대체 버전을 이용한 펌 실시간 데이터베이스 동시성 제어 방법

홍 동 권†

요 약

펌 실시간 데이터베이스 시스템에서 수행되는 펌 실시간 트랜잭션은 트랜잭션이 마감시간을 넘길 경우 그 트랜잭션의 의미가 없어지므로 제한시간을 넘기는 즉시 시스템에서 제거된다. 이런 트랜잭션들을 지원하는 펌 실시간 데이터베이스 시스템에서 데이터베이스의 성능을 향상시키기 위한 여러 가지 방법들이 연구되고 있다. 지금까지 연구된 방법들은 잠금 방식을 사용하거나 또는 낙관적 방식을 사용하고 있다. 그러나 잠금 방식과 낙관적 방법에서 사용되는 방식은 시스템의 부하에 따라서 서로 다른 특성을 보이고 있다. 따라서 본 논문에서는 잠금 방식과 낙관적 방식의 장점만을 채택한 *보류된 정지/재개 재시작* 방법을 제안하고 이 방법을 동시성 제어에 적용한 ***대체 버전을 이용한 펌 실시간 동시성 제어 방법***을 소개한다. 이 논문에서 제안하는 동시성 제어 방법은 우선 순위에 의한 데이터 접근 충돌이 발생할 경우 트랜잭션의 *즉시 재시작* 버전과 *정지* 버전을 동시에 유지함으로써 트랜잭션이 마감시간 안에 수행을 마칠 수 있도록 한다. 끝으로 이 논문에서는 제안하는 알고리즘이 직렬 가능한 스케줄을 생성함을 보이고, 모의실험을 통하여 기존의 방법과 성능을 비교하여 우수성을 증명한다.

Alternative Version Concurrency Control Method for firm real-time database systems

Dong-Kweon Hong†

ABSTRACT

Firm real-time transactions on firm real-time database systems are discarded when they miss their deadlines, as there is no value to completing them after they miss their deadlines. Several approaches that exploit the semantics of firm deadlines to improve the performance of firm real-time database systems have been proposed in the literature. They are based on locking or optimistic concurrency control. The performance comparisons of the two approaches differ with systems load. In this paper, we develop a novel policy termed stop/resume deferred restart policy, and a concurrency control algorithm based on the policy (termed Alternative Version Concurrency Control). When conflicts (due to priority) occur, our algorithm maintains the immediately restarted version as well as the stopped version of a transaction in order to use one of the two to meet the firm deadline. At last we show that our policy generates serializable schedules and show that our algorithm performs better than the traditionally used method for wide ranges of the system load for firm deadline transaction.

※ 본 연구는 1997년도 계명대학교 비사연구 기금으로 이루어 졌음.
† 정 회 원 : 계명대학교 컴퓨터·전자공학부 교수
논문접수 : 1997년 7월 30일, 심사완료 : 1998년 4월 27일

1. 서 론

트랜잭션을 빠르게 또는 빨리 수행하라고 하는 것은 매우 추상적이다. 얼마나 빠르게라는 것이 명시되어 있지 않기 때문이다. 실시간 개념은 빠르게와 일치하는 개념이라기보다 실제로 어떤 시스템에서 수행되는 작업 중에서 구체적으로 명시된 작업들이 수행 시작 된 후부터 언제까지 종료되어야 한다는 명백한 요구를 나타낸다. 시스템 내에서 수행되는 작업들이 시작된 후 언제까지 끝나야 한다는 것은 일반적으로 해당 작업의 마감시간(deadline)[16]으로 표시되며 주기적으로 반복되는 일이 아닌 경우 그 작업이 시작되면서 부여된다. 또 해당 작업이 종료시점을 지키지 못한 경우 시스템 내에서 처리하는 방법과 그 시스템의 효용성에 따라 실시간 시스템을 하드, 펌, 소프트 실시간 시스템으로 분류한다[6,7,8].

실시간 데이터베이스 시스템은 실시간 시스템에 포함(embedded)되어 있거나 또는 독립적으로 사용될 수도 있는 데이터베이스의 새로운 연구 분야이다. 시스템 내의 모든 작업이 전부 마감시간 이내에 끝나야 하는 하드 실시간 시스템으로는 데이터의 공유라는 데이터베이스 특성으로 인하여 잘 적용되지는 않지만 펌 또는 소프트 실시간 분야로 많이 연구되고 있다. 이 분야는 지난 몇 년 동안 급격히 연구되다가 최근에는 응용 부분으로의 적용 기술 부족과 데이터 웨어하우징, 데이터 마이닝, 멀티미디어 자료 처리 등의 유행에 밀려 그 성장 기세가 많이 약화되었다. 하지만 SQL3에서도 지원되는 트리거 개념의 원천인 능동 데이터베이스[18], 가상 현실 등의 기술과 함께 곧 연구가 다시 활발히 진행될 분야이다. 특히 국방, 원자로, 항공 등과 같이 많은 양의 데이터를 사용하면서 마감시간 내에 필요한 처리를 해야 하거나, 사람이 직접 참여하기 힘든 우주, 해저, 위험한 장소의 제어 등에 사용될 실시간 데이터베이스는 실시간 시스템의 한 분야로 꾸준히 연구되고 있으며, 능동 데이터베이스, 시간지원 데이터베이스 등과의 공동 연구도 많이 진행되고 있다[18].

펌 실시간 데이터베이스는 네트워크 서비스 데이터베이스[17,18,20], 주식 시장 응용[19] 등에서 많이 사용되고 있는 것으로 마감시간 내에 종료하지 못한 작업은 그 의미가 없어지므로 마감시간을 종료하는 즉시 데이터베이스 시스템에서 제거된다. 예를 들어, 컨베이어벨트에 물체가 올려져서 일정한 속도로 지나가고 한쪽 구석에 장치된 카메라에 의해 그 물체의 이미지를 포착한 후 포착한 이미지를 데이터베이스에서 찾아서 해당 물체에 따라서 적절한 액션 (로봇 팔에 의해 필요한 장소로 이동)을 취하는 동시에 어떤 물체를 처리했다는 통계 기록을 유지하는 무인 공장 자동 시스템의 일부분을 살펴보자[18]. 카메라에 포착된 이미지를 데이터베이스에서 찾는 시간은 컨베이어벨트에 실려 있는 다음 물체가 카메라에 포착되는 시간으로 제한된다. 만약 다음 물체가 컨베이어벨트에 실려 나타날 때까지 이미지를 데이터베이스에 찾아서 필요한 액션을 취한 후 그 결과를 기록하는 트랜잭션이 수행되지 못하면 그 트랜잭션은 즉시 취소되고 다음에 나타난 물체에 대한 서비스를 즉시 시작하여야 한다. 이런 실시간 데이터베이스 시스템에 대한 연구 분야는 동시성 제어[1,2,4,7,10,13,14,15], 우선 순위 부여 방식[8,9,11,12,15], 회복 기능, 주기억장치 상주형 데이터베이스[20], 실시간 인덱스 기법 등 다양하지만 현재 가장 집중적으로 연구되고 있는 분야는 동시성 제어와 우선 순위 방식을 포함하는 실시간 데이터베이스 트랜잭션 스케줄링에 대한 연구이다[6]. 이 분야는 수행되는 작업에 대한 모든 정보를 가정하고 있는 기존의 많은 실시간 시스템의 스케줄링[16]과는 달리 부정확한 수행 시간 정보와 데이터베이스의 무결성을 동시에 수용해야 하는 많은 어려움을 내재하고 있다. 이 논문에서는 기존의 펌 실시간 데이터베이스의 트랜잭션 스케줄링 기법에 대한 자세한 분석을 통하여 그 장단점과 문제점 등을 자세히 분석하고 장점들을 효과적으로 수용할 수 있는 새로운 방법을 제시하고 모의실험을 통하여 새로운 방법의 우수성을 검증한다.

1.1 기존 연구에 대한 조사

펌 실시간 데이터베이스에서 제안된 트랜잭션 스케줄링 기법으로는 2PL-HP와 낙관적 동시성 제어 방법이 있는데, 이들 각각에 대하여 살펴보면 다음과 같다. 2PL-HP(2 Phase Locking with High Priority[1])는 데이터베이스 시스템에서 널리 사용되는 2단계 잠금 방식에 높은 우선 순위를 가장 우대하는 기법을 적용한 방식이다. 이 방식에서는 낮은 우선 순위를 가진 트랜잭션이 높은 우선 순위를 가진 트랜잭션이 현재 보유하고 있는 데이터를 서로 충돌되는 잠금 모드로 접근하려

고 할 때 2단계 잠금 방식에 따라 낮은 우선 순위를 가진 트랜잭션을 블록시킨다. 반대로 높은 우선 순위를 가진 트랜잭션이 낮은 우선 순위를 가진 트랜잭션이 현재 보유하고 있는 데이터를 서로 충돌되는 잠금 모드로 요구했을 경우 시간적으로 급한 트랜잭션 즉 우선 순위가 높은 트랜잭션을 먼저 수행시키기 위해 낮은 우선 순위의 트랜잭션을 롤백시킨다. 이 때 높은 우선 순위의 트랜잭션은 요구한 데이터를 가지게 되고 계속 수행을 한다. 한편 낮은 우선 순위의 트랜잭션은 롤백 후에 다시 처음부터 수행을 시작한다.

데이터에 접근하는 순간에 즉시 그 잠금 모드 충돌을 검사하는 2단계 잠금 방식과는 달리 낙관적 동시성 제어 방법(Optimistic Concurrency Control)은 트랜잭션이 정상 종료(commit)하는 시점에 그 트랜잭션을 정상 종료해도 좋은지를 검사하는 검사 단계(validation phase)를 가지고 있다. 이 검사 단계에 만약 여러 트랜잭션들이 서로 충돌될 경우를 해결하는 OCC-commit, OCC-priority-abort, OCC-priority-wait, OCC-Wait-50 등의 방법이 제안되었다[7,8,13, 15].

실시간 데이터베이스와 일반 데이터베이스에서 잠금 방식과 낙관적 방식의 성능 비교는 여러 논문에서 많이 비교되었으며 그 결과를 요약하면 아래와 같다.

1) 일반 데이터베이스 환경에서 잠금 방식은 낙관적 방식보다 우수한 성능을 보이고 있다[3].
2) 소프트 실시간 데이터베이스에서는 잠금 방식을 이용한 2PL-HP가 낙관적 방식보다 마감시간을 어기는 트랜잭션의 수가 적다[1,2,13].
3) 펌 실시간 데이터베이스에서는 낙관적 방식이 2PL-HP보다 우수하다고 하는 논문도 있고[15], 반대로 2PL-HP가 거의 모든 환경에서 우수하다고 하는 상반된 결과를 보이고 있다[13].

위의 결과는 시스템이 일반, 소프트, 또는 펌으로 완전히 구분된다고 가정한 실험적인 결과이다. 그러나 실제 데이터베이스 운영 환경은 일반, 소프트, 펌 실시간 데이터베이스로 완전히 분리되는 것이 아니라 때로는 소프트와 펌이 함께 혼합되기도 하고 또 일반, 펌이 함께 혼합되기도 한다[18]. 이 같은 상황을 고려한다면 실시간 데이터베이스는 일반 트랜잭션, 소프트, 그리고 펌 실시간 트랜잭션의 혼합을 쉽게 수용할 수 있는 형태로 구성되어져야 한다[10]. 따라서 이 논문에서는 일반 데이터베이스 환경과 소프트 실시간 데이터베이스를 잘 지원하는 잠금 방식을 근간으로 하여 펌 실시간 데이터베이스도 잘 지원하는 동시성 제어 방법을 제시한다.

2. 기본적인 착상

잠금 방식이 우수한 성능을 보이는 일반 데이터베이스와는 달리 펌 실시간 데이터베이스에서는 낙관적 방식이 2PL-HP에 버금가는 또는 상당히 가능성을 보이는 방법으로 평가되고 있다. 일반 데이터베이스와 소프트 실시간 데이터베이스에서 항상 낙관적 방식 보다 뛰어난 성능을 보이던 2PL-HP가 갑자기 펌 실시간에서 그 우위를 지키지 못하는 이유는 2PL-HP의 *낭비적인 재시작*(wasted restart)과 *낭비적인 기다림*(wasted wait)이라는 문제점의 영향이 낙관적 방식의 큰 단점인 *낭비적인 수행*(wasted execution)보다 더 커졌기 때문이다.

낭비적인 재시작은 높은 우선 순위의 트랜잭션이 낮은 우선 순위의 트랜잭션이 가지고 있는 데이터를 사용하기 위해 낮은 우선 순위의 트랜잭션을 롤백, 재시작하도록 한 후 자기 자신은 종료시점을 어겨서 시스템에서 제거되는 경우 생긴다. 곧 종료시점을 지키지 못해 시스템에서 제거될 트랜잭션이 낮은 우선 순위의 트랜잭션을 롤백, 재시작 하도록 하는 경우 발생하는 현상이다.

낭비적인 기다림은 낮은 우선 순위의 트랜잭션이 높은 우선 순위의 트랜잭션이 가지고 있는 데이터를 기다리면서 블록되어 있는 동안 높은 우선 순위의 트랜잭션이 종료시점을 지키지 못해 시스템에서 제거되는 경우 발생한다. 낮은 우선 순위의 트랜잭션이 곧 시스템에서 제거될 높은 우선 순위의 트랜잭션이 정상적으로 종료하기를 기대하면서 기다릴 경우 발생하는 현상이다.

낭비적인 수행은 트랜잭션이 끝까지 수행된 후 마지막 순간의 검사 단계에서 다른 트랜잭션들과 충돌이 생겨 다시 재시작 하는 경우 발생한다. 이 현상은 데이터의 충돌을 데이터에 접근하는 순간에 검사하지 않고 마

하며 순간까지 기다리는 낙관적 방식에서 발생한다.

2PL HP에서는 만약 잠금을 요구한 트랜잭션의 우선 순위가 현재 잠금을 가지고 있는 트랜잭션의 우선 순위보다 높을 경우 현재 잠금을 가지고 있는 트랜잭션을 즉시 재시작(롤백을 포함) 시킨다. *즉시 재시작* 방법은 높은 우선 순위를 가진 트랜잭션이 정상적으로 종료할 확률이 높은 경우 유리하다. 소프트 실시간 트랜잭션의 정상적인 종료 확률은 95% 이상이다. 하지만 펌 실시간 트랜잭션의 경우 정상 부하상태에서도 80% 이하의 정상 종료 확률을 가지고 있다. 이런 상태에서 즉시 재시작 방법을 사용하는 것은 낭비적인 재시작 때문에 전체 성능에 악영향을 줄 수도 있다. 따라서 이 논문에서는 낙관적 방식처럼 보류된 갱신 방식(deferred update)을 채택하여 높은 우선 순위의 트랜잭션이 낮은 우선 순위의 트랜잭션을 즉시 재시작 시키지 않으면서 계속 진행할 수 있는 여지를 만들어 놓는다. 이런 상황하에서 만약 높은 우선 순위의 트랜잭션이 낮은 우선 순위의 트랜잭션과 충돌이 생길 경우 낮은 우선 순위의 트랜잭션을 잠시 정지시켰다가 높은 우선 순위의 트랜잭션이 비정상적으로 종료할 경우 낮은 우선 순위의 트랜잭션을 다시 재개시킨다. 이 방법을 우리는 정지/재개 방식이라 칭한다. 여기서 트랜잭션을 정지시킨다는 것은 다음과 같은 의미를 갖는다.

트랜잭션 정지(Transaction stopping)는 높은 우선 순위의 트랜잭션이 낮은 우선 순위의 트랜잭션이 가지고 있는 데이터에 접근할 때 발생하는 낮은 우선 순위의 트랜잭션의 블로킹을 의미한다. 이 용어는 낮은 우선 순위의 트랜잭션이 높은 우선 순위의 트랜잭션이 보유하고 있는 데이터에 접근하려고 하다가 블로킹되는 현상과는 다르다. 블로킹과는 달리 트랜잭션 정지는 트랜잭션 수행 중 아무 곳에서나 가능한 것이 아니다. 즉 트랜잭션이 만약 임계영역 (critical section) 안에 있다면 그 트랜잭션은 임계영역을 빠져 나온 직후 정지된다.

2.1 충돌 해결방식의 비교

즉시 재시작 방식과 보류 재시작 방식의 장단점을 비교하기 위하여 4가지의 경우를 (그림 1,2,3,4)에서 고려해 보았다. 각각의 경우에 서로 다른 3가지 방식, 1)낙관적 보류 재시작 방식(DR-OCC), 2) 즉시 재시작 방식(IR), 3) 정지/재개 재시작 방식(DR-SR)을 비교하였다.

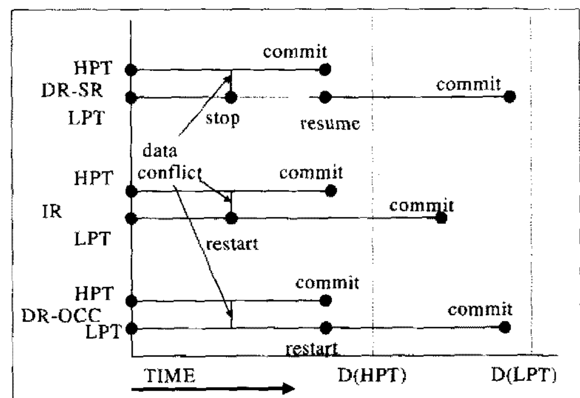
1) DR-OCC: 이 방식은 낙관적 방식에서 사용하는 방법이다. 트랜잭션이 정상 종료할 때 데이터베이스의 일치성을 해치는 트랜잭션들을 찾아서 재시작 시키는 방식을 사용한다.

2) IR: 이 방식은 2PL-HP에서 사용하는 방법이다. 데이터베이스의 일치성을 위해서 데이터에 접근하는 순간에 충돌을 검사하고 만약 충돌이 되면 즉시 재시작을 하는 방식이다.

3) DR-SR: 이 방식은 이 논문에서 새로 제안하는 방법이다. 데이터의 충돌은 데이터에 접근하는 순간에 검사하지만 즉시 재시작 대신에 보류 재시작을 적용한 방법이다. 높은 우선 순위의 트랜잭션이 낮은 우선 순위의 트랜잭션과 데이터 충돌을 일으켰을 경우 낮은 우선 순위의 트랜잭션은 정지되고 그 재시작은 높은 우선 순위의 트랜잭션이 정상 종료하는 순간까지 보류된다. 만약 그 중간에 높은 우선 순위의 트랜잭션이 마감시간을 지키지 못해 비정상적으로 종료하는 경우 낮은 우선 순위의 트랜잭션은 다시 재개된다.

아래의 경우들은 트랜잭션 수행 결과에 따라 분류한 경우로, 각각의 경우에서 위 3가지 방법들의 성능을 살펴보면 다음과 같다.

1) case 1: 높은 우선 순위의 트랜잭션과 낮은 우선 순위의 트랜잭션이 함께 마감시간 이전에 정상적으로

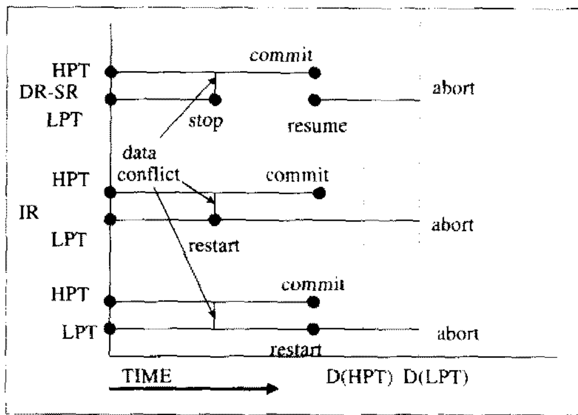


HPT:High Priority Transaction, LPT: Low Priority Transaction, D(HPT): Deadline of HPT

(그림 1) 모든 트랜잭션이 정상적으로 종료
(Fig. 1) Both transactions finished successfully within their deadlines

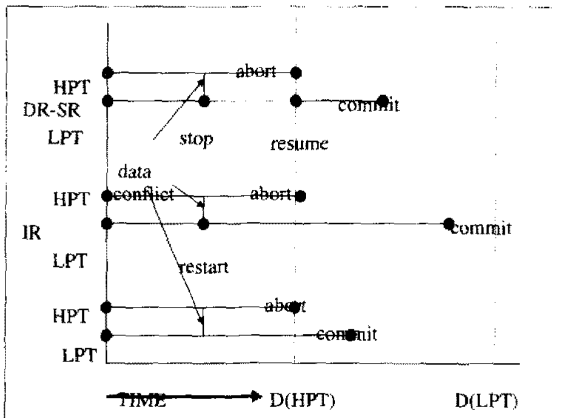
종료하는 경우이나 (그림 1), IR의 경우가 낮은 우선 순위의 종료 시간이 가장 빠르게 나타나고 있다. 3가지 방법 중 IR이 이 경우에 가장 좋은 방식이라는 것이 명백하게 나타나고 있다.

2) case 2: 높은 우선 순위의 트랜잭션은 마감시간 내에 정상으로 종료를 하고 낮은 우선 순위의 트랜잭션은 마감시간을 지키지 못해서 비정상적으로 끝나는 경우이다 (그림 2). 이 경우 시스템 자원을 가장 적게 낭비한 DR-SR과 낮은 우선 순위의 실질적 서비스 시간 (effective service time)이 가장 긴 IR 방식이 좋은 방식으로 평가된다.



(그림 2) HPT는 정상 종료, LPT는 비정상 종료
(Fig. 2) HPT completed successfully, LPT missed deadlines

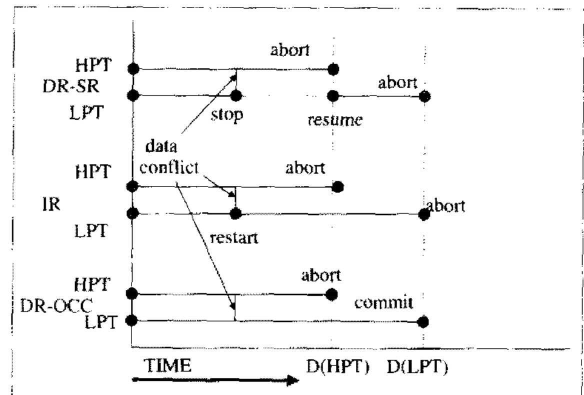
3) case 3: 높은 우선 순위의 트랜잭션은 마감시간 내에 종료하지 못하고 낮은 우선 순위의 트랜잭션은 정상적으로 종료를 하는 경우이다(그림 3). 이 경우는



(그림 3) HPT는 비정상 종료, LPT는 정상 종료
(Fig. 3) HPT missed, LPT completed successfully

며, HPT의 낭비적인 재시작을 명백하게 설명해주고 있다. 따라서 IR이 가장 나쁜 방식으로 평가되며 DR-SR과 DR-OCC가 좋은 방식으로 평가된다.

4) case 4: 높은 우선 순위의 트랜잭션과 낮은 우선 순위의 트랜잭션이 함께 마감 시간을 지키지 못하는 경우이다 (그림 4). 시스템의 자원을 낭비하는 측면에서 본다면 DR-SR이 가장 좋은 방식이다.



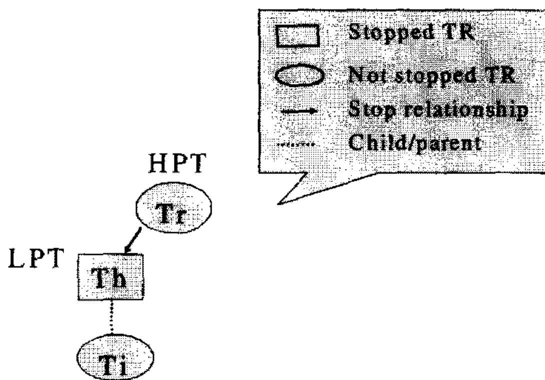
(그림 4) HPT와 LPT가 함께 비정상적으로 종료
(Fig. 4) Both of HPT and LPT missed their deadlines

위의 결과들을 분석해 보면 각 경우마다 IR과 DR-SR을 적절히 사용하는 것이 한 방식만 계속해서 사용하는 것 보다 더 좋은 방법이라는 것을 알 수 있다. 이 사실을 바탕으로 이 논문에서는 어떤 트랜잭션의 즉시 재시작 버전과 정지 버전을 동시에 유지하는 AVCC (Alternative Version Concurrency Control) 방법을 제시한다. AVCC는 우선 순위를 빠른 마감시간 우선 방식 (EDF - Earliest Deadline First) 방식을 사용하고 보류된 갱신 방식을 사용한다.

3. 새로운 방법에 대한 알고리즘

DR-SR 방식과 IR 방식의 장점을 함께 취하기 위해서 이 논문에서는 각각의 방식에 의한 트랜잭션 버전을 동시에 유지하는 방법을 제안한다. 여기서 트랜잭션 버전이란 트랜잭션 프로그램이 수행되기 위해서 만들어지는 트랜잭션 프로세스를 말한다. 만약 데이터를 요구하는 높은 우선 순위의 트랜잭션(Tr)이 현재 데이터를 가지고 있는 낮은 우선 순위의 트랜잭션(Th)과 충돌을 일으킬 경우 먼저 Th를 정지시키고 동시에 즉시 시작

된 트랜잭션 버전(Ti)을 만들어 수행시킨다. 즉 AVCC에서는 낮은 우선 순위의 트랜잭션에 대한 정지 버전과 즉시 재시작 버전을 함께 유지하다가 상황에 따라 유리한 쪽을 택하는 방식이다. 이것이 AVCC의 기본 착상이다. 정지된 버전 Th는 다시 재개될 때까지 약간의 프로세스 정보를 저장하기 위한 공간(메모리)만 필요할 뿐 전혀 수행 시간을 요구하지 않는다. 반면에 Ti는 Tr과 충돌을 일으켰던 시점까지 계속 진행될 수 있으며 만약 Tr이 정상 종료하면 Th의 처음부터 시작하는 것이 아니라 이미 어느 정도 진행된 Ti를 계속 진행시킨다. 반대로 Tr이 마감시간 내에 종료하지 못하면 더 이상 Ti는 필요하지 않으므로 시스템에서 제거하고 정지되어 있던 Th를 다시 재개시킨다. 이 방법은 저장점(savepoint)을 사용하지 않는 부분 롤백(partial rollback) 방법으로 간주될 수도 있다. 여기서 Ti는 Tr이 정상 종료하는 경우 Th가 부분적으로 롤백한 것과 같은 결과를 가지고 있다.

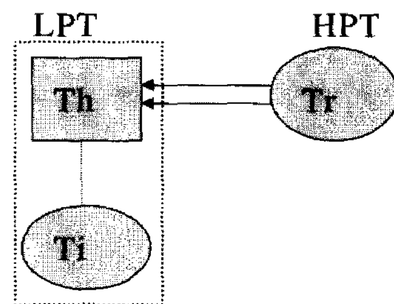


(그림 5) Tr의 보류 재시작 버전과 즉시 재시작 버전
(Fig. 5) Deferred restart and immediate restart version of Tr

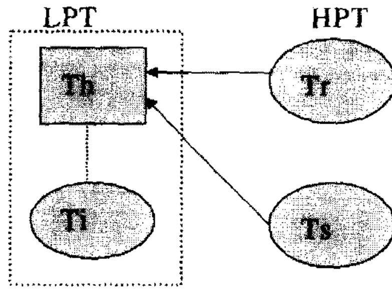
AVCC는 위와 같이 정지 버전과 재시작 버전을 함께 유지함으로써 낭비적인 재시작과 낭비적인 수행을 줄일 수 있다. 또 AVCC에서는 Tr이 시스템에 존재하는 동안은 Th와 Ti의 수행 경로(execution path)가 똑같다. 그 이유는 공유되는 데이터에 대해서는 Th가 계속 잠금을 가지고 있으므로 Ti외의 다른 트랜잭션이 그 값에 접근하여 변경할 수 없으며 외부 입력 값은 Th의 마감시간 동안에는 바뀌지 않는 것이 의미상으로 타당하기 때문이다. 정지된 버전 Th와 재시작 버전 Ti는 부모/자식(parent/child) 관계를 가지기 때문에 Ti는 Th의 마감시간과 우선 순위를 부여받으며 또 Th가 보유한 데이타에 자유롭게 접근할 수 있다. 한 트랜잭션의 수행 중 그 트랜잭션의 즉시 재시작 버전은 또 다른 높은 우선 순위의 트랜잭션에 의해 정지될 수 있으며 이때 새로운 즉시 재시작 버전이 다시 형성된다. 따라서 AVCC는 보류된 재시작 버전과 즉시 재시작 버전에 의한 트리를 형성하며 각 트랜잭션은 어떤 시점에 여러 개의 보류 재시작 버전과 오직 1개의 즉시 재시작 버전을 가질 수 있다.

3.1 알고리즘

위의 기본 착상을 바탕으로 조금 더 복잡한 경우를 살펴보자. 먼저 (그림 6)의 높은 우선 순위를 가진 트랜잭션(Tr)이 정지된 버전(Th)과 1번 이상 충돌을 일으킬 경우를 살펴보자. 이 경우에 아무 문제도 일으키지 않는다. 만약 Tr이 정상적으로 종료하면 우리는 Ti를 채택하고 정지된 버전 Th를 시스템에서 제거하면 된다. 그와는 반대로 Tr이 마감시간을 지키지 못하면 Tr은 Ti와 함께 시스템에서 즉시 제거되고 정지된 버전 Th를 재개시키면 된다. (그림 7)의 경우는 높은 우선 순위의 트랜잭션 Tr이 먼저 Th를 정지 시켜서 Ti가 새로 만들어진 후 또 다른 트랜잭션 Ts (Ts는 Th보다 우선 순위가 높다)가 Th와 충돌을 일으킨 경우이다. 이 경우에 Tr과 Ts가 전부 마감 시간을 지키지 못해 시스템에서 제거되면 정지된 버전 Th를 다시 재개시킨다. 그렇지 않은 경우(즉 적어도 1개가 정상 종료하면) 정지된 버전 Th를 시스템에서 제거하고 Ti를 계속 진행시킨다.



(그림 6) 한 트랜잭션과 1번 이상 충돌
(Fig. 6) Conflicts with the same transactions more than once



(그림 7) 다른 트랜잭션과 2번 이상 충돌
(Fig. 7) Conflicts with different transactions more than once

AVCC의 중요한 함수는 Lock_acquire, Commit, Discard이다. 이 함수들은 공통으로 사용되는 공유 잠금 테이블에 접근하며 각 트랜잭션들은 2단계 잠금을 사용한다. 잠금 테이블은 객체 식별자 (OID), 잠금 모드, 잠금을 기다리는 트랜잭션의 수, 현재 잠금이 허가된 트랜잭션의 리스트 등을 가지고 있다. 그리고 각 트랜잭션들은 〈표 1〉과 같은 변수들을 유지하고 있다.

〈표 1〉 AVCC에서 사용되는 변수와 의미
〈Table 1〉 Fields and their meanings in AVCC

사용되는 변수	의미
Ti.state	트랜잭션의 상태 (READY, STOPPED, BLOCKED)
Ti.stop_list	Ti가 정지 시킨 트랜잭션들의 TID와 정지를 유도한 데이터 oid들의 리스트이다. 이 리스트는 보류된 재시작을 구현하기 위해서 필요하다.
Ti.av_stopped_list	Ti가 정지된 트랜잭션일 때 Ti를 정지시킨 트랜잭션들의 TID와 정지에 사용된 데이터 oid의 리스트
Ti.stop_cnt	Ti에 의해서 정지된 트랜잭션의 수
Ti.av_stopped_cnt	Ti를 정지시킨 트랜잭션의 수
Ti.parent	Ti의 부모 트랜잭션을 가리키는 포인터
Ti.child	Ti의 자식 트랜잭션을 가리키는 포인터

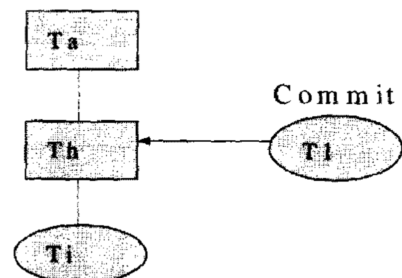
3.1.1 Lock acquire 함수

한 트랜잭션이 데이터에 접근하려고 할 때 먼저 잠금의 호환성과 부모/자식의 관계를 검사 해야 한다. 만약 잠금을 요구하는 높은 우선 순위의 트랜잭션 Tr이 낮은 우선 순위의 트랜잭션 Th가 가지고 있는 데이터를 요구하는 경우 1) Th는 정지되고, 2) Tr은 필요한 데이터를 가진 후 계속 수행을 하며, 3) Th의 즉시 재시작 버전인 Ti가 만들어져서 수행을 한다.

```
Procedure Lock_acquire(Tr, oid, lockmode)
{
  if (lockmode is NOT compatible)
  {
    IF(Th is Tr's ancestor)
      Lock_granted(Tr,oid,lockmode);
    ELSE
    {
      IF (Pr(Tr) is greater than or equal to Pr(Th))
      {
        IF (Th.state is NOT STOPPED)
        {
                 stop Th and generate Ti which is
        restarted version of Th;
          Add STOPPED flag to Th;
        }
        /* Now it is stopped */
        Add oid, and TID of Th to Tr's stop_list;
        Add oid, and TID of Tr to Th's av_stopped_list;
        Change lock owner of oid from Th to Tr;
      }     ELSE
        block(Tr);
    }
  }
  ELSE /* compatible */
    lock_granted(Tr, oid, lockmode);
} /* End of procedure */
```

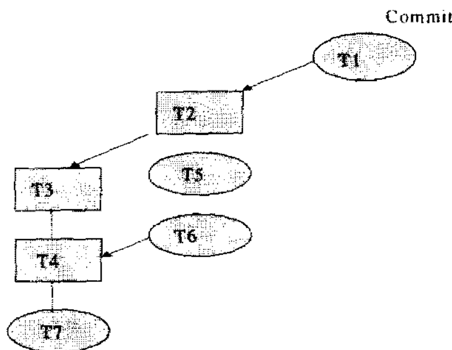
3.1.2 Commit 함수

이 시점에 트랜잭션의 지역 메모리 영역에 있는 데이터를 디스크로 출력하며 더 이상 사용되지 않는 정지된 버전들을 전부 제거한다. 트랜잭션 T1이 (그림 8)과 같은 상태에서 정상 종료를 하면, T1은 T1의 stop_list를 이용하여 T1에 의해서 정지된 트랜잭션과 그 트랜잭션들의 조상(ancestor) 트랜잭션들을 제거한다. 따라서 (그림 8)에서 T1이 정상 종료를 하면 Th



(그림 8) 트랜잭션 정상 종료
(Fig. 8) Transaction commit

와 Ta가 시스템에서 제거된다. (그림 9)는 연속 정지(chain stop) 상황을 보여준다. 트랜잭션 T2가 T3를 데이터 D1 때문에 정지시켰으며 얼마 뒤에 T1이 같은 데이터 D1에 접근하기 위해 T2를 정지시킨 경우이다. 이런 경우 T1이 정상적으로 종료를 하면 T2와 T3은 시스템에서 즉시 제거된다.



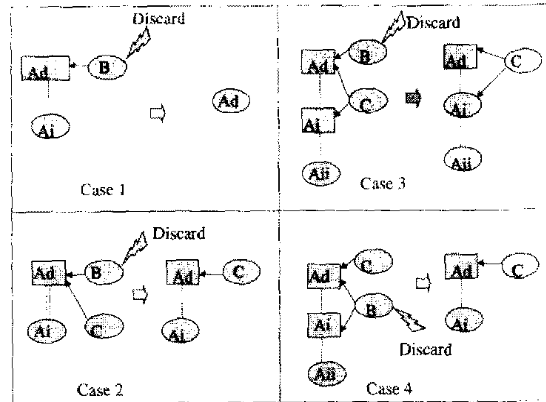
(그림 9) 트랜잭션의 연속 정지
(Fig. 9) Transaction chain stop

```
Procedure Commit(T1)
{
  Make local copies global;
  if (T1.stop_cnt is greater than zero)
  {
    FOREACH (Ti in T1.stop_list)
    {
      check chain stop;
      remove all stopped versions in T1.stop_list and
their ancestors;
      remove all locks held by T1;
    }
  }
}
```

3.1.3 Discard 함수

이 함수는 다음과 같은 경우에 호출된다.

- o 어떤 트랜잭션이 마감시간을 지키지 못해 시스템에서 제거되는 경우
- o 어떤 트랜잭션이 정상 종료하면서 그 트랜잭션의 stop_list에 있는 정지된 트랜잭션을 제거하는 경우
- o 어떤 트랜잭션의 즉시 재시작 버전이 즉시 재시작을 만든 트랜잭션의 제거로 인해 시스템에서 제거되는 경우 (그림 10)은 한 트랜잭션이 마감시간을 지키지 못해 시스템에서 제거될 때 트랜잭션들의 관계 변화를 설명하고 있다.



(그림 10) 트랜잭션 제거에 의한 관계 변화
(Fig. 10) Changes of transaction relationship after discard

```
Procedure Discard(Ta)
{
  if (Ta.stop_cnt is greater than zero)
  {
    FOREACH (Ti in Ta.stop_list)
    {
      Change lock owner of oid, which caused stop of Ti
      from Ta, to Ti
      and wake up transactions blocked due to oid;
      Delete Ta from the av_stopped_list of Ti;
      if (Ti.av_stopped_cnt is zero)
      {
        if (Tc.state is STOPPED)
          Remove Tc and adjust parent/child field of
          Tc's parent and child;
        else
        {
          Remove Tc and adjust child field of Tc's
          parent;
          Resume Ti;
        }
      }
    } /* end of foreach */
  } /* stop_cnt is NOT zero */
  Remove all locks held by Ta;
  Remove all data structure of Ta from the system;
}
```

3.2 직렬화에 대한 증명

이 절에서는 제안하는 방법의 정확성을 증명하기 위해 다음과 같은 정리를 증명하고자 한다.

Theorem 1 AVCC는 직렬성을 유지한다.

Proof: 어떤 히스토리 H가 직렬성을 유지한다는 것을 증명하기 위해서는 직렬 그래프 SG(H)에 사이클이 없다는 것을 보여야 한다. T_1^n과 T_2^m를 AVCC에 의해서 만들어진 히스토리 H에 있는 2개의 정상 종료된 트랜잭션이라고 하자 (여기서 아래 첨자는 트랜잭

션 식별자이며 위 첨자는 그 트랜잭션의 버전 번호를 말한다). 만약 SG(H)에 $T_1^n \rightarrow T_2^m$ 의 에지(edge)가 존재한다면 서로 충돌을 일으키는 연산 p와 q가 존재해서 $q_1^n[x] < p_2^m[x]$이다 (여기서 a<b 는 a가 b보다 먼저 수행된 연산임을 나타낸다).

1) IF $Pr(T_1^n) > Pr(T_2^m)$,

$q_1^n[x] < \ldots < c_1^n < p_2^m[x] < \ldots < c_2^m$

case 1: 만약 T_1^n이 수행 중 한번도 정지되지 않았다면 T_1^n은 정상 종료 시점에 가지고 있던 모든 잠금을 해제할 것이다. 따라서 T_2^m은 데이터 x에 T_1^n이 정상 종료 하기 전까지는 접근할 수 없다.

case 2: 만약 T_1^n이 데이터 x 때문에 T_h^l에 의해서 정지되었다면 T_h^l은 T_1^n보다 높은 우선 순위를 가지고 있다. 그 동안에는 T_h^l이 데이터 x에 대해서 잠금을 가지고 있으므로 T_2^m이 데이터 x에 접근 할 수 없다. T_h^l이 시스템에서 제거된 후 (T_1^n이 정상 종료하기 위해서는 T_h^l이 반드시 시스템에서 제거되어야 한다) 데이터 x에 대한 잠금은 T_1^n에게 다시 전달되므로 T_2^m이 T_1^n이 정상 종료하기 전에는 데이터 x에 접근 할 수 없다.

case 3: 만약 T_1^n이 x와는 다른 데이터 y 때문에 T_h^l에 의해서 정지되었다면 T_h^l은 T_1^n보다 높은 우선 순위를 가지고 있다. 그 동안에는 T_1^n이나 T_1^n의 자손중의 하나인 T_1^{n+l}이 데이터 x에 대한 잠금을 가지고 있으므로 T_2^m이 데이터 x에 접근할 수 없다. T_1^{n+l}이 종료하면서 데이터 x에 대한 잠금이 T_1^n에게 다시 전달 되므로 T_2^m이 T_1^n이 정상 종료하기 전에는 데이터 x에 접근할 수 없다.

2) If $Pr(T_1^n) < Pr(T_2^m)$, $q_1^n[x] < \ldots$

$< c_1^n < p_2^m[x] < \ldots < c_2^m$

만약 p와 q가 T_1^n과 T_2^m사이의 첫 번째 충돌되는 연산이었다고 가정하자. 만약 $p_2^m[x]$가 c_1^n보다 먼저 나타났다면 T_2^m이 T_1^n보다 우선 순위가 높고 T_2^m은 정상 종료한 트랜잭션이기 때문에 T_1^n은 정상 종료할 수가 없었을 것이다.

위의 2가지 결과를 염두에 두고 다음 시나리오를 가정해보자. 먼저 SG(H)에 $T_1^n \rightarrow T_2^m \rightarrow \ldots \rightarrow T_n^k \rightarrow T_1^n$와 같은 사이클이 있다고 생각해보자.

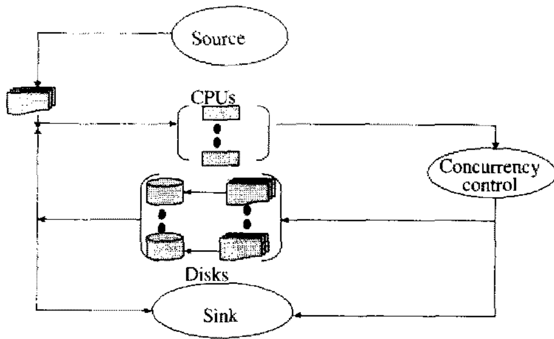
case 1: 만약 $Pr(T_1^n) < Pr(T_n^k)$이면 $T_1^n \rightarrow T_2^m \rightarrow \ldots \rightarrow T_n^k$는 $c_1^n < c_n^k$를 나타내고 있으며 $T_n^k \rightarrow T_1^n$는 $c_n^k < c_1^n$를 나타내고 있다. 이것은 모순이므로 일어나지 않는다.

case 2: 만약 $Pr(T_1^n) > Pr(T_n^k)$이면 $T_1^n \rightarrow T_2^m \rightarrow \ldots \rightarrow T_n^k$는 $c_1^n < c_n^k$를 나타내고 있으며 $T_n^k \rightarrow T_1^n$는 $c_n^k < c_1^n$를 나타내고 있다. 이것은 모순이므로 일어나지 않는다.

따라서 AVCC가 만드는 SG(H)에는 사이클이 존재하지 않으므로 AVCC는 오직 직렬성을 보장하는 히스토리만 만들어 낸다.□

4. 성능측정

AVCC의 성능을 비교하기 위하여 SIMPACK[5] 시뮬레이션 패키지를 이용하여 실시간 트랜잭션 스케쥴러의 시뮬레이터 (그림 11)를 구현하였다. 이 실험에서는 다중 프로세서와 다중 디스크를 가정하였다. 다중 프로세서는 공동으로 사용하는 1개의 큐를 가지며 우선 순위를 바탕으로 하는 선점-재개(preemptive-resume) 방식을 사용한다. 디스크의 경우 각 디스크는 각각의 독립적인 큐를 가지며 그것들은 비선점 방식인 도착 우선 순위 우선 방식을 사용한다.



(그림 11) 시뮬레이션 모델
(Fig. 11) Simulation Mode

이 시뮬레이션에 사용된 데이터는 〈표 2〉와 같고 트랜잭션들은 시스템에 포아송 형태 (Poisson Process)의 도착율 (도착 트랜잭션들 사이의 간격은 지수함수 분포)로 도착한다. 트랜잭션들은 시스템에 들어가는 즉시 준비 상태가 된다. 트랜잭션이 수행 중 갱신할 데이터의 수는 min_size와 max_size 사이에서 균등 분포 함수로 선택되며 실제로 선택할 개개의 데이터는 db_size 값 내에서 균등하게 선택된다. 트랜잭션은 데이터에 접근한 후 cpu_time 값 만큼 데이터 처리 시간이 필요하며 그 후 다음 데이터에 접근한다.

각 트랜잭션의 마감시간은 트랜잭션의 자원시간 (접근하는 데이터의 수 * 수행시간)에 여유시간 (slack time)을 더하여 결정된다. 여유 시간은 min_slack과 max_slack에 의하여 조정된다.

$$\text{마감시간 } (Deadline) = \text{도착시간} + resource\ time \times \left(1 + \frac{slack\ percent}{100}\right)$$

데이터를 읽을 때 디스크에서 읽어야할 확률은 disk_prob에 의해서 조정된다. disk_prob를 조정함으로써 시스템 내에 버퍼가 존재하는 것을 모델링할 수 있다. 그리고 시스템의 중요한 자원인 프로세서는 no_of_cpu로, 디스크는 no_of_disk로 쉽게 조정이 가능하다.

위의 입력 데이터를 바탕으로 성능 평가를 위한 실험이 행해졌다. 성능 비교를 위한 척도로는 실시간 시스템에서 가장 많이 사용되는 트랜잭션 실패율 (transaction miss percent 즉, 시스템 내로 들어간 트랜잭션 중 마감시간 내에 종료하지 못하는 트랜잭션의 백분율)을 비교한다.

$$\text{실패율 } (Miss\ percent) = \frac{\text{실패한 트랜잭션의 수}}{\text{전체 트랜잭션의 수}} \times 100$$

이 실험에서는 도착율을 변경시키며 각 도착율의 값마다 10,000개의 트랜잭션을 수행하여 95%의 신뢰도를 얻었으며 처음 수행된 1,000개의 트랜잭션에 대한 결과는 초기의 warm-up문제를 피하기 위하여 의도적으로 제외하였다.

4.1 실험결과 분석

이 실험에서 각 방법을 구현할 때 수반되는 오버헤드는 시뮬레이션의 한계점으로 완벽히 고려되지는 않았다. AVCC를 구현할 때 생길 수 있는 오버헤드는 각 트랜잭션마다 유지하는 몇 개의 리스트와 그 리스트에 대한 메모리 내에서의 연산들이다. 이들 연산들은 실험결과에 영향을 미칠 만큼 많은 처리시간을 요구하지는 않는다. 그러나 AVCC는 1개의 즉시 재시작 버전과 여러 개의 정지 버전을 가지고 있으므로 EDF-HP (Earliest Deadline First를 사용한 2PL-HP)보다 더 많은 메모리를 요구한다. 이 메모리 문제는 우리가 급속히 떨어지는 메모리의 가격과 일반적으로 약간의 여유자원을 항상 보유하는 실시간 시스템의 특성, 마감시간 동안만 존재하는 펌 실시간 트랜잭션의 특성을 고려하면 무시될 수도 있다.

〈표 2〉 시뮬레이션 데이터
〈Table 2〉 Parameters for the simulations

Parameters	value
db_size	1000
max_size	24
min_size	8
i/o_time	20 ms
cpu_time	10 ms
disk_prob	0.5
min_slack	100 %
max_slack	650 %
no_of_cpu	8
no_of_disk	16

〈표 3〉 실험결과
〈Table 3〉 Experimental results

	도착율(trs/sec) / 종 류	10	20	30	40	50	60	70	80	90	100
Transaction miss percent(%)	EDF-HP	4.5	18	40	57	69	74	79	84	89	94
	AVCC	0.1	8	33	44	59	69	75	81	88	93

이 실험에서 우리는 트랜잭션의 도착율을 초당 10개부터 초당 100개까지로 변화시키면서 비교하였다. 우선 순위는 마감시간 우선 방식을 사용하였으며 그 실험 결과는 〈표 3〉에 있다. 〈표 3〉의 결과를 보면 아주 높은 부하를 제외한 거의 모든 시스템의 부하에서 AVCC가 EDF-HP보다 마감 시간을 넘긴 트랜잭션의 수가 감소함으로써 더 좋은 성능을 보인다. 아주 높은 부하에서는 앞의 (그림 4)와 같은 경우가 많이 생겨 낭비된 재시작을 줄일 수 있는 기회가 거의 없다. 그러나 정상 부하에서는 대부분의 경우가 (그림 1), (그림 2) 또는 (그림 3) 과 같은 경우로서 낭비적인 재시작과 낭비적인 수행을 줄일 수 있었다.

5. 결 론

실시간 데이터베이스 시스템에서 트랜잭션 스케줄링의 기능은 트랜잭션들이 마감시간 내에 수행을 종료할 수 있도록 트랜잭션에 적당한 우선 순위를 부여하는 것과 트랜잭션 사이에 데이터 접근 충돌이 생겼을 때 효과적으로 해결하는 방법을 제공하는 것이다. 실시간 DB에서 데이터 접근 충돌을 해결하는 방법으로 1) EDF-HP는 블로킹과 데이터 충돌이 생기는 즉시 우선 순위를 이용하여 즉시 재시작 방법을 사용하고 있고, 2) 낙관적 방식은 트랜잭션이 정상 종료되는 순간에 데이터베이스의 무결성을 해치는 트랜잭션을 발견하고 그 트랜잭션을 재시작 시키는 기법을 사용하고 있다. 위의 방법에서 EDF-HP는 충돌을 일으키는 트랜잭션들이 전부 정상 종료할 확률이 높은 경우 장점을 보이고 있다. 반면에 낙관적 방식은 낮은 우선 순위를 가진 트랜잭션을 재시작시킨 높은 우선 순위의 트랜잭션이 비정상적으로 종료하는 경우 장점을 보이고 있다. 따라서 본 논문에서는 *즉시 재시작* 방법과 *보류된 정지/재개* 방법을 혼합하여 사용하는 새로운 기법을 제시하여 기존에 제시된 방법들의 장점만을 가지려고 시도하였다.

본 논문에서 제시하는 이 방법은 기존의 방법들보다 더 많은 메모리 영역을 사용하여 실시간 트랜잭션의 성능을 떨어뜨리는 *낭비적인 재시작*과 *낭비적인 수행*을 줄임으로써 마감시간을 넘기는 트랜잭션의 수를 줄이려고 노력하였다. 시뮬레이션을 이용한 성능 평가를 요약하면 아래와 같다.

1. 본 논문에서 제시한 AVCC 방법은 모든 부하 영역에서 기존의 EDF-HP보다 마감시간을 넘기는 트랜잭션 수가 더 적다.

2. 본 논문에서 제시한 방법은 실시간 데이터베이스 시스템의 정상 부하 영역에서 좋은 성능을 보이고 있다. 일반적으로 실시간 데이터베이스 시스템에서의 정상 부하 영역은 시스템 내로 들어온 트랜잭션 중 마감시간을 넘기는 트랜잭션의 비율이 약 20%정도인 경우를 말하며 실제적인 실시간 데이터베이스 시스템은 하루 중 많은 시간을 정상 부하 영역에서 동작하도록 구성되어 있다.

우리가 실제 응용인 네트워크 관리 데이터베이스나 주식시장의 판매 응용을 살펴보면 시스템내의 트랜잭션은 일반 트랜잭션, 소프트 실시간 트랜잭션, 펌 실시간 트랜잭션이 혼합되어 있는 것을 알 수 있다. 따라서 실시간 데이터베이스 시스템의 스케줄러는 일반, 소프트, 펌 실시간의 3가지 종류를 전부 잘 지원하는 형태가 되어야 한다. AVCC는 일반 트랜잭션과 소프트 실시간 트랜잭션을 잘 지원하는 2PL-HP를 바탕으로 하여 펌 실시간 트랜잭션 처리기능을 강화한 형태이므로 일반 트랜잭션과 소프트 실시간 트랜잭션을 잘 지원하지 못하는 낙관적 형태의 방식에 비해서 장점을 가지고 있다.

참 고 문 헌

[1] Robbert Abbott and Hector Garcia_Molina "Scheduling real-time transactions: a perfor-

mance evaluation" in Proc. of 14th VLDB, pp.1-12,1988.

[2] Robbert Abbott and Hector Garcia_Molina "Scheduling real-time transactions with disk resident data" in Proc. of 15th VLDB, pp. 385-396, 1989.

[3] Rakesh Agrawal, Michael J. Carey and Miron Livny "Concurrency control performance modeling: alternatives and implications" ACM TODS, Vol.12, No.4, pp.609-654, 1987.

[4] A. Buchmann, D.R. McCarthy and M. Hsu "Time-critical database scheduling: A framework for integrating real-time scheduling nad concurrency control" in Proc. of 5th Data Engineering, pp.470-480, 1989.

[5] Paul A. Fishwick "SIMPACK: C-based simulation tool package", University of Florida, 1992.

[6] Marc H. Graham "Issues in real-time data management" Journal of Real-Time Systems, Vol.4, pp.185-202, 1992.

[7] Jayant R. Haritsa, Michael J.Carey and Miron Livny "Dynamic real-time optimistic concurrency control" in Proc. of Real-Time Systems Symposium, pp.94-103, 1990.

[8] Jayant R. Haritsa, Michael J.Carey and Miron Livny "Earliest Deadline Scheduling for real-time database systems" in Proc. of Real-Time Systems Symposium, pp.232-242, 1991.

[9] D. Hong, M. Kim, and S. Chakravarthy "Incorporating load factor into the scheduling of soft real-time transactions for main memory databases" in Proc. of RTCSA'96, pp.60-66, 1996.

[10] D. Hong, S. Chakravarthy and T.Johnson "Locking based concurrency control for integrated real-time database systems" in Proc. of the 1996 RTDB'96, pp.144-149, 1996.

[11] D. Hong, T. Johnson and S. Chakravarthy " Real-Time transaction scheduling: A cost conscious approach", In Proc. of 1993 ACM SIGMOD, pp.197-206, 1993.

[12] S.Chakravarthy, D. Hong and T. Johnson "Real-Time Transaction scheduling: A Framework for synthesizing static and dynamic factors" Journal of Real-Time Systems, Vol. 14, No.2, pp.135-170, 1998.

[13] Jiandong Huang, John A. Stankovic, Krithi Ramamritham and Don Towsley "Experimental evaluation of real-time optimistic concurrency control schemes" in Proc. of 17th VLDB, pp. 35-46, 1991.

[14] Woosaeng Kim and Jaidepp Srivastava "Enhancing real-time DBMS performance with multiversion data and priority based disk scheduling" in Proc. of Real-Time Systems Symposium, pp.222-231, 1991.

[15] J. Lee and Sang H. Son "Using dynamic adjustment of serialization order for real-time database systems" in Proc. of Real-Time Systems Symposium, pp.66-75, 1993.

[16] C.L Liu and J.W. Layland "Scheduling algorithms for multiprogramming in a hard real-time environment" Journal of ACM, Vol. 20, pp.46-61, 1973.

[17] N. Redding "Network Services Databases" in Proc. of IEEE Global Telecommunications Conference, pp.1336-1340, 1986.

[18] R.M. Sivasankaran, B. Purimetla, J.A.Stankovic and K. Ramamritham "Network Services Databases - A distributed active real-time database applications" in Proc. of IEEE workshop on Real-Time Applications, pp.184-187, 1993.

[19] John Voelcker "How computers helped stampede the stock market" IEEE Spectrum, Vol. 24, pp.30-33, 1987.

[20] S. Hvasshovd, O. Torbjornsem "The ClustRa Telecom Databasedases High Availability, High Throughput, and real-time response" in Proc. of 21th VLDB, pp.469-477, 1995.

홍 동 권

1985년 경북대학교 전자공학과 졸업(학사)
1992년 University of Florida 전자계산학과 졸업(석사)
1995년 University of Florida 전자계산학과 졸업(박사)
1985년~90년 한국전자통신연구원
1996년~97년 한국전자통신연구원
1997년~현재 계명대학교 컴퓨터·전자공학부 전임강사
관심분야 : 능동 실시간 데이터베이스, 병렬 처리, 성능 평가, 시뮬레이션, 멀티미디어 처리