Journal of Korea Game Society 2021 Feb; 21(1): 55-66
http://dx.doi.org/10.7583/JKGS.2021.21.1.55

게임 프로그래밍

# C++11 멀티스레드 프로그래밍을 위한 Lock-Free shared_ptr와 weak_ptr의 구현

구태균*, 정내훈**
한국산업기술대학교 게임공학과*, 한국산업기술대학교 게임공학부**
snrn2426@gmail.com*, nhjung@kpu.ac.kr**

# Implementation of Lock-Free shared_ptr and weak_ptr for C++11 multi-thread programming

TaeKyun Ku*, NaiHoon Jung**
Dept. of Game Engineering, Korea Polytechnic Univ.* , Dept. of Game & Multimedia Engineering, Korea Polytechnic Univ.**

## 요 약

고성능이 요구되는 게임 프로그래밍에서 멀티스레드 프로그래밍은 필수이다. 하지만 널리 사용되는 C++11의 shared_ptr와 weak_ptr는 멀티스레드 환경에서 오작동 및 성능 문제를 가지고 있다. 본 논문에서는 기존의 오작동 방지 방법보다 높은 성능을 가지는 Lock-Free shared_ptr와 weak_ptr를 제안한다. 제안하는 두 객체는 논블로킹 알고리즘을 이용하여 멀티스레드에서의 데이터 레이스를 방지하였으며, 8스레드 환경에서 실험한 결과 스레드 사이의 경쟁이 낮은 상황에서 기존의 방법보다 최대 7424% 향상되었고, 경쟁이 높은 상황에서 최대 3767% 향상된 성능을 보여준다.

## ABSTRACT

Multi-thread programming is essential in high performance game programming. But, the widely used C++11 shared_ptr and weak_ptr have malfunction and performance problems in multi-thread environments. In this paper, we propose Lock-Free shared_ptr and weak_ptr, which have higher performance than current error preventing methods. These use a non-blocking algorithm to prevent data race in multi-thread environments. As a result of experimenting in an 8 thread environment, performance has improved up to 7424% in a situation where competition between threads is low, and 3767% in high competition.

**Keywords** : Lock-Free(무잠금), C++11 shared_ptr(쉐어드 포인터), C++11 weak_ptr(위크 포인터), Data race(데이터 레이스)

Received: Nov. 11. 2020. Revised: Jan. 29. 2021.
Accepted: Jan. 29. 2021.
Corresponding Author: NaiHoon Jung(Korea Polytechnic Univ.)
E-mail: nhjung@kpu.ac.kr

ISSN: 1598-4540 / eISSN: 2287-8211

## 1. 서 론

최근 멀티코어 프로세서가 발전하면서 게이밍 PC의 CPU의 코어 수도 증가하고 있다. 스팀에서 제공하는 하드웨어 통계[1]를 보면 현재 4코어(40.28%), 6코어(32.56%)와 8코어(11.65%)인 CPU 제품이 84.49%를 차지하고 있고, 6코어와 8코어 CPU 제품의 비율이 꾸준히 증가하고 있는 것을 볼 수 있다. 이처럼 게임에 이용되는 CPU가 증가함에 따라, 이에 필요한 효율적인 멀티스레드 프로그래밍이 필수적으로 요구되고 있다.

멀티스레드 프로그래밍으로 구현되는 대용량 다중 접속 게임서버와 고성능 3D 게임엔진은 주로 처리 속도가 빠른 C++를 이용해 구현된다. 하지만 낮은 레벨(low level) 언어인 C++는 가비지 컬렉션(garbage collection)을 제공하지 않아 동적 메모리를 사용할 때 사용자가 직접 메모리를 할당(new)/해제(delete)해야 하는 단점을 갖는다. 이로 인해 게임서버와 게임엔진을 구현할 때 메모리 관리에 대한 어려움이 동반된다.

C++11에서는 메모리를 쉽게 관리할 수 있도록 std::shared_ptr와 std::weak_ptr 클래스를 제공한다[2]. 하지만 두 객체는 싱글스레드에서만 정상적으로 동작하도록 구현되어있어 멀티스레드에서 사용하는 경우 데이터 레이스(Data race)[3]로 인한 프로그램의 오동작을 야기한다. 그러므로 두 객체를 멀티스레드에서 사용하기 위해서는 C++11 std::atomic 템플릿의 atomic_load()와 atomic_store()를 이용해 데이터 레이스를 해결해야 한다. 하지만 두 메소드들은 잠금을 이용하기 때문에 멀티스레드에서 병렬성이 떨어지고 이로 인해 성능이 낮아지는 단점을 가진다. 이러한 문제점을 개선한 std::experimental::atomic_shared_ptr가 있지만 이는 C++20부터 제공되기 때문에 현재 이용할 수 없다. 따라서 멀티스레드 게임서버와 게임엔진 프로그램에서 C++11 std::atomic과 함께 이용되는 std::shared_ptr와 std::weak_ptr의 사용은 게임의 성능을 악화시키는 요인이 된다.

멀티스레드에서의 효율적인 메모리 관리를 위해 Hazard Pointer[4]와 EBR(Epoch Based memory Reclamation)[5] 등 여러 가지 시스템이 제안되었다. 참고문헌 [6]은 효율적인 EBR인 DEBRA를 소개하며, Hazard pointer와 DEBRA를 이용한 코드를 보여준다. 두 코드를 살펴보면 알고리즘과 두 시스템이 밀접하게 연동되어 있어 해당 알고리즘에 대한 이해가 낮으면 적용하기 어려운 것을 볼 수 있다. 기존 프로그램에서 포인터 타입의 변수를 shared_ptr로 단순히 대체하기만 하면 되는 shared_ptr에 비해, EBR과 Hazard Pointer는 알고리즘에서 메모리의 사용과 해제를 고려해 고유의 API들을 적절한 위치에 추가해야 하는 높은 사용 난이도를 단점으로 가지는 것을 이를 통해 알 수 있다.

본 논문에서는 C++11 std::atomic 템플릿을 이용한 C++11 shared_ptr와 weak_ptr보다 높은 성능을 가지는 Lock-Free shared_ptr와 weak_ptr를 제안한다. 제안하는 두 객체는 논블로킹 알고리즘의 일종인 Lock-Free 알고리즘을 통해 데이터 레이스를 해결하고, 메모리 재사용 관리 객체인 Recycle Linked List를 이용해 객체 내부에서 발생하는 메모리 관리 문제를 해결하였다.

논문의 구성은 다음과 같다. 2장에서 관련연구에 대해 설명하고, 3장에서 Lock-Free shared_ptr와 weak_ptr를 구현하며, 4장에서는 구현된 알고리즘의 성능을 측정하고 분석한다. 마지막으로 5장에서 결론과 향후 연구방향을 기술한다.

## 2. 관련연구

이번 장에서는 Lock-Free shared_ptr와 weak_ptr의 기반이 되는 Lock-Free 알고리즘을 설명한 후, C++11 shared_ptr와 weak_ptr의 구현 방식과 문제점에 관해 설명한다.

### 2.1 Lock-Free 알고리즘

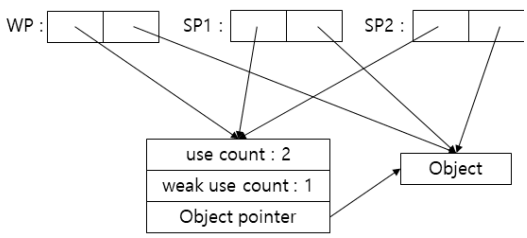
멀티스레드에서 사용되는 병렬 자료구조는 원자

적으로 동작해야 하며 이를 위해 잠금을 사용하거나 [1]CAS(Compare And Swap)와 같은 원자 연산자를 이용하여 구현된다. 이 중 잠금을 이용하는 경우 멀티스레드 프로그램은 블로킹이 발생하며, 블로킹은 컨보잉(convoying) 등의 문제점들[7]을 야기한다. 이러한 이유로 인해 블로킹이 발생하지 않도록 원자 연산자만을 이용하여 구현되는 논블로킹 알고리즘이 선호되고 있다. 여러 논블로킹 알고리즘 중 구현이 쉽고 성능이 높은 장점을 가진 Lock-Free 알고리즘[8]이 현재 가장 많이 사용되고 있으며, 본 논문에서도 Lock-Free 알고리즘을 통해 Lock-Free shared_ptr와 weak_ptr를 구현하였다.

Lock-Free 알고리즘에서는 메모리를 해제할 때 다른 스레드가 해제되는 메모리에 접근하여 발생하는 메모리 해제 타이밍 문제와 [2]ABA문제[9]가 발생한다. 본 논문에서는 이러한 문제점들을 해결하기 위해 메모리 재사용 관리 객체인 Recycle Linked List(RLL)를 통해 메모리를 재사용하는 방법을 이용하였으며, RLL에 대한 자세한 내용은 3.2.4절에서 다룬다.

## 2.2 C++11 shared_ptr/weak_ptr

C++11 shared_ptr(SP)/weak_ptr(WP)는 스마트 포인터의 일종으로 참조 횟수가 0이 되면 원본 객체가 해제되는 계수 포인터로 구현된다. 본 논문에서는 원본 객체를 공유 객체라 하였고, 여러 개의 shared_ptr/weak_ptr가 같은 객체를 가리킬 때 이를 공유한다고 표현하였다.




[Fig. 1] C++11 shared_ptr/weak_ptr

[Fig. 1]은 2개의 SP와 1개의 WP가 객체 1개를 공유하는 상황을 표현한다. SP/WP는 내부적으로 2개의 포인터를 가지며, 각각 공유 객체와 control_block을 참조한다[10]. control_block은 공유 객체 당 하나씩 짝을 이루는 자료 구조로 참조 횟수와 관련된 use count와 weak use count 변수를 가진다. 또, use count가 0인 경우 공유 객체를 해제하고 use count와 weak use count가 모두 0인 경우 control_block을 해제하여 메모리를 관리한다.

SP/WP의 객체 공유 과정은 다음과 같다.

1) SP/WP가 공유 객체를 참조한다.

2) SP/WP가 control_block을 참조한다.

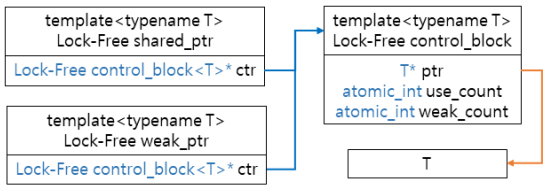
3) 참조한 control_block의 use count/weak use count를 증가시킨다.

하지만 SP/WP는 위의 과정들이 동시에 실행되지 않아 멀티스레드 환경에서 데이터 레이스가 발생하며, 이를 해결하기 위해 std::atomic 템플릿의 atomic_store()와 atomic_load()를 이용해야 한다.

# 3. Lock-Free shared_ptr/weak_ptr

## 3.1 구조

SP/WP의 데이터 레이스를 해결하기 위해 Lock-Free shared_ptr(LFSP)/weak_ptr(LFWP)는 다음과 같은 구조를 가진다.



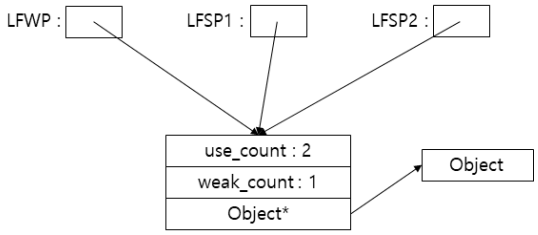

[Fig. 2] Organization of Lock-Free shared_ptr/weak_ptr

1) CAS는 메모리 값을 주어진 값과 비교하고 동일한 경우에만 해당 메모리 값을 주어진 값으로 수정하는 단일 원자 연산이며, 그 동안 다른 스레드에서 값을 수정하는 경우 실패한다.

2) ABA는 약어가 아니며, 공유 위치의 값이 A에서 B로 변경된 다음 다시 A로 변경될 수 있음을 나타낸다.

LFSP/LFWP는 SP/WP와는 달리 메모리 관리 객체인 Lock-Free control_block(LFCB)을 참조하는 ctr 포인터만을 가지며, LFCB는 공유 객체 T를 가리키는 ptr 포인터와 control_block의 use count와 weak use count에 해당하는 use_count(uc)와 weak_count(wc)를 가진다. SP/WP와 동일하게 LFSP/LFWP도 uc가 0인 경우 공유 객체를 해제하며, uc와 wc가 0인 경우 LFCB를 재사용하여 메모리를 관리한다.

다음은 [Fig. 1]과 동일한 상황에서 LFSP와 LFWP를 이용한 경우를 표현한다.




[Fig. 3] Lock-Free shared_ptr/weak_ptr

이처럼 LFSP/LFWP는 LFCB를 통해 공유 객체에 접근할 수 있으며, 공유 객체를 직접 참조하지 않는 이러한 구조는 2.2절에서 설명한 SP/WP의 3가지 과정을 2가지 과정으로 줄여 Lock-Free 알고리즘으로 구현하기 위함이다.

## 3.2 Lock-Free 구현

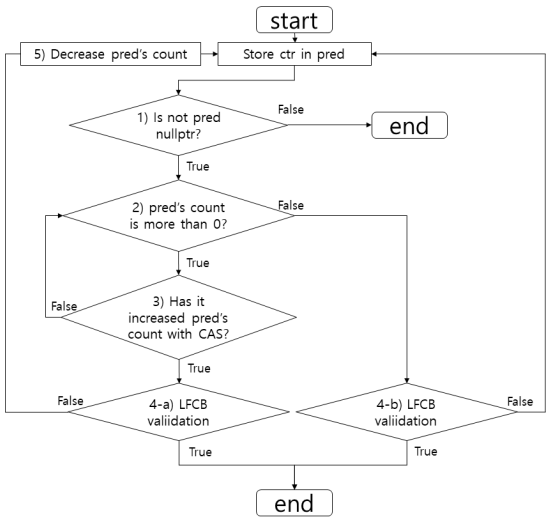
LFSP/LFWP의 2가지 과정은 addcopy 알고리즘과 '='operator 알고리즘을 이용하여 구현된다. 참조할 LFCB의 count(uc 혹은 wc)를 증가시키는 과정은 addcopy 알고리즘에서 실행되며, 다른 LFCB를 참조하는 ctr을 수정하는 과정은 '='operator 알고리즘에서 실행된다.

LFSP와 LFWP에서 사용하는 두 알고리즘은 count 변수의 차이만 있으므로 3.2.1절과 3.2.3절에서는 LFSP에 대해서만 논하였으며, 이해를 돕기 위해 count를 uc라 하였다.

### 3.2.1 addcopy 알고리즘

addcopy 알고리즘은 LFSP가 참조하는 LFCB를 다른 LFSP가 참조하려 할 때, 메모리 재사용으로부터 LFCB를 안전하게 사용하기 위해 LFCB의 사용 가능 여부를 반환하는 알고리즘이다. LFCB를 사용할 수 있는(uc가 1 이상인) 경우 uc를 증가시킨 후 성공을 반환하며, LFCB를 사용할 수 없는(uc가 0인) 경우 실패를 반환한다.

다음은 addcopy 알고리즘의 순서도이다.




[Fig. 4] addcopy algorithm Flow chart

위와 같이 addcopy 알고리즘은 1)부터 4-a)까지의 과정을 모두 만족하여야 count의 증가를 보장하며, 1)의 False와 2)의 False 이후 4-b)의 True는 count 증가의 실패를 의미한다. 여기서 pred는 LFSP가 참조하고 있는 LFCB를 뜻한다.

1)은 LFSP가 참조하는 LFCB가 있는지 확인하기 위해 pred를 검사하며, pred가 nullptr라면 알고리즘은 실패로 종료된다. pred가 nullptr가 아닌 경우 2)와 3)의 순서와 같이 LFCB의 uc가 0보다 큰 값인지 검사한 뒤 CAS를 이용해 원자적으로 증가시켜야 한다. 이는 사용 가능한(uc가 1 이상인) LFCB의 uc를 증가시킴으로써 다른 스레드에서 uc를 감소시키더라도 항상 1 이상인 uc를 가져 LFCB가 재사용되지 않도록 보장할 수 있기 때문

이다. 하지만 2)와 3)은 다른 스레드가 ctr을 수정하는 것을 반영할 수 없다. 3)에서 이용되는 LFCB는 pred이므로 수정 이전의 ctr을 의미하기 때문이다. 따라서 4-a)에서는 LFSP의 ctr이 수정되었는지 확인하는 LFCB의 유효성을 검사하며, LFCB가 유효한 경우에만 알고리즘이 성공으로 종료된다. LFCB가 유효하지 않은 경우에는 초기 상태로 돌아가 수정된 ctr로 다시 시도해야하며, 5)에서는 3)에서 증가한 pred의 uc를 감소시킨다. LFCB가 재사용되는 상황에서도 LFCB 유효성 검사는 유효하며, 이에 대한 내용은 3.2.2절에서 자세히 다룬다.

2)에서 uc가 0인 경우 LFSP는 사용할 수 없는 LFCB를 참조하고 있음을 의미하며, 4-b)에서 LFSP가 참조하는 LFCB가 없는 경우 LFCB가 유효하다. 하지만 이때는 사용 불가능한 LFCB이므로 알고리즘은 실패로 종료된다. 4-b)에서 LFSP가 LFCB를 참조하고 있는 경우는 다른 스레드가 LFSP의 ctr을 수정했음을 의미한다. 따라서 LFSP가 사용 가능한 LFCB를 참조하고 있으므로 LFCB가 유효하지 않다. 이때는 초기 상태로 돌아가 수정된 ctr로 알고리즘을 다시 시도해야하며, 3)을 실행하지 않아 uc가 증가되지 않았으므로 uc를 감소시키는 5)의 과정은 생략한다.

[Fig. 5]는 addcopy 알고리즘을 이용한 LFSP의 멤버 메소드 add_shared_copy()의 의사코드를 보여준다.

```
1: method add_shared_copy() : control_block*
2:     while :
3:         pred = ctr
4:         if pred is nullptr :
5:             ret nullptr
6:         ret_ctr = pred→add_use_count()
7:         curr = ctr
8:         if ret_ctr is curr :
9:             return ret_ctr
10:        else :
11:            if ret_ctr is not nullptr :
12:                pred→release()
13: method add_use_count() : control_block*
14:    while :
15:        curr = use_count
16:        if curr > 0 :
17:            if CAS(use_count, curr, curr + 1) is true :
18:                ret this
19:        else :
20:            ret nullptr
21: method release()
22:    while :
23:        curr = use_count
23:        if CAS(use_count, curr, curr - 1) is true :
24:            if curr = 1 :
25:                delete ptr
26:            return
```

[Fig. 5] add_shared_copy() Operation in LFSP

add_shared_copy()는 addcopy 알고리즘이 성공한 경우 pred의 uc를 증가시킨 후 pred를 가리키는 curr을 반환하며, 알고리즘이 실패한 경우 nullptr를 반환한다.

이해를 돕기 위해 [Fig. 4]와 [Fig. 5]를 함께 보자. 4-5라인에서는 1)과 같이 pred를 검사하며, pred가 nullptr인 경우 nullptr를 반환해 LFCB의 사용 불가를 알린다. 6라인의 add_use_count()는 2)와 3)에 해당하는 LFCB의 멤버 메소드이다. 13-20라인을 살펴보면 add_use_count()는 16라인에서 uc가 0보다 큰지 검사하고 17라인에서 CAS를 이용하여 uc가 1 증가할 때까지 반복하는 것을 볼 수 있다. 여기서 add_use_count()는 CAS가 성공하는 경우 LFCB를 반환하고 uc가 0인 경우 nullptr를 반환하며, 이는 각각 3)의 True와 2)의 False를 의미한다. 6-7라인에서의 ret_ctr과 curr은 add_use_count()의 결과와 pred→add_use_count() 이후의 ctr을 가리키며, 8라인에서는 ret_ctr과 curr을 비교하여 LFCB의 유효성을 검사한다. add_shared_copy()는 LFCB가 유효한 경우 9라인에서 curr을 반환하지만 LFCB가 유효하지 않는

경우 수정된 ctr로 다시 시도해야한다. 따라서 11라인에서 ret_ctr이 nullptr가 아닌(pred의 uc가 증가한) 경우 12라인에서 pred→release()를 통해 증가한 uc를 감소시킨다.

여기서 release()는 LFCB의 멤버 메소드이다. 21-26라인을 살펴보면 release()는 CAS를 이용해 uc를 1 감소시키고 uc가 0이 된 경우 공유 객체의 메모리를 해제하며, 이러한 과정은 5)의 과정임을 확인할 수 있다.

### 3.2.2 LFCB 유효성 검사

이번 장에서는 LFCB가 재사용되는 상황에서 LFCB 유효성 검사의 유효성을 논증한다. 이를 위해 [Fig. 5]의 8라인 LFCB 유효성 검사에서 발생할 수 있는 모든 상황을 가정한다.

3라인에서 pred가 LFCB A를 참조하고 T1 객체를 공유한다고 가정해보자. 이때 6라인에서의 ret_ctr은 pred→add_use_count()가 성공한 경우 LFCB A, 실패한 경우 nullptr를 참조할 수 있다. 또, 7라인에서의 curr은 LFSP의 ctr이 다른 스레드에 의해 수정되지 않은 경우 LFCB A를 참조하지만, LFCB A가 재사용된 경우 다른 객체(T2)를 가리킬 수 있다. 따라서 curr이 LFCB A를 참조하는 경우 LFCB A는 T1 객체나 T2 객체를 가리킬 수 있다. 반대로 LFSP의 ctr이 다른 스레드에 의해 수정된 경우 curr은 nullptr나 T3 객체를 가리키는 LFCB B를 참조할 수 있다. 지금까지의 모든 경우의 수를 표현하면 다음과 같다.

| pred | |
|---|---|
| LFCB A | T1 |

| ret_ctr \ curr | | nullptr | LFCB A | | LFCB B |
|---|---|---|---|---|---|
| | | - | T1 | T2 | T3 |
| nullptr | - | (1) | (2) | (3) | (4) |
| LFCB A | T1 or T2 | (5) | (6) | (7) | (8) |

[Fig. 6] LFCB validation

[Fig. 6]에서 ret_ctr이 nullptr인 경우 LFSP가 사용할 수 없는 LFCB를 참조하고 있음을 의미한다. 따라서 curr이 nullptr인 (1)에서는 LFCB가 유효하며, LFSP가 다른 LFCB를 참조하는 (2), (3)과 (4)에서는 LFCB가 유효하지 않다.

ret_ctr가 LFCB A인 경우 LFCB A의 uc가 증가하였음을 의미한다. 따라서 curr이 LFCB A를 가리키는 (6)과 (7)에서는 LFCB가 유효하며, nullptr와 LFCB B를 가리키는 (5)와 (8)은 LFCB가 유효하지 않다.

이때 (6)에서는 LFCB A가 T1 객체를 가리킬 때 uc가 증가하였음을 알 수 있다. 하지만 (7)에서는 LFCB A가 T1 객체와 T2 객체 중 어떤 객체를 가리킬 때 uc가 증가했는지 알 수 없다. 그러므로 LFCB 유효성 검사의 유효성을 확인하기 위해 (7)의 상황에서 LFCB A가 어떤 객체를 가리킬 때 uc가 증가 하였는지 확인해야한다.

pred가 LFCB A를 가리키는 경우는 [Fig. 4]에서 1)의 True를 뜻하며, 이를 통해 LFCB A의 재사용은 1)과 4) 사이에서 발생했음을 알 수 있다.

다음은 LFCB A 재사용 시점에 대한 가정이다.

가정 1) LFCB A가 1)과 3) 사이에서 재사용되었다고 가정해보자. 이는 LFCB A가 재사용되어 T2 객체를 가리킨 이후에 uc가 증가했음을 의미한다. 따라서 LFCB A의 uc는 T2 객체를 가리킬 때 증가했으며, curr의 LFCB A 또한 T2 객체를 가리키므로 LFCB가 유효한 것을 확인할 수 있다.

가정 2) LFCB A가 3)과 4) 사이에서 재사용되었다고 가정해보자. 이는 T1 객체를 가리키는 LFCB A의 uc가 증가한 이후에 LFCB A가 재사용되는 것을 의미한다. 하지만 1 이상인 uc를 가지는 LFCB A는 3) 이후 재사용될 수 없다. 따라서 가정 2)는 발생하지 않는 가정임을 알 수 있다.

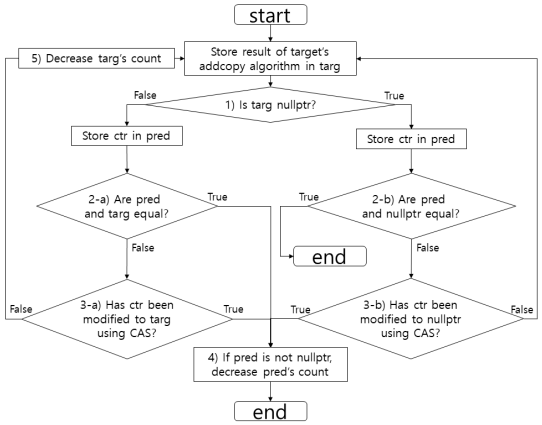
가정 1)과 가정 2)를 통해 (7)에서 LFCB A의 재사용은 1)과 3) 사이에서 발생하며, LFCB A의 uc는 T2 객체를 가리킬 때 증가하는 것을 알 수 수 있다. 따라서 ret_ctr과 curr이 재사용된 LFCB A를 참조하는 (7)에서도 LFCB가 유효한 것을 알 수 있다.

본 절을 통해 LFCB 유효성 검사는 (1), (6)과

(7) 상황에서 성공하는 것을 확인할 수 있으며, LFCB가 재사용되는 (3)과 (7) 상황에서도 LFCB 유효성 검사가 유효한 것을 확인할 수 있다.

### 3.2.3 '='operator 알고리즘

'='operator 알고리즘은 addcopy 알고리즘 이후 LFSP의 ctr을 수정하여 다른 객체를 공유하는 알고리즘이며, 객체를 공유하는 다른 LFSP(target)를 인자로 가진다.




[Fig. 7] '='operator algorithm Flow chart

[Fig. 7]은 '='operator 알고리즘의 순서도를 보여준다. 여기서 pred는 참조하고 있는 LFCB를 의미하며, targ는 target의 addcopy 알고리즘 결과를 의미한다.

'='operator 알고리즘은 1)에서 4)까지의 과정을 통해 LFSP의 ctr을 수정한다. target의 addcopy 알고리즘 이후 1)에서는 targ를 통해 참조할 LFCB의 사용 가능 여부를 확인하며, LFCB가 사용 가능한 경우 ctr은 targ를 참조하여 객체를 공유하며, 반대의 경우 nullptr를 참조한다.

2-a)와 2-b)는 알고리즘 최적화 과정으로 ctr이 이미 targ나 nullptr인 경우 ctr 수정 연산을 생략한다. 위의 과정이 불가능한 경우 ctr은 3-a)와 3-b)에서 CAS를 이용해 수정된다.

최적화가 가능한 2-a)의 True와 CAS가 성공한 3-a)와 3-b)는 ctr이 targ를 참조하는 것을 의미한다. 따라서 4)에서는 더 이상 사용하지 않는 pred의 uc를 감소시킨다. targ와 pred가 nullptr인 2-b)의 True에서는 pred가 가리키는 LFCB가 없으므로 4)의 과정을 생략할 수 있다.

3-a)와 3-b)에서의 CAS 실패는 다른 스레드에 의해 ctr이 수정되었음을 의미한다. 이때는 초기 상태로 돌아가 수정된 ctr을 이용해 알고리즘을 다시 실행하며, 1)의 성공으로 증가한 targ의 uc는 5)에서 감소시킨다.

[Fig. 8]은 '='operator 알고리즘을 이용한 LFSP::operator=()의 의사코드를 보여준다. 3라인에서는 target의 addcopy 알고리즘이 실행되며, targ를 통해 addcopy 알고리즘의 결과를 확인할 수 있다. [Fig. 8]을 살펴보면 LFCB가 사용 가능한 경우 5-15라인에서 2-a)와 3-a)를 실행하며, 반대의 경우 16-24라인에서 2-b)와 3-b)를 실행하는 것을 볼 수 있다.

```
1: function operator=(target: LFSP) : LFSP*
2:       while :
3:         targ = target→add_shared_copy()
4:         pred = ctr
5:         if targ is not nullptr :
6:           if pred is targ :
7:             pred→release()
8:             return *this
9:           else :
10:            if CAS(ctr, pred, targ) is true :
11:              pred→release()
12:              return *this
13:            else :
14:               targ→release()
15:               continue
16:       else :
17:         if pred is nullptr :
18:           return nullptr
19:         else :
20:           if CAS(ctr, pred, targ) is true :
21:             pred→release()
22:             return *this
23:           else :
24:             continue
```

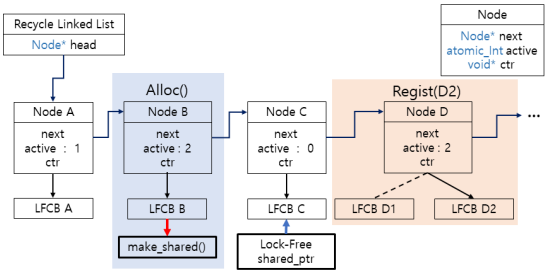
[Fig. 8] operator=() Operation in LFSP

### 3.2.4 Recycle Linked List

본 논문에서는 안전한 LFCB의 재사용을 위해 전용 메모리 재사용 관리 객체인 Recycle Linked List(RLL)를 구현해 이용하였다.

RLL은 Lock-Free 알고리즘으로 동작하며, 리스트를 구성하는 노드(Node)를 삭제하지 않고 재사용하여 내부적으로 메모리 누수와 ABA 문제가 발생하지 않도록 하였다. RLL의 멤버 메소드는 Alloc()과 Regist()가 있으며, Alloc()은 RLL에 등록된 재사용 가능한 LFCB를 반환하고 Regist()는 재사용할 LFCB를 RLL에 등록한다.

RLL을 구성하는 노드는 3가지 변수를 가지며 리스트를 연결하는 next 포인터, 재사용할 LFCB를 참조하는 ctr 포인터와 노드의 상태를 나타내는 active 변수가 있다. active는 0, 1과 2의 값을 가질 수 있으며, 각각 재사용 가능(활성), 재사용 불가(비활성)와 수정을 의미한다.




[Fig. 9] Recycle Linked List

[Fig. 9]는 재사용 가능한 LFCB A를 참조하는 Node A(활성 노드), 이미 재사용한 LFCB C를 참조하는 Node C(비활성 노드)와 Alloc()/Regist()에서 사용 중인 Node B와 Node D(수정 노드)를 보여준다. 수정 노드는 Alloc()과 Regist()를 실행 중인 스레드에서만 이용될 수 있으며, 이외의 다른 스레드에서는 이용될 수 없다.

다음 [Fig. 10]은 RLL의 Alloc()과 Regist()의 의사코드를 보여준다.

```
1: method Alloc(): void*
2:     curr = head→next
3:     while curr is not nullptr :
4:       if curr→active = 1 :
5:         if CAS(curr→active, 1, 2) is true :
6:           ret_ctr = curr→ctr
7:           curr→active = 0
8:           ret ret_ctr
9:         curr = curr→next
10:    ret nullptr
11: method Regist(p : void*)
12:    pred = head
13:    while :
14:      if pred→next is nullptr :
15:        break
16:      curr = pred→next
17:      if curr→active is 0 :
18:        if CAS(curr→active, 0, 2) is true :
19:          curr→ctr = p
20:          curr→active = 1
21:          ret
22:        pred = pred→next
23:    n = new Node(p)
23:    while :
24:      if pred→next is nullptr :
25:        if CAS(pred→next, nullptr, n) is true :
26:          ret
27:      pred = pred→next
```

[Fig. 10] Alloc()/Regist() Operations in RLL

Alloc()은 연결리스트에 등록된 재사용 가능한 LFCB를 반환하고 재사용 가능한 LFCB가 없는 경우 nullptr를 반환한다. 재사용 가능한 LFCB를 검색하기 위해 3-9라인과 같이 연결리스트를 순회하며, CAS를 이용해 활성 노드를 수정 노드로 변경한다. 노드에 등록된 재사용 가능한 LFCB는 5-8라인에서 지역변수 포인터를 이용해 안전하게 반환되며, 수정 노드를 재사용하기 위해 반환하기 전 비활성 노드로 변경한다.

13-22라인은 Regist()가 비활성 노드를 검색하여 수정 노드로 변경하고, LFCB를 등록한 후 활성 노드로 변경하는 것을 보여준다. 연결리스트를 순회하며 비활성 노드를 검색하지 못하여 재사용할 LFCB를 등록하지 못하는 경우 메모리 누수 문제가 발생한다. 따라서 이를 방지하기 위해 LFCB를 등록한 새로운 노드를 생성하여 연결리스트의 말단에 삽입해야한다. 이때 연결리스트의 말단에 노드를 삽입하는 작업은 다른 스레드와 경쟁을 야기할 수 있으므로 23-26라인과 같이 CAS를 이용해 삽입한다. 여기서 CAS의 실패는 다른 스레드가 새로운 노드를 삽입했음을 의미한다. 그러므로 27라인과 같이 말단 노드를 계속 검색하여 삽입한다.

## 4. 실험

실험에 앞서 std::weak_ptr와 Lock-Free weak_ptr의 구현의 차이는 std::shared_ptr와 Lock_free shared_ptr와 동일하므로 실험에서는 std::shared_ptr와 Lock_free shared_ptr만 비교하였다.

실험은 intel® coreTM i7-8700K 3.7GHz, 16G RAM 환경에서 진행하였으며, JAVA로 구현된 참조문헌[7]의 게으른 동기화 연결 리스트(laZy Synchronization Linked list: ZSL)를 이용하였다.

메모리가 자동으로 관리되지 않는 언어인 C++에서 일반 포인터를 이용하여 ZSL을 구현하는 경우 메모리 누수 문제를 가진다. 실험에서는 이러한 문제를 해결하기 위해 메모리 관리 클래스인 shared_ptr를 이용하여 ZSL을 구현하였으며, 실험에 이용된 ZSL은 다음과 같다.

[Table 1] Type of ZSL in the experiment

| ZSL | pointer |
|---|---|
| PZSL | base pointer |
| SPZSL | C++11 shared_ptr |
| ATSPZSL | C++11 shared_ptr using std::atomic template |
| LFSPZSL | Lock-Free shared_ptr |

이와 같이 ZSL을 구성하는 노드 자료구조의 next 포인터를 4가지 종류의 포인터 타입으로 구현하여 각 ZSL의 성능을 측정하였다. 여기서 메모리 누수의 문제를 가지는 PZSL은 성능 비교에서 제외하였으며, 이를 통해 메모리를 관리하는 shared_ptr 클래스의 필요성을 알 수 있다.

다음은 ZSL의 성능 측정 방법이다.

1) 0에서 L까지의 무작위 값을 선택한 후 노드 삽입/삭제/검색 중 하나의 메소드를 실행한다. 이때 L은 ZSL의 최대 길이이며, 각 메소드의 실행 확률은 모두 1/3이다.

2) 1)의 동작을 1,000,000번 실행한다.

3) 위의 과정을 스레드의 수를 늘려가며 실행하고, 각 ZSL의 실행 소요 시간을 측정한다.

실험은 실행 소요 시간을 5번씩 측정하여 초당 평균 메소드 실행 횟수(operations per seconds: ops)를 계산해 비교하였다.

실험의 구성은 [Table 2]에서 확인할 수 있다.

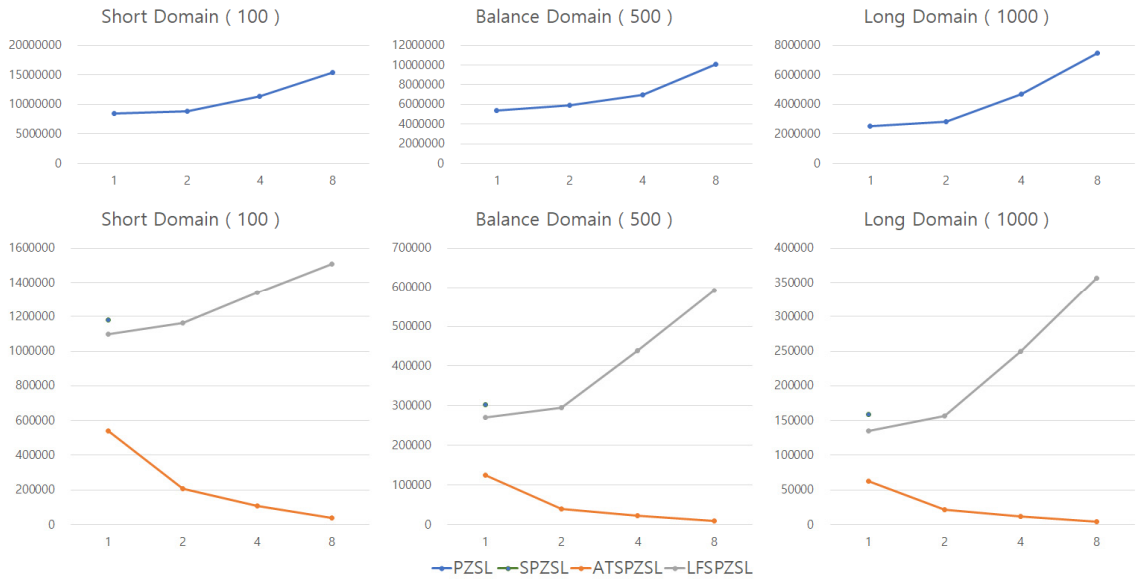
각 ZSL에서 동일한 노드에 접근하는 스레드 사이의 경쟁은 스레드 수와 비례하게 높아지며, ZSL의 최대 길이와는 반비례하게 낮아진다. 따라서 ZSL의 최대 길이가 가장 짧은 Short Domain은 동일한 노드에 접근하는 스레드 사이의 경쟁이 높은 상황, ZSL의 최대 길이가 가장 긴 Long Domain은 스레드의 경쟁이 낮은 상황에서의 성능을 측정할 때 사용할 수 있다. Balance Domain은 Short/Long Domain과 함께 비교하여 ZSL을 구성하는 각 포인터가 성능에 얼마나 영향을 미치는지 알 수 있다.

[Table 2] Configuration of the experiment

| | | |
|---|---|---|
| Number of thread | 1 | |
| | 2 | |
| | 4 | |
| | 8 | |
| Maximum length of ZSL (L) | Short Domain | 100 |
| | Balance Domain | 500 |
| | Long Domain | 1000 |

[Fig. 11]은 실험 결과를 보여준다. PZSL은 모든 상황에서 다른 ZSL들보다 가장 높은 ops를 보이지만 메모리를 관리하지 못하는 문제를 가지고 있다. 또, PZSL을 제외한 ZSL 중 스레드가 1개일 때 가장 높은 ops를 보이는 SPZSL은 C++11 shared_ptr를 이용해 메모리를 관리할 수 있지만 데이터 레이스로 인해 멀티스레드에서 동작하지 않는 문제를 가진다.

SPZSL의 데이터 레이스를 해결하기 위해 std::atomic을 이용한 ATSPZSL은 멀티스레드에서 정상적으로 동작하며, 메모리도 관리할 수 있다. 하지만 경쟁이 가장 적은 Long Domain에서 스레드 수에 반비례하게 ops가 낮아지며(63K, 22K, 11K, 4.7K), 이는 Short Domain(540K, 206K,

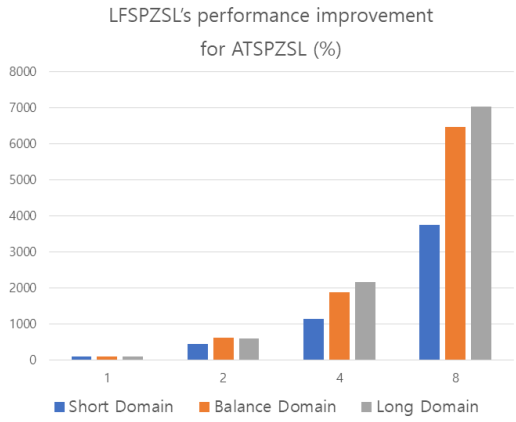



[Fig. 11] Result of the experiment

107K, 39K)과 Balance Domain(125K, 41K, 22K, 9.3K)에서도 발생하는 것을 볼 수 있다. 이를 통해 스레드 수가 많아질수록 잠금으로 인한 블로킹이 자주 발생하여 병렬성이 떨어지는 ATSPZSL의 특징이 멀티스레드 프로그램에서 성능을 악화시키는 것을 확인할 수 있다.

반면 LFSPZSL은 Long Domain에서 스레드 수에 비례하게 ops가 급격하게 증가하며(135K, 157K, 250K, 357K), Balance Domain(270K, 296K, 439K, 591K)과 경쟁이 높은 Short Domain(1097k, 1162k, 1340k, 1508k)에서도 ops가 완만하게 증가하는 것을 확인할 수 있다. 이를 통해 블로킹이 발생하지 않는 Lock-Free shared_ptr를 사용한 LFSPZSL가 ATSPZSL보다 더 높은 병렬성을 가지는 것을 확인할 수 있다.

[Fig. 12]는 ATSPZSL에 대한 LFSPZSL의 성능 향상을 보여준다. LFSPZSL은 스레드 사이의 경쟁이 낮은 Long Domain에서 ATSPZSL보다 최대 7424% 만큼의 성능을 향상시킬 수 있었고, 경쟁이 높은 Short Domain에서도 성능을 최대 3767% 만큼 향상시킬 수 있었다.




[Fig. 12] LFSPZSL's performance improvement for ATSPZSL

이를 통해 멀티스레드 프로그램의 여러 경쟁 상황에서 논블로킹 알고리즘으로 동작하는 Lock-Free shared_ptr가 스레드 수에 비례하게 성능이 향상되는 것을 볼 수 있었고, 블로킹 알고리즘으로 동작하는 std::atomic을 이용한 C++11 shared_ptr보다 높은 성능을 보이는 것을 확인할 수 있었다.

## 5. 결론 및 향후연구

본 논문은 C++11에서 std::atomic 템플릿을 이용한 std::shared_ptr와 std::weak_ptr보다 높은 성능을 가지는 Lock-Free shared_ptr와 weak_ptr를 제안하였고, 핵심 구현 알고리즘들을 논증하였다. Lock-Free shared_ptr와 weak_ptr는 기존 객체들의 데이터 레이스를 Lock-Free 알고리즘으로 해결하였고, 실험 결과 경쟁 심한 상황에서도 기존의 방법보다 성능이 향상되는 것을 확인하였다. 이를 통해 Lock-Free shared_ptr와 weak_ptr가 멀티스레드로 구현된 대용량 다중 접속 게임서버와 고성능 3D 게임엔진에 활용되어 프로그램의 성능을 높일 수 있을 것이라 기대한다.

향후 연구방향은 Lock-Free shared_ptr와 weak_ptr를 활용하여 논블로킹 멀티스레드 환경의 게임에 필요한 여러 자료구조를 제작하고 이를 활용하여 게임의 성능을 개선시키는 것이다.

## REFERENCES


[1] Steam Hardware&Software Survey: July 2020, https://store.steampowered.com/hwsurvey/cpus/

[2] INCITS/ISO/IEC, "Information technology-Programming-C++", New York: American National Standards Institute, INCITS/ISO/IEC 14882-2011[2012], pp. 540-565, 2012.

[3] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. "Eraser: A dynamic data race detector for multithreaded programs", ACM Transactions on Computer Systems, 15(4):391-411, 1997.

[4] M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects", IEEE Transactions on Parallel and Distributed Systems, 15 (6): 491 - 504. 2004.

[5] H. Wen, J. Izraelevitz, W. Cai, H. A. Beadle and M. L. Scott. "Interval-based memory reclamation". ACM SIGPLAN Notices. 2018.

[6] Trevor Alexander Brown, "Reclaiming Memory for lock-free Data Structures: There has to be a Better Way", PODC '15: Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, July 2015, https://doi.org/10.1145/2767386.2767436

[7] M. Herlihy and N. Shavit. "The Art of Multiprocessor Programming Revised Reprint", Morgan Kaufmann, 2012.

[8] M. Herlihy. "Wait-Free Synchronization". ACM Transactions on Programming anguages and Systems (TOPLAS), 13 (1): 124-149, Jan. 1991.

[9] D. Dechev, P. Pirkelbauer and B. Stroustrup. "Understanding and Effectively Preventing the ABA Problem in Descriptor-based Lock-free Designs". 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, 2010.

[10] Bjarne Stroustrup, "The C++ Programming Language (Fourth Edition)", Addison-Wesley, pp. 990-995, 2013



구 태 균 (Ku, Tae kyun)

약 력 : 2014-현재
한국산업기술대학교 게임공학과 학사과정

관심분야 : 병렬처리, 온라인 게임서버

정 내 훈 (Jung, NaiHoon)

약 력 : 2002 KAIST, 전산학박사
2002-2008, NCSoft, MMORPG 프로그램팀장
2008-현재 한국산업기술대학교 게임공학부 교수

관심분야 : 병렬처리, 컴퓨터 구조, 대용량 온라인 게임서버