클라이언트-서버 DBMS 환경에서 콜백 잠금 기반 다중 버전의 활용

(Exploitation of Multi-Versions based on Callback Locking in a Client-Server DBMS Environment)

강 흠 근 † 민 준 기 †† 전 석 주 ††† 정 진 완 ††††
(Heum-Geun Kang) (Jun-Ki Min) (Seok-Ju Chun) (Chin-Wan Chung)

요 약 데이타 캐쉬를 관리하는 알고리즘의 효율성은 클라이언트에 데이타 캐쉬가 있는 시스템의 성능에 매우 큰 영향을 미친다. 클라이언트에 데이타 캐쉬가 있는 시스템에서는 서버에 추가 스페이스 부담 없이 한 데이타에 대해서 두 개의 버전을 유지 관리할 수 있다. 그 이유는 하나의 데이타가 동시에 서버의 버퍼와 클라이언트의 캐쉬에 저장될 수 있기 때문이다. 본 논문에서는 버전 기법을 활용하는 캐쉬 일관성 알고리즘인 Two Versions-Callback Locking(2V-CBL)을 제안하고, 실험을 통해서 2V-CBL과 기존에 가장 높은 성능을 보였던 Asynchronous Avoidance-based Cache Consistency(AACC)의 성능을 비교하였다. 실험 결과에 의하면 2V-CBL은 최소한 AACC와 같은 성능을 보이며 일부의 트랜잭션이 데이타를 수정하지 않고 읽기만 하는 경우에는 AACC 보다 훨씬 높은 성능을 보인다.

키워드 : 클라이언트/서버,캐쉬 일관성, 동시성 제어, 트랜잭션

Abstract The efficiency of algorithms managing data caches has a major impact on the performance of systems that utilize client-side data caching. In these systems, two versions of data can be maintained without an additional space overhead of the server by exploiting the replication of data in the server's buffer and clients' caches. In this paper, we present a new cache consistency algorithm employing versions: Two Versions-Callback Locking (2V-CBL). Our experimental results indicate that 2V-CBL provides good performance, particularly outperforms a leading cache consistency algorithm, Asynchronous Avoidance-based Cache Consistency, when some clients run only read-only transactions.

Key words : client/server, cache consistency, concurrency control, transactions

1. 서 론

클라이언트-서버 시스템(client-server system)에서 클라이언트에 데이타 저장을 위한 캐쉬(cache)를 두는 기법은 분산 시스템, 객체 지향 데이타 베이스 시스템 등의 분야에서 널리 사용되고 있다[1-4]. 이러한 시스템의 클라이언트는 필요한 데이타를 서버로부터 가져와 자신의 캐쉬에 저장한 다음 사용한다. 다음에 같은 데이타가 또 필요하면 다시 서버에서 가져오지 않고, 캐쉬에 저장되어 있는 데이타를 사용한다. 클라이언트에 있는 데이타 캐쉬의 활용은 데이타 접근 시간을 줄이며, 클라이언트 컴퓨터의 자원을 효율적으로 활용할 수 있도록 하여 서버의 병목 현상을 감소시킬 수 있다.

클라이언트에 데이타 캐쉬가 있는 시스템의 성능은 캐쉬에 저장된 데이타의 일관성을 유지시키는 알고리즘의 효율성에 의해서 많은 영향을 받는다. 캐쉬를 관리하는 방법은 인트라-트랜잭션(intra-transaction) 방법과 인터-트랜잭션(inter-transaction) 방법이 있다. 인트라-트랜잭션 방법에서는 캐쉬에 저장되어 있는 데이타를 하나의 트랜잭션 내에서만 사용하고 그 트랜잭션이 끝나면 캐쉬에 저장된 모든 데이타를 삭제한다. 그러므로, 트랜잭션이 시작할 때에는 캐쉬가 비어 있게 된다. 인터

· 본 연구는 정보통신부의 대학 IT연구센터(ITRC) 지원을 받아 수행되었습니다.
† 비 회 원 : 우송공업대학 전자계산과 교수
hgkang@wst.ac.kr
†† 비 회 원 : 한국과학기술원 전산학과
jkmin@islab.kaist.ac.kr
††† 정 회 원 : 서울교육대학교 컴퓨터교육과 교수
chunsj@snue.ac.kr
†††† 종신회원 : 한국과학기술원 전산학과 교수
chungcw@cs.kaist.ac.kr
논문접수 : 2004년 1월 27일
심사완료 : 2004년 6월 11일

-트랜잭션 방법에서는 캐쉬에 저장된 데이타가 트랜잭션의 범위와 관계없이 사용된다. 따라서, 이미 종료된 트랜잭션이 사용하던 데이타도 계속 캐쉬에 저장되어 있으므로 다음 트랜잭션이 캐쉬에 저장된 그 데이타를 사용할 수 있다. 인터-트랜잭션 방법은 캐쉬에 저장된 데이타의 일관성을 유지하기 위해 추가의 자료구조 및 알고리즘을 필요로 한다. 인터-트랜잭션 방법은 그러한 추가의 부담에도 불구하고 인트라-트랜잭션 방법에 비해 월등히 높은 성능을 보이는 것으로 알려졌다[5,6].

인터-트랜잭션 방법에서 필요한 캐쉬 일관성 알고리즘은 크게 회피 기반과 탐지 기반으로 구분할 수 있다[4]. 회피 기반 방법들은 일관성이 결여된 데이타의 사용을 원천적으로 방지한다. 반면에, 탐지 기반 방법에서는 일관성이 결여된 데이타가 사용될 수도 있다. 그러한 경우, 트랜잭션의 종료 시 탐지하여 해결한다. 대표적인 회피 기반 방법에는 ACBL(Adaptive Callback Locking)[7]과 AACC(Asynchronous Avoidance-Based Cache Consistency)[8] 등이 있고 대표적인 탐지 기반 방법에는 AOCC(Adaptive Optimistic Concurrency Control)[9,10]가 있다.

본 논문에서는 버전 기법을 활용하는 캐쉬 일관성 알고리즘을 제안하고 기존의 알고리즘과 제안한 알고리즘의 성능을 비교 분석한다. 객체의 크기는 페이지의 크기보다 작은 것으로 가정한다. 본 논문에서는 클라이언트의 데이타 캐쉬에 있는 데이타를 서버에 있는 같은 데이타의 한 버전으로 간주할 수 있다는 가정에서 연구를 시작하였다. 하나의 데이타에 대해 여러 버전을 관리하면, 데이타가 수정 중이어도, 다른 클라이언트가 그 데이타의 다른 버전을 사용할 수 있으므로 동시성을 크게 높일 수 있다. 다중 버전 기법은 하나의 데이타에 대해서 여러 버전을 관리해야 하기 때문에 앞에서 언급한 장점에도 불구하고 널리 사용되지는 않고 있다. 그러나, 클라이언트에 데이타 캐쉬를 두어서 데이타를 저장하는 구조에서는 추가 부담 없이 하나의 데이타에 대해서 두 개의 버전을 관리할 수 있다. 본 논문에서는 버전 기법을 활용하는 캐쉬 일관성 알고리즘인 2V-CBL(Two Version-Callback Locking)을 제안한다. 또한, 가장 높은 성능을 보이는 것으로 알려진 AACC 알고리즘과의 성능을 비교한다. 실험 결과에 의하면, 2V-CBL은 AACC에 비해 같은 시간 동안에 더 많은 트랜잭션을 처리하고 더 작은 수의 트랜잭션들이 교착 생태에 빠진다.

본 논문의 구성은 다음과 같다. 제2절에서는 관련 연구로 회피 기반 캐쉬 일관성 알고리즘인 ACBL과 AACC에 대해서 설명한다. 제3절에서는 본 논문에서 제안하는 2V-CBL에 대해 자세히 기술한다. 제4절에서는 실험에 사용될 시뮬레이션 모델, 실험 환경 및 실험 결과를 제시한다. 제5절에서는 실험 결과에 대해 논의를 하고 결론을 내린다.

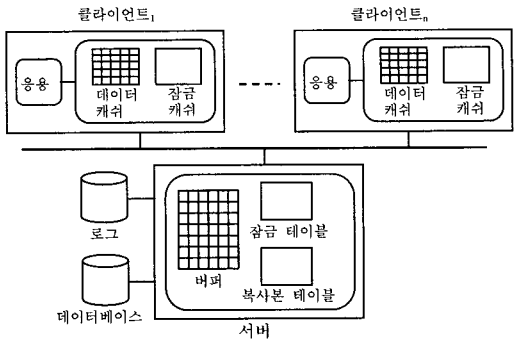

그림 1 클라이언트에 데이타 캐쉬가 있는 시스템의 일반적인 구조

2. 관련 연구

그림 1은 클라이언트에 데이타 캐쉬가 있는 시스템의 일반적인 구조이다. 그림 1에서 서버와 클라이언트는 근거리 통신망(Local-Area Network)을 통해서 메시지를 주고 받는다. 서버는 데이타의 일관성 유지의 책임을 갖는다. 이를 위해 데이타에 설정된 잠금(lock)을 저장하는 잠금 테이블과 어느 데이타가 어느 클라이언트의 데이타 캐쉬에 저장되어 있는지 저장하는 복사본 테이블을 사용한다. 클라이언트는 응용이 필요로 하는 데이타를 서버에게서 받아 데이타 캐쉬에 저장하여 사용한다. 또한 잠금 캐쉬에는 데이타 캐쉬에 저장된 데이타에 설정된 잠금을 저장한다.

2.1 ACBL(Adaptive Callback Locking)

ACBL은 잠금(lock)을 사용하여 데이타의 일관성을 유지한다[7,11-13]. 그리고, 데이타의 일관성 유지를 위해서 콜백(callback)을 사용한다. 한 클라이언트로부터 배타 잠금(exclusive lock) 설정을 요청 받은 서버는 수정될 데이타를 저장하고 있는 모든 클라이언트에게 콜백을 요청하여 수정될 데이타를 사용하지 않도록 한다. 콜백을 받은 클라이언트는 자신의 캐쉬에서 수정될 데이타를 삭제한다. ACBL은 잠금과 콜백을 할 때 상황에 따라 가변적으로 페이지 단위 또는 객체 단위로 한다. ACBL 알고리즘을 상황별로 간단히 기술하면 다음과 같다.

첫 상황은 클라이언트 C_1이 자신의 캐쉬에 없는 객체 O_1를 사용하려는 경우이다. 이 경우 C_1은 서버에게 O_1를 요청하는 메시지를 보낸다. 서버는 O_1과 O_1이 속한 페이지 P_1에 배타 잠금이 설정되어 있는지 확인한다. 배

타 잠금이 설정되어 있지 않으면, C_1에게 P_1를 보낸다. 만일 P_1에는 배타 잠금이 없지만 O_1에는 배타 잠금이 있으면 서버는 배타 잠금이 해제되기를 기다린 다음 C_1에게 P_1을 보낸다. 이때, P_1에 속한 객체들 중에서 배타 잠금이 설정된 객체는 삭제하고 전송한다. P_1에 배타 잠금이 설정되어 있는 경우 배타 잠금을 설정한 클라이언트에게 P_1에 설정된 배타 잠금을 P_1에 속한 객체들 중에서 수정하고 있는 객체에 대한 배타 잠금으로 전환할 것을 요청한다. 서버는 클라이언트로부터 잠금 전환의 결과를 받으면 O_1에 배타 잠금이 설정되는지 여부를 파악하여 앞서 설명한 대로 처리한다.

다음 상황은 클라이언트 C_1이 객체 O_1을 수정하려고 하는 경우이다. 이 경우 C_1은 O_1를 수정하기 전에 먼저 서버에게 O_1에 대한 배타 잠금을 요청한다. 서버는 O_1과 O_1이 속한 페이지 P_1에 배타 잠금이 없고, P_1을 캐쉬하고 있는 클라이언트들이 없을 때 C_1에게 P_1에 대한 배타 잠금을 보낸다. P_1과 O_1에 배타 잠금은 없지만 P_1을 캐쉬하고 있는 클라이언트가 있으면, 그 클라이언트에게 P_1를 캐쉬에서 삭제할 것을 요청한 다음 삭제가 되면 P_1에 대한 배타 잠금을 C_1에게 보낸다. P_1에 배타 잠금이 설정된 경우에는 배타 잠금을 설정한 클라이언트에게 P_1에 설정된 배타 잠금을 P_1에 속한 객체들 중 수정하고 있는 객체에 대한 배타 잠금으로 전환할 것을 요청한다. 요청이 처리된 결과를 받은 서버는 O_1에 배타 잠금이 설정되는지 파악한다. O_1에 배타 잠금이 설정되지 않으면 C_1에게 배타 잠금이 설정되는 개체를 삭제한 P_1을 보낸다. O_1에 배타 잠금이 설정되면 O_1에 설정된 배타 잠금이 해제될 때까지 기다린 다음 P_1를 C_1에게 보낸다.

2.2 AACC(Asynchronous Avoidance-Based Cache Consistency)

AACC는 비동기 방식을 적용하여 ACBL을 확장한 것이다[8]. ACBL에서는 객체를 수정하기 위해서는 먼저 서버에게서 배타 잠금을 얻어야 했다. 그러나, AACC에서는 서버에게 배타 잠금을 요청한 다음 결과를 기다리지 않고 바로 원하는 수정을 한다. 원래는 배타 잠금이 설정된 다음 수정을 해야 하지만 배타 잠금이 설정되는 시간 동안 미리 수정을 해두는 것이다. 하지만 트랜잭션이 종료될 때까지도 도착하지 않은 배타 잠금이 있을 수 있다. 그러한 경우에는 모든 배타 잠금이 도착한 후에 트랜잭션이 종료된다.

3. 새로운 캐쉬 동시성 제어 알고리즘 : 2V-CBL

본 절에서는 새로운 캐쉬 일관성 알고리즘인 2V-CBL을 소개한다. 2V-CBL에서는 서버에 있는 복사본과 클라이언트의 캐쉬에 있는 복사본을 한 데이타의 두 버전으로 취급한다. 하나의 데이타에 대해서 두 개의 버전이 관리되므로 하나의 클라이언트에 의해서 수정되고 있는 객체의 이전 버전을 다른 클라이언트가 읽을 수 있도록 허용하여 동시성을 높인다. 2V-CBL에서는 두 종류의 트랜잭션이 있다. 하나는 데이타를 읽고 수정하는 일반 트랜잭션(read-write transaction)이고 다른 하나는 데이타를 읽기만 하는 읽기 전용 트랜잭션(read-only transaction)이다.

표 1 잠금 호환 표

요청하는 잠금	설정된 잠금								
	S'	IS'	S	IS	X	IX	SIX	C	IC
S'	Y	Y	Y	Y	Y	Y	Y	N	N
IS'	Y	Y	Y	Y	Y	Y	Y	N	Y
S	Y	Y	Y	Y	N	N	N	N	N
IS	Y	Y	Y	Y	N	Y	Y	N	Y
X	Y	Y	N	N	N	N	N	N	N
IX	Y	Y	N	Y	N	Y	N	N	Y
SIX	Y	Y	N	Y	N	N	N	N	N
C	N	N	N	N	N	N	N	N	N
IC	N	Y	N	Y	N	Y	N	N	Y

3.1 동시성 제어

2V-CBL에서의 트랜잭션은 생성되고 소멸되는 과정에서 두 단계로 나뉘어 실행된다. 트랜잭션은 시작해서부터 정상적인 실행을 하는 동안에는 활동 단계이다. 트랜잭션이 실행이 끝나면 종료를 위한 준비에 들어간다. 이 때부터 종료까지가 종료 단계이다. 종료 단계에서 트랜잭션은 모든 배타 잠금을 종료(commit) 잠금으로 변경한다. 그리고, 트랜잭션들의 직렬성(serializability)을 위해 일반 트랜잭션이 수정한 객체의 이전 값을 읽은 읽기 전용 트랜잭션이 있으면, 그 읽기 전용 트랜잭션이 종료한 다음에 일반 트랜잭션은 종료할 수 있다.

읽기 전용 트랜잭션은 활동 단계의 트랜잭션이 수정하고 있는 객체의 이전 값을 읽을 수 있다. 그러나, 일반 트랜잭션이 읽기 전용 트랜잭션을 끝없이 기다라는 상황을 방지하기 위해서 종료 단계의 일반 트랜잭션이 수정한 객체를 읽기 위해서는 그 일반 트랜잭션이 종료하기를 기다리도록 하였다.

다음은 잠금의 종류와 각각에 대한 설명이다.

S' : 읽기 전용 트랜잭션이 데이타를 읽을 때 사용하는 잠금이다.

IS' : 읽기 전용 트랜잭션이 더 작은 단위의 데이타에 S' 잠금을 설정하려 할 때 사용하는 잠금이다.

S : 일반 트랜잭션이 데이타를 읽을 때 사용하는 잠금이다.

IS : 일반 트랜잭션이 더 작은 단위의 데이타에 S 잠금을 설정하려 할 때 사용하는 잠금이다.

X : 일반 트랜잭션이 데이타를 수정할 때 사용하는 잠금이다.

IX : 일반 트랜잭션이 더 작은 단위의 데이타에 **S** 또는 **X** 잠금을 설정하여 할 때 사용하는 잠금이다.

SIX : 일반 트랜잭션이 데이타를 읽으면서 일부는 수정하고자 할 때 설정하는 잠금이다.

C : 일반 트랜잭션의 활동 단계에서 설정한 **X** 잠금은 종료 단계에서 **C** 잠금으로 변환된다.

IC : 일반 트랜잭션의 활동 단계에서 설정한 **IX** 잠금은 종료 단계에서 **IC** 잠금으로 변환된다.

표 1은 잠금 간의 호환성 여부를 표시하고 있다. 각 열은 어떤 트랜잭션이 특정 데이타에 설정 한 잠금의 종류이고 각 행은 다른 트랜잭션이 그 데이타에 설정하려는 잠금의 종류이다. "Y"는 설정하려는 잠금과 기존에 설정된 잠금이 호환됨을 의미한다. "N"은 잠금이 호환되지 않음을 의미하고 이 경우에는 기존의 잠금이 해제될 때까지 요청한 잠금은 설정될 수 없다.

잠금 기반의 동시성 제어 알고리즘에서는 교착상태가 발생할 수 있다. 교착상태는 시간 종료나 교착상태 탐지 알고리즘에 의해서 탐지될 수 있다. 비록 교착상태 탐지 알고리즘은 시간 종료 방법에 비해서 CPU시간을 더 소모하지만, WFG(Waits-for graph)를 점진적으로 생성하면 필요한 CPU 시간을 매우 적게 할 수 있다 [14]. 2V-CBL에서는 교착 상태 탐지를 위해서 교착 상태 탐지 알고리즘을 사용한다. 트랜잭션의 잠금 요청이 다른 잠금을 기다리는 트랜잭션 때문에 허용되지 않으면 교착 상태가 발생하였는지 탐지한다.

3.2 트랜잭션 실행

설명을 간단히 하기 위해서, **IS**, **IX**, **SIX**, **IC** 등과 같은 의도 잠금(Intent Lock)에 관한 것은 제외하고 설명한다. 다음은 트랜잭션의 실행 과정을 상황 별로 나누어 설명한 것이다.

- **읽기 전용 트랜잭션의 데이타 요청**: 클라이언트 C_1이 캐쉬에 없는 객체 O_1을 읽기 위해서는 먼저 서버에게 O_1을 요청하는 메시지를 보낸다. 메시지를 받은 서버는 O_1 또는 O_1이 속한 페이지 P_1에 배타 잠금이 설정되어 있는지 확인한다.
  - 만일 P_1과 P_1에 속하는 모든 객체에 배타 잠금이 설정되어 있지 않으면 서버는 C_1에게 P_1를 보낸다.
  - 만일 P_1과 O_1에는 배타 잠금이 설정되어 있지 않지만 P_1에 속하는 다른 객체에 배타 잠금이 설정되어 있으면 서버는 그 배타 잠금이 설정된 객체들을 삭제한 P_1를 C_1에게 보낸다.
  - 만일 P_1에 클라이언트 C_2에 의해 배타 잠금이 설정되어 있으면 서버는 C_2에게 P_1에 설정된 배타 잠금을 P_1에 속한 객체들 중에서 수정되고 있는 객체에 대한 배타 잠금으로 변환할 것을 요청한다. 서버로부터 메시지를 받은 C_2는 페이지 수준의 배타 잠금을 객체 수준의 배타 잠금으로 변환한다. C_2는 서버에게 P_1에 속한 객체들 현재 실행 중인 트랜잭션이 수정한 객체들의 OID(object identifier)를 보내서 배타 잠금을 설정하고 P_1에 설정된 배타 잠금은 해제한다. C_2로부터 메시지를 받은 서버는 O_1이 C_2가 수정한 객체이면 C_1의 요청의 처리를 지연시키고 아니면 C_2가 수정한 객체를 P_1에서 삭제한 다음 C_1에게 보낸다.
  - 만일 O_1에 C_2에 의해서 배타 잠금이 설정되어 있으면 C_2로부터 C_1으로의 간선(edge)을 WFG에 입력한 다음 WFG에 순환경로(cycle)가 있는지 확인한다.
    * WFG에 순환경로가 없고 C_2가 종료 단계에 있으면 서버는 C_1의 요청의 처리를 C_2가 종료할 때까지 지연시킨다.
    * WFG에 순환경로가 없고 C_2가 활동 단계에 있으면 서버는 P_1를 C_1에게 보낸다.
      O_1의 값은 지금 다른 클라이언트에 의해서 수정되고 있다. 현재의 트랜잭션에서는 수정되기 전의 값을 사용하지만 다음 트랜잭션에서는 수정된 후의 값을 사용해야 한다. 그러므로, 서버는 C_1에게 현재 실행 중인 트랜잭션이 끝나면 P_1를 캐쉬에서 삭제할 것을 통지한다.
    * WFG에 순환경로가 있으면 서버는 C_1에게 현재 실행 중인 트랜잭션을 중단시킬 것을 지시한다.

  C_1은 서버로부터 받은 P_1를 캐쉬에 저장하고 P_1에 있는 O_1을 사용한다.
- **일반 트랜잭션의 데이타 요청**: 일반 트랜잭션의 데이타 요청은 읽기 전용 트랜잭션의 데이타 요청과 비슷하게 처리되며 요청한 객체에 배타 잠금이 설정되어 있는 경우만 다르게 처리된다. 그러므로, 요청한 객체에 배타 잠금이 설정된 경우만 설명한다. 설명하지 않은 부분은 읽기 전용 트랜잭션의 경우와 같다. 클라이언트 C_1이 서버에게 O_1를 요청한 경우 클라이언트 C_2에 의해서 O_1에 배타 잠금이 설정되어 있을 수 있다. 이 경우 서버는 C_1으로부터 C_2로의 간선을 WFG에 입력하고 WFG에 순환경로가 생성되는지 검사한다.
  - WFG에 순환경로가 없으면 서버는 단순히 C_2가 종료할 때가지 C_1의 데이타 요청 처리를 지연시킨다.
  - WFG에 순환경로가 있으면 서버는 C_1에게 실행 중인 트랜잭션의 종료를 지시한다.
- **일반 트랜잭션의 배타 잠금 요청**: 클라이언트 C_1가 자신의 캐쉬에 저장되어 있는 객체 O_1을 수정하려면 서버에게서 O_1에 대한 배타 잠금을 얻어야 한다. C_1은 서버에게 O_1에 대한 배타 잠금을 요청하는 메시지를

보내고 O_1을 수정하기 시작한다. 서버는 받은 메시지를 아래와 같이 처리한다.

- O_1에 배타 잠금이 설정되어 있으면 배타 잠금이 해제된 후에 C_1에게 O_1이 속한 페이지 P_1에 대한 배타 잠금을 보낸다.
- 클라이언트 C_2에 의해서 P_1에 배타 잠금이 설정되어 있는 경우 C_2에게 페이지 수준의 배타 잠금을 객체 수준의 잠금으로 전환할 것을 요청한다. O_1에 배타 잠금이 설정되면 배타 잠금이 해제된 후에 C_1에게 O_1이 속한 페이지 P_1에 대한 배타 잠금을 보낸다. O_1에 배타 잠금이 설정되지 않으면 C_1에게 O_1에 대한 배타 잠금을 보낸다.
- O_1과 P_1에 배타 잠금이 설정되어 있지 않고 P_1이 다른 클라이언트의 캐쉬에 없는 경우 서버는 P_1에 대한 배타 잠금을 C_1에게 보낸다.
- O_1과 P_1에 배타 잠금이 설정되어 있지 않고 P_1이 다른 클라이언트의 캐쉬에 저장되어 있는 경우 서버는 그 클라이언트에게 P_1를 캐쉬에서 삭제하라는 메시지를 보낸다. 서버에게서 메시지를 받은 클라이언트 C_2는 만일 P_1를 사용하지 않았으면 캐쉬에서 P_1를 삭제한 다음 서버에게 이를 알린다. 만일 C_2가 P_1에 있는 O_1를 제외한 다른 객체를 사용했으면 캐쉬에 있는 P_1에서 그 객체들을 삭제하고 이를 서버에게 알린다. 만일 C_2에서 실행 중인 읽기 전용 트랜잭션이 O_1를 사용했으면 C_2는 서버에게 이를 알린다. 만일 C_2에서 실행 중인 일반 트랜잭션이 O_1를 사용했으면 C_2는 실행 중인 트랜잭션이 종료한 다음 캐쉬에서 P_1를 삭제하고 이를 서버에게 알린다. 모든 클라이언트로부터 답장이 왔으면 서버는 다음과 같이 처리한다.
  * 만일 모든 클라이언트가 자신들의 캐쉬에서 P_1를 삭제하였으면 서버는 P_1에 대한 배타 잠금을 C_1에게 보낸다.
  * 어떤 클라이언트가 자신의 캐쉬에서 P_1를 삭제하지 않고 대신 O_1를 삭제한 경우 서버는 O_1에 대한 배타 잠금을 C_1에게 보낸다.
  * C_2가 O_1를 사용하고 있음을 알렸으면 서버는 C_1으로부터 C_2로의 간선을 WFG에 입력하고 WFG에 순환경로가 생성되는지 확인한다. 순환경로가 없으면 서버는 C_1에게 P_1에 대한 배타 잠금을 보낸다. 만일 순환경로가 있으면 서버는 C_1에게 실행 중인 트랜잭션의 종료를 지시한다.

• **일반 트랜잭션 종료 처리**: 일반 트랜잭션은 활동 단계를 끝내고 종료 단계로 가기 전에 배타 잠금에 대한 요청의 답장을 모든 받아야 된다. 아직 도착하지 않은 것이 있으면 기다린다. 모든 답장이 도착하였으면 그 트랜잭션이 수정한 페이지들을 서버에게 보내면서 트랜잭션의 종료를 요청한다. 트랜잭션 종료 요청을 받은 서버는 그 트랜잭션 보다 먼저 종료되어야 하는 다른 트랜잭션이 있는지 확인한다.

- 만일 먼저 종료되어야 하는 트랜잭션이 있으면 서버는 종료를 요청한 트랜잭션의 종료를 지연시킨다.
- 만일 먼저 종료되어야 하는 트랜잭션이 없으면 서버는 종료하기를 원하는 트랜잭션의 종료를 허용한다.

3.3 알고리즘의 증명

알고리즘의 증명은 두 부분으로 이루어진다. 첫 부분은 2V-CBL에 의해서 생성되는 히스토리가 직렬화가 될 수 있음을 보인다. 다음 부분에서는 2V-CBL이 생성하는 직렬화 가능한 히스토리가 1SR(one-copy serializable)임을 보인다. 다중 버전 직렬화 이론의 가장 기본적인 정리는 트랜잭션들의 선후 관계를 표시하는 그래프에 순환경로가 없으면 다중 버전 트랜잭션들의 수행을 직렬화할 수 있다는 것이다[15]. 그러므로, 트랜잭션들이 2V-CBL에 따라 실행하면 항상 직렬화가 된다는 것을 증명하기 위해서는 트랜잭션들에 의해서 생성되는 WFG에 순환경로가 생성되지 않음을 보이는 것이다.

다중 버전 히스토리인 H는 트랜잭션 집합인 $T = \{T_1, \cdots, T_n\}$에 속한 트랜잭션들의 연산의 선후 관계를 표현한다. 트랜잭션 연산에는 다음과 같은 세 가지 연산이 있다. $r_i[x_j]$, $w_i[x_i]$, c_i가 그들이다. $r_i[x_j]$는 트랜잭션 T_i가 트랜잭션 T_j가 수정한 x의 값을 읽는다는 의미이다. $w_i[x_i]$는 T_i가 x의 새로운 값을 저장한다는 의미이다. c_i는 T_i의 종료를 의미한다.

다중 버전 히스토리 H가 주어졌을 때, MVSG(H)는 직렬화 그래프 SG(H)에 다음의 조건을 만족하는 간선을 추가한 것이다 [16].

1. 모든 데이타 x에 대해서, MVSG(H)는 x을 수정한 모든 트랜잭션들 간에 전 순서 (total order)를 갖는다. 전 순서는 $\ll_x$로 표현한다.
2. 모든 데이타 x에 대해서, 만일 $T_i \ll_x T_j$이고 T_k가 T_i가 수정한 x의 값을 읽었으면 MVSG(H)에는 T_k로부터 T_j로의 간선이 존재한다. 그렇지 않고 $T_j \ll_x T_i$인 경우 MVSG(H)는 T_j로부터 T_i로의 간선을 갖는다.

Theorem 1 2V-CBL에 의해서 생성된 히스토리 H는 직렬화될 수 있다.

Proof: 히스토리 H를 위한 MVSG(H)에 순환경로가 생기지 않음을 보이면 히스토리 H가 직렬화될 수 있음이 증명된다. MVSG(H)에는 세 종류의 간선이 있다 [15]. 첫번째인 $T_i \rightarrow T_j$는 reads-from관계를 나타내는 것으로 T_i가 수정한 x의 값을 T_j가 읽음을 의미한다. T_i가 종료해야 T_i가 수정한 데이타를 다른 트랜잭션이

읽을 수 있으므로 $c_i < r_j[x_i]$. 트랜잭션은 모든 읽기와 수정이 끝난 후에 종료 연산을 수행한다. 따라서, $r_j[x_i] < c_j$. 그러므로, $c_i < c_j$.

두 번째 간선인 $T_i \rightarrow T_j$는 버전의 순서를 표시하는 간선이다. 즉, $x_i \ll_x x_j$ (i와 j는 같지 않다). T_i는 x에 배타 잠금을 설정하고 x의 값을 쓴다. 그리고, 종료 연산 c_i를 수행하고 나서 x에 설정된 배타 잠금을 해제한다. T_i가 배타 잠금을 해제해야 T_j가 x에 배타 잠금을 설정할 수 있다. T_j는 배타 잠금을 설정한 다음에 x의 값을 쓴다. T_j는 모든 쓰기와 읽기가 끝나야 종료 연산 c_j를 수행한다. 따라서, $c_i < c_j$.

세 번째 간선 종류인 $T_k \rightarrow T_j$는 $x_i \ll_x x_j$와 $r_k[x_j]$를 의미한다. 여기에서, i, j, k는 서로 같지 않다. 만일 T_k가 읽기 전용 트랜잭션이면, $r_k[x_i]$가 실행되는 동안에 T_j는 $w_j[x_j]$를 실행할 수 있다. 그러나, T_k가 종료한 후에야 T_j는 종료 연산인 c_j를 실행할 수 있다. 그러므로, $c_k < c_j$. 만일 T_k가 일반 트랜잭션이면, $w_j[x_j]$는 $r_k[x_i]$와 충돌을 일으킨다. $w_j[x_j]$는 T_k에 의해서 x에 설정된 읽기 잠금이 해제될 때까지 지연된다. T_k는 종료 연산 c_k가 수행된 다음에 설정한 모든 잠금을 해제한다. 그리고, c_j는 $w_j[x_j]$가 수행된 다음에 수행된다. 그러므로, $c_k < c_j$.

위의 논의에 의해서 MVSG(H)에 있는 모든 간선은 해당 트랜잭션의 종료 순서로 연결됨을 알 수 있다. 종료 순서는 히스토리 H에 포함되고 히스토리는 정의에 의해서 순환경로가 없기 때문에 MVSG(H) 역시 순환경로가 존재하지 않는다. □

Theorem 2 2V-CBL에 의해서 생성된 히스토리 H는 1SR이다.

Proof: 다음의 조건을 만족하는 직렬화될 수 있는 다중 버전 히스토리는 1SR이다[15]. 모든 i, j, x에 대해서 만일 트랜잭션 T_j가 수정한 x의 값을 트랜잭션 T_i가 읽었다면 i = j이거나 T_j는 T_i에 앞선 모든 트랜잭션들 중에서 마지막으로 x의 값을 수정한 트랜잭션이다. Theorem 1에 의하면 2V-CBL이 생성하는 히스토리 H는 직렬화될 수 있다. 트랜잭션 T_i, T_j, T_k가 있고, $c_j < c_k < c_i$ 라고 하자. T_j와 T_k는 x의 값을 수정하고 T_i는 x의 값을 읽는다고 가정하자. 2V-CBL에서는 한 데이타 아이템에 대해서 최대 두 개의 값을 저장한다. 하나는 가장 최근에 종료한 트랜잭션이 수정한 값이고 다른 하나는 실행 중인 트랜잭션이 수정 중인 값이다. 2V-CBL에서는 가장 최근에 종료한 트랜잭션이 수정한 값만을 저장하며 그전에 종료한 트랜잭션이 수정한 값은 저장하지 않는다. 따라서, 2V-CBL에서는 T_j가 수정한 값을 저장하고 있지 않으므로 T_i는 T_j가 수정한 x의 값을 읽을 수 없다. T_i는 가장 최근에 종료한 T_k가 수정한 x의 값을 읽기 때문에 위의 조건을 만족시키며, 따라서, 2V-CBL이 생성하는 히스토리 H는 1SR이다. □

4. 실 험

4.1 기본 시스템 모델과 작업부하 모델

그림 2는 실험에서 사용할 시뮬레이션 모델을 보여주고 있다. 그림 2의 모델에서 서버나 클라이언트에 큐가 하나씩 있다. 컴퓨터에 있는 큐는 그 컴퓨터에 도착한 메시지들을 저장한다. 네트워크에도 큐가 있어서 네트워크를 통해서 전송될 메시지들을 저장한다. 서버에 부착된 디스크도 각기 큐를 갖는다. 디스크의 큐는 그 디스크에 요청된 디스크 입출력 명령을 저장한다.

그림 2의 모델에서 클라이언트는 캐쉬 관리자, 동시성 제어 관리자, 클라이언트 관리자, 트랜잭션 생성자 등으로 구성된다. 캐쉬 관리자는 LRU 페이지 교체 기법으로 캐쉬를 관리한다. 동시성 제어 관리자는 일관성을 유지하기 위한 기능을 수행한다. 클라이언트 관리자는 트랜잭션의 수행과 메시지를 보내고 받는 작업들을 조정한다. 트랜잭션 생성자는 한번에 하나씩 트랜잭션을 생성한다. 하나의 트랜잭션은 객체에 대한 읽기와 쓰기의 흐름이다.

그림 2의 모델에서 서버는 버퍼 관리자, 동시성 제어 관리자, 서버 관리자, 디스크 관리자 등으로 구성된다. 버퍼 관리자는 버퍼 관리 및 데이타의 일관성 유지를 위한 기능을 수행한다. 동시성 제어 관리자는 잠금 테이블을 사용하여 잠금에 대한 정보를 관리하며, 복사본 테

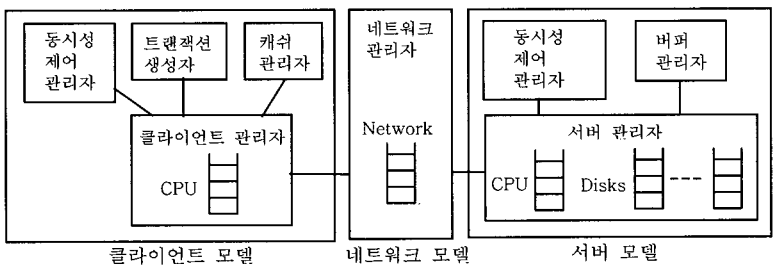

그림 2 클라이언트에 데이타 캐쉬가 있는 시스템의 모델

이블을 사용하여 어떤 페이지들이 어느 클라이언트의 캐쉬에 저장되어 있는지 그 위치를 저장하고 관리한다. 또한, 알고리즘에 따라, WFG를 관리하기도 한다. 서버가 클라이언트에게 페이지 삭제 명령이나 잠금 변환 명령을 보낼 때는 복사본 테이블을 참조하여 어느 클라이언트에게 보낼지 판단한다. 서버 관리자는 클라이언트의 요청들을 바탕으로 하여 서버의 동작을 조정한다.

네트워크를 통해서 메시지를 전송하는데 걸리는 시간은 CPU 시간과 메시지가 네트워크 선을 통과하는 시간으로 구성된다. CPU 시간은 메시지의 크기에 관계없이 일정하게 필요한 시간과 메시지의 크기에 따라 달라지는 시간으로 구성된다.

표 2는 실험에서 사용하는 연산들의 처리 시간을 보여주고 있다. ClientCPU는 클라이언트 CPU의 처리 속도를 나타낸다. ServerCPU는 서버 컴퓨터의 CPU 속도를 나타낸다. 연산을 처리하는데 걸리는 시간은 그 연산의 처리에 필요한 명령의 수와 CPU의 처리 속도에 의해서 결정된다.

본 논문에서는 Sh-Hotcold라는 작업 부하를 사용하여 캐쉬 일관성 알고리즘의 성능을 측정하였다. Sh-Hotcold의 데이타 공유 수준은 클라이언트의 캐쉬에 데이타를 저장하는 응용들의 일반적인 데이타 공유 수준이다[8]. 실험에 사용된 데이타베이스는 하나의 클라이언트가 주로 사용하는 독립 영역, 여러 클라이언트들이 공유하는 공유 영역, 그리고 나머지 부분 및 다른 클라이언트의 독립 영역인 기타 영역으로 구성된다. 독립 영역은 클라이언트 마다 하나씩 존재한다. 클라이언트는 데이타 사용의 80%을 자신의 독립 영역에 있는 데이타를 사용하고, 10%을 공유 영역에 있는 데이타를 사용하다. 나머지 10%는 기타 영역의 데이타를 사용한다.

표 3은 실험에서 사용한 작업 부하의 매개 변수들을 표시하고 있다. TransSize는 하나의 트랜잭션이 실행되는 동안 접근하는 객체의 수를 나타낸다. 트랜잭션은 접근하는 모든 객체들을 읽고 그 중에서 일부를 수정한다. WrtProb는 하나의 객체를 수정할 확률을 나타낸다. 예를 들어, TransSize가 200이고 WrtProb가 10%이면 하나의 트랜잭션은 200개의 객체를 읽고 그 중에서 20개를 수정한다. 하나의 페이지에는 여러 개의 객체가 저장될 수 있다. ClusterSize는 하나의 트랜잭션이 수행되는 동안 하나의 페이지에서 몇 개의 객체가 사용되는지를

표 2 시스템 매개변수

매개 변수	설 명	값
ClientCPU	클라이언트의 CPU가 초당 처리하는 명령의 수	100 MIPS
ServerCPU	클라이언트의 CPU가 초당 처리하는 명령의 수	250 MIPS
ClientCacheSize	클라이언트에 있는 데이타 캐쉬의 크기	375 pages
ServerBufferSize	서버에 있는 버퍼의 크기	750 pages
Server Disks	서버에 있는 디스크의 수	2 disks
DiskAccessTime	디스크 접근 시간	1600 microsecs/Kbyte
NetworkBandwidth	네트워크 대역폭	80 Mbps
NumClients	클라이언트의 수	20
PageSize	페이지의 크기	4,096 bytes
DatabaseSize	데이타베이스의 페이지 수	1500 pages
ObjectsPerPage	한 페이지에 저장되는 객체의 수	20 objects
FixedMsgInst	메시지 전송에서 고정적으로 소요되는 명령의 수	6000 instructions
PerByteMsgInst	메시지 전송에서 가변적으로 소요되는 명령의 수	7.17 inst./byte
ControlMsgSize	제어 메시지의 크기	256 bytes
LockInst	잠금을 설정하고 해제하는데 소요되는 명령의 수	300 instructions
RegisterCopyInst	페이지의 위치를 저장하는데 소요되는 명령의 수	300 instructions
DiskOverheadInst	디스크 입출력을 유발하기 위한 명령의 수	5000 instructions

표 3 작업 부하 매개변수

매개 변수	설 명	설 정
TranSize	한 트랜잭션에서 사용하는 객체의 수	200, 400
ClusterSize	페이지 당 사용되는 객체의 수	8 - 18
WrtProb	객체가 수정될 확률	20%, 40%
PerReadObjInst	객체를 읽는데 필요한 CPU 명령의 수	5000
PerWrtObjInst	객체를 수정하는데 필요한 CPU 명령의 수	10000
SharedRgnSize	공유 영역의 페이지 수	50
ClientRgnSize	독립 영역의 페이지 수	50

나타낸다. TransSize가 200이고 ClusterSize가 10이면, 하나의 트랜잭션은 20개의 페이지를 사용한다. PerReadObjInst는 하나의 객체를 읽는데 소요되는 CPU 명령의 개수를 나타낸다. PerWrtObjInst는 하나의 객체를 수정하는데 소요되는 CPU 명령의 개수를 나타낸다. SharedRgnSize는 모든 클라이언트들이 공유하는 영역의 크기를 나타낸다. ClientRgnSize는 특정 클라이언트가 주로 사용하는 독립 영역의 크기를 나타낸다.

4.2 실험 결과

이 절에서는 2V-CBL과 AACC의 성능을 앞 절에서 설명한 작업 부하를 사용하여 비교한 결과를 보인다. 실험에서 클라이언트들은 읽기 전용 트랜잭션만 실행하는 클라이언트와 일반 트랜잭션만 실행하는 클라이언트로 구분된다. 본 논문에서는 읽기 전용 트랜잭션만을 실행하는 클라이언트의 수를 0부터 20으로 변화시켜가면서 실험을 하였다. 초당 종료되는 트랜잭션의 수는 성능을 측정하는 주된 항목이다. 부수적으로, 정상적으로 종료되지 못하고 중간에 중단되는 트랜잭션의 수도 성능 측정에 사용하였다. 모든 트랜잭션이 읽기 전용 트랜잭션이거나 모든 트랜잭션이 일반 트랜잭션인 경우에는 2V-CBL과 AACC 사이에 차이는 없다. 그러므로, 위와 같은 경우에 대해서는 실험 결과를 언급하지 않았다.

그림 3은 TransSize가 200일 때의 트랜잭션 처리량을 보여준다. 왼쪽의 그래프는 WrtProb가 20%인 경우이고 오른쪽 그래프는 WrtProb가 40%인 경우이다. 두 그래프는 읽기 전용 트랜잭션만을 실행하는 클라이언트의 수가 증가할수록 트랜잭션 처리량도 증가함을 보여주고 있다. 2V-CBL은 AACC보다 더 많은 트랜잭션을 처리하는 것을 볼 수 있으며 객체를 수정하는 비율이 큰 경우 그 차이가 더 커진다. 2V-CBL에서는 일반 트랜잭션이 수정 중인 데이타를 읽기 전용 트랜잭션이 읽을 수 있고, 읽기 전용 트랜잭션이 읽고 있는 데이타를 일반 트랜잭션이 수정할 수 있기 때문에 잠금에 의한 트랜잭션들의 지연이 AACC에 비해서 적다. 따라서, 2V-CBL에서는 AACC에서 보다 더 많은 트랜잭션들이 처리될 수 있다.

그림 4는 TransSize가 200일 때 트랜잭션이 정상적으로 종료하지 못하고 중간에 중단되는 수를 보여주고 있다. 왼쪽의 그래프는 WrtProb가 20%인 경우이고 오른쪽 그래프는 WrtProb가 40%인 경우이다. 두 그래프는 읽기 전용 트랜잭션만을 실행하는 클라이언트의 수가 증가할수록 중단되는 트랜잭션의 수가 증가함을 보여주고 있다. 2V-CBL은 AACC에 비해서 중단되는 트랜잭션의 수가 적음을 알 수 있다. 2V-CBL은 AACC에 비해서 잠금에 의해서 트랜잭션들이 충돌하는 확률이 적으며, 이에 따라, 트랜잭션들이 교착 상태에 빠질 확률이 적다. 따라서, 2V-CBL의 환경에서 수행되는 트랜잭션들은 정상적으로 종료되지 못하고 중간에 중단될 확률이 AACC에 비해서 적다.

그림 5는 읽기 전용 트랜잭션의 크기가 200이고 일반 트랜잭션의 크기가 400인 경우의 트랜잭션 처리량을 보여 주고 있다. 왼쪽의 그래프는 WrtProb가 20%인 경

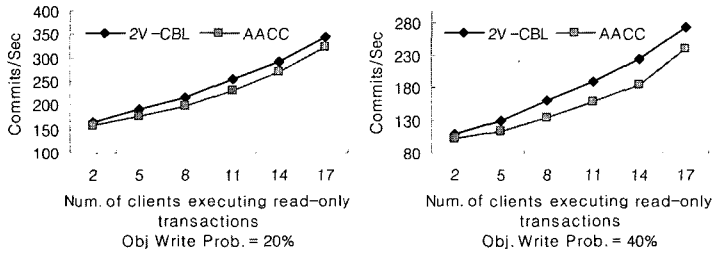

그림 3 TranSize = 200인 경우의 트랜잭션 처리량

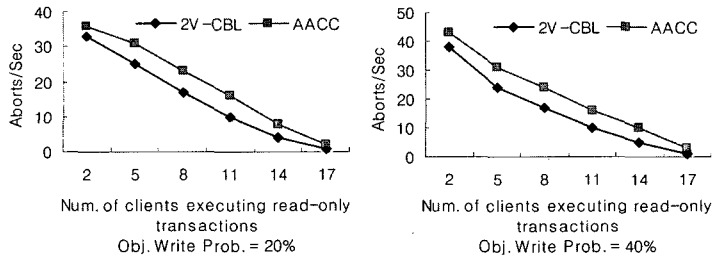

그림 4 TranSize = 200인경우의 트랜잭션 중단 비율

우이고 오른쪽 그래프는 WrtProb는 40%인 경우이다. 이러한 환경에서, 2V-CBL은 AACC보다 훨씬 많은 트랜잭션을 처리함을 알 수 있다.

일반 트랜잭션의 크기가 읽기 전용 트랜잭션의 크기보다 큰 경우, 2V-CBL에서는 일반 트랜잭션이 수정하고 있는 데이타를 읽기 전용 트랜잭션이 읽은 다음 일반 트랜잭션과의 간섭 없이 종료하므로 AACC에 비해서 훨씬 많은 트랜잭션을 처리할 수 있다. 일반 트랜잭션의 wrtProb가 증가하면 읽기 전용 트랜잭션과 충돌이 발생할 확률이 높지만 반대로 일반 트랜잭션의 실행 시간이 길어져서 충돌이 발생하기 전에 읽기 전용 트랜잭션이 종료할 가능성도 높아진다.

그림 6은 읽기 전용 트랜잭션의 크기가 400이고 일반 트랜잭션의 크기가 200인 경우의 트랜잭션 처리량을 보여 주고 있다. 왼쪽의 그래프는 WrtProb가 20%인 경우이고 오른쪽은 WrtProb가 40%인 경우이다. 이 환경에서도 2V-CBL은 AACC에 비해서 더 많은 트랜잭션을 처리함을 알 수 있다. 읽기 전용 트랜잭션만을 실행하는 클라이언트의 수가 2에서 8인 경우에는 처리되는 트랜잭션의 수에 큰 변화가 없다. 그 이유는 읽기 전용 트랜잭션의 크기가 일반 트랜잭션의 크기에 비해 크기 때문이다.

읽기 전용 트랜잭션이 읽은 데이타를 일반 트랜잭션이 수정하면 일반 트랜잭션이 종료하기 전에 읽기 전용 트랜잭션이 종료되어야 한다. 따라서, 일반 트랜잭션은 긴 읽기 전용 트랜잭션이 종료하기를 기다릴 가능성이 높다. 읽기 전용 트랜잭션은 일반 트랜잭션이 수정하고 있는 데이타도 읽을 수 있으므로 잠금에 의한 지연의 가능성은 적다. 그러므로, 2V-CBL에서는 일반 트랜잭션 보다 읽기 전용 트랜잭션들이 우대된다. 이러한 이유로 해서, 읽기 전용 트랜잭션들이 많이 실행되면 AACC와 2V-CBL의 트랜잭션 처리량은 증가한다. 읽기 전용 트랜잭션만 실행되는 경우에는 두 방법 사이에 차이가 없다. 따라서, 일반 트랜잭션들의 수가 아주 적어지면 두 방법 간의 차이가 적어진다.

5. 논의 및 결론

2V-CBL의 장점 중의 하나는 정상적으로 종료하지 못하고 실행되는 도중에 중단되는 트랜잭션의 수가 적다는 것이다. 그림 7의 시나리오 1은 AACC에서 교착상태가 발생되어 트랜잭션이 중단되는 상황이다. 그러나, 2V-CBL라면 트랜잭션들이 중단되지 않고 정상적으로 종료된다. 시나리오 1에서 일반 트랜잭션은 읽기 전용 트랜잭션의 데이타 A에 대한 읽기 잠금이 해제되기를 기다리고, 읽기 전용 트랜잭션은 일반 트랜잭션의 데이타 B에 대한 쓰기 잠금이 해제되기를 기다린다. 그러나, 2V-CBL인 경우에는 읽기 전용 트랜잭션이 일반 트랜잭션이 설정한 쓰기 잠금의 해제를 기다릴 필요가 없

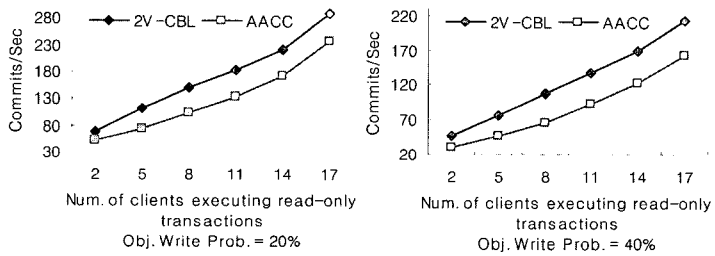

그림 5 읽기 전용 트랜잭션의 크기 = 200, 일반 트랜잭션의 크기 = 400인 경우의 트랜잭션 처리량

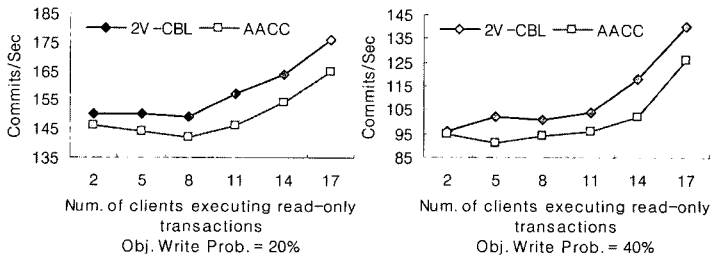

그림 6 읽기 전용 트랜잭션의 크기 = 400, 일반 트랜잭션의 크기 = 200인 경우의 트랜잭션 처리량

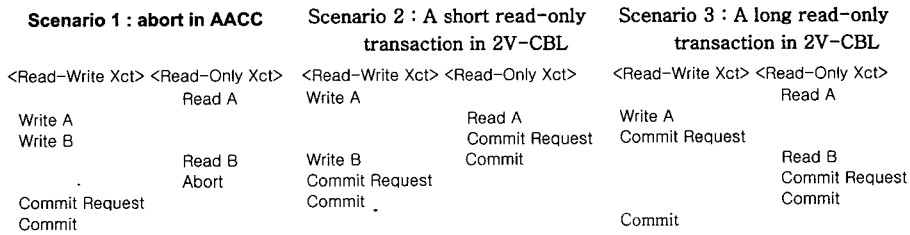

그림 7 트랜잭션 실행 시나리오

다. 다만, 읽기 전용 트랜잭션이 일반 트랜잭션이 수정하기 전의 데이타 A의 값을 읽었기 때문에 읽기 전용 트랜잭션이 종료한 후에야 일반 트랜잭션은 종료할 수 있다.

읽기 전용 트랜잭션의 크기와 일반 트랜잭션의 크기 비율은 2V-CBL의 성능에 영향을 줄 수 있다. 읽기 전용 트랜잭션의 크기가 일반 트랜잭션보다 작을 경우 2V-CBL은 AACC 보다 더 많은 읽기 전용 트랜잭션을 처리할 수 있다. 그림 7의 시나리오 2는 일반 트랜잭션과의 간섭 없이 일반 트랜잭션이 수정하는 객체를 사용하는 읽기 전용 트랜잭션이 실행될 수 있음을 보여주고 있다. 그러나, 읽기 전용 트랜잭션의 크기가 큰 경우에는 일반 트랜잭션이 종료하기 위해서 읽기 전용 트랜잭션의 종료를 기다려야 하는 경우가 발생할 수 있다. 그림 7의 시나리오 3은 이것을 보여주고 있다.

본 논문에서는 캐쉬 일관성을 유지하기 위한 알고리즘에서 버전 기법의 활용이 성능에 미치는 영향을 연구하였다. 이를 위하여, 버전 기법을 활용하는 캐쉬 일관성 알고리즘인 2V-CBL을 제안하였다. 2V-CBL은 읽기 전용 트랜잭션이 사용하는 읽기 잠금과 일반 트랜잭션이 사용하는 읽기 잠금을 구분한다. 읽기 전용 트랜잭션의 읽기 잠금은 일반 트랜잭션의 쓰기 잠금과 호환이 된다. 따라서, 읽기 전용 트랜잭션은 일반 트랜잭션이 수정하고 있는 객체를 읽을 수 있다. 그러나, 일반 트랜잭션이 사용하는 읽기 잠금은 쓰기 잠금과 호환되지 않는다. 본 논문에서는 2V-CBL과 가장 성능이 높은 알고리즘으로 알려진 AACC를 실험을 통해 비교하였다.

실험 결과에 의하면 2V-CBL은 AACC에 비해 더 높은 트랜잭션 처리량을 갖으며 더 낮은 수의 트랜잭션이 중단된다. 특히, 읽기 전용 트랜잭션의 크기가 작은 경우에는 2V-CBL은 AACC에 비해 훨씬 큰 성능을 보인다.

참 고 문 헌

[1] C. Osthoff, C. Bentes, D. Ariosto, M. Mattoso, and C.L. Amorim, "Evaluating the DSMIO Cache-Coherence Algorithm in Cluster-Based Parallel ODBMS," 8th International Conference on OOIS, pp. 286-297, 2002.

[2] Z. Tari, H. Hamidjadja, "A CORBA Cooperative Cache Approach with Popularity Admission and Routing Mechanism," Thirteenth Australasian Database Conference Australasian Database Conference, 2002.

[3] Z. Tari, Q.T. Lin, and H. Hamidjaja, "Cache Management in CORBA Distributed Object Systems," IEEE Concurrency, vol. 8, num. 3, pp. 48-55, 2000.

[4] M.J. Franklin, M.J. Carey, and M. Livny, "Transactional Client-Server Cache Consistency: Alternatives and Performance," ACM Transactions on Database Systems, vol. 22, num. 3, pp. 315-363, 1997.

[5] M.J. Carey, M.J. Franklin, M. Livny, and E.J. Shekita, "Data Caching Tradeoffs in Client-Server DBMS Architecture," ACM SIGMOD Conference, pp. 357-366, June, 1991.

[6] Y. Wang and L. Rowe, "Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture," ACM SIGMOD Conference, pp. 367-376, June, 1991.

[7] M.J. Carey, M.J. Franklin, and M. Zaharioudakis, "Fine-Grained Sharing in a Page Server OODBMS," ACM SIGMOD Conference, pp. 359-370, June, 1994.

[8] M.T. Ozsu, K. Voruganti, and R.C. Unrau, "An Asynchronous Avoidance-Based Cache Consistency Algorithm for Client Caching DBMSs," Proceedings of the 24th VLDB Conference, pp. 440-451, 1998.

[9] B. Liskov, M. Castro, L. Shrira, and A. Adya, "Providing Persistent Objects in Distributed Systems," Proc. 13th European Conference on Object-Oriented Programming, pp. 230-257, 1999.

[10] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari, "Efficient Optimistic Concurrency Control Using Loosely Synchronous Clocks," ACM SIGMOD Conference, pp. 23-34, June, 1995.

[11] G. Hughes, "Scalability of Avoidance-Based Transactional Cache Consistency," Proc. 6th IDEA Workshop, pp. 51-56, 1999.

[12] M. Zaharioudakis, M.J. Carey, and M.J. Franklin, "Adaptive, Fine-Grained Sharing in a Client-

Server OODBMS: A Callback-Based Approach," ACM Transactions on Database Systems, vol. 22, num. 4, pp. 570-627, 1997.
[13] M. Zaharioudakis and M.J. Carey, "Hierarchical, Adaptive Cache Consistency in a Page Server OODBMS," IEEE Transactions on Computers, vol 47, num. 4, pp. 427-444, 1998.
[14] E. Panagos and A. Biliris, "Synchronization and Recovery in a Client-Server Storage System," The VLDB Journal, vol. 6, num. 3, pp. 209-223, 1997.
[15] P.A. Bernstein, V. Hadzilacos, and N. Goodman, "Concurrency Control and Recovery in Database Systems," Addison-Wesley, 1987.
[16] D. Agrawal and S. Sengupta, "Modular Synchronization in Distributed, Multiversion Databases: Version Control and Concurrency Control," IEEE Transactions on Knowledge and Data Engineering, vol. 5, num. 4, pp. 126-137, 1993.

강 홈 근

1992년 한국과학기술원 전산학과(석사) 1992년~1995년 한국전자통신연구원 연구원. 1995년~현재 한국과학기술원 전산학과 박사과정. 관심분야는 데이타웨어하우스, OLAP, 분산시스템

민 준 기

정보과학회논문지 : 데이타베이스
제 31 권 제 1 호 참조

전 석 주

1987년 경북대학교 전자공학과 컴퓨터공학 전공(학사). 1989년 경북대학교 대학원 전자공학과 컴퓨터공학전공(석사). 2002년 한국과학기술원 정보및통신공학과(박사). 1989년~1995년 현대중공업 중앙연구소 주임연구원. 1997년~2004년 안산1대학 인터넷정보과 조교수. 2004 년~현재 서울교육대학교 컴퓨터교육과 전임강사. 관심분야는 데이타 마이닝, 데이타웨어하우스와 OLAP, 멀티미디어 데이타베이스

정 진 완

정보과학회논문지 : 데이타베이스
제 31 권 제 1 호 참조