http://dx.doi.org/10.5392/JKCA.2014.14.08.022

락의 실제 : 멀티코어 상의 데이터베이스 성능 분석

Locking in Practice : Performance of a Database System on a Multicore Machine

한혁
동덕여자대학교 컴퓨터학과

Hyuck Han(hhyuck96@dongduk.ac.kr)

요약

락은 멀티프로세서 환경에서 공유 데이터에 대한 접근을 안전하게 하는 잘 알려진 일반적인 방법이다. 1960년대에 상호 배제가 소개된 후에 많은 스핀락 알고리즘이 제안되었고 운영체제나 데이터베이스 시스템에 사용되어 왔다. 이 연구에서 고성능 멀티코어 시스템 상에서 락 알고리즘이 데이터베이스 시스템에 미치는 영향을 측정하였다. 평가를 위해 그 동안 멀티코어 상에서 성능 개선을 위해 재구조화된 최신 MySQL 5.6 및 MySQL에 탑재된 InnoDB 엔진을 사용하였다. InnoDB의 스핀락 함수를 수정하여 다양한 락 알고리즘들을 구현하였고 구현된 락 알고리즘들을 멀티코어 환경에서 평가하였다.

■ 중심어 : | 락 | 멀티코어 | 데이터베이스 |

Abstract

A lock is a general and popular way of serializing accesses to shared data in multiprocessor environments. After the mutual exclusion was first introduced in 1960s, many spinlock algorithms have been proposed and deployed to real systems such as operating systems and (transactional) database systems. In this study, we measure impacts of a lock mechanism on a database system under various CPU configurations using a high-end multicore system. For the evaluation, we use the most up-to-date version of MySQL (version 5.6) with InnoDB engine, which has been substantially re-architected to improve scalability on multicore machines. We changed the original spinlock function of InnoDB to evaluate various spinlock mechanisms on multicore machines.

■ keyword : | Lock | Multicore | DBMS |

I. 서 론

상호 배제는 (Mutual Exclusion) 여러 개의 쓰레드 혹은 프로세스들이 동작하는 곳에서 올바른 수행을 도와주는 프로그램의 일부이다. 상호 배제 문제는 1965년에 E. W. Dijkstra가 해결책과 함께 처음 제시하였다 [1]. 그 후에 상호 배제를 위한 락과 같은 많은 알고리즘들이 제안되었고, 공유 메모리 환경에서 제안된 알고리즘들이 평가되었다. 그리고 스핀락 계열의 알고리즘들이 운영 체제 및 데이터베이스 시스템에 사용되었다. 예를 들어, 스핀락의 일종인 티켓 락[2]은 2008년에 Linux에 채용되었다.

페터슨 락[3]과 같은 초기 스핀락 알고리즘들의 일부는 선형화된 (linearizable) 메모리 연산을 가정하였다.

접수일자 : 2014년 04월 18일
수정일자 : 2014년 06월 18일
심사완료일 : 2014년 06월 18일
교신저자 : 한혁, e-mail : hhyuck96@dongduk.ac.kr

하지만, 이런 메모리에 관한 가정은 최신 컴퓨터 구조에서 더 이상 유효하지 않다. 그래서 그 후의 스핀락 알고리즘들은 CPU의 원자 연산들을 이용하여 구현되었다. 대표적인 CPU 원자 연산이 test-and-set (TAS)와 fetch-and-add (FAA)이다. 이러한 연산들은 비용이 비싸기 때문에 일반적으로 딜레이 혹은 로컬 스핀과 같은 오버헤드를 줄일 수 있는 방법과 함께 쓰인다. 예를 들어 T. E. Anderson은 지수 백오프 (exponential backoff)와 test-and-test-and-set (TTAS)를 함께 쓰는 알고리즘이 스핀락 알고리즘들 중에서 가장 좋은 성능을 가짐을 보였다[4]. 많은 스핀락 알고리즘들이 이론적으로 참여하고 있는 프로세서들 간에 공평하지 않은 성질을 가지기 때문에 이러한 점을 개선한 MCSLock[5], ALock[4]과 같은 알고리즘들이 제안되었다. 특히 MCSLock은 자바 가상 머신의 모니터 락에서 사용되고 있는 유명한 큐 기반 스핀락 알고리즘이다[9].

데이터베이스 및 운영체제와 같은 소프트웨어에서는 크리티컬 섹션의 길이가 일정하지 않다. 일반적으로 스핀은 크리티컬 섹션의 길이가 짧은 경우에 유의미하고, 반대로 블록은 그 반대의 경우에 유의미하기 때문에 최신 락 알고리즘은 일정하지 않은 크리티컬 섹션에 대응하기 위해 스핀과 블록을 혼합하여 사용한다. 즉, 처음 어느 정도는 스핀을 하고 어느 정도 시간을 지나도 락을 얻지 못 하면 CPU 제어를 놓고 지금 락을 점유하고 있는 프로세스가 깨워줄 때까지 기다린다. 오픈 소스 데이터베이스 시스템 중에서 가장 유명한 MySQL의 InnoDB 엔진이 이러한 방법을 사용한다.

이 논문에서 본 저자는 멀티코어 환경에서 락이 성능에 미치는 영향을 분석하고자 한다. [6][7]에서 락이 운영체제에 미치는 영향을 분석하고 개선하는 방법을 제안하였다. 본 연구는 락이 데이터베이스 시스템에 미치는 영향을 분석하고자 하였다. 이를 위해 오픈 소스 데이터베이스 시스템인 InnoDB가 탑재된 MySQL를 이용하였다. 이 연구를 위해 CPU 설정 및 InnoDB의 락 알고리즘을 변경하면서 데이터베이스 트랜잭션 처리량이 어떻게 변하는지 측정하였다.

이 논문의 구성은 다음과 같다. 2장에서는 이 연구에서 사용된 MySQL의 InnoDB의 락 알고리즘과 이 연구에 사용될 락 알고리즘에 대해 설명하고 3장에서는 실험 방법을 설명한다. 4장에서는 성능 결과와 그 분석 결과를 설명하고 5장에서는 이 논문의 결론을 제시한다.

II. MySQL InnoDB의 락 알고리즘

Algorithm 1 Lock/Unlock in InnoDB of MySQL

```
Lock ( Lock_Variable )
SpinRound : maximum number of spin rounds
MaxDelay : maximum number of delays

1: i = 0
2: while( Lock_Variable == true || i < SpinRound )
3:     delay( rand( 0, MaxDelay ) );
4:     i++;
5: endwhile
6:
7: if ( test-and-set( Lock_Variable ) == false )
8:     return;
9:
10: if ( i < SpinRound ) goto line 2;
11:
12: cond_wait( Lock_Variable );
13: goto line 1;

Unlock ( Lock_Variable )

14: Lock_Variable = false;
15: cond_signal( Lock_Variable );
```

그림 1. InnoDB의 Lock/Unlock 알고리즘

[그림 1]은 MySQL InnoDB의 락 알고리즘을 보여준다. MySQL InnoDB의 락/언락 알고리즘은 TTAS (test-test-and-set) 기반이다. 기본 TTAS 기반 락 알고리즘은 성능에 악영향을 주는 캐쉬 무효화 폭풍을 (cache invalidation storm) 유발할 수 있기 때문에 [그림 1]의 세 번째 줄과 같은 랜덤 딜레이가 사용된다. 즉, [그림 1]의 두 번째 줄과 같이 락 변수가 참이면 (test) 딜레이 시간만큼 아무 의미 없는 CPU 연산을 수행하고 다시 test 연산을 수행한다. 이렇게 test-test-and-set의 첫 번째 test가 성공하면 7번째 줄과 같이 CPU의 원자적 연산인 test-and-set 연산을 이용하여 락 변수를 참으로 변경한다. 변경이 성공하면 락을 얻은 것이 되고 실패하면 첫 test 연산을 시도한다. 만약 스핀의 회수가 일정한 수를 (SpinRound) 넘으면 해당 쓰레드 혹은 프로세스는 CPU 제어를 놓고 현재 락을 점유하고

표 1. 실험에 사용된 락 알고리즘

락 알고리즘	락 알고리즘 설명
Double Delay	딜레이를 기존보다 두 배로 한다.
Exponential Backoff	딜레이를 지수적으로 크게 한다.
Proportional Backoff	딜레이를 선형적으로 증가시킨다.
Sleep	딜레이 대신에 CPU 컨텍스트 전환을 유발한다. 이를 위해 sleep 함수를 사용한다.
Half SpinRound	기존 최대 스핀 횟수를 1/2으로 하여 빠르게 CPU 컨텍스트 전환에 이르게 한다.

있는 쓰레드 혹은 프로세스의 신호를 기다린다. 즉, InnoDB의 락 알고리즘은 SpinRound 횟수만큼 스핀을 하고 그 후에는 블록 되어 신호를 기다린다.

서론에서 언급했듯이 TTAS와 지수 백오프를 함께 쓰는 것이 TAS 기반 스핀락 알고리즘들 중에서 가장 성능이 우수하다고 알려져 있다. 하지만, [4]에서 사용된 알고리즘은 크리티컬 섹션이 비어있는 환경에서 평가되어서 데이터베이스 시스템과는 거리가 멀다. 또한, 데이터베이스 시스템은 로그 처리, 트랜잭션 처리, B-tree 처리 등에서 스핀락이 사용되기 때문에 크리티컬 섹션의 길이가 일정하지 않다. 따라서 본 연구에서 스핀락이 데이터베이스 시스템에 미치는 성능 영향을 알아보기 위해 멀티코어 서버에서 CPU의 코어의 개수, 트랜잭션의 로드 등을 변경시키면서 평가하였다.

III. 실험 평가 환경

3.1 평가를 위한 다양한 스핀락 알고리즘

스핀 기반의 락 알고리즘들은 스핀을 짧은 시간 동안 하고 락을 획득할 수 있을 때 좋은 성능을 보인다. 즉 크리티컬 섹션이 짧을 때 좋은 성능을 보인다. 하지만 같은 워크로드라도 CPU 코어의 수가 늘어나면 락 경합 정도가 커지게 된다. 이것은 크리티컬 섹션을 길게 하는 것과 동일한 효과를 가진다. 이러한 상황의 일반적인 대응책은 딜레이를 ([그림 1]의 세 번째 줄) 크게 하는 것이다.

InnoDB의 딜레이는 CPU 컨텍스트 전환 오버헤드를 줄이기 위해 CPU의 의미 없는 연산들로 구성된 연산을 수행함으로써 CPU 제어를 유지하고 있다. 코어가 많아지면 크리티컬 섹션이 짧아도 락 경합 정도 때문에 오히려 CPU 컨텍스트를 전환하여 다른 쓰레드 혹은 프로세스가 작업을 진행하는 것이 더 좋은 성능을 가지게 할 수도 있다.

따라서 저자는 락 알고리즘이 데이터베이스 시스템의 성능에 미치는 영향을 분석하기 위해서 락 알고리즘을 변경하여 비교 평가하였다. 변경한 부분은 CPU 컨텍스트 전환 없이 딜레이를 다양하게 하는 것과 강제로 CPU 컨텍스트 전환을 유발하는 것이다. [표 1]과 같이 5가지 락 알고리즘들을 실험에 사용하였다. 이러한 알고리즘을 MySQL InnoDB에 구현하여 실험에 활용하였다. 구현은 락 알고리즘 별로 모드 값을 할당하여 알고리즘을 동작하게 구현하였고 세부 구현은 [그림 2]와 같다.

3.2 실험 환경

본 연구를 위해 MySQL 5.6이 사용되었으며, MySQL은 4개의 Intel Xeon MP Processor 8000 series CPU가 탑재된 서버에서 동작한다. Intel Xeon MP Processor 8000 series는 CPU당 6개의 코어를 탑재하고 있으며 1.86 GHz로 동작한다. 각각의 코어는 48 KiB의 전용 L1 캐쉬 그리고 256 KiB의 L2 캐쉬를 가지며, 하나의 CPU에 있는 6개의 코어는 18 MiB의 L3 캐쉬를 공유한다. 또한 실험을 위하여 512 GiB의 DRAM과 500 GiB의 7200 RPM의 SATA II 인터페이스로 연결되는 하드 디스크를 장착했다. 그리고 운영체제는 Linux 3.1.5가 사용되었다.

실험을 위해 MySQL은 [표 2]의 설정을 제외하고는 모두 기본 설정을 이용하여 구동하였다. [표 2]의 innodb_flush_log_at_trx_commit 값이 2라는 것은 트랜잭션 로그의 플러쉬 정책 중에서 “로그는 매 커밋시에 로그 플러쉬는 1초에 한 번”이라는 의미로, 그룹 커밋

Algorithm 2 Lock/Unlock in InnoDB of MySQL

```
Lock ( Lock_Variable, mode )
SpinRound : maximum number of spin rounds
MaxDelay : maximum number of delays

1: if ( mode == Double_Delay )
2:    MaxDelay = 2 * MaxDelay;
3: if ( mode == Half_SpinRound )
4:    SpinRound = SpinRound / 2;
5:
6: i = 0
7: while( Lock_Variable == true || i < SpinRound )
8:    if ( mode == Sleep ) sleep( 0 );
9:    else delay( rand( 0, MaxDelay ) );
10:   if ( mode == Exponential_Backoff )
11:      MaxDelay = MaxDelay * 2;
12:   else if ( mode == Proportional_Backoff )
13:      MaxDelay =MaxDelay + 1;
14:   i++;
15: endwhile
16:
17: if ( test-and-set( Lock_Variable ) == false )
18:    return;
19:
20: if ( i < SpinRound ) goto line 7;
21:
22: cond_wait( Lock_Variable );
23: goto line 6;

Unlock ( Lock_Variable )

24: Lock_Variable = false;
25: cond_signal( Lock_Variable );
```

그림 2. 실험에 사용한 Lock/Unlock 알고리즘

에서 사용된다. innodb_buffer_pool_size를 10 GiB로 충분히 크게 하는 것은 락 알고리즘 관련된 함수들의 경합 정도를 측정하기 위해서이다.

모든 실험은 클라이언트 서버 환경 모델 하에서 이루어졌으며 MySQL 서버는 위에서 설명한 한 대의 서버에서 하나의 MySQL 프로세스가 동작하게 했으며, 클라이언트를 위해 위의 머신과 동일한 재원을 가진 머신을 추가로 사용했다. 서버와 클라어인트는 1 Gbps 이더넷으로 연결되었다. 실험에 사용된 벤치마크는 Java 1.6으로 작성되어 구동되었으며 JDBC는 MySQL Connector JDBC driver version 5.1.1이 사용되었다.

표 2. MySQL 설정

max-connection	300
innodb_buffer_pool_size	10 GiB
innodb_log_file_size	256 MiB
innodb_buffer_size	16 MiB
innodb_flush_method	fsync
innodb_flush_log_at_trx_commit	2

벤치마크를 위한 데이터베이스는 bench-1/2/3의 이름을 가지는 세 개의 테이블을 가지고 있으며 각각의 테이블은 두 개의 integer 칼럼과 열 개의 varchar 칼럼을 가지고 있다. 두 개의 integer 칼럼 중의 하나가 (b_int_key) 프라이머리 키로 사용된다. 각 테이블은 랜덤하게 생성된 10만개의 레코드를 가지고 있으며, 평가를 위해 아래와 같이 두 종류의 트랜잭션을 정의했다.

i) read-only 트랜잭션 : bench-i 테이블에서 100개의 연속된 레코드를 읽어서 프라이머리 키가 아닌 다른 integer 칼럼 값을 합한다.

ii) update-after-read 트랜잭션 : bench-i 테이블에서 100개의 연속된 레코드를 읽어서 프라이머리 키가 아닌 다른 integer 칼럼 값을 합한다. 그 후에 bench-((i+1)%3) 테이블에서 랜덤하게 선택된 20개의 레코드를 갱신한다.

3.3 데이터베이스 시스템 성능 분석 방법

벤치마크를 수행하는 동안에 시스템의 동작을 분석하기 위해 OProfile[8], vmstat 등의 툴을 사용하였다. 시스템을 프로파일한 결과표기를 위해 프로파일 결과를 다음의 세 개의 범주로 나누었다.

i) kernel : CPU가 운영체제 함수를 수행하는 부분.

ii) MySQL : CPU가 락과 관련되지 않는 MySQL 함수들을 수행하는 부분.

iii) Lock : CPU가 MySQL의 락 함수를 수행하는 부분.

서버의 CPU 코어 개수를 변경하기 위해 코어를 논리적으로 오프라인 시키는 명령어를 사용하였다. 예를 들어 코어 ID가 X인 코어를 오프라인 시키기 위해 아래의 명령어를 수행한다. 이러한 명령어를 통해 다양한 CPU 설정 아래에서 실험을 진행한다.

```
# echo 0 > /sys/devices/system/cpu/cpuX/online
```

IV. 평가 및 분석 결과

4.1 MySQL 기본 락 알고리즘 평가 결과

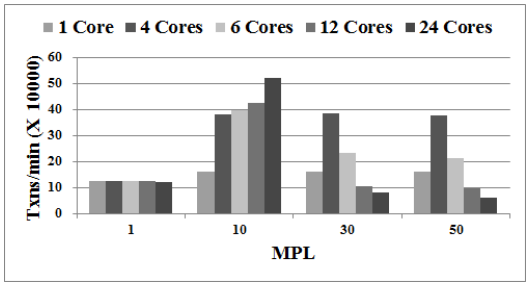

그림 3. Read-only 트랜잭션만 사용한 경우의 성능

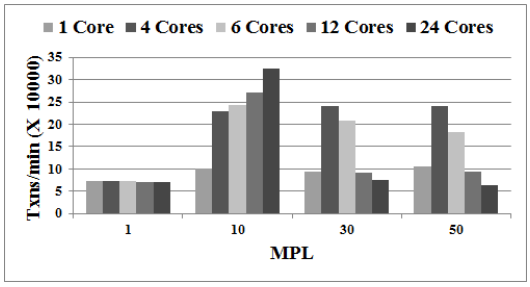

그림 4. 75%의 Read-only 트랜잭션을 사용한 경우의 성능

MySQL InnoDB의 락을 평가하기 위해 서버의 코어 개수와 트랜잭션을 요청하는 터미널의 수를 (MPL, Multi Programming Level) 변경시키면서 트랜잭션 처리량을 측정하였다. [그림 3]은 read-only 트랜잭션만 사용한 경우의 성능을 [그림 4]는 read-only 트랜잭션과 update-after-read 트랜잭션의 비율을 3:1로 했을 경우의 성능을 보여준다. 두 경우 모두 성능의 최대치 외에는 비슷한 성능 추이를 보인다. 성능의 최대치는 클라이언트 터미널의 개수가 10이고 CPU 코어의 개수가 24개인 경우에 얻어진다. 트랜잭션 로드가 작은 경우인 10 MPL인 경우에는 CPU의 코어의 개수가 증가함에 따라 성능이 증가하는 결과를 보였다. 하지만, 30/50 MPL인 경우에는 CPU 코어의 개수가 증가하면 오히려 성능이 떨어짐을 알 수 있다. 이것은 CPU 코어의 개수를 증가하면 락 관련 함수의 경합이 심해져서 성능의 악영향을 미치기 때문이다.

이를 확인하기 위해 3.3절에서 소개한 시스템 프로파일 툴을 사용하여 분석을 진행하였다. [그림 5]는 read-only 트랜잭션만 사용했을 때의 시스템 분석 결과이다. 전체적으로 CPU 코어의 개수가 증가함에 따라 락 관련 함수들을 수행하기 위한 CPU 사용율이 증가함을 알 수 있다. 30/50 MPL이고, 코어의 개수가 24개인 경우에는 CPU 사용의 87%를 락 함수를 수행하는데 쓰고 있다. 이것은 더 많은 데이터베이스 트랜잭션을 수행하기 위해서 CPU 코어를 늘리더라도 데이터베이스 시스템의 내부 자료 구조 보호를 위한 락 함수 경합으로 락 함수 수행에 더 많은 CPU 자원을 소모함을 알 수 있다. 본 실험에서는 서버의 메모리가 충분히 크고 MySQL InnoDB의 버퍼 크기가 충분히 크므로 디스크 관련 IO 혹은 버퍼 관리에서 사용하는 락이 차지하는 비중은 거의 없고, B-tree, 동시성 제어에 필요한 락 관련 함수가 대부분의 수행 시간을 차지하였다.

[그림 6]은 read-only 트랜잭션만 사용했을 때의 L3 캐쉬의 미스 횟수를 측정한 결과이다. 락 함수 수행을 위한 CPU 사용률이 올라가는 추세에 맞추어서 L3 캐쉬의 미스 횟수가 올라감을 확인할 수 있다. 이것은 락 함수 수행을 위한 공유 데이터를 여러 CPU에서 동시에 접근하기 때문이다.

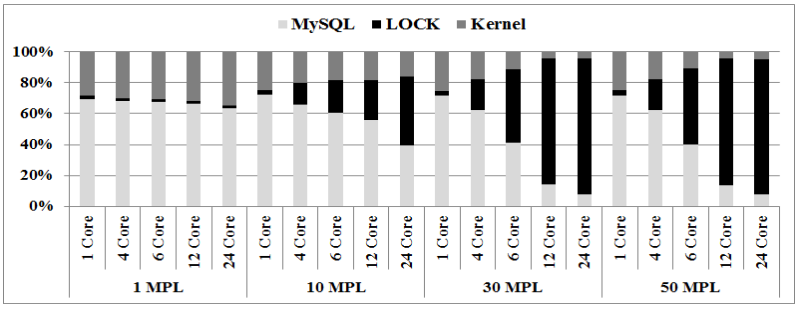

그림 5. Read-only 트랜잭션만 사용한 경우의 시스템 프로파일 결과

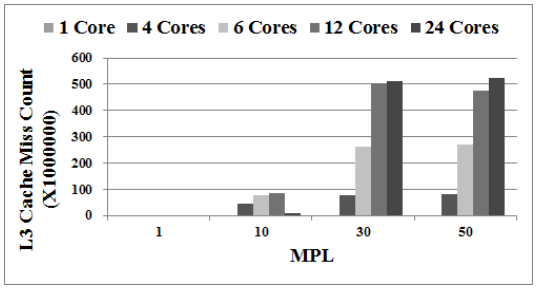

그림 6. Read-only 트랜잭션만 사용한 경우의 L3 Cache 미스 횟수 (LLC Miss Count)

4.2 다양한 락 알고리즘 평가 결과

이번 장에서는 3.1장에서 제안한 다양한 락 알고리즘들을 비교 평가한다. 이를 위해 CPU 코어의 수를 24개로 고정하고 락 알고리즘과 트랜잭션을 요청하는 터미널의 수를 변화시켰다.

[그림 7]은 read-only 트랜잭션들만 요청했을 때의 여러 락 알고리즘들의 성능 결과이다. Default는 MySQL InnoDB의 기본 락 알고리즘을 나타낸다. Double Delay, Exponential/Proportional Backoff, Half SpinRound 방법들은 모두 CPU의 컨텍스트 전환을 막기 위해 CPU의 무의미한 연산을 TTAS의 첫 테스트 사이의 딜레이에 활용하는 것을 기반으로 한다. 이 방법은 크리티컬 섹션의 길이가 짧으면 효과적이지만 코어가 많아져서 락 경합이 심해지면 딜레이를 길게 하여도 성능에 악영향을 미친다. 반면, CPU 컨텍스트를 강제로 전환시키는 방법인 Sleep은 최고 성능은 다른 방법에 비해 떨어지지만 성능을 유지하는 정도가 더 좋다. 그 이유는 워크로드가 낮은 상황에서 컨텍스트 전환은 컨텍스트 전환 비용이 딜레이 비용보다 커서 컨텍스트 전환이 트랜잭션 처리에 오버헤드가 되고 워크로

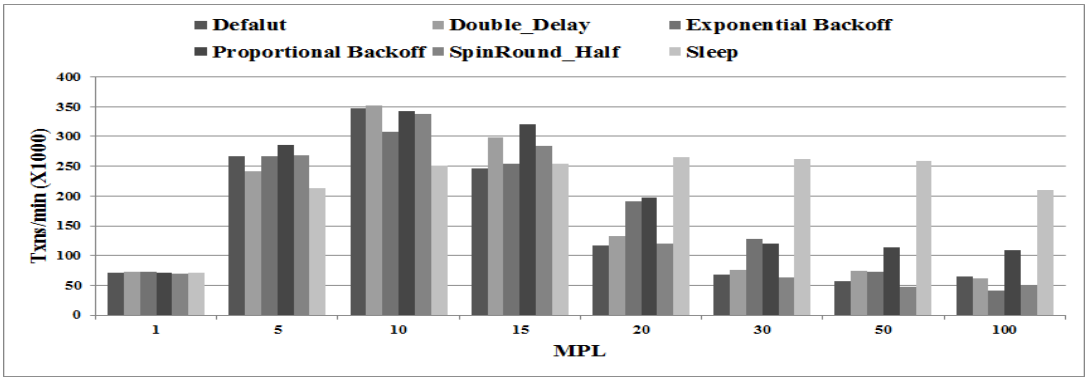

그림 7. 다양한 락 알고리즘을 사용했을 때의 트랜잭션 처리 성능 (24 코어, read-only 트랜잭션)

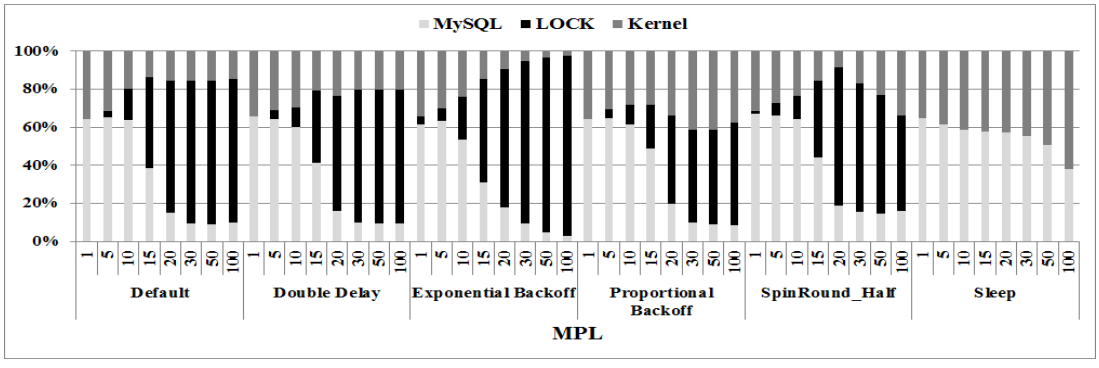

그림 8. 다양한 락 알고리즘을 사용했을 때의 시스템 프로파일 결과 (24 코어, read-only 트랜잭션)

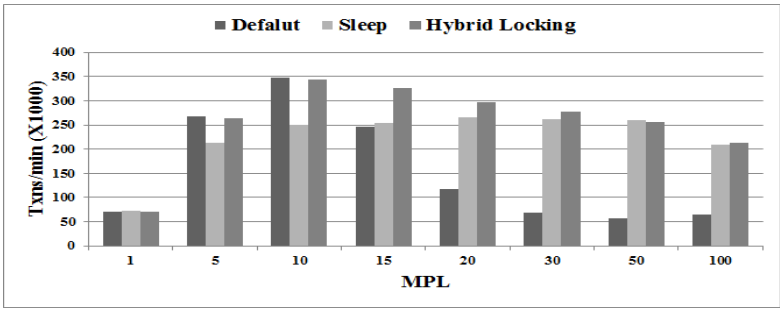

그림 9. 하이브리드 락 알고리즘의 성능 (24 코어, read-only 트랜잭션)

드가 높은 상황에서는 락 획득 실패 시에 CPU 제어를 다른 쓰레드에 넘겨주어서 다른 트랜잭션 처리를 가능하게 하기 때문이다.

[그림 8]은 시스템 프로파일 결과이다. Sleep 방법을 제외한 다른 모든 방법에서 락 함수를 수행하는 부분이 워크로드가 심해질수록 커짐을 확인할 수 있다. 특히, 락 함수에서 CPU의 의미 없는 연산들을 수행하는데 보내는 시간이 대부분이다. 반면 Sleep 방법은 CPU 컨텍스트를 전환하기 때문에 락 관련 함수 수행 시간이 매우 적음을 알 수 있고, 이것이 최고 성능은 비록 떨어지지만 성능을 꾸준히 유지시켜주는 역할을 한다.

위의 실험 결과는 락 경합의 정도에 따라 CPU 컨텍스트를 전환 여부를 결정하는 것이 중요하다는 것을 보여준다. 따라서 MySQL InnoDB의 기본 락 알고리즘에 워크로드가 낮을 때는 CPU 컨텍스트를 전환하지 않고 워크로드가 높을 때는 CPU 컨텍스트를 강제로 전환하는 부분을 첨가하였다. 이를 위해 sleep 함수의 기본 수행 시간을 측정 즉, sleep(0)의 수행 시간을 측정하여 그 시간만큼은 CPU 컨텍스트를 전환하지 않고 그 후부터는 sleep 함수를 통해 강제로 CPU 컨텍스트를 전환하도록 변경하였다. [그림 9]의 Hybrid Locking이 그 결과이다. 이 방법에서의 트랜잭션 처리량은 기본 알고리즘의 트랜잭션 처리량의 최대치와 같고, 높은 워크로드에서의 성능을 유지하는 성질을 가지고 있다.

V. 결론

스핀을 기반으로 하는 락은 크리티컬 섹션이 짧은 경우에 일반적으로 효과가 크다. 그래서 그 동안 락을 사용하는 많은 소프트웨어가 크리티컬 섹션을 짧게 하는 방향으로 개발이 진행되어 왔다. 그러나 멀티코머 환경에서는 락 경합이 심해지면 크리티컬 섹션의 길이와 상관없이 성능에 악영향을 미치게 되고, 락 경합시에 스핀을 하는 방법이 성능에 중요한 변수가 된다.

본 연구에서는 멀티코어 환경에서 스핀락이 데이터베이스 시스템에 미치는 영향을 MySQL의 InnoDB 엔진에 다양한 스핀락 알고리즘을 구현 및 평가하여 분석하였다. 실험을 통해서 락 알고리즘의 스핀이 i) CPU 컨텍스트 전환을 하지 않는 경우에는 최대 성능이 높으나 코어 및 워크로드가 많아지면서 성능이 크게 하락하고, ii) CPU 컨텍스트 전환을 하면 최대 성능은 다소 떨어지나 코어 및 워크로드가 많아져도 성능이 떨어지는 폭이 상대적으로 적음을 확인하였다. 이것은 멀티코어 환경에서는 크리티컬 섹션이 짧더라도 락 경합 정도에 따라 CPU 컨텍스트 전환 여부를 선택하는 것이 더 좋다는 것을 보여준다. 이를 위해 CPU 컨텍스트를 전환하는 것과 전환하지 않는 것을 같이 사용하는 간단한 방법을 고안했고, 실험을 통해 이러한 방법이 최대 성능을 높게 그리고 높은 성능을 유지해주는 효과가 있음을 확인하였다.

참 고 문 헌

[1] E. W. Dijkstra, “Solution of a problem in concurrent programming control,” Communications

of the ACM, Vol.8, No.9, p.569, 1965.
[2] D. P. Reed and R. K. Kanodia, "Synchronization with event-counts and sequencers," Communications of the ACM, Vol.22, No.2, pp.115-123, 1979.
[3] G. L. Peterson, "Myths about the mutual exclusion problem," Information Processing Letters, Vol.12, No.3, pp.115-116, 1981.
[4] T. E. Anderson, "The performance of spin lock alternatives for shared-memory multiprocessors," IEEE Transactions on Parallel and Distributed Systems, Vol.1, No.1, pp.6-16, 1990.
[5] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," ACM Transactions on Computer Systems, Vol.9, No.1, pp.21-65, 1991.
[6] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich, "An analysis of Linux scalability to many cores," in Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI), pp.1-8, 2010.
[7] A. T. Clements, F. Kaashoek, and N. Zeldovich, "Scalable Address Spaces Using RCU Balanced Trees," in Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2012.
[8] http://oprofile.sourceforge.net/
[9] http://www.podc.org/dijkstra/2006-dijkstra- prize/

저 자 소 개

한 혁(Hyuck Han) 정회원

- 2003년 8월 : 서울대학교 컴퓨터공학부(공학사)
- 2006년 2월 : 서울대학교 컴퓨터공학부(공학석사)
- 2011년 2월 : 서울대학교 컴퓨터공학부(공학박사)
- 2011년 3월 ~ 2012년 8월 : 서울대학교 컴퓨터공학부 박사후 연구원
- 2012년 9월 ~ 2014년 2월 : 삼성전자 메모리 사업부 책임연구원
- 2014년 3월 ~ 현재 : 동덕여자대학교 컴퓨터학과 조교수

<관심분야> : 데이터베이스 시스템, 병렬 프로그래밍, 분산 시스템