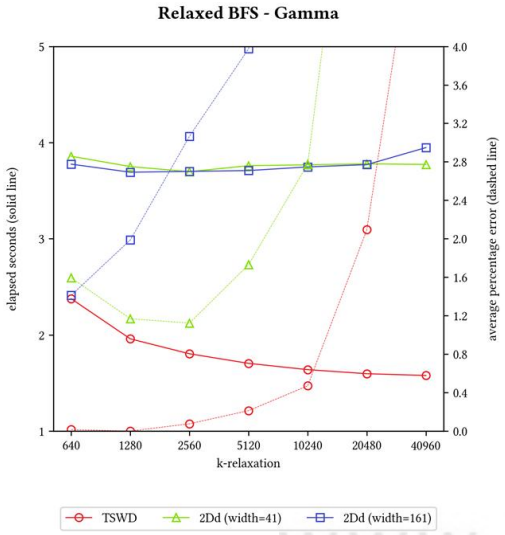

Fig. 26. Relaxed BFS – Gamma (relaxation scalability)

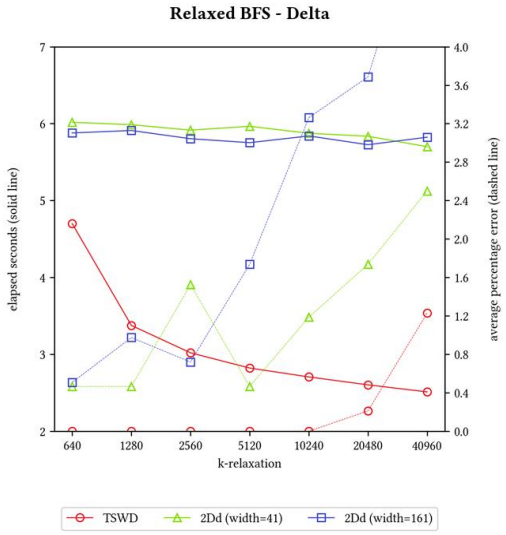

Fig. 27. Relaxed BFS – Delta (relaxation scalability)

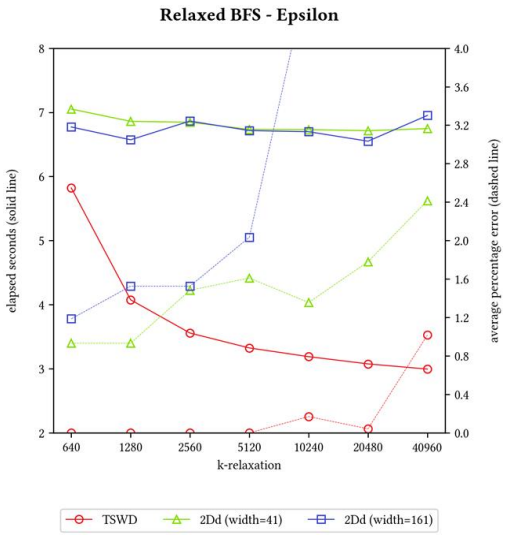

Fig. 28. Relaxed BFS – Epsilon (relaxation scalability)

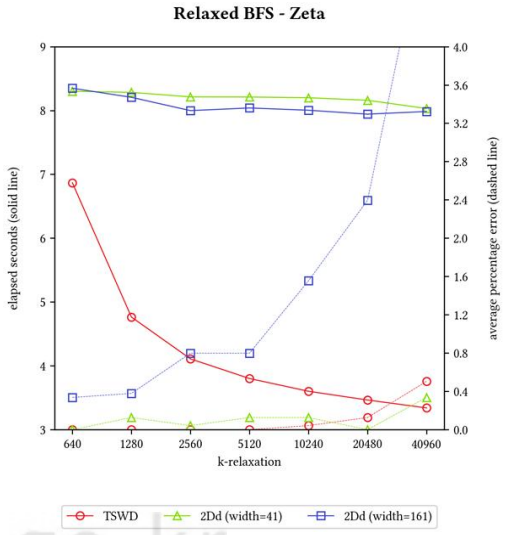

Fig. 29. Relaxed BFS – Zeta (relaxation scalability)

REFERENCES

[1] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. "*The art of multiprocessor programming*". Newnes. 2020.

[2] Michael, Maged M., and Michael L. Scott. "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms." Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing. 1996. DOI: 10.1145/248052.248106

[3] Henzinger, Thomas A., et al. "Quantitative relaxation of concurrent data structures." Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 2013. DOI: 10.1145/2429069.2429109

[4] Haas, Andreas, et al. "Distributed queues in shared memory: multicore performance and scalability through quantitative relaxation." Proceedings of the ACM International Conference on Computing Frontiers. 2013. DOI: 10.1145/2482767.2482789

[5] Von Geijer, Kåre, et al. "Balanced Allocations over Efficient Queues: A Fast Relaxed FIFO Queue." Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming. 2025. DOI: 10.1145/3710848.3710892

[6] Rukundo, Adones, Aras Atalar, and Philippas Tsigas. "Relaxing Concurrent Data-structure Semantics for Increasing Performance: A Multi-structure 2D Design Framework." arXiv preprint arXiv:1906.07105 (2019). DOI: 10.48550/arXiv.1906.07105

[7] Dodds, Mike, Andreas Haas, and Christoph M. Kirsch. "A scalable, correct time-stamped stack." ACM SIGPLAN Notices 50.1 (2015): 233-246. 2015. DOI: 10.1145/2775051.2676963

[8] Bar-Nissan, Gal, Danny Hendler, and Adi Suissa. "A dynamic elimination-combining stack algorithm." International Conference On Principles Of Distributed Systems. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011. DOI: 10.48550/arXiv.1106.6304

[9] Dechev, Damian, Peter Pirkelbauer, and Bjarne Stroustrup. "Understanding and effectively preventing the ABA problem in descriptor-based lock-free designs." 2010 13th IEEE international symposium on object/component/service-oriented real-time distributed computing. IEEE, 2010. DOI: 10.1109/ISORC.2010.10

[10] Fraser, Keir. "Practical lock-freedom". No. UCAM-CL-TR-579. University of Cambridge, Computer Laboratory, 2004. DOI: 10.48456/tr-579

[11] Wen, Haosen, et al. "Interval-based memory reclamation." ACM SIGPLAN Notices 53.1 (2018): 1-13. 2018. DOI: 10.1145/3200691.3178488

[12] Michael, Maged M. "Hazard pointers: Safe memory reclamation for lock-free objects." IEEE Transactions on Parallel and Distributed Systems 15.6 (2004): 491-504. 2004. DOI: 10.1109/TPDS.2004.8

[13] Herlihy, Maurice. "Wait-free synchronization." ACM Transactions on Programming Languages and Systems (TOPLAS) 13.1 (1991): 124-149. 1991. DOI: 10.1145/114005.102808

[14] Herlihy, Maurice P., and Jeannette M. Wing. "Linearizability: A correctness condition for concurrent objects." ACM Transactions on Programming Languages and Systems (TOPLAS) 12.3 (1990): 463-492. 1990. DOI: 10.1145/78969.78972

[15] Khyzha, Artem, et al. "Proving linearizability using partial orders." European Symposium on Programming. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017. DOI: 10.1007/978-3-662-54434-1_24

[16] Von Geijer, Kåre, and Philippas Tsigas. "How to relax instantly: Elastic relaxation of concurrent data structures." European Conference on Parallel Processing. Cham: Springer Nature Switzerland, 2024. DOI: 10.1007/978-3-031-69583-4_9

Authors

Yong-Joon Cho has been pursuing the B.S. degree in Game & Multimedia Engineering at Tech University of Korea since 2019. He is interested in parallel computing and large-scale online game servers.

Yong-Joon Cho is currently a student in the Department of Game & Multimedia Engineering, Tech University of Korea. He is interested in parallel computing and large-scale online game servers.

Nai-Hoon Jung received the Ph.D. degree in Computer Science from KAIST, Korea, in 2002. Since 2008, he has been with the Department of Game & Multimedia Engineering, Tech University of Korea,

where he is currently a Professor. Dr. Jung joined the faculty of the Department of Game & Multimedia Engineering at Tech University of Korea, Siheung, Korea, in 2008. He is currently a Professor in the Department of Computer Science, Tech University of Korea. He is interested in parallel computing and large-scale online game servers.