(KCC2023 우수발표논문) 매니코어 환경에서 스킵 리스트에 적용된 논블로킹 동기화 기법의 성능비교[†]

(Performance Comparison of Non-Blocking Synchronization Techniques Applied to a Skip List in a Many-Core Environment)

[‡] [§] [¶] [¶]

() () () ()

요 약 매니코어 프로세서 도입의 빠른 확산으로 다양한 종류의 소프트웨어 성능이 동기화 기법에 큰 영향을 받는다. 전통적인 락(lock) 기반의 블록킹 동기화 기법이 갖는 성능 한계를 극복하기 위해, 여러 종류의 논블로킹 방식의 알고리즘들이 제안되었다. RCU는 대표적인 논블로킹 방식 동기화 기법으로 널리 사용되고 있지만 프로그래밍 복잡성과 쓰기연산의 성능 저하 문제가 있다. 이와 같은 한계를 극복하기 위해 최근 Multi Version Concurrency Control (MVCC) 기반으로 동작하며 트렌잭션 방식의 보다 쉬운 API를 제공하는 MV-RLU 동기화 기법이 소개되었다. 본 논문에서는 LSM 트리기반 Key-Value Store에서 각광받고 있는 Skip list 자료구조에 RCU와 MV-RLU 동기화 기법을 적용하여 매니코어 환경에서 성능 평가 및 분석을 하였다. 대부분 읽기연산(읽기 연산 98%) 워크로드에서 MV-RLU가 RCU에 비해 최대 5.04배의 스레드 동시성 향상 결과를 보였다.

키워드 : 메니코어, 논블록킹 동기화 기법, MV-RLU, RCU, 스킵리스트

Abstract The rapid adoption of many-core processors has a significant impact on various types of software performance due to synchronization techniques. To overcome the performance limitations of traditional lock based synchronization techniques in many-core systems, algorithms based on non-blocking methods have been proposed. While RCU is a representative non-blocking synchronization technique widely used, it has issues with programming complexity and performance degradation during write operations. To address these limitations, a more recent synchronization technique called MV-RLU has been introduced, which operates based on Multi Version Concurrency Control (MVCC) and provides a simpler API in a transactional manner. In this paper, we apply RCU and MV-RLU synchronization techniques to the Skip list data structure, which is gaining attention in LSM tree-based Key-Value Stores, and evaluate and analyze their performance in a many-core environment. Under a read mostly workload(Read Operation 98%), MV-RLU demonstrated about 5.04s higher thread concurrency improvement compared to RCU.

Key words : Manycore Environment, Non-Blocking Synchronization Technique, MV-RLU, RCU, Skiplist

[†] 사사

[‡] 비 회 원 :

[§] 비 회 원 :

[¶] 종신회원 :

[¶] 종신회원 :

논문접수 : 2023년 ??월 ??일
심사완료 : 20??년 ??월 ??일

정보과학회논문지: 정보통신 제31권 제6호(2004.12)

1. 서 론

매니코어 프로세서의 도입으로 컴퓨터 시스템의 병렬성은 증가되었지만 전통적 동기화 기법의 한계로 코어 수에 비례하는 성능 향상을 보이지 못한다. Spinlock, Mutex, Readers-writer lock 등의 전통적 동기화 기법을 사용하면 암달의 법칙에 따라 코어가 증가하여도 Lock에 의한 임계영역으로 작업들이 블록 되기 때문에 성능 향상에 한계가 생긴다[1].

매니코어 시스템에서 성능 한계가 발생하는 또 다른 이유는 매니코어 하드웨어의 캐시 동기화로 인한 부하이다. 전통적 동기화 기법이 스레드 간에 공유하는 동기화 객체는 스레드가 임계영역에 진입할 때마다 그 값이 변

한다. 이는 임계영역에 진입하기 위해 블록된 스레드들의 캐시를 모두 갱신해야 함을 뜻한다. 코어수가 많은 시스템에서 이러한 부하는 성능에 치명적이다[2].

이와 같은 전통적 동기화 기법의 문제를 해결하기 위해 논블로킹 방식의 동기화 기법이 제안되었다[3, 4]. 대표적 논블로킹 알고리즘인 Read-Copy Update(RCU)는 한 스레드가 공유 자료구조를 변경하는 중에도 읽기 연산을 수행하는 스레드들의 동시 실행을 가능하게 한다. 이러한 특성으로 인해 RCU는 쓰기 연산의 비율이 적은 상황에서는 성능 확장성을 보이지만 쓰기 연산을 Mutex로 동기화하기 때문에 쓰기 연산의 비율이 증가하면 성능이 감소하는 문제가 있다. 또한, 수정 연산 시에 단일 포인터에 대한 원자적 연산만 지원하므로 복잡한 자료구조의 수정 시 프로그래밍이 복잡해지는 단점이 있다. MV-RLU (Multi Version Read Log Update)는 다중 읽기뿐만 아니라 다중 쓰기 연산도 블로킹 없이 수행할 수 있는 논블로킹 기법이다. 다중 버전을 생성하여 여러 스레드가 각자 해당 버전을 동시에 접근하는 방식으로 스레드 동시성을 높이는 구조이다. 그리고 트랜잭션 방식의 쉬운 프로그래밍 인터페이스를 제공한다[4, 5].

Skip list는 탐색, 삭제, 삽입이 확률적으로 O(log(N))이면서도 B+Tree에서 발생하는 자료구조 재조정의 부하가 없기 때문에 LSM 트리 기반 Key-Value Store를 포함하여 응용프로그램에서 각광받고 있다[6]. 매니코어 환경에서 전통적인 동기화 기법에 기반한 Key-Value Store를 사용하는 경우, Skip list는 성능 확장성이 없다.

대표적인 논블로킹 기법인 RCU, MV-RLU를 Skip list에 적용하는 것으로 동기화 성능 향상을 기대할 수 있음에도 관련 선행 연구가 적다. 본 논문에서는 Skip list에 논블로킹 동기화 기법인 RCU, MV-RLU를 적용하여 매니코어 환경에서 삽입, 삭제, 탐색 연산의 성능을 평가하였다. 성능 평가는 읽기 집중(읽기 80%), 쓰기 집중(읽기 20%), 대부분 읽기연산(읽기 98%)의 워크로드로 진행했다. MV-RLU를 적용한 Skip list는 대부분 읽기집중 워크로드에서 RCU보다 최대 5.04배의 성능 향상을 보였다.

2. 관련 연구

본 연구 이전에도 매니코어 환경에서 확장성 있게 동작하는 Skip list를 구현, 실험한 연구들이 있다[7, 8]. JellyFish는 Key-Value Store인 RocksDB의 Skip list를 MVCC (Multi Version Concurrency Control) 방식의 기법을 적용해 성능을 개선했다[7]. Jiffy 역시 MVCC 기반의 Skip list로써 배치 연산과 자료구조의 스냅샷을 이용해 최적화된 범위 탐색을 구현하였다[8]. 논블로킹 방식의 동기화 기법인 MV-RLU와 RCU도 Skip list 자료구조의 동시성을 향상시킬 수 있는 방안임에도 불구하고 선행 연구가 많지 않아 본 논문에서 이를 평가 및 분석한다.

3. 구현

3.1 RCU Skip list

Skip list는 한 노드에 접근할 수 있는 경로가 하나 이상이므로 RCU가 요구하는 단일 포인터 연산으로는 Skip list의 수정을 마칠 수 없다. 대신 노드별 Mutex와 Garbage Collector를 이용한 알고리즘[9]을 응용해 구현하였다. RCU는 읽기 연산을 위해 임계영역에 진입하는 쓰레드에게 RCU_READ_LOCK을 호출하게 한다. RCU는 쓰레드가 RCU_READ_UNLOCK을 호출하기 전에 참조한 객체가 제거되지 않는 것을 보장한다. 참고한 알고리즘[9]은 Java 언어를 대상으로 개발한 것으로 참조가 존재하면 그 객체가 제거되지 않는 것을 전제로 한다. 이를 RCU_READ_LOCK과 RCU_READ_UNLOCK API를 이용해 Java의 Garbage Collector를 대체하였다. 사용한 알고리즘[9]은 자료구조의 무결성을 보장하기 위해 삽입 중인 노드, 삭제 중인 노드를 두 개의 플래그를 사용해 구분한다. 노드별로 할당된 두 개의 플래그는 원자적 연산을 사용해 동기화한다. RCU로 적용한 알고리즘[9]은 대기할 필요 없이 Skip list를 탐색할 수 있지만 Skip list 수정연산은 Mutex를 얻기위해 대기하므로 블록킹 방식이다.

3.2 MV-RLU Skip list

MV-RLU는 API와 구현 가이드라인이 명확히 정의되어 있다. 객체를 읽기 위해서는 MVRLU_READ_LOCK을 호출해 스레드 자신이 자료구조에 접근한 상태임을 알리고 자료구조에 접근한 시점을 측정한다. 이후 객체를 읽기 위해서는 MVRLU_DEREF를 이용해 자신이 자료구조에 접근한 시점에 부합하는 객체의 버전을 탐색한다. 객체를 수정하는 경우에는 MVRLU_TRY_LOCK을 이용해 객체의 수정 권한을 얻는다. 만약 객체의 수정 권한을 얻는데 실패하면 MVRLU_ABORT를 호출해 수행한 모든 작업을 취소하고 수정을 재시도한다. 객체의 수정권한을 얻은 후에는 스레드의 로그에 복사본을 생성한다. 스레드는 복사본을 수정 완료한 뒤 다른 스레드에게 공개(commit)한다.

4. 성능 평가

실험 환경은 Intel® Xeon® Gold CPU 두 개를 이용해 구성하였다. 자세한 사항은 표 1에 나타내었다.

표 1 실험 환경 / Experiment Environment

CPU	Intel® Xeon® Gold 6258RCPU@2.70GHz
# of cores	2 x CPUs (112 logical cores)
Memory	376GB
OS	5.15.0-67-generic #74~20.04.1-Ubuntu
Compiler	clang version 10.0.0-4ubuntu1

실험을 위해 마이크로 벤치마크를 구현하였다. 벤치마크에서는 Skip list에 삽입, 삭제, 탐색 세 가지 연산을

10초 동안 수행한 횟수를 측정하였다. 읽기와 쓰기연산의 비율에 따른 성능 추이를 관찰하기 위하여 탐색 연산의 비율을 조절하여 읽기집중(탐색 80%), 쓰기집중(탐색 20%), 대부분읽기(탐색 98%) 워크로드로 구분하여 실험을 진행했다. 또한 각 연산별 노드의 접근 분포에 따른 성능 영향을 관찰하기 위해 균등(Uniform) 분포, Zipfian 분포(지수값 s: 0.3, 0.6, 0.9)로 구분하여 총 열두 종류의 실험을 진행했다. 그림 1은 워크로드 구성별 탐색, 삽입, 삭제 연산의 초당 평균수를 모두 더한 실험결과이다. 각 실험 결과에 대한 설명을 위해 Perf를 이용해 CPU cycle을 측정하여 표 2에 나타냈다. 표 2 MV-RLU 항목에 기재된 find는 Skip list에 삽입, 삭제할 때 수정할 노드들을 찾는 함수다. mutex_trylock은 MVRLU_TRY_LOCK 함수가 노드의 수정권한을 얻기 위해 사용하는 함수이다.

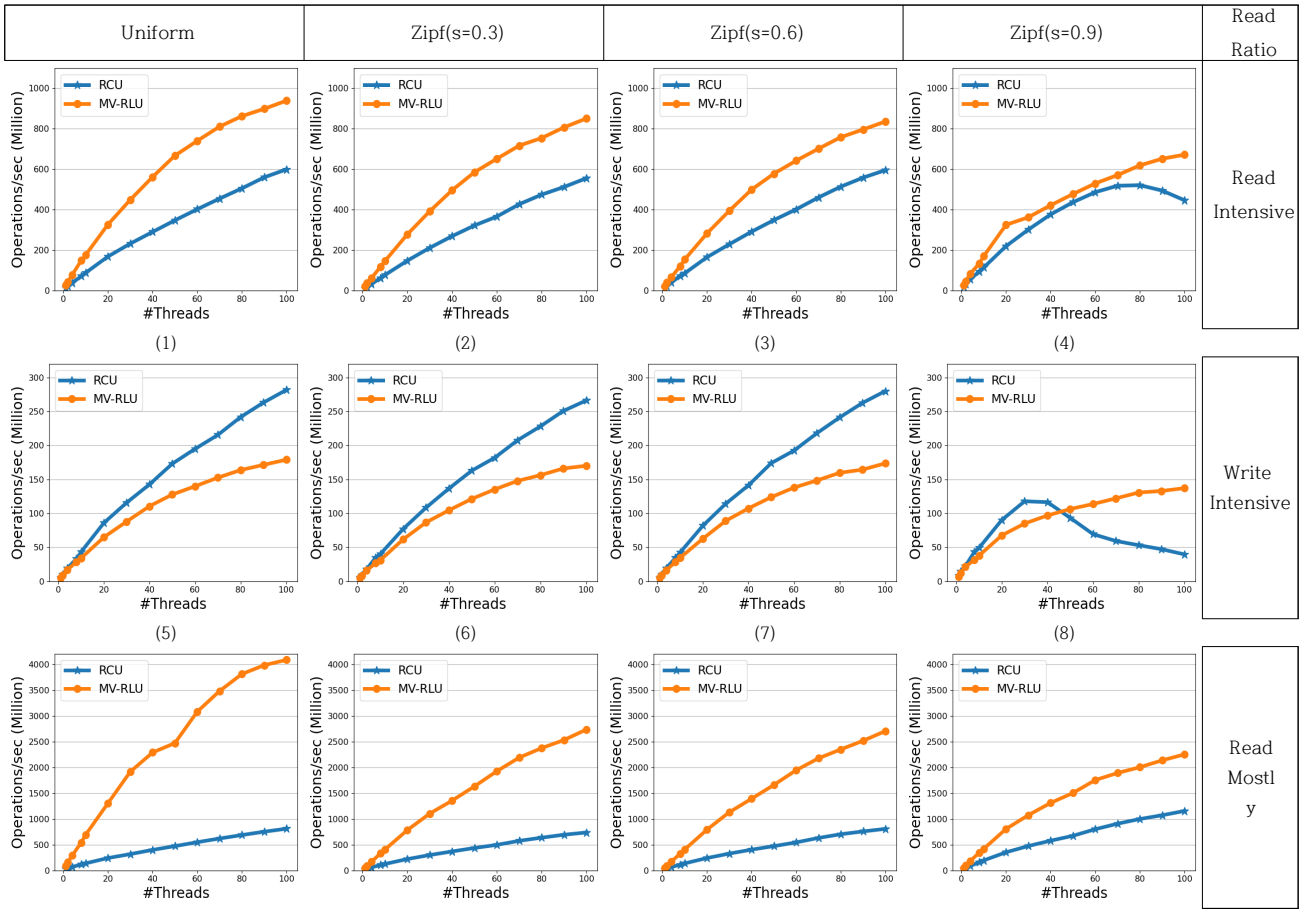

그림 1: Skip list에 가해진 시간당 연산 수행 횟수 / The number of operations per second applied to the skip list

읽기 집중 워크로드에서 MV-RLU는 항상 RCU보다 높은 성능을 보여준다. MV-RLU가 Skip list를 수정하면 버전이 생성된다. 생성된 버전은 MV-RLU가 Skip list를 탐색할 때 확인해야 하는데 버전이 거의 없거나 적으면 버전 탐색 부하가 감소한다. RCU는 상대적으로 균일한 랜덤 키가 생성되는 그림1-(1), (2), (3) 실험결과에서 스레드 개수에 비례해 성능이 증가함을 확인할 수 있다. 이러한 결과는 RCU를 Skip list에 적용하기 위해 사용한 알고리즘[9]이 노드별 Mutex (Fine-grained lock 방식)을 사용해 최적화하였기 때문이다. 수정이 필요한 노드면 Mutex를 얻으므로 스레드 간 경합이 낮으면 RCU의 수정 연산을 병렬적으로 처리할 수 있어 확장성있는 성능을 낼 수 있다. 그림1-(4)에서 RCU는 70개 스레드를 기점으로 스레드 개수에 성능이 반비례하는 현상을 관찰할 수 있다. 원인은 Zipf 분포의 편향이 심해짐에 따라 스레드간 연산이 동일 노드에서 경합 가능성이 높으므로 Mutex에 소모하는 CPU 사이클이 증가하기 때문이다.

쓰기 집중 워크로드인 그림1-(5), (6), (7)에서 RCU는 MV-RLU보다 높은 성능을 보여준다. 하지만 그림1-(8)에서 RCU는 스레드 30개를 기점으로 스레드 수에 성능이 반비례한다. 이 현상의 원인은 그림1-(4)와 동일한 것으로 랜덤 키 값의 Zipf 분포가 심하게 편향됨에 따라 Mutex에 소모하는 CPU 사이클이 증가하기 때문이다.

표 2에서 RCU의 쓰기 집중 워크로드를 보면 40개 스레드, Zipf(s=0.9)에서 Mutex에 소모하는 CPU 사이클이 11.49%이다. 이후 70개 스레드에서는 Mutex의 CPU 사이클 소모율은 42.90%까지 치솟는다. 이러한 현상으로 인해 스레드가 증가하여도 오히려 성능이 감소하게 된다.

표 2 Perf 상위 세 가지 CPU 소모 항목 / Perf top three cpu cycle consuming items

		40 Threads		70 Threads	
		RCU	MV-RLU	RCU	MV-RLU
Read Intensive	Uniform	Search(67.84%) Insert(10.28%) Delete(9.68%)	mvrlu_deref(38.62%) find(9.37%) Thread Func(4.69%)	Search(67.24%) Insert(10.28%) Delete(9.74%)	mvrlu_deref(37.22%) find(8.50%) Thread Func(4.33%)
	Zipf (s=0.3)	Search(61.43%) Insert(9.33%) Delete(8.82%)	mvrlu_deref(32.69%) find(7.79%) Gen Zipf(5.81%)	Search(61.70%) Insert(9.48%) Delete(8.93%)	mvrlu_deref(31.89%) find(7.25%) Gen Zipf(5.56%)
	Zipf (s=0.6)	Search(60.10%) Insert(9.21%) Delete(8.70%)	mvrlu_deref(32.91%) find(7.52%) Gen Zipf(5.92%)	Search(60.21%) Insert(9.36%) Delete(8.80%)	mvrlu_deref(45.35%) find(5.21%) mutex_trylock(4.81%%)
	Zipf (s=0.9)	Search(53.66%) Insert(8.78%) Delete(7.98%)	mvrlu_deref(43.18%) find(5.65%) Gen Zipf(4.93%)	Search(48.72%) Insert(8.96%) Delete(7.41%)	mvrlu_deref(32.51%) find(7.00%) Gen Zipf(5.56%)
Write Intensive	Uniform	Insert(33.50%) Delete(31.73%) Search(14.28%)	mvrlu_deref(43.80%) find(10.75%) mutex_trylock(5.43%)	Insert(32.46%) Delete(30.61%) Search(13.62%)	mvrlu_deref(41.63%) find(9.88%) mutex_trylock(5.09%)
	Zipf (s=0.3)	Insert(31.23%) Delete(29.32%) Search(13.06%)	mvrlu_deref(41.66%) find(10.07%) mutex_trylock(5.21%)	Insert(30.54%) Delete(28.80%) Search(12.76%)	mvrlu_deref(39.55%) find(9.30%) mutex_trylock(4.83%)
	Zipf (s=0.6)	Insert(30.43%) Delete(28.70%) Search(12.91%)	mvrlu_deref(41.04%) find(9.86%) mutex_trylock(5.35%)	Insert(29.81%) Delete(27.98%) Search(12.39%)	mvrlu_deref(39.66%) find(8.99%) mutex_trylock(4.97%)
	Zipf (s=0.9)	Insert(22.91%) Delete(18.80%) Mutex(11.49%)	mvrlu_deref(42.80%) find(8.38%) mutex_trylock(6.90%)	Mutex(42.90%) Insert(14.86%) cpu idle(8.41%)	mvrlu_deref(43.99%) mutex_trylock(9.15%) find(7.24%)
Read Mostly	Uniform	Search(88.48%) Gen Random(2.61%) Thread Func(2.39%)	mvrlu_deref(18.23%) Thread Func(16.28%) Search(14.55%)	Search(88.94%) Gen Random(2.32%) Thread Func(2.14%)	mvrlu_deref(18.96%) Thread Func(15.27%) Search(15.10%)
	Zipf (s=0.3)	Search(78.26%) Gen Zipf(4.22%) __expm1(3.33%)	Gen Zipf(14.79%) __expm1(12.13%) mvrlu_deref(11.09%)	Search(80.13%) Gen Zipf(3.83%) __expm1(2.96%)	Gen Zipf(14.47%) __expm1(11.66%) mvrlu_deref(11.57%)
	Zipf (s=0.6)	Search(76.49%) Gen Zipf(4.65%) __expm1(3.66%)	Gen Zipf(14.90%) __expm1(12.50%) mvrlu_deref(10.93%)	Search(78.38%) Gen Zipf(4.22%) __exmp1(3.21%)	Gen Zipf(14.62%) mvrlu_deref(12.42%) __expm1(11.72%)
	Zipf (s=0.9)	Search(68.33%) Gen Zipf(6.61%) __expm1(4.55%)	mvrlu_deref(19.69%) Gen Zipf(14.29%) __expm1(10.03%)	Search(70.24%) Gen Zipf(6.05%) __expm1(4.09%)	mvrlu_deref(28.09%) Gen Zipf(12.51%) __expm1(8.61%)

반면에 MV-RLU는 그림1-(8)에서 50개 스레드를 기점으로 RCU의 성능을 역전한 뒤에도 스레드 개수에 비례해 성능이 증가하는 것을 확인할 수 있다. 표 2에 쓰기 집중 워크로드에서 측정한 MV-RLU의 CPU 사이클 소모 항목 중에 Zipf(s=0.9)를 보면 40개 스레드에서 mvrlu_trylock이 6.90%이고 이후 70개 스레드에서 9.15%로 증가한다. 하지만 RCU가 70개 스레드에서 Mutex에 소모한 CPU 사이클이 42.90%인 것을 감안하면 MV-RLU는 RCU에 비해 상대적으로 Mutex에 소모하는 비용이 적다. 이는 MV-RLU의 연산은 수정/읽기 여부와 무관하게 항상 논블록킹인 반면 RCU의 수정 연산은 Mutex를 얻기 위해 대기해야 하는 블록킹 방식이기 때문이다.

대부분 읽기 연산 워크로드(읽기 연산 98%)에서는 MV-RLU가 RCU보다 항상 높은 성능을 보여준다. 그림 1-(9)에서 스레드 100개일 때 MV-RLU는 RCU에 비해 5.04배 높은 성능을 냈다. 수정 연산의 비율이 2%로 적기 때문에 다른 워크로드와 비교해 MV-RLU, RCU 모두 연산의 절대값은 증가했다. 수정 연산이 적을 때 MV-RLU는 Skip list를 탐색하는데 단일 스레드로 탐색하는 수준에 가까운 비용만 요구한다. 반면 RCU는 Skip list의 무결성을 보장하기 위해 수정 연산과 무관하게 항상 찾은 노드의 플래그를 원자적 연산으로 확인해야 한다. MV-RLU와 RCU의 탐색 비용 격차는 표 2의 대부분 읽기 연산 워크로드 항목에 나타난다. 표 2에 대부분 읽기 워크로드 항목에 RCU는 Search에 소모된 CPU 사이클이 60%를 상회하지만 MV-RLU는 mvrlu_deref에 소모한 CPU 사이클이 최대 30%를 넘지 않는다. RCU는 대부분 읽기 연산 항목에서 유일하게 스레드 수 증가에 반비례해 성능이 감소하는 현상을 격지 않았다. 수정 연산이 2%로 극히 적기 때문에 랜덤 키의 분포가 Zipf(s=0.9)로 편향되어도 Mutex의 소모한 CPU 사이클이 적었다. 표 2에도 상위 세 개의 항목에 Mutex는 등장하지 않는다.

공간제약 상 읽기/쓰기 혼합(읽기 연산 50%) 워크로드는 제시하지 못했다. 읽기/쓰기 혼합 워크로드에서는 Uniform, Zipf(s=0.3, 0.6)의 실험에서 MV-RLU와 RCU

는 유사한 성능을 보여줬다. 하지만 Skip list 특정 구역에 연산이 집중되는 Zipf((s=0.9) 실험 결과에서는 그림 1-(8)과 유사한 특징이 나타났다. Zipf(s=0.9) 결과에서 RCU는 40개 스레드를 기점으로 성능이 하락했다. 반면에 MV-RLU는 스레드 개수에 비례한 확장성있는 성능을 보여줬다.

5. 결론

본 논문에서는 널리 사용되는 자료구조인 Skip list에 대표적인 논블로킹 동기화 알고리즘인 RCU, MV-RLU를 적용하고 그 성능을 비교하였다. 쓰기 집중 워크로드를 제외한 모든 실험에서 MV-RLU는 RCU보다 높은 성능을 보였다. 쓰기 집중 워크로드의 Zipf(s=0.9) 실험에서 RCU는 Mutex의 영향으로 30개 스래드부터 성능이 감소한 반면 MV-RLU는 확장성 있는 성능을 보여줬다. 읽기 집중 워크로드와 대부분 읽기 연산 워크로드에서 RCU와 MV-RLU의 성능의 절대값은 쓰기 집중 워크로드에 비해 증가하였지만 RCU는 탐색에 사용하는 원자적 연산으로 인해 MV-RLU 보다 저조한 성능을 보였다. 특히 대부분 읽기 연산 워크로드에서 MV-RLU는 RCU 보다 5.04배 더 높은 성능을 보였다.

참 고 문 헌

[1] Eyerman, S., & Eeckhout, L. Modeling Critical Sections in Amdahl's Law and Its Implications for Multicore Design. SIGARCH Comput. Archit. News, 38(3). doi:10.1145/1816038.1816011

[2] Boyd-Wickizer, S., Kaashoek, M. F., Morris, R., & Zeldovich, N. Non-scalable locks are dangerous. In Proceedings of the Linux Symposium.

[3] McKenney, P. E., & Walpole, J. What is RCU, fundamentally?. Linux Weekly News (LWN.net).

[4] Kim, J., Mathew, A., Kashyap, S., Ramanathan, M.K., & Min, C. Mv-rlu: Scaling read-log-update with multi-versioning. In Proceedings of the Twenty-Fourth International Conference on ASPLOS.

[5] Kim, C., Choi, E., Han, M., Lee, S., & Kim, J. Performance Analysis of RCU-Style Non-Blocking Synchronization Mechanisms on a Manycore-Based Operating System. Applied Sciences, 12(7), 3458.

[6] Luo. Chen, "LSM-based storage techniques: a survey", The VLDB Journal, 2020.

[7] Yeon, J., Kim, L., Han, Y., Lee, H. G., Lee, E., & Kim, B. S. JellyFish: A Fast Skip List with MVCC. In Proceedings of the 21st International Middleware Conference.

[8] Kobus, T., Kokociński, M., & Wojciechowski, P. T. (2022, April). Jiffy: a lock-free skip list with batch updates and snapshots. In Proceedings of the 27th ACM SIGPLAN.

[9] Herlihy, M., Lev, Y., Luchangco, V., & Shavit, N. A simple optimistic skiplist algorithm. 14th International Colloquium, SIROCCO 2007, Castiglioncello, Italy, June 5-8, 2007. Proceedings 14. Springer Berlin Heidelberg.