□ 기술해설 □

주기억 장치 데이타베이스 시스템 환경하에서의 실시간 트랜잭션 관리자 설계 및 구현

인하대학교 박동선*
대림전문대학 이순조**
인하대학교 배해영***

● 목 차 ●

1. 서 론

실시간 특성을 갖는 업무의 처리를 지원하기 위한 실시간 데이타베이스 시스템에서 트랜잭션을 관리하는 목적은 시간 제약과 일치성 기준을 준수하면서 동시에 수행되는 트랜잭션의 수를 증가시키는 것으로서 기존 데이타베이스 시스템에서의 목적인 평균 응답시간 최소화와는 커다란 차이가 있다. 최근에 실시간 트랜잭션을 처리하는 데에 있어서는 디스크장치 데이타베이스 시스템(disk-based database systems)의 가장 큰 문제점인 디스크 입출력으로 인한 예측 불가능한 트랜잭션 처리 지연을 제거할 수 있으며, RAM 가격의 하락으로 인하여 어느 정도의 경제성도 갖는 주기억 장치 데이타베이스 시스템(main memory database systems)이 가장 좋은 해결 방안으로 제시되고 있다. 그러나 주기억 장치 데이타베이스 시스템은 실시간 응용업무에 대해 충분히 빠른 응답 속도를 제공할 수 있음에도 불구하고, 그들은 고장 회복과 트랜잭션 관리 및 데이타 관리 등에서 새로운 문제점을 가지고 있다[1,2,3]. 본 논문에서는 이러한 문제점 중에서 실시간 트랜잭션의 관리를 해결하기 위한 방법을 고려한다.

실시간 데이타베이스 시스템에서 실시간 트랜잭션을 관리하기 위해서는 우선순위 할당과 동시성 제어에 있어서 긴급하고 중요성이 큰 트랜잭션을 우선해서 처리하도록 하는 알고리즘을 적용하여야 한다. 현재까지, 실시간 트랜잭션 관리를 위하여 연구된 동시성 제어 알고리즘의 대부분은 주기억 장치 데이타베이스 시스템의 특성을 고려하지 못한 일반 데이타베이스 시스템에서 사용해왔던 로킹과 낙관적 기법에 기초를 두고 있다[4,5,6,7]

먼저, 실시간 데이타베이스 시스템에서 사용되어온 초기의 우선순위 할당 기법에는 ED (Earliest Deadline) 기법과 LS(Least Slack time) 기법 그리고 AED(Adaptive Earliest Deadline) 기법 등이 있다. 이들 기법은 실시간 트랜잭션의 데드라인만을 고려함으로써 중

*학생회원
**정 회 원
***종신회원

요도가 높은 트랜잭션의 처리에 문제가 있었다. 따라서 최근에는 이들의 변형인 HED(Hierarchical Earliest Deadline) 기법이 제안되었다[4,8].

동시성 제어 기법에는 일반 데이타베이스 시스템에서 사용된 2PL을 실시간 데이타베이스 시스템에 변형하여 적용한 연구로 Abott 등이 제안한 2PL-HP 알고리즘과 Agrawal 등이 제안한 2PL-OS 알고리즘 등이 있으며, 낙관적인 동시성 제어 기법을 변형 적용한 연구로는 Haritsa 등이 제안한 OPT-SACRIFICE, OPT-WAIT-50 등의 알고리즘이 있다[4,6,9,10].

지금까지 제안된 실시간 트랜잭션 관리 알고리즘들의 대부분이 전체 데이타베이스가 주기억 장치에 상주한 주기억 장치 데이타베이스 시스템의 특성을 충분하게 활용하지 못하고 있으며 또한 적용 환경이 소프트나 펌 실시간이라는 특정 환경에 대해서만 수행된 것으로서 소프트와 펌 및 하드 실시간 트랜잭션이 동시에 존재하는 실제 환경에 대해서는 미비한 점이 있다. 즉, 하드 트랜잭션은 항상 처리가 가능하도록 준비되어 있어야만 하고, 데이타에 대한 충돌 해결 방법으로서 데이타를 로크하는 방법은 실시간 데이타가 기존 데이타와는 달리 빠른 처리를 위해 어느 정도의 불일치성은 인정한다는 점을 고려하지 못하고 있다[16].

본 논문에서는 실시간 트랜잭션을 관리하기 위해서 전체 데이타베이스가 주기억 장치에 상주하고 있는 주기억 장치 데이타베이스 시스템의 특성과 실시간 데이타의 완화된 무결성을 반영하고, 하드 실시간 트랜잭션(hard real-time transaction)의 처리를 보장하면서도 펌(firm)과 소프트(soft) 실시간 트랜잭션의 처리율에 미치는 영향을 최소화하는 다중 큐(multi queue)와 이완 로크(loosely lock) 기법을 제안한다.

2. 실시간 데이타와 트랜잭션

2.1 실시간 데이타

실시간 데이타베이스 시스템을 구성하는 데이타에는 연속성 특성을 갖는 형태와 이산성 특성을 갖는 형태가 있다. 연속성 데이타는 외부 환경에 따라 연속적으로 변경되는 데이타로서, 그 데이타의 값은 센서로부터 직접 입력받거나 데이타 집합의 값을 규칙적인 주기로 계산한다. 이산성 데이타의 값은 시간이 지나도 폐기되어지지 않는다는 점에서 고정적이며, 갱신 트랜잭션이 그 값을 변경할 때까지 유효하다.

연속성 데이타는 그 데이타의 값이 상대와 절대 일시적 일치성 둘다를 만족한다면 정확한 상태에 있다고 말할 수 있는 반면에, 이산성 데이타는 그 값이 논리적 일치성을 갖는한 정확한 상태에 있다고 말할 수 있다. 각 연속성 데이타에 대해 단지 한 기록자만이 있음을 준수하고 그값을 일시적 일치성이 유지되어지는 동안만 사용한다면, 대부분의 기존 데이타베이스에서 정확성을 유지하기 위해 사용되는 트랜잭션의 직렬성과 회복성이 이러한 종류의 데이타에 대해서는 필요하지 않다[7,11].

2.2 실시간 트랜잭션

실시간 데이타베이스 시스템 내의 트랜잭션은 여러가지 속성에 의해서 특징지워진다. 첫째, 지정된 시간 제약 조건인 데드라인을 위반하는 경우 시스템에 미치는 영향도에 따라 하드, 펌, 그리고 소프트 트랜잭션으로 구분할 수 있다.

표 1 트랜잭션의 형태

형 태	특 징
타입 I 트랜잭션	• 기록전용 • 주기적 • 하드
타입 II 트랜잭션	• 판독/기록 • 주기적 • 하드 • 실행시간과 접근 데이타 사전 인지
타입 III 트랜잭션	• 판독전용 • 주기적 • 하드
타입 IV 트랜잭션	• 비주기적 • 실행시간과 접근 데이타가 항상 사전에 인지할 수 없음 • 이산 데이타 접근 • 다양한 보장 레벨
타입 V 트랜잭션	• 위의 어느 타입에도 속하지 않음 • 소프트 또는 펌 • 실행시간 항상 인지할 수 없음 • 영속성 또는 이산성 데이타 접근

하드 트랜잭션이 데드라인을 위반한다면 시스템은 큰 손실을 볼 수 있다. 그렇지만, 펌이나 소프트 트랜잭션이 데드라인을 위반하는 것은 성능이 떨어질 뿐 큰 손실은 초래하지 않는다. 둘째, 트랜잭션의 발생 형태에 의한 분류로 주기성과 비주기성 및 산발성으로 구분할 수 있다. 규칙적인 발생 시간을 갖는 트랜잭션을 주기적이라고 하고 발생 시간이 규칙적이지 않다면 비주기적이라고 한다. 일반적으로 비주기적 트랜잭션은 소프트나 펌 데드라인을 갖는다. 마지막으로 산발적(sporadic) 트랜잭션은 비주기적이면서 하드 데드라인을 갖는 트랜잭션을 말한다. 셋째, 데이타 접근 형태에 의한 분류로 미리 정의된 트랜잭션인지 무작위로 접근하는 트랜잭션인지의 여부를 구분한다. 넷째, 데이타 요구 조건에 의한 분류와 다섯째, 실행 시간 요구 조건을 인지하는지의 여부로 구분한다. 여섯째, 접근한 데이타 타입에 의한 분류로 연속성(continuous), 이산성(discrete) 또는 둘 다 접근하는지의 여부로 구분한다.

위의 특성을 고려하면 실시간 데이타베이스 적용업무에는 표 1에서 볼 수 있는 바와 같이 5가지 형태의 트랜잭션이 있다[11].

대부분의 실시간 데이타베이스 시스템은 위의 타입의 부분 집합만을 포함하는 모델을 대상으로 구현하며, 시스템 내의 트랜잭션들을 결코 구분할 수 없다. 그렇지만, 실질적으로 모든 종류의 트랜잭션은 하나의 시스템 내에 공존할 수 있다.

3. 실시간 트랜잭션 관리자 설계시 고려사항

주기억 장치 데이타베이스 시스템 환경하에서 데이타베이스에 대한 접근은 디스크장치 데이타베이스 시스템 환경하에서의 데이타베이스 접근보다 매우 빠르기 때문에 실시간 트랜잭션들이 보다 신속하게 완료될 수 있다. 이것은 로크 기법과 낙관적 기법을 이용한 기존의 동시성 제어 알고리즘의 변화를 초래한다.

먼저 로크 기법을 이용한 동시성 제어를 사용하는 시스템에서는 로크가 오래동안 유지되지 않는다는 것을 의미하며, 따라서 로크의 경쟁은 데이타가 디스크에 상주한 경우보다 중요성이 훨씬 줄어들었음을 제시한다. 또한, 로크 단위에 있어서 작은 단위(필드 또는 레코드)를 선택한 시스템은 데이타베이스가 주기억 장치에 상주함으로써 작은 로크 단위가 갖는 원래의 장점은 사라졌다고 볼 수 있다. 이러한 이유 때문에 매우 큰 로크 단위(즉, 릴레이션)가 주기억 장치 데이타베이스를 위해서 가장 적절하다고 제시되었다[12]. 극단적으로 로크 단위를 전체 데이타베이스로 선택할 수도 있다[13,14]. 이것은 트랜잭션의 수행을 순차 수행으로 간주할 수 있기 때문에 동시성 제어의 비용(로크의 확보와 해제 그리고 교착상태의 해결)을 거의 완전하게 제거할 수 있다. 그리고 낙관적 기법에서 검증을 위해 필요한 시간은 실시간 주기억 장치 데이타베이스 시스템의 트랜잭션 처리에 있어서 너무 크기 때문에, 많은 트랜잭션이 그들의 데드라인을 위반하도록 한다[5]. 또한 주기억 장치 데이타베이스 시스템에서는 기억 공간이 중요함으로 다중버전에 의한 동시성 제어 및 임시적으로 데이타 공간을 생성하는 동시성 제어 방법 등은 적합하지 않다. 따라서, 기존 디스크 장치 데이타베이스 시스템에서 효율적으로 수행되었던 접근 방법이나 질의 처리가 주기억 장치 데이타베이스 시스템에서는 더이상 효율적일 수 없을 것이다.

일반적인 데이타베이스 시스템에서 우선순위 역행 문제를 해결하기 위한 방법으로는 우선순위 상속과 우선순위에 의한 중단 그리고 조건부 우선순위 상속 등이 있다. 우선순위 상속 방법은 우선순위가 낮은 트랜잭션이 공유 자원을 사용하고 있을 때, 충돌이 발생하면 충돌이 발생한 트랜잭션 중에서 우선순위가 가장 높은 트랜잭션의 우선순위를 상속시킴으로 해서 보다 빨리 자원을 해제할 수 있도록 한다. 그리고 우선순위에 의한 중단 방법은 충돌이 발생했을 때 우선순위가 낮은 트랜잭션을 무조건 중단시키는 방법이다. 마지막으로 조건부 우선순위 상속 방법은 위의 두 방법의 장점 만을 차용한 것이다. 이 세가지 방법 중에서 주기억 장치 데이타베이스 시스템 환경하에서 적합한 방법은 우선순위 상속 기법이다. 그 이유는 주기억 장치 데이타베이스 시스템 환경하에서의

그 수행 시간이 매우 짧아진 실시간 트랜잭션에 비해 우선순위가 낮은 트랜잭션을 취소하기 위해 발생하는 취소 작업 시간이 너무 길기 때문이다.

마지막으로 고려해야할 사항으로는 기존 트랜잭션 처리의 정확성 기준인 순차성은 제한시간이 결부되어 있는 실시간 트랜잭션에 적용하기에는 너무나 엄격한 조건이라는 것이다. 순차성에 의한 트랜잭션 스케줄링의 목적은 데이타 일치성을 유지하면서 높은 동시성을 제공함으로써 트랜잭션의 평균 응답 시간을 단축하는데 있다. 그러나, 실시간 트랜잭션의 데드라인 내의 처리를 중요시 하는 실시간 데이타베이스 시스템에서 일부 데이타는 엄격한 일치성을 어느정도 완화하여야 한다.

이러한 환경에 적용되어야 하는 실시간 트랜잭션 관리자를 위해 본 논문에서 결정한 설계 기준은 다음과 같다[16].

- 발생되는 트랜잭션의 종류는 하드, 펌, 소프트, 그리고 비실시간 트랜잭션이 있다. 일반적으로 실시간 시스템에는 펌이나 소프트와 같은 특정한 종류의 트랜잭션만이 존재하는 경우는 거의 없다. 따라서, 본 논문에서 제안한 실시간 주기억 장치 데이타베이스 시스템의 트랜잭션 관리자는 모든 종류에 대한 방안을 제시한다.
- 발생되는 트랜잭션에 대한 정보를 알 수 있다고 가정한다. 대부분의 실시간 시스템에서는 그 특성상 발생되는 트랜잭션의 종류가 제한되는 경우가 대부분이므로 트랜잭션을 사전 분석하여 그에 대한 정보를 얻을 수 있다.
- 발생되는 실시간 트랜잭션의 수행 시간은 매우 짧다고 가정한다. 실시간 시스템에서 발생되는 실시간 트랜잭션은 긴 트랜잭션이 아닌 대부분이 짧은 트랜잭션이고, 대부분의 긴 수행 시간을 갖는 트랜잭션은 비실시간 트랜잭션들이다.
- 동시성 제어는 트랜잭션이 요구하는 데이타 항목 전부에 대한 로크를 획득할 때까지 수행을 보류하고 로크 해제는 트랜잭션의 마지막 연산이 처리되었을 때 전부 해제하는 Static 2PL 기법[5]을 변형한 Static 2PL-HP 기법을 적용한다. Static 2PL-HP는 수행 전까지 우선순위에 의해서 데이타 자원을 경쟁하도록 함으로써 우선순위가 높은 트랜잭션이 먼저 처리되도록 하였다. 이 기법은 교착 상태를 방지할 수 있고 취소된 트랜잭션으로 인한 다른 트랜잭션의 연속적인 취소를 방지한다.
- 기록 로크와 판독 로크 이외에 이완 로크를 제공한다. 이완 로크가 필요한 이유는 실시간 데이타 중에 일부는 정확한 데이타의 일치성 보다는 데드라인 내의 처리 속도를 중요시하기 때문이다.
- Static 2PL-HP 기법의 적용으로 인해서 발생하는 우선순위 역행 문제는 우선순위 상속 기법으로 해결한다. 이 방법을 선택한 이유는 다른 방법이 수행하고 있는 트랜잭션의 취소를 기반으로 하기 때문이다. 실시간 주기억 장치 데이타베이스 시스템에서 취소로 인해 발생되는 작업 수행 시간은 충돌이 발생한 트랜잭션을 대기하고 있는 시간보다 대부분 길기 때문이다.
- 로크의 단위는 하나의 릴레이션으로 한다. 앞에서 설명한 바와 같이 주기억 장치 데이타베이스 시스템에서 로크 시간은 디스크 입출력의 제거로 상당히 감소하였기 때문에 심각하지 않다. 따라서 작은 로크 단위는 그 장점이 사라지고 관리를 위한 정보 유지 부하만 커졌다. 따라서 실시간 주기억 장치 데이타베이스 시스템에서는 로크 단위를 크게하는 것이 적합하다.

이들 기준에 의해 설계 구현된 트랜잭션 관리자는 다음 절에서 다룬다.

4. 실시간 트랜잭션 관리자의 설계 및 구현

4.1 트랜잭션 관리 큐

주기억 장치 데이타베이스 시스템 환경하에서 발생되는 실시간 트랜잭션을 처리하기 위해 본 논문에서 제안한 시스템 모델은 그림 1과 같다.

4.1.1 대기 큐

제안된 트랜잭션 관리 시스템에서 데이타베

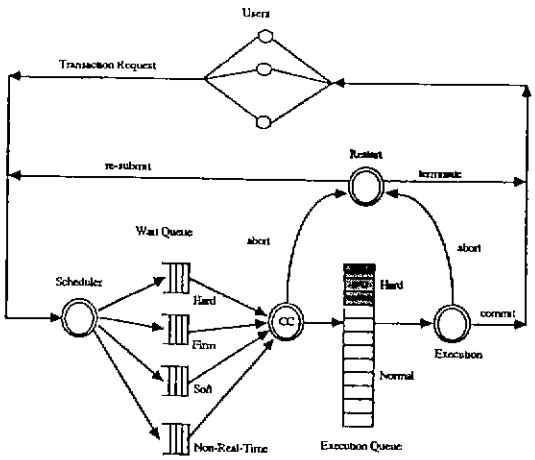


그림 1 제안 시스템 모델

이스에 연산을 요청한 트랜잭션은 트랜잭션의 종류에 따라 스케줄러에 의해서 우선순위를 할당받고 해당 대기 큐에서 대기한다. 대기 큐는 트랜잭션의 종류에 따라서 4개의 큐로 구분되어 있고, 해당 큐 내에서는 우선순위가 높은 트랜잭션이 먼저 처리되는 우선순위 대기 큐이다.

대기하고 있는 트랜잭션은 요청한 자원을 모두 할당받아야 수행 큐로 들어갈 수 있다. 이와 같이 전부(all)아니면 전무(nothing)인 Static 2PL 기법을 선택한 이유는 실시간 트랜잭션 특성상 다수개의 자원을 요구하는 경우가 드물기 때문에 이 방법을 선택하면 구현이 간단해지고 관리하기 위한 시스템 부하도 줄어든다. 또한, 교착상태 문제를 피할 수 있어 교착상태 문제로 인해 발생되는 시스템 부하를 제거할 수 있다. 기아 현상은 계속 존재하지만 실시간 시스템에서 트랜잭션의 처리는 우선순위가 높은 실시간 트랜잭션부터 처리하기 때문에 실시간 시스템에서 이 문제는 심각하지 않다.

대기 큐에서 대기하는 트랜잭션이 획득한 로크는 그 트랜잭션보다 우선순위가 높은 트랜잭션이 요청한 로크와 충돌하면 로크하고 있던 해당 자원을 넘겨준다. 그러나 수행 큐로 들어가면 우선순위가 높은 트랜잭션과 충돌해도 자원을 넘겨주지 않는다. 이와 같은 방법을 사용한 이유는 수행 중인 트랜잭션의 자원을 넘겨받음으로 해당 트랜잭션이 수행했던 작업을 취소하는데 수반되는 시스템 부하를 제거하기 위함이다.

4.1.2 수행 큐

트랜잭션 수행 큐는 모든 자원을 할당받은 트랜잭션이 실제 수행을 위해서 대기하고 있는 큐이다. 일단 수행 큐에 들어오면 자신 보다 높은 우선순위를 갖은 트랜잭션에 의해서 취소되지 않는다. 즉, 수행이 보장된다. 제안된 수행 큐의 크기는 무한정 확장시키지 않고 유한개로 한정한다. 그 이유는 너무 많은 트랜잭션이 수행 큐에서 대기함으로써 대부분의 실시간 트랜잭션이 자신에게 부여된 데드라인을 초과하는 일을 제거하기 위함이다. 그리고 유한 개의 큐에서 일부는 트랜잭션에 할당하지 않고 하드 트랜잭션을 위해 여분으로 유지한다. 이렇게 함으로써 하드 실시간 트랜잭션과 같은 긴급하고 중요한 트랜잭션이 시스템에서 발생했을 경우 여분의 큐가 없음으로해서 수행되지 못하는 상황을 제거한다. 따라서, 일반적인 트랜잭션과 소프트 및 펌 트랜잭션에 대한 수행 큐 할당은 Normal 큐에 하고 하드 트랜잭션도 Normal 큐의 여분이 있을 경우는 Normal 큐를 할당한다.

이외에 시스템의 스케줄러가 하는 역할은 대기 큐에 대기하고 있는 트랜잭션들을 주기적으로 검사하여 데드라인 내에 수행이 가능한가를 판단한다. 만일 데드라인 내의 수행이 불가능하다면 해당 트랜잭션을 취소한다. 이렇게 함으로써 의미없는 수행으로 인한 자원의 낭비를 방지한다.

4.2 동시성 제어기

동시성 제어기의 역할은 충돌이 발생하는 트랜잭션들의 자원 할당 문제를 해결하여 데이타베이스의 일치성을 유지하는 것이다. 그러나 실시간 데이타베이스 시스템을 구성하고 있는 데이타 중에는 어느 정도의 불일치성을 인정하는 데이타가 있다. 즉, 연속성을 가진 데이타는 트랜잭션의 수행시점에 어느 정도의 불일치성은 감수하고 처리 속도를 빠르게 해도 문제가 없다. 이러한 데이타에 대한 로크를 기존 동시

성 기법에서 지원되는 로크 기법을 사용하면 시스템 전체의 동시성은 저하되고 처리 속도도 지연될 것이다. 따라서 본 논문에서는 이러한 데이타 유형을 위해 이완 로크(loosely lock)를 지원하는 동시성 제어 기법을 제안한다. 다음 표 2는 이완 로크를 반영하였을 경우의 로크 양립성 테이블이다. 기존에 제안된 공유 로크와 배타 로크는 표 2의 판독 로크와 기록 로크에 해당된다.

표 2 이완 로크를 반영한 양립성 테이블

		Lock Already Set by Ti			
		Read	Write	L_Read	L_Write
Lock Request by Tj	Read	Y	N	Y	N
	Write	N	N	N	N
	L_Read	Y	Y	Y	Y
	L_Write	Y	N	Y	N

제안된 동시성 제어 기법에서 기록 로크와 판독 로크는 기존 Static 2PL에서와 마찬가지이고 다른 부분은 이완 판독 로크와 이완 기록 로크 부분이다. 만일 트랜잭션 T_i가 자원에 이완 판독 로크를 했을 경우, 그 자원을 요청하는 모든 트랜잭션의 로크를 허용한다. 그리고 트랜잭션 T_i가 자원에 이완 기록 로크를 했을 경우, 그 자원을 요청하는 모든 트랜잭션은 해당 트랜잭션의 우선순위에 관계없이 판독 로크와 이완 판독 로크를 허용한다. 이렇게 함으로써 연속성을 갖는 실시간 데이타에 대한 동시성을 증가시킬 수 있다.

동시성 제어기의 또다른 역할은 본 논문에서 적용한 Static 2PL-HP의 문제점인 우선순위 역행 문제를 해결하는 것이다. 제안된 트랜잭션 관리기에서의 자원 경쟁은 대기 큐에서 이루어지기 때문에, 발생되는 우선순위 역행 예는 그림 2와 같다.

그림 2에서 트랜잭션의 우선순위는 TL < TM < TH의 순이다. 트랜잭션 TL이 자원 S2를 로크하고 있고 있을 때, 트랜잭션 TH와 TM이 TL이 로크하고 있는 자원 S2를 요구한다. 그러나 TH는 S1과 S2를 요구하고 있기 때문에 TL이 완료되어 자원 S2의 로크를 해제하였을 경우 S1에 대한 로크를 획득하지 못

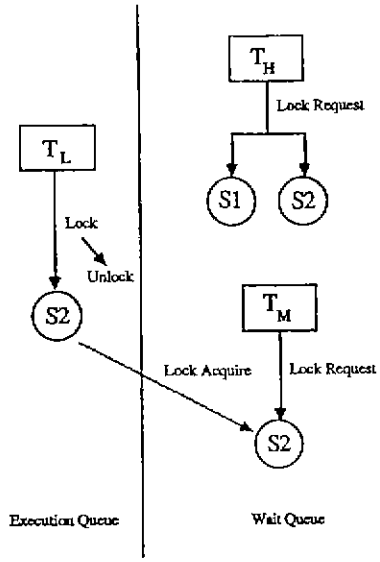


그림 2 우선순위 역행의 예

했다면 자원 S2는 TH에 할당되지 않는다. 즉, 제안된 트랜잭션 관리자는 전부 아니면 전무인 방법을 선택하기 했기 때문에 자원 S2는 TM에 할당된다. 이것은 우선순위가 낮은 TM에 의해서 우선순위가 높은 TH가 대기하게 되는 우선순위 역행을 발생한다. 따라서, 제안된 트랜잭션 관리자에서는 충돌이 발생한 자원에 대해서 우선순위가 높은 트랜잭션의 우선순위를 현재 자원을 로크하고 있는 트랜잭션에 상속하여 이 문제를 해결하였다. TL은 완료후 자신에게 상속된 우선순위를 갖는 트랜잭션에게 자원을 할당한다. 이 방법은 위에서 언급한 우선순위가 높은 트랜잭션이 우선순위가 낮은 트랜잭션을 취소함으로써 발생하는 시스템 부하의 문제점을 제거하면서 우선순위 역행 문제를 해결했다.

트랜잭션 관리기는 이상에서 언급한 기능 이외에 수행이 끝난 트랜잭션이 데드라인 내에 처리되었는지를 검사하여 완료시킬 것이지 아니면 취소시킬 것인지를 결정한다. 그림 1에서 재시작 루틴은 취소된 트랜잭션을 재요청할 것인지 아니면 종료시킬 것인지를 결정한다. 대부분의 재요청되는 트랜잭션은 비실시간 트랜잭션일 것이다. 실시간 트랜잭션은 부여된 시간제약으로 인하여 재요청해도 데드라인을 벗어나기 때문에 재요청의 의미가 없다.

4.3 제안 기법의 검증

제안된 트랜잭션 관리자에서 적용하고 있는 Static 2PL-HP 기법은 트랜잭션이 요구하는 모든 데이타 자원을 확보할 경우 수행이 시작되고, 수행이 완료된 후에야 모든 데이타 자원을 해제하는데, 수행 전까지는 우선순위가 높은 트랜잭션에게 데이타 자원을 양도하는 기법을 사용한다. 따라서 이 기법은 수행하고 있는 트랜잭션의 자원에 대해서는 비선점(non-preemptive)이다. 다음 정리는 본 논문의 제안 기법에 대한 검증[7,15] 내용이다.

정리 1. 프로세서 이용률이 1보다 큰 트랜잭션 집합에 대해서는 모든 실시간 트랜잭션의 데드라인을 보장하는 스케줄링을 할 수 없다.

증명 : 각각의 트랜잭션이 시간 구간 [0, P]에서 최대의 비율 (maximum rate)로 수행을 요청한다고 하자. 트랜잭션 Ti의 수행시간이 Ci이고 주기가 P_i일때 $P=P_1P_2\cdots P_n$라고 하고, 프로세서의 이용률이 1보다 큰 것으로부터

$$\frac{C_1}{P_1}+\frac{C_2}{P_2}+...+\frac{C_n}{P_n}>1 \qquad (1)$$

을 얻는다. 이것은

$$P_1P_2...P_n\frac{C_1}{P_1}+P_1P_2...P_n\frac{C_2}{P_2}+P_1P_2...P_n\frac{C_n}{P_n}>P_1P_2...P_n \qquad (2)$$

와 같다. 결국, 시간 구간 [0, P]에서 트랜잭션들이 필요로 하는 시간의 총량이 사용 가능한 프로세서보다 크다는 뜻이 되므로 데드라인을 넘기지 않는 스케줄링은 존재하지 않는다.

정리 1을 검토해 볼 때 시스템 부하가 $\sum_{i=1}^{n}C_i/P_i>1$인 경우에는 데드라인 내에 처리되지 못하는 트랜잭션이 발생되기 때문에 이를 위한 해결책이 제시되어야 한다. 본 논문에서 제안한 기법은 이를 위한 해결책으로서 수행 큐의 개수 n을 $\sum_{i=1}^{n}C_i/P_i\leq 1$이 되도록 정하여 과부하로 인한 연속적인 데드라인 위반 문제를 제거하였다. 물론, 적용한 기법은 기아 현상이 발생하지만 기아 상태가 되어 데드라인을 위반하는 트랜잭션은 우선순위가 낮은 트랜잭션으로 우선순위가 높은 트랜잭션의 처리를 우선으로 하는 실시간 시스템에서는 적합한 방법이라고 볼 수 있다.

정리 2. 제안된 기법은 모든 충돌에 대해서 충돌 직렬성을 유지한다.

증명 : 트랜잭션 집합 $T=\{T_1, T_2, ..., T_n\}$에서 임의의 트랜잭션 T_i, T_j를 구성하는 연산이 각각 $\{O_{i1},O_{i2},...,O_{im}\}$, $\{O_{j1},O_{j2},...,O_{jm}\}$이라고 할 경우, 우선순위에 의해 경쟁하는 Static 2PL-HP에서 트랜잭션의 수행은 $\{O_{i1},O_{i2},...,O_{im}\}<\{O_{j1},O_{j2},...,O_{jm}\}$ 또는 $\{O_{j1},O_{j2},...,O_{jm}\}<\{O_{i1},O_{i2},...,O_{im}\}$의 순서에 의해서 수행되거나 두 트랜잭션 집합 내에 충돌되는 연산이 없다면 $\{O_{i1},O_{i2},...,O_{im}\}=\{O_{j1},O_{j2},...,O_{jm}\}$, 즉 동시에 수행 가능하다. 따라서 이 기법에서의 트랜잭션 수행은 각 연산간의 상호 간섭이 없기 때문에 원자성이 보장되며 그 수행 결과는 일관된 데이타베이스의 결과를 생성한다. 따라서 제안된 기법은 모든 충돌에 대해서 충돌 직렬성을 유지할 수 있다.

정리 3. 제안된 기법은 교착 상태가 발생하지 않는다.

증명 : 교착 상태는 다음과 같은 대기 사이클이 존재하여야만 발생한다.

T1→T2→···→Tn→T1 (대기 사이클)

본 논문에서 제안한 트랜잭션 관리 기법에서 Ti→Tj인 상황이 발생하는 경우는 Tj가 수행 큐에서 수행 중이거나 Tj가 우선순위가 높은 경우이다. 그렇지만 후자의 경우에도 수행이 시작되기 전까지는 실질적으로 자원에 로크하는 것이 아니기 때문에 Tj보다 우선순위가 더 높은 트랜잭션이 항상 자원에 대한 로크 우선권을 갖는다. 따라서 교착 상태의 원인이 되는 대기 사이클이 제안된 기법에서는 발생하지 않는다.

4.4 구성 알고리즘

```
알고리즘-1] AssignPriority
    트랜잭션의 우선순위를 할당한다.
        Input  : 트랜잭션 레코드
        Output : void
        Method :
        AssignPriority(TRANSREC trans)
        {
         FeasibleTest(trans); /* 도착한 트랜잭션이 타당한가를 검사 */
```

```
        /* 트랜잭션의 타입에 따라 우선순위 할당 */
        switch (trans->type) {
        case HARD   :
                        assignHardTrans(trans);
                        break;
        case FIRM   :
                        assignFirmTrans(trans);
                        break;
        case SOFT   :
                        assignSoftTrans(trans);
                        break;
        case NORMAL :
                        assignNormalTrans(trans);
                        break;
        }
    }
```

알고리즘-2] FeasibleTest
트랜잭션 수행의 타당성을 검사한다.

```
    Input  : 트랜잭션 레코드
    Output : integer
     트랜잭션이 데드라인을 초과하는 경우      - OVERDEADLINE  0
     트랜잭션이 데드라인 내에 처리가능한 경우 - SUCCESS       1
    Method :
    FeasibleTest(TRANSREC trans)
    {
      TIME                  currentTime;
      CurrentTime = getTime();
      if(trans->deadline < trans->exectime + CurrentTime)
              /* 트랜잭션이 데드라인을 초과 */
              return OVERDEADLINE;
              /* 데드라인 내에 처리가 가능 */
              else    return SUCESS;
    }
```

알고리즘-3] WaitQueueMng
트랜잭션 대기 큐를 관리한다.

```
    1. 클라이언트 프로세서로부터 메시지를 받는다.
    2. 대기 큐에 트랜잭션을 우선순위를 고려하여 삽입한다.
    Input  : void
    Output : void
    Method :

    WaitQueueMng( void )
    {
      int           wsize;
      MSGTRANS      msgTrans;
      classMsg      typelMsg( MSG_TYPE1_KEY );

      wsize = sizeof(MSGTRANS) - sizeof(long);

      /* 트랜잭션 대기 큐 관리 루틴 */
      for(;;) {
    if( typelMsg.receive( &msgTrans, wsize )<0 ) {
              exit( 0 );
    }
              /* 트랜잭션을 디스패치 시킴 */
    Dispatch( &msgTrans );
              /* 트랜잭션 대기 큐의 카운트를 증가시킴 */
    WaitCntSem.V();
      }
    }
```

알고리즘-4] WaitQeueMng
트랜잭션의 우선순위를 기준으로 트랜잭션을 대기 큐에 삽입한다.

```
    Input  : 트랜잭션 우선순위, 트랜잭션 정보
    Output : SUCCESS - 1
             FAIL    - 0
    Method :
    InsertTransWQ( int priority, TRANSREC trans)
    {
      TRANSLIST   *ins, *tmp;
      int num;
      /* 동기화를 위한 세마포 설정 */
      P();
      /* 사용하지 않는 리스트를 얻는다 */
      ins = getFreeRec();
      /* 할당된 리스트의 값 설정 */
      ins->elementLtransRec = trans;
      num = head->numTrans++;
      /* 트랜잭션의 우선순위를 기준으로 트랜잭션을 삽입 */
      if( head->numTrans == 1 ) {
              /* 트랜잭션을 큐의 맨처음에 삽입 */
              insertFirstWQ(ins);
      } else {
    for( tmp = head->First; tmp != NULL; tmp = tmp->next ) {
    if( priority > tmp->key ) {
                      /* 트랜잭션을 큐의 중간에 삽입 */
                      insertTransWQ(ins);
    }
    if( tmp->next == NULL ) {
              /* 트랜잭션을 큐의 마지막에 삽입 */
                      insertLastWQ(ins);
    }
      }
      V();
      /* 동기화를 위한 세마포 해제 */
      return 1;
    }
```

알고리즘-5] DeleteTransWQ
대기 큐에 대기 중인 트랜잭션중에 우선순위가 가장 높은 트랜잭션을 큐에서 제거한다.

```
    Input  : void
    Output : integer
                SUCCESS - 1
                FAIL    - 0
    Method :
    DeleteTransWQ(void)
    {
      TRANSLIST *del;
      int num;

      /* 동기화를 위한 세마포 값 설정 */
      P();

      del =  head->Current;
      if( del==NULL ) return 0;

      /* 대기 큐에서 해당 트랜잭션을 제거 */
      if( del->prev==NULL ) head->First     = del->next;
      else                  del->prev->next = del->next;
      if( del->next==NULL ) head->Last = del->prev;
      else                  del->next->prev = del->prev;

      /* 제거된 트랜잭션의 리스트를 자유영역으로 되돌려줌 */
      listFree(del);
      num = head->numTrans--;

      /* 동기화를 위한 세마포 값 해제 */
      V();

      return 1;
    }
```

알고리즘-6] InsertHoldLock
WaitQueue에 있는 트랜잭션이 사용하는 자원의 목록을 Hold Lock Table에 삽입한다.

```
    Input  : TRANSREC     *pTrans
    Output : integer
                SUCCESS - 0
                FAIL    - -1
    Method :
    InsertHoldLock( TRANSREC *pTrans )
    {
      RES_REC *res;
      /* 트랜잭션의 자원목록의 이상 여부 검사 */
      if( pTrans==NULL ) return -1;
      /* 트랜잭션의 자원 목록을 Hold Lock Table에 삽입 */
      for(res=oWaitL.fpReadFirst( pTrans->tid ); res!=NULL;
              res=oWaitL.fpReadNext() ) {
    if( oHoldL.fnInsertRecord( *res )!=0 ) {

          fprintf( stderr, "Error : Holdlock insert !!\n" );
              return -1;
    }
      }
      return 0;
    }
```

알고리즘-7] InsertWaitLock
Wiat Lock Table에 트랜잭션을 삽입한다.

```
  Input :
      TID       tid
      RES_REC *Res
```

```
        int       nNum
Output: integer
            SUCCESS - 0
            FAIL    - -1
Method :
InsertWitLock( TID tid, RES_REC pRes[], int nNum )
{
   int          nHashIndex, nTIDMemCell, nRESMemCell ;
        LOCK_TID    stTmpTID ;
        LOCK_RES    stTmpRES ;

        if( nNum <= 0 ) return -1;
   /*  동기화를 위한 세마포 설정 */
        P() ;

   /* Hash 값을 구함 */
        nHashIndex = fnHashFunction( tid ) ;
   /*  자유공간에서 리스트를 할당받는다 */
        nTIDMemCell = fnAlloc() ;

   /*  할당받은 리스트에 값을 대입한다. */
        stTmpTID.tid = tid ;
        stTmpTID.nNext = pWaitTbl->pHashPtr[nHashIndex] ;
        pWaitTbl->pHashPtr[nHashIndex] = nTIDMemCell ;

   /*  Hold Lock Table에 리스트를 삽입한다 */
        stTmpTID.nResLink = -1;
        for( int i = 0 ; i < nNum ; i ++ )  {
         nRESMemCell = fnAlloc() ;
         if( nRESMemCell < 0 )              {
          WriteRecord(nTIDMemCell,(void *)&stTmpTID,sizeof( LOCK_TID));
                        V();
            return( -1 ) ;
         }
         stTmpRES.stResRec = pRes[i] ;
         stTmpRES.nResLink = stTmpTID.nResLink ;
         stTmpTID.nResLink = nRESMemCell ;
        fnWriteRecord(nRESMemCell, (void*)&stTmpRES, sizeof(LOCK_RES));
        }
        V() ;
        return( 0 ) ;
}
```

알고리즘-8] DeleteWaitLock
    Wiat Lock Table에서 트랜잭션을 제거한다.
   Input :
        TID   tid
Output: integer
            SUCCESS - 0
            FAIL   - -1
Method :
DeleteWaitLock(TID tid)
{
      int   nHashIndex ;
      int   nTIDMemCell, nPreTIDMemCell, nTmpMemCell, nRESMemCell;
       LOCK_TID   stTmpTID1, stTmpTID2 ;
       LOCK_RES   stTmpRES ;

  /* 동기화를 위한 세마포 설정 */
       P() ;
   /* Hash 값을 구함 */
       nHashIndex = fnHashFunction(tid) ;
   /* 테이블에서 해당 트랜잭션을 찾는다 */
       fnFindTID(tid, nTIDMemCell, nPreTIDMemCell) ;
       if(nTIDMemCell < 0) {
        V() ;
        return(-1) ;   /* not found. */
       }
   /* 트랜잭션의 정보를 판독한다 */
       fnReadRecord(nTIDMemCell, (void *)&stTmpTID1, sizeof(LOCK_TID));
       nRESMemCell = stTmpTID1.nResLink ;

   /* 테이블에서 트랜잭션을 제거한다 */
       while(nRESMemCell >= 0) {
        fnReadRecord(nRESMemCell, (void *)&stTmpRES, sizeof(LOCK_RES));
        nTmpMemCell = stTmpRES.nResLink ;
        fnFree(nRESMemCell) ;
        nRESMemCell = nTmpMemCell ;
       }
       if(nPreTIDMemCell < 0) {
        pWaitTbl->pHashPtr[nHashIndex] = stTmpTID1.nNext ;
       } else {
        fnReadRecord(nPreTIDMemCell, &stTmpTID2, sizeof(LOCK_TID));
        stTmpTID2.nNext = stTmpTID1.nNext ;
        fnWriteRecord(nPreTIDMemCell, &stTmpTID2, sizeof(LOCK_TID));
       }
   /* 자유공간으로 돌려준다 */
       fnFree(nTIDMemCell) ;

       V();
       return(0) ;
}
```

알고리즘-9] FindRES
    Hash Table에서 해당 자원의 위치를 찾는다.
          Input :
          char *szResName
          int &nRetMemCell
        int &nPreMemCell*
Output: integer
          자원의 위치
Method :
```
FindRES(char *szResName, int &nRetMemCell, int &nPreMemCell)
{
       int   nHashIndex, nCurMemCell ;
       EXEC_REC   stTmp ;

   /* Hash 값을 구함 */
       nHashIndex = fnHashFunction(szResName) ;
       nCurMemCell = pExecTbl->pHashPtr[nHashIndex] ;

   /* 자원의 위치를 검색 */
       while(nCurMemCell >= 0) {
        fnReadRecord(nCurMemCell, &stTmp, sizeof(EXEC_REC)) ;
        if(strcmp(stTmp.stHoldRec.stResRec.szResName, szResName)==0)
            break ;   /* found. */
        nPreMemCell = nCurMemCell ;
        nCurMemCell = stTmp.nExecLink ;
       }
       nRetMemCell = nCurMemCell ;

   /* 자원의 위치 값을 리턴함 */
       return(nRetMemCell) ;
}
```

알고리즘-10] IsExecutable
    트랜잭션이 실행가능한 가를 로크 테이블을 검사한다.
Input :
          RES_REC stResRec
Output: integer
          실행가능     - 1
          충돌발생     - 0
Method :
```
IsExecutable(RES_REC stResRec)
{
       int   nMemCell, nPreMemCell, nFO1, nFO2, test ,
       static int   nLockTbl[2][2] = { { 0, 1 }, { 1, 1 } } ;
       EXEC_REC   stExecRec ;
       RES_REC   *pResRec = &stExecRec.stHoldRec.stResRec ;

   /* 동기화를 위한 세마포 설정 */
       P() ;
   /* 로크 테이블에서 자원을 사용하는 트랜잭션이 존재하는 가를 검사 */
       if(fnFindRES(stResRec.szResName, nMemCell, nPreMemCell)< 0) {
          V() ;
          /* 자원을 발견하지 못함 <-- 실행가능*/
       return(1) ;
       }
   /* 해당 자원을 사용하는 트랜잭션 발견 */
   /* 자원에 대한 Lock의 종류를 판독하여 양립성 여부를 검사 */
       fnReadRecord(nMemCell, &stExecRec, sizeof(EXEC_REC)) ;
       if(stExecRec.stHoldRec.tagDelete == 1)   {
        V();
          /* Unlock <-- 실행가능 */
        return 1;
       } else if(stExecRec.stHoldRec.tagDelete != 0)   {
        V();
          /* Lock <-- 충돌발생 */
        return 0;
       }
}
```
```

5. 성능 평가

본 장에서는 제안된 트랜잭션 관리자에 대해 성능을 SUN Sparc 10에서 평가한다. 제안된 실시간 트랜잭션 관리자는 3가지 다른 기법과 다음과 같은 상황에서 비교 분석한다. 스케줄러1은 우선순위, 트랜잭션의 종류, 그리고 큐의 구분이 없이 트랜잭션을 수행한다. 스케줄러2는 우선순위를 고려하지만 트랜잭션의 종류, 그리고 큐의 구분이 없이 트랜잭션을 수행한다. 스케줄러3은 우선순위와 트랜잭션의 종류를 고려하지만 하드 트랜잭션을 위한 큐의 구분이 없이 트랜잭션을 수행한다. 스케줄러4는 트랜잭션의 우선순위와 종류를 구분하며, 하드 트랜잭션을 위한 별도의 큐를 두는 스케줄러이다.

① 기존 로크 기법 상황에서 다중 큐 기법의 성능 비교

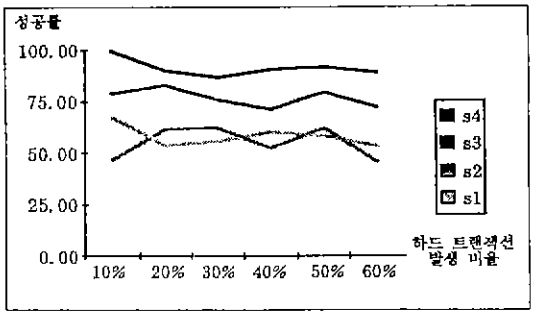
② 다중 큐 기법 상황에서 기존 로크와 이완 로크 기법의 성능 비교

5.1 평가 항목 ①의 결과

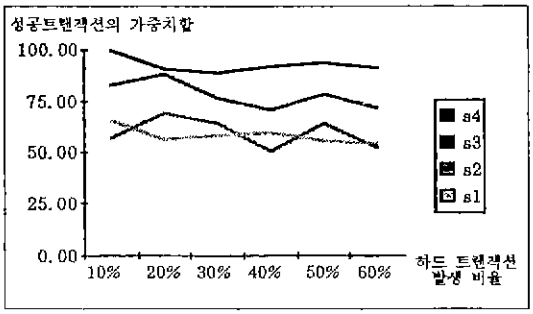
그림 3은 하드 트랜잭션의 증가에 따른 트랜잭션의 처리율과 트랜잭션 가중치의 평균을 구한 그래프이다. S1은 스케줄러1, S2는 스케줄러2, S3는 스케줄러3 그리고 S4는 스케줄러4를 나타낸 것으로, 하드 트랜잭션의 발생 비율을 10%에서 60%까지 증가시키면서 트랜잭션의 성공률과 가중치를 구한 것이다.

그림 3(a)에서와 같이 발생되는 하드 트랜잭션의 발생 비율이 적은 경우에는 제안된 S4 스케줄러가 좋은 성공률을 나타내고 있다. 또한 하드 트랜잭션의 발생 비율이 증가해도 하드 트랜잭션의 성공률이 높게 나타난다. 가중치에 대한 평가도 성공률과 비슷하게 나타난다.

그림 4부터 그림 6까지는 트랜잭션에서 판독 연산의 비율이 증가함에 따라 트랜잭션의 성공률과 가중치의 변화를 보여주고 있다. 그림 4에서는 하드 트랜잭션의 처리율을 나타내고, 그림 5는 펌 트랜잭션의 처리율을 나타내고, 그림 6은 소프트 트랜잭션의 처리율을 나타낸다. 그림에서 볼 수 있듯이 전반적으로 제

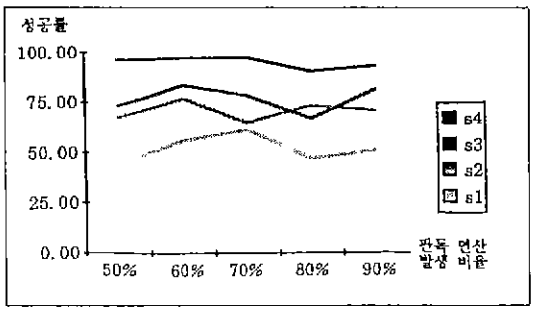


a) 하드 트랜잭션의 성공률

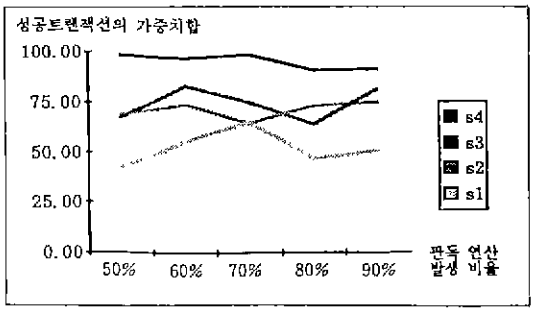


b) 하드 트랜잭션의 가중치

그림 3 하드 트랜잭션의 발생 비율에 따른 트랜잭션 처리율

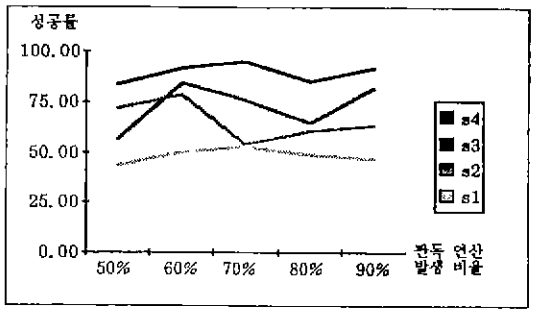


a) 하드 트랜잭션의 성공률

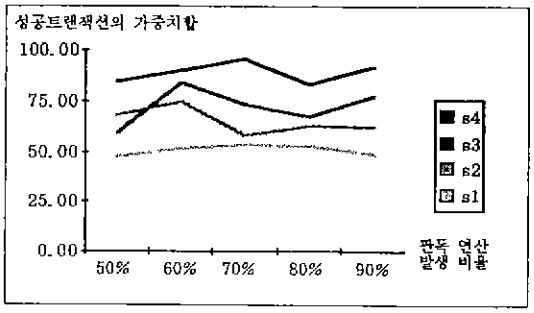


b) 하드 트랜잭션의 가중치

그림 4 판독 트랜잭션의 발생 비율에 따른 하드 트랜잭션 처리율

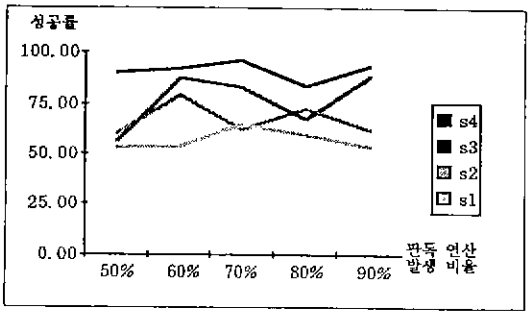


a) 펌 트랜잭션의 성공률

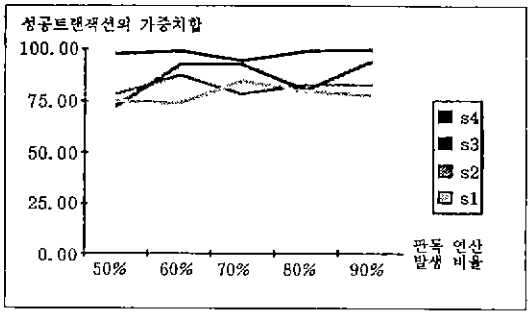


b) 펌 트랜잭션의 가중치

그림 5 판독 트랜잭션의 발생 비율에 따른 펌 트랜잭션 처리율



a) 소프트 트랜잭션의 성공률



b) 소프트 트랜잭션의 가중치

그림 6 판독 트랜잭션의 발생 비율에 따른 소프트 트랜잭션 처리율

안된 스케줄러가 성공률과 가중치가 높은 것을 볼 수 있다.

5.2 평가 항목 ②의 결과

그림 7은 다중 큐 기법 상황에서 기존 로크와 이완 로크 기법의 성능을 평가한 것이다. 그림에서 보는 바와 같이 이완 로크는 동시성을 증가시키기 때문에 처리율이 일반적으로 증가한다. 그러나 트랜잭션의 처리율 증가가 큰 차이를 보이자 않는 것은 제안 시스템에서 수행 큐를 유한개로 제한했기 때문이다.

이상의 비교 평가를 살펴보면, 본 논문에서 제안한 실시간 트랜잭션 관리자가 주기억 장치 데이타베이스 시스템 환경하에서 효율적이라는 것을 알 수 있다.

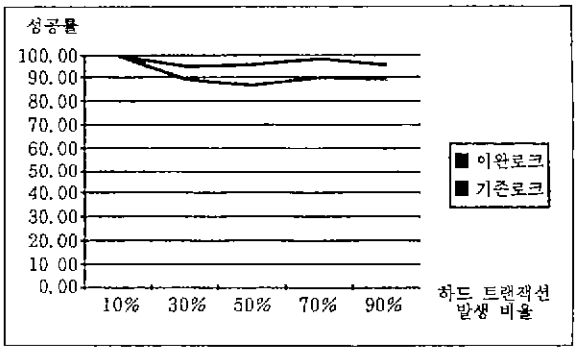


그림 7 다중 큐 기법에서 기존 로크와 이완 로크의 성능 평가

6. 결 론

본 논문에서는 실시간 트랜잭션을 관리하기 위해서 전체 데이타베이스가 주기억 장치에 상주하고 있는 주기억 장치 데이타베이스 시스템의 특성과 실시간 데이타의 완화된 무결성을 반영하기 위하여 다중 큐와 이완 로크 기법을 이용한 트랜잭션 관리자를 구현하였다. 이 기법은 기존의 실시간 트랜잭션 관리자들이 특정한 환경(소프트 또는 펌 실시간) 만을 고려하여 구현된데 반하여, 하드 트랜잭션의 데드라인 내의 처리를 보장하면서도 펌과 소프트 트랜잭션의 처리에 미치는 영향을 최소화한 기법이다. 이 기법은 일반적으로 실시간 시스템에서 하드, 펌, 소프트 트랜잭션이 공존하기 때문

에 실제 실시간 시스템 환경에 적용하는데 용이한 이점이 있다. 또한 기존 실시간 동시성 제어가 실시간 데이타의 연속성을 고려하지 않고 자원에 로크를 하기 때문에 발생하는 문제점을 이완 로크를 사용해서 해결했다. 제안된 기법의 성능은 평가를 통해 타 기법보다 우수함을 증명하였다.

향후 연구되어야 할 부분은 성능 평가에서도 볼 수 있었듯이 예측할 수 없는 트랜잭션 처리율과 같은 문제점을 해결하여야 한다. 이러한 문제점은 실시간 트랜잭션 관리자의 구현이 기존의 운영체제인 SUN OS release 4.3.1에 의존하여 수행됨으로써 운영체제 영역의 역할을 파악할 수 없기 때문에 발생한다고 본다. 따라서 실시간 주기억 장치 데이타베이스 시스템이 완벽할 수 있기 위해서는 실시간 운영체제도 같이 개발되어야 할 것으로 본다.

참고문헌

[1] A. Ammann, M. Hanrahan and R. Krishnamurthy, "Design of a Memory Resident DBMS," Proceedings of a IEEE COMPCON, pp.54-57, 1985.

[2] H. Garcia-Molina, K. Salem, "Main Memory Database Systems : An Overview," Transaction on Knowledge and Data Engineering, IEEE, Vol.4, No.6, pp.509-516, 1992.

[3] T. J. Lehman, DESIGN AND PERFORMANCE EVALUATION OF A MAIN MEMORY RELATIONAL DATABASE SYSTEM, Ph. D. thesis, University of Wisconsin-Madison, 1986.

[4] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions : A Performance Evaluation," Proceedings of the 14th International Conference on VLDB, pp.1-12, 1988.

[5] L. Gruenwald and S. Liu, "A Performance Study of Concurrency Control in a Real-Time Main Memory Database System," SIGMOD RECORD, ACM, Vol. 22, No.4, pp.38-44, 1993.

[6] J. R. Haritsa, M. J. Carey and M. Livny, "Dynamic Real-Time Optimistic Concurrency Control," Proceedings 11th Real-Time Systems Symposium, IEEE, pp.94-103, 1990.

[7] L. Shu and M. Young, CORRECTNESS CRITERIA AND CONCURRENCY CONTROL FOR REAL-TIME SYSTEMS : A SURVEY, Purdue Univ. Dept. of Computer Sciences TR # SERC-TR-131-P, 1992.

[8] J. R. Haritsa, M. Livry and M. J. Carey, "Earliest Deadline Scheduling for Real-Time Database Systems," Proceedings 12th Real-Time System Symposium, IEEE, pp.232-242, 1991.

[9] D. Agrawal, A. E. Abbadi, and R. Jeffers, "Using Delayed Commitment in Locking Protocols for Real-Time Databases," Proceedings of ACM-SIGMOD International Conference on Management of Data, pp. 104-113, 1992.

[10] D. Agrawal, et al., "Ordered Shared Locks for Real-Time Databases," The International Journal on Very Large Data Bases, Vol. 4, No. 1, pp.87-126, 1995.

[11] S. H. Son and Y. K. Kim, "Predictability and Consistency in Real-Time Database Systems," Proceedings InforScience, pp. 225-232, 1993.

[12] D. Ries and M. Stonebraker, "Locking Granularity Revisited," Transactions on Database Systems, ACM, Vol.4, No.2, pp. 210-227, 1979.

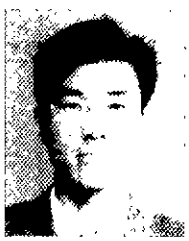
[13] K. Salem and H. Garcia-Molina, "Checkpointing Memory-Resident Databases," Proceedings 5th International Conference on Data Engineering, IEEE, pp. 452-462, 1989.

[14] J. A. Stankovic, "Real-Time Computing Systems : The Next Generation," Tutorial Hard Real-Time Systems, pp.14-37, 1988.

[15] 백 윤철, 고 건, "연속적인 미디어의 검색을 위한 비중단 실시간 태스크의 스케줄 가능성 분석," 한국정보과학회 논문지, 제21권, 제7호, pp.1252-1260, 1994.

[16] 이 순조, KORED/RT : 실시간 주기억장치 데이타베이스 관리 시스템의 설계 및 구현, 공학박사학위 논문, 인하대학교, 1995.

박 동 선



1992 인하대학교 전자계산학과 (공학사)
1994 인하대학원 전자계산학과 (공학석사)
1994～현재 인천과학아카데미 전임연구원
1996～현재 인하대학원 전자계산학과 박사과정
관심분야 : 주기억장치 데이터베이스 시스템, 멀티미디어 데이터베이스 시스템

이 순 조

1985년 인하대학교 전자계산학과 (이학사)
1987년 인하대학원 전자계산학과 (이학석사)
1995년 인하대학원 전자계산학과 (공학박사)
1995년～현재 대림전문대학 전자계산과 교수
관심분야 : 실시간 데이터베이스 시스템, 주기억장치 데이터베이스 시스템, 멀티미디어 데이터베이스 시스템

배 해 영

인하대학교 공과대학을 졸업하고 연세대학교 공학석사, 숭실대학교에서 전자계산학으로 공학박사(1989) 학위를 취득.
미국 휴스턴(Houston) 대학의 객원 교수, 인하대학교 전자계산소 부장, 소장을 역임하고 현재 인하대학교 전자계산학과에 교수로 재직중. 관심 분야로는 멀티미디어 데이터베이스 시스템, 실시간 데이터베이스 시스템, 지리 정보 시스템.