Journal of Korea Game Society 2020 Dec; 20(6): 13-22
http://dx.doi.org/10.7583/JKGS.2020.20.6.13

게임 프로그래밍

RIO와 HTM을 이용한 MMO 게임서버의 성능 개선

강수빈*, 정내훈**
한국산업기술대학교 디지털엔터테인먼트학과*, 한국산업기술대학교 게임공학부**
{chochang01, nhjung}@kpu.ac.kr

Performance Improvement of MMO Gameservers Using RIO and HTM

Subin Kang*, NaiHoon Jung**
Dept. of Digital Entertainment, Korea Polytechnic Univ.*,
Dept. of Game & Multimedia Engineering, Korea Polytechnic Univ.**

요 약

RIO는 윈도우의 최신 네트워크 API로 낮은 부하와 지연을 통해 높은 IO 성능을 발휘하도록 설계되었으며. 고성능의 네트워크 IO를 요구하는 대규모 동시접속(MMO) 게임 서버에 적합할 것으로 기대된다. 또한 HTM은 기존의 멀티스레드 동기화 방식보다 생산성과 성능이 우수하여 MMO 게임 서버에 적용 시 성능향상이 예상된다. 본 논문에서는 MMO 게임 서버에 RIO를 적용함과 동시에 RIO의 성능을 최대한 끌어내도록 구조를 개선하고, 기존의 시야 처리 알고리즘을 HTM 방식으로 변경하여 서버의 성능을 향상시켰다. 결과적으로 동시 접속자 수를 19%가량 증가시켰으며, 벤치마킹 프로그램을 사용하여 이를 검증하였다.

ABSTRACT

RIO is a new network API for Windows that is designed to have high I/O performance through low overhead and latency. Using RIO, MMO game servers may have much performance benefits. In addition, HTM has better productivity and performance compared to existing synchronization methods, so adopting it may produce better performance, also. In this paper, we improved server performance by implementing a new MMO game server architecture optimized with RIO and HTM. The performance of the server was verified through a benchmark program, and the number of concurrent users increased by 19%.

Keywords : Game Server(게임서버), IOCP, Registered I/O, Hardware Transactional Memory

Received: Sep. 20. 2020. Revised: Nov. 03. 2020.
Accepted: Nov. 05. 2020.
Corresponding Author: NaiHoon Jung(Korea Polytechnic Univ)
E-mail: nhjung@kpu.ac.kr

ISSN: 1598-4540 / eISSN: 2287-8211

1. 서 론

대규모 멀티플레이어 온라인(MMO) 게임은 많은 수의 플레이어가 하나의 월드에 접속하여 서로 상호작용하는 특징이 있다. 이러한 게임의 서버는 플레이어의 원활한 게임 경험을 유지해 주기 위하여 일정한 응답시간을 보장해 주어야한다. 이를 위해서는 멀티스레딩을 통해 다중 CPU와 다중 코어의 컴퓨팅 파워를 적극 활용함과 동시에 운영체제에서 별도로 제공하는 고성능 네트워크 I/O API를 사용하여야 한다.

우리나라의 경우 MMO게임 서버는 대부분 Windows에서 C++로 구현되고 있으며, 일반적으로 MICROSOFT사에서 개발한 멀티스레드 비동기 I/O처리 API인 IOCP(Input Output Completion Port)를 사용하여 고성능을 유지한다[1]. IOCP는 커널레벨에서 구현되어 있으며 네트워크 I/O의 요청 완료 시 커널에서 스레드 풀에서 대기 중인 스레드에게 완료를 통지하는 방식으로 대용량 메시지를 병렬적으로 처리한다.

새로 추가된 통신 API인 Registered I/O[2]의 경우 사용할 공유 자료구조를 커널에 미리 등록하여 IOCP에 비해 적은 시스템 호출로 I/O를 수행할 수 있으며, I/O 요청을 모아서 수행하는 기능을 지원하여 시스템 호출을 추가적으로 경감시킬 수 있는 기능이 있다.

멀티스레드로 작성된 프로그램들은 스레드 사이의 동기화가 가장 큰 문제이고 이를 공유 자료구조를 통해 해결한다. 이 때 성능을 위해 잠금 방식의 자료구조 보다 Lock-free방식의 자료구조를 선호하게 된다. 하지만 Lock-free자료구조의 난해함으로 인한 생산성 문제가 제기되며 HTM은 이를 해결해 주는 최신 기법이다[3].

본 연구에서는 RIO의 효율적인 적용을 위해 기존의 IOCP 게임서버의 클라이언트와 작업자 스레드의 할당 구조를 변경하였으며, 이를 통해 RIO API에 사용되는 공유 자료구조에 접근 시 유발되는 데이터 레이스 문제를 해결할 수 있었다. 또한 빈번하게 사용하는 공유 자료구조이나 Lock-free구현이 어려워 임계영역을 잠금 방식으로 보호하던 시야 처리 자료구조를 HTM를 이용하여 임계영역을 보호하도록 구현하여 높은 병렬성을 얻도록 하였다. 이를 통해 궁극적으로 동시 접속자수를 극대화할 수 있는 MMO 게임서버를 구현하였다.

본 논문의 구성은 다음과 같다. 2장에서는 IOCP, Registered I/O, HTM와 시야처리에 대한 관련 연구에 대하여 소개한다. 3장에서는 기준이 되는 서버에 RIO를 적용하는 방법과 최적화기법, HTM으로 구현한 알고리즘에 대하여 기술한다. 4장에서는 구현한 서버의 스트레스 테스트를 통해 수치화 된 성능을 비교하고, 5장에서는 결론 및 향후 연구 과제를 제시한다.

2. 관련연구

2.1 IOCP

IOCP[4]는 커널레벨로 구현된 멀티스레딩 비동기 I/O처리 API이다. 멀티스레딩의 효율성을 높일 수 있도록 스레드 풀을 미리 구성하고 대기중인 스레드에게 I/O의 완료통지를 넘겨주어 I/O를 병렬처리하도록 구현되어 있다.

유저는 완료를 넘겨줄 컴플리션 포트(Completion Port)를 미리 만들고 소켓과 연결하여 해당 소켓의 I/O 완료가 커널단의 컴플리션 큐 (Completion Queue)에 저장되도록 한다. 작업자 스레드들은 GetQueuedCompletionPort()함수를 호출하여 자신을 스레드 풀에 등록하며, 처리할 I/O완료 작업이 있는 경우 대기 스레드 큐에서 깨어나 컴플리션 큐에 있는 I/O 완료작업을 받아 수행한다.

IOCP는 위와 같이 기본적으로 멀티스레드 프로그래밍을 전제로 하고 있기 때문에 I/O완료 작업 수행 시에 공유 자료구조에 대한 데이터 레이스가 매우 일어나기 쉽고 이를 각별히 유의하여 사용해야한다. 또한, 오래전에 설계된 API이기

때문에 커널과의 통신을 전적으로 시스템 호출에 의존해야 한다는 단점이 있다.

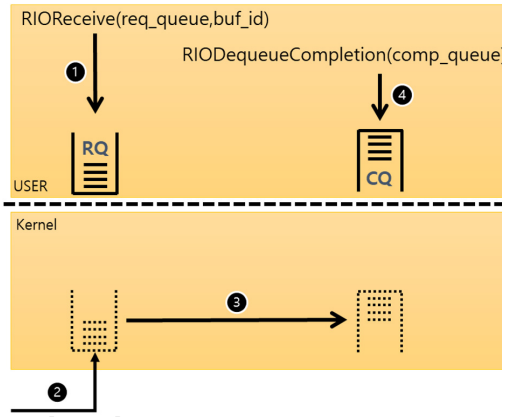
IOCP를 사용한 서버에 관한 연구는 다수 진행된 바 있으며[5,6,7,8], NDC2012에서는 IOCP를 사용한 기본적인 서버의 구조를 설명하고, 멀티스레드를 활용하면서 생기는 동기화에 관한 문제들을 함께 언급하였다. 추가적으로 이러한 동기화를 해결하기 위한 Lockless 방식의 서버 구조를 제안하였다[5]. 또한 김기남 외 1명이 진행한 연구에서는 단일 서버로 인한 부하 처리의 한계점을 인지하고 이를 개선하기 위하여 IOCP를 사용한 서버와 부하분산을 위한 로드밸런스 모듈을 적용하였다. 또한 이를 직접 제작한 테스트용 더미 클라이언트를 활용하여 서버의 부하 경감을 수치적으로도 증명하였다[6].

2.2 Registered I/O

Registered I/O(RIO)는 Winsock의 확장 API로 Windows Server 2012부터 추가되었다[2]. RIO API는 낮은 레이턴시와 적은 지터(jitter)로 고성능이 필요한 어플리케이션에서 높은 초당 I/O 처리률 (IOPS)을 달성할 수 있도록 설계되었다.

RIO를 사용하기 위해서는 I/O에 사용하는 버퍼와 I/O요청을 저장하는 RQ(Request Queue), 완료를 저장하는 CQ(Completion Queue)를 미리 할당해야 한다. 버퍼의 경우 유저가 별도로 버퍼를 할당하여 이를 커널에 등록하는 과정에서 RIO API에 사용할 ID를 부여받으며, RQ와 CQ는 할당과 동시에 커널에 등록된다. API에서 사용하는 자료구조를 모두 미리 할당하고 커널에 등록하게 되어 있으므로 유저모드와 커널모드 양쪽에서 동일한 자료구조에 접근이 가능해진다.

[Fig. 1]은 RIO를 사용한 통신이 어떻게 수행되는지 RIOReceive함수를 예시로 하여 간략한 도식으로 나타낸 것이다. RIO의 데이터 송신과 수신 작업 수행은 미리 지정해둔 버퍼에서 이루어진다.



[Fig. 1] RIO Communication Diagram

1) I/O 작업 요청은 소켓에 연결된 RQ에 저장된다.

2) 외부로부터 해당 소켓을 통해 데이터가 들어오면 저장된 I/O요청이 있는지 확인한다.

3) 일치하는 경우 I/O완료를 CQ에 저장한다.

4) 각 스레드는 지정된 CQ에서 완료통지를 꺼내어 처리하며 연관된 버퍼의 ID를 통해 데이터에 접근이 가능하다.

이를 통해 완료통지의 폴링을 별도의 시스템 호출 없이 유저모드에서 구현할 수 있는 장점이 있다. 하지만 멀티스레드를 적용한 프로그램에서는 자료구조 자체가 스레드 안전하지 않기 때문에 RQ와 CQ에 접근할 때 적절한 동기화 처리기능을 추가로 구현해 주어야 한다.

2.3 Hardware Transactional Memory

트랜잭셔널 메모리[3]는 잠금 기반의 프로그래밍 모델을 유지하면서 논 블로킹(Non-blocking)으로 병렬 수행이 가능하도록 하여 임계영역의 처리 성능을 높이는 것을 목적으로 한 병렬처리 기법이다. 익숙한 잠금 모델을 기반으로 하기 때문에 Lock-free방식에 비해 구현 난이도가 낮아 알고리즘의 생산성이 높은 것이 가장 큰 장점이

다. HTM은 이를 HW로 구현한 것으로 소프트웨어로 구현하는 방식에 비해 현저한 성능상의 우위를 가진다[9].

트랜잭셔널 메모리는 공유 메모리의 임계영역을 트랜잭션으로 취급하며 트랜잭션 영역의 실행이 완료되면 트랜잭션 대상인 공유 메모리를 다른 스레드가 변경했는지 검사하여 충돌이 없는 경우 결과물이 반영(Commit)되고 충돌이 있다면 결과물을 모두 취소(Abort)하여 동기화를 구현한다. Abort되는 경우에는 트랜잭션을 재시도하여야 하며, 이때 특정 상황에서 트랜잭션이 무한히 재시도 될 수 있다. 이를 해결하기 위해 Fall back path를 설정하며, 재시도 횟수가 일정 값을 넘어가면 Fall back path를 통해 무조건 성공시킬 수 있도록 해주어야 한다.

Intel사는 4세대 프로세서부터 HTM을 사용할 수 있도록 Transactional Synchronization Extensions(TSX)[3]라는 추가 확장 기능을 제공하고 있으며 본 논문에서는 이 API를 활용하였다.

Marios Kardaras외 4인은 Skiplist 자료구조에 HTM을 적용하고 트랜잭션 범위를 최적화 하는 연구를 진행하였다. Skiplist의 메서드 실행 시 메서드 전체를 하나의 트랜잭션으로 감싸는 Coarse-grained HTM 버전과 실제 메모리 변경부분만을 트랜잭션으로 감싸는 Fine-grained HTM 버전등으로 나누어 Lock-free, Lock-based, 이상치(Ideal)등과 비교분석하였으며 Fine-grained HTM버전의 경우 성능이 Lock-Free버전과 큰 차이를 보이지 않음을 보였다[10].

2.4 시야처리

MMO 게임서버는 하나의 월드에 접속한 매우 많은 수의 플레이어의 동기화를 지원해야 한다. 이때 접속한 플레이어의 수를 N이라 하면 유저 전체에게 전송 시 필요한 동기화 메시지 전송 회수는 $O(N^2)$에 달한다. 이 횟수를 줄이기 위해 게임 서버는 시야처리[1][7][11] 기법을 사용하며, 결과적으로 메시지 전송 회수를 $O(N^2)$에서 $O(K\times N)$ (K는 시야처리 영역에 포함되는 플레이어의 수)으로 최적화할 수 있다.

시야처리는 게임서버의 부하를 경감시키는 대표적인 방법으로 클라이언트의 정보를 모두에게 전송하는 것이 아닌 주위 일정 범위 이내에 존재하는 클라이언트들에게만 보내게 하는 방식이다[12,13].

조슈아 글레이저, 산제이 마드히브가 저술한 “멀티플레이어 게임 프로그래밍”에서는 대규모 접속 서버에서 대역폭과 지연시간의 문제로 오브젝트 전체를 리플리케이션하는 것이 한계점이 있다는 것을 지적하였으며, 이를 최적화하기 위하여 서버상의 클라이언트 시야 절두체를 활용한 가시성기법들을 설명하였다[11]. 또한 이러한 가시성 기법을 활용함에 있어 시야의 범위와 공간분할의 크기에 따른 동시 접속자 수에 관한 연구가 이루어진 바 있으며, 시야를 공간 분할 크기의 1.5배로 설정하였을 때 최적의 성능을 얻을 수 있는 것을 벤치마크를 통하여 증명하였다[7].

3. RIO, HTM을 적용한 게임서버 성능향상

본 논문에서 설계한 게임서버는 RIO의 효율적인 사용을 위해 전용 스레드를 사용한 잠금 제거 방식을 도입하였고, RIO의 deferred I/O를 활용하기 위해 브로드캐스트 시스템을 개선하였다. 또한 기존의 시야처리에 사용되던 동기화 알고리즘의 구현 방식을 HTM으로 변경하였다.

3.1 전용 스레드를 사용한 잠금 제거

RIO API와 RQ (Request Queue), CQ (Completion Queue)등의 자료구조는 스레드 안전하지 않기 때문에 사용 시 각별히 주의를 요하도록 되어있다. 잠금을 이용하여 임계영역을 보호하는 것이 일반적이나, 잠금으로 인한 성능 저하가 클 수 있기 때문에 본 논문에서는 잠금을 제거하

도록 구조를 최적화하였다.

I/O완료 통지를 받는 CQ를 각 스레드마다 한 개씩 할당하고, RIODequeueCompletion 메서드 호출 시 자신의 스레드에 할당된 CQ만 사용하도록 하여 데이터 레이스를 제거하였다. 이를 위해 각 클라이언트들은 여러 스레드 중 하나에 고정되며 그 클라이언트가 사용하는 CQ는 할당된 스레드에서만 접근 가능하다.

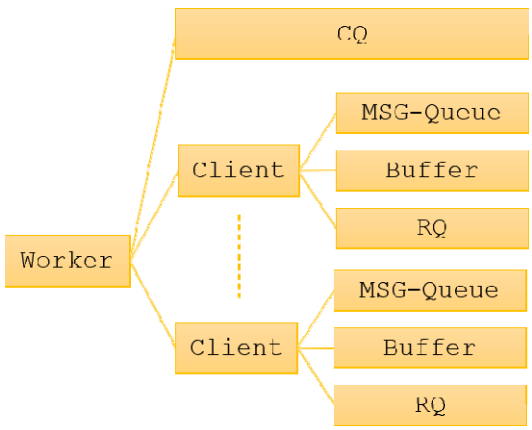
RQ는 조금 더 복잡하다. 클라이언트의 I/O의 동작 결과를 상대 클라이언트에 전송하려면 상대의 RQ에 접근해야 하는데 이때 데이터 레이스가 발생할 수 있고, 기존에는 이를 잠금을 사용하여 해결하였다. 이를 피하기 위해 각 RQ의 접근을 클라이언트가 고정된 스레드에서만 하도록 하였고, 작업을 저장하는 메시지 큐를 클라이언트마다 두어 이를 통해서만 RQ의 접근을 하도록 하였다. 다른 클라이언트의 RQ접근이 필요한 경우 이 요청을 해당 클라이언트에 메시지 큐에 남기도록 하였으며, 이후 각 스레드가 I/O작업을 처리할 때 자신에 고정된 클라이언트들을 순회하며 저장된 요청을 RQ에 넘기도록 하였다.

결과적으로 메시지 전송시마다 다른 스레드와 중복되는 RQ접근을 없애고 Lock-Free Set[14]을 활용하여 구현한 메시지 큐에 대신 접근하도록 하여 RQ의 잠금을 제거하였다.

3.2 Deferred I/O

다른 I/O API와 동일하게 RIO의 Send역시 시스템 호출을 유발한다. 하지만 RIO는 Send API호출 시 시스템 호출을 하지 않고 요청을 RQ에 모아 놓기만 하는 RIO_MSG_DEFER 플래그를 제공한다. 사용자는 RQ에 모인 요청을 RIO_MSG_COMMIT_ONLY 플래그를 통해 하나의 시스템 호출로 모두 전송할 수 있고, 이를 통해 여러 개의 버퍼를 하나의 시스템 호출로 전송하는 Scatter/Gather I/O[15] 효과를 내는 것으로 호출 횟수를 줄일 수 있다.

본 논문에서는 이를 활용하기 위해 3.1에서 언급한 메시지 큐를 사용하였다. Scatter/Gather I/O효과를 최대한 이끌어 낼 수 있도록 전체 메시지 Commit 이후 다음 메시지 전송 사이에 20ms의 시간 간격을 세팅하여 RQ에 1개 이상의 메시지가 모이도록 하였다. 이때 시간 간격은 4.1절의 스트레스 테스트 클라이언트를 이용한 벤치마크를 사용하였으며 전송 시간간격을 0ms에서 100ms까지 10ms간격으로 변경하며 테스트하였다. 최종적으로 최대 동시 접속자 수치가 가장 높은 결과인 20ms를 시간간격으로 선정하였다.



[Fig. 2] Worker Thread Data Structure Tree

[Fig. 2]는 3.1과 3.2에서 서술한 최적화 기법을 모두 적용하였을 때 하나의 스레드가 처리하는 자료구조들을 구조화한 그림이다.

3.3 HTM을 활용한 병렬성 증가

본 논문의 기준이 되는 서버는 시야처리를 사용하여 메시지 전송의 부하를 경감시켰다. 이때 시야에 들어온 클라이언트들의 동기화시에 어긋남을 막기 위하여 Viewlist라는 자료구조를 사용하였고 이는 unordered_set으로 구현되어 있다. 이 Viewlist는 모든 스레드에서 접근하여 갱신 할 수 있어야 하기 때문에 데이터 레이스에 매우 취약하며, 이를 스레드 안전하게 접근 및 수정하기 위하여 잠금을

통해 데이터 레이스를 해결하였다. Viewlist는 add(), remove(), search(), copy()의 4개의 메소드를 필요로 한다. 여기서 add(), remove(), search()의 경우 Lock-free로 구현된 자료구조가 많이 있으나, copy()까지 구현된 자료구조는 찾을 수 없었다. 이는 copy() 메서드의 경우 한번에 많은 객체에 접근하므로 Lock-free구현이 어렵기 때문이라고 추측할 수 있다.

본 논문에서는 Intel사의 HTM구현인 TSX를 사용하여 잠금을 사용하지 않는 알고리즘을 구현하였고 이를 이용하여 기존의 잠금 기반 프로그램의 구조를 크게 변경하지 않으면서 병렬성을 확보하여 성능상의 이점을 가져올 수 있었다.

HTM을 적용할 때 무한히 재시도하는 것을 막을 수 있는 Fall Back Path의 구현이 필요한데, 본 논문에서는 이를 잠금을 사용하여 구현하였으며 HTM과의 동기화를 위해 abortcnt와 isLocked라는 상태 변수를 추가하였다.

```
1  Start(mutex lock) {
2      int abortcnt = 0;
3      while(true) {
4          while(isLocked);
5
6          if (abortcnt > MAX_ABORT_CNT) {
7              isLocked = true;
8              lock.lock();
9              return;
10         }
11
12         if (_XBEGIN_STARTED == _xbegin()) {
13             if (isLocked) _xabort();
14             return;
15         } else {
16             abortcnt++;
17         }
18     }
19 }
```

[Fig. 3] HTM Start Method

트랜잭션은 영역의 시작과 끝을 나타내는 Start, End 로 구성된다. Start 메서드([Fig. 3])는 트랜잭션 실패 시 재시도한 회수를 저장할 abortcnt 변수 (2 line)를 가진다. abortcnt가 MAX_ABORT_CNT를 넘기 전까지 (6 line) 트랜잭션을 계속 시도 (12 line)하며 실패한 경우 abortcnt를 증가시킨다. 만일 트랜잭션이 성공했으나 다른 트랜잭션이 Fallback Path에 들어간 경우 (13 line) 트랜잭션을 abort한다. Abortcnt가 MAX_ABORT_CNT를 넘을 때까지 트랜잭션을 실패한 경우 인자로 받은 lock을 걸고 Fallback Path를 실행(6-10 line)하여 강제로 진행하게 하였다.

```
1  End(mutex lock) {
2      if (true == _xtest()) {
3          if (true == isLocked)
4              _xabort();
5          else
6              _xend();
7      }
8
9      else {
10         lock.unlock();
11         isLocked = false;
12     }
13 }
```

[Fig. 4] HTM End Method

트랜잭션 영역의 끝에서 불리는 End([Fig. 4])는 조금 더 간단하다. 트랜잭션을 수행 중인지 검사(2 line)하고, 만약 트랜잭션 중에 Fallback Path로 진행된 경우 abort, 문제가 없다면 그대로 종료한다. 트랜잭션이 수행 중이지 않은 경우(9-12 line)는 Fallback Path로 진행되었기 때문에 역시 인자로 받은 lock을 해제하여 임계영역 보호를 종료한다.

```
1 Client->HTMStart(Client->Viewlist_lock);
2 old_viewlist = Client->curr_viewlist;
3 Client->HTMEnd(Client->Viewlist_lock);
4
5 viewlist_process(old_viewlist);
```

[Fig. 5] Viewlist copy with HTM

[Fig. 5]는 실제로 Start와 End메서드를 사용하여 Viewlist의 복사를 수행하는 코드이다. 일반적인 잠금과 동일한 방식으로 사용하며 Viewlist의 복사등과 같이 데이터 레이스를 유발하기 쉬운 작업을 스레드 안전하게 처리할 수 있다.

4. 서버 성능 평가

본 논문에서 제시하는 Registered I/O와 HTM을 적용한 서버와 기존의 IOCP로 구현된 서버와의 성능차이를 측정하기 위하여 벤치마크를 수행한다. 서버의 구동에 사용된 머신의 성능은 [Table 1]과 같다.

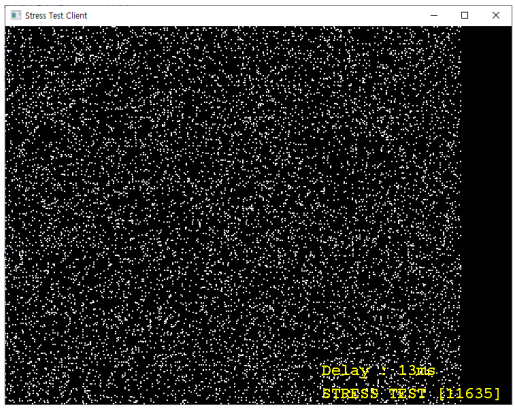
[Table. 1] Hardware Specification

CPU	Intel Xeon Gold 6240@2.6GHz
RAM	256GB
OS	Windows Server 2019

4.1 스트레스 테스트 클라이언트

스트레스 테스트 클라이언트는 서버에 부하를 주어 최대 동시 접속자를 테스트 하는 목적으로 직접 제작하여 사용하였다. 스트레스 테스트 클라이언트는 하나의 프로세스에서 서버와 최대한 많은 접속을 유지하도록 설계하였다. 최초 지정한 10ms마다 서버에 접속시도를 하고 접속에 성공한 클라이언트는 별도의 작업자 스레드에서 서버와 통신을 지속한다. 접속에 성공한 클라이언트는 매 초마다 서버에게 랜덤한 이동방향 메시지를 송신하고 해당 메시지가 처리된 메시지를 서버로부터 수령하여 화면에 출력하도록 한다. 이때, 서버로 전송하는 메시지마다 고유한 번호를 부여하여 서버의 반응까지 걸리는 시간을 측정한다.

서버의 반응이 100ms가 넘는 메시지가 도착하면 접속시도 간격을 점차적으로 증가시키고 평균 반응시간이 100ms가 넘는 경우 접속시도를 중지하고 서버에게 접속종료를 요청한다. 서버의 반응시간을 100ms이하가 될 때 까지 클라이언트의 접속을 하나씩 끊는 것을 반복하며 접속자수가 일정 수치에 수렴하면 해당 수치를 최대 동시 접속자 수치로 판단한다. [Fig. 7]은 스트레스 테스트 클라이언트가 구동중인 화면이다.



[Fig. 7] Benchmark Program Image

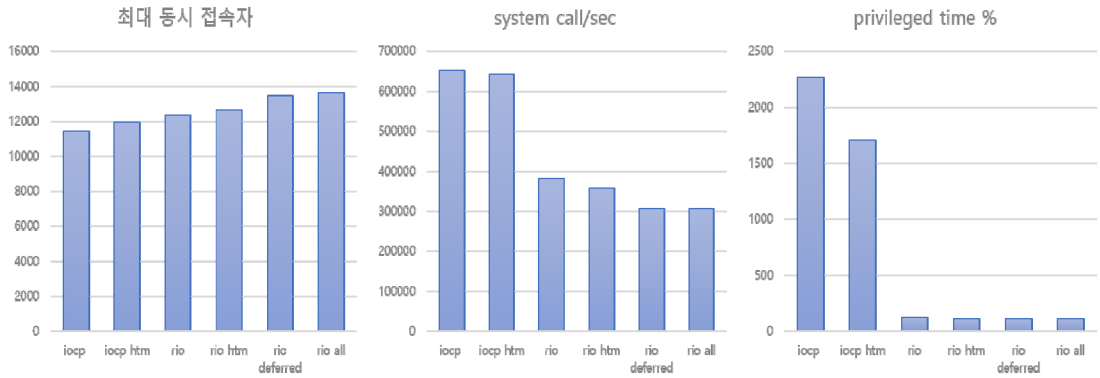
벤치마크는 구현한 서버에 스트레스 테스트 클라이언트를 최대한 접속시켜 최대 동접수를 측정한다. 테스트 PC와 서버는 기가비트 공유기를 통하여 내부망으로 연결하였다. 각각의 테스트 PC는 최대 동시 접속자 수가 2만명인 스트레스 테스트 클라이언트를 실행한다. 최대 동시 접속자 벤치마크의 서버의 작업자 스레드는 36개로 설정하였다. 또한 최대 동시 접속자 이외의 수치를 측정하기 위하여 Windows의 성능 모니터 툴을 이용하여 % Privileged Time과 System Call/sec를 측정하였다. % Privileged Time은 커널모드에서 프로세스 스레드가 코드를 실행하는데 소비한 경과 시간의 백분율을 나타내며 낮을수록 좋은 것으로 간주한다.

[Fig. 6], [Table 2]는 각 구현방식에 따른 요소별 벤치마크 결과이다. [Fig. 6]는 [Table 2]의 수치를 막대그래프로 표현한 것이다.

[Fig. 6]의 결과를 보면 IOCP를 RIO로 변경하는 것만으로도 시스템 호출 회수와 % Privileged Time이 눈에 띄게 개선 된 것을 확인 할 수 있다. 이는 실제 통신에 사용되는 버퍼의 접근과 I/O요청 완료를 확인하는 작업등에서 IOCP보다 RIO의 시스템 호출회수가 낮기 때문이다. 또한 추가적으로 RIO의 Deferred 옵션을 사용하는 것으로 시스템 호출 회수를 더욱 더 개선할 수 있는 것을 알 수 있다. HTM은 Viewlist의 동기화에

[Table 2] Benchmark Result

	최대 동시 접속자	System Call/SEC	% Privileged Time
IOCP	11627	651321	2265
IOCP + HTM	12128	643395	1706
RIO	12274	382253	125
RIO HTM	13416	358143	115
RIO Deferred	13597	307306	113
RIO Deferred HTM	13764	307562	114



[Fig. 6] Benchmark Result Graph

사용이 되었으며 HTM을 실패 시 재시도 하는 경우등의 모듈 자체의 오버헤드로 인하여 IOCP를 RIO로 변환한 만큼의 성능향상은 보이지 않은 것으로 판단되지만, IOCP와 잠금을 사용한 서버에서 잠금을 HTM으로 변경하는 것만으로도 %Privileged Time과 초당 시스템 호출 회수를 감소시켜 동시접속자 수를 향상시킬 수 있었다.

최종적으로 기준이 되는 IOCP서버와 본 논문에서 제안하는 모든 방법을 적용한 RIO서버를 비교하였을 때 Privileged Time은 최대 95%, 초당 시스템 호출 회수는 최대 53% 감소하였고, 최종적으로 최대 동시 접속자는 19% 증가하였다.

5. 결론 및 향후연구

본 연구에서는 최신 I/O API인 Registered IO와 Hardware Transactional Memory를 활용하는 새로운 MMO서버 구조를 제안하고, 이를 구현하였으며, 기존의 IOCP와 잠금방식의 게임서버와의 성능 차이를 측정하였다. 기존의 IOCP방식에 비하여 % Privileged Time은 최대 95%감소, 초당 시스템 호출 회수는 최대 53% 감소하였고, 궁극적으로 최대 동시 접속자 수는 최대 19% 상승하였다. 향후 연구 과제로는 본 구현에서 제안한 Registered I/O의 사용법이 성능향상을 위해 메모리를 많이 사용하여 서버의 사양에 민감한 문제가 있어, 이를 최소화하는 방법에 대하여 연구하고자 한다.

REFERENCES

[1] Hyun-jik Bae. “Game Server Programming Textbook”, Gilbut. 2019.

[2] Microsoft. “Registered Input/Output (RIO) API Extensions”. https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-r2-and-2012/hh997032(v=ws.11)
[3] Intel. “Intel® 64 and IA-32 Architectures Software Developer’s Manual”. Chap.16 Intel TSX Recommendations
[4] Microsoft. “I/O Completion Ports”. https://docs.microsoft.com/en-us/windows/win32/fileio/i-o-completion-ports
[5] Sung-Ik Kim, "Lockless 게임서버 설계와 구현", NDC2012, 2012
[6] KiNam KIM, Hye-Young Kim. "A study on gaming server Implementation as IOCP model applying load balance scheme in MMORPG." Korean Institute of Information Technology, Proceedings of KIIT Conference 2019.6 pp.612-615. 2019
[7] Youngsik Kim, "A study of optimal spatial partition size and field of view in massively multiplayer online game server". International Journal of Applied Engineering Research. 12. pp.8174-8178. 2017.
[8] Kim Y., Kim KN. "Design and Evaluation of a MMO Game Server". Software Engineering Research, Management and Applications (SERA 2018). Studies in Computational Intelligence, vol 789. Springer, Cham. 2019.
[9] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M. Michael, and Hisanobu Tomari. “Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, Intel Core, and POWER8.” SIGARCH Comput. Archit. News 43, 3S. pp. 144 - 157. June 2015.
[10] Marios Kardaras, Dimitrios Siakavaras, Konstantinos Nikas, Georgios Goumas, Nectarios Koziris. “Fast Con-current Skip Lists with HTM”, Intl Symposium on High-Level Parallel Programming and Applications (HLPP). 2018.
[11] Glazer Joshua L, Madhav Sanjay, “Multiplayer Game Programming”, Gilbut, 2017
[12] Tae-hyun Lim. "MMO 서버 개발 포스트 모템", NDC2012, 2012
[13] Heung-sub Lee, "[야생의 땅: 듀랑고] 서버 아키텍처 Vol. 2", NDC2016, 2016
[14] M. Herlihy and N. Shavit. “The Art of Multiprocessor Programming revised edition” Morgan Kaufmann Publishers Inc. 2012.
[15] Microsoft. “Scatter/Gather I/O”. https://docs.microsoft.com/en-us/windows/win32/winsock/scatter-gather-i-o-2

강 수 빈 (Kang, Su-bin)

약 력 : 2019 한국산업기술대학교 게임공학과 학사
2019-현재 한국산업기술대학교 일반대학원 디지털엔터테인먼트학과 석사과정

관심분야 : 병렬처리, 대용량 온라인 게임 서버

정 내 훈 (Jung,NaiHoon)

약 력 : 2002 KAIST 전산학과 박사
2002-2008 NCSoft MMORPG 프로그램팀장
2008-현재 한국산업기술대학교 게임공학부 부교수

관심분야 : 병렬처리, 컴퓨터 구조, 대용량 온라인 게임 서버